The use of Java in large scientific applications in HPC environments

Aidan Fries

ADVERTIMENT. La consulta d'aquesta tesi queda condicionada a l'acceptació de les següents condicions d'ús: La difusió d'aquesta tesi per mitjà del servei TDX (**www.tdx.cat**) ha estat autoritzada pels titulars dels drets de propietat intel·lectual únicament per a usos privats emmarcats en activitats d'investigació i docència. No s'autoritza la seva reproducció amb finalitats de lucre ni la seva difusió i posada a disposició des d'un lloc aliè al servei TDX. No s'autoritza la presentació del seu contingut en una finestra o marc aliè a TDX (framing). Aquesta reserva de drets afecta tant al resum de presentació de la tesi com als seus continguts. En la utilització o cita de parts de la tesi és obligat indicar el nom de la persona autora.

ADVERTENCIA. La consulta de esta tesis queda condicionada a la aceptación de las siguientes condiciones de uso: La difusión de esta tesis por medio del servicio TDR (**www.tdx.cat**) ha sido autorizada por los titulares de los derechos de propiedad intelectual únicamente para usos privados enmarcados en actividades de investigación y docencia. No se autoriza su reproducción con finalidades de lucro ni su difusión y puesta a disposición desde un sitio ajeno al servicio TDR. No se autoriza la presentación de su contenido en una ventana o marco ajeno a TDR (framing). Esta reserva de derechos afecta tanto al resumen de presentación de la tesis como a sus contenidos. En la utilización o cita de partes de la tesis es obligado indicar el nombre de la persona autora.

WARNING. On having consulted this thesis you're accepting the following use conditions: Spreading this thesis by the TDX (**www.tdx.cat**) service has been authorized by the titular of the intellectual property rights only for private uses placed in investigation and teaching activities. Reproduction with lucrative aims is not authorized neither its spreading and availability from a site foreign to the TDX service. Introducing its content in a window or frame foreign to the TDX service is not authorized (framing). This rights affect to the presentation summary of the thesis as well as to its contents. In the using or citation of parts of the thesis it's obliged to indicate the name of the author.

Universitat de Barcelona
Departament d'Astronomia i Meteorologia

The use of Java in large scientific applications in HPC environments

Memòria presentada per

Aidan Fries

per optar al grau de
Doctor per la Universitat de Barcelona

Barcelona, novembre de 2012

Programa de doctorat en Física

The use of Java in large scientific applications in HPC environments

Memòria presentada per

Aidan Fries

per optar al grau de
Doctor per la Universitat de Barcelona

Barcelona, novembre de 2012

Directors de la tesi:
Dr. Jordi Portell
Dr. Raül Sirvent

Tutor:
Dr. Xavier Luri

Acknowledgements

Firstly, and most importantly, I would like to acknowledge the help and advise that I received from my two advisors — Jordi Portell and Raül Sirvent. Jordi, thank you so much for all your help and encouragement during my PhD. You have much responsibility on your shoulders, but I have never seen you in a bad mood, you always had good advice and knew the best way forward. Raül, thanks for your help and advice, it was a great benefit for me to get some guidance and input from outside of the Gaia team. You both always made time for me, and I will be forever in your debt.

Jordi T., thanks for giving me the opportunity to work on this thesis. Thanks to everyone in the Gaia team at the UB (past and present). It is a fantastic group of people — hard working, dedicated to the success of the Gaia mission, and very welcoming. Javier "el crack" Castañeda, you helped me get to know Barcelona, and you always had time to explain Gaia software to me. Marysia, we started on our PhD journeys at the same time, and it has been a pleasure to have shared some experiences with you, thanks for helping me along at times, congratulations on your achievements, and I wish you every happiness in the future. Lola, thanks for your help, and for putting up with my talent for losing boarding passes. Yago, thanks for sharing your experience and ideas. Cesca, thanks for being so nice, and eager to help. Dani, thanks for fixing every problem that Miranda and I experienced over the years! I would also like to mention Xavier, Edward, Mercè, Eva, Raul B., Nadia, Marwan, Maria M., Carme, Josep Manel, Claus, Holgar, Teresa, Oscar, and all of the other members of the Gaia *family* at the UB.

Thanks to Lennart Lindegren, and everyone else who was involved with the ELSA programme, for all your efforts. I benefited a lot from this programme. I would like to mention the ELSA fellows: Berry, Daniel, Michael, Ummi, Alex, Paola, Maya, Dagmara, Ester, Thibaut, Tenay, Luca and Mihály. It was a pleasure getting to know you all.

Thanks to everyone working within DPAC on the Gaia data reduction. It has been an honour and a privilege to have been involved with this project, and I wish it every success in the future. Thanks to everyone working at the BSC and at CESCA for the facilities and support that they provided. Thanks to Guillermo and Roberto, at the University of A Coruña for their collaboration, support and advise on Java-based high-performance data communication.

Thanks to everyone in the department of Astronomy and Meteorology, and the members of *cafedam* who have helped me or offered words of encouragement, including JR, Gaby, Jordi.

I would like to mention Ben and Julia, Sean and Mallika. I wish you lots of happiness in the future. I will always remember our runs up Montjuïc, and the view from the top.

Anna, dankon pro via apogo kaj konsolo. Vi estas tre speciala persono, vi ĉiam estos en mia koro, vi faras min volas esti bona persono. Flugi vian freak flago alta kaj fiera.

And finally, thanks to my Mum and Dad. Thanks to you, I have had many opportunities in my life, which you didn't have. The hard part was being away from you.

Acknowledgement of funding support

The author was supported as an Early Stage Researcher in the Marie Curie Research Training Network of “European Leadership in Space Astrometry” (ELSA) MRTN-CT-2006-033481. This work was also supported by the MICINN (Spanish Ministry of Science and Innovation) - FEDER through grant AYA2009-14648-C02-01 and CONSOLIDER CSD2007-00050.

Resumen

A pesar de que Java es un lenguaje de programación muy utilizado, su uso entre las comunidades científica y de Computación de Altas Prestaciones (*High-Performance Computing* o HPC) sigue siendo relativamente bajo. Uno de los factores que ha contribuido a su limitada adopción es la percepción de que ofrece menos rendimiento que los lenguajes utilizados tradicionalmente en HPC, tales como C y Fortran. Esta percepción está basada, principalmente, en el rendimiento de las primeras versiones de la *Java Virtual Machine* (JVM). Además, existe una falta de conocimiento por parte de esas comunidades de las herramientas y librerías disponibles para ayudar al desarrollo y la ejecución de aplicaciones Java en entornos HPC. En esta tesis investigamos la opción de utilizar Java para desarrollar aplicaciones científicas destinadas a ser ejecutadas en entornos HPC.

El tratamiento de datos para la misión astrométrica espacial *Gaia* es un claro ejemplo de un gran proyecto software desarrollado en Java para su ejecución en entornos HPC. La ejecución eficiente de este tratamiento de datos ha sido una de las principales motivaciones de esta tesis. A pesar de que en diversas ocasiones discutimos cosas específicas de este proyecto, la gran mayoría de esta tesis es de aplicación general, siendo pues considerada como una investigación de la utilización de Java en HPC de forma general (enfocado a la computación científica).

El rendimiento del software es siempre importante en entornos HPC, hasta el punto que suele ser la principal preocupación cuando se elige un lenguaje de desarrollo para las aplicaciones. En esta tesis, el rendimiento es uno de los criterios que utilizamos para comparar Java con otros lenguajes. Sin embargo, debe también tenerse en cuenta que hay muchos otros factores que deben ser considerados al comparar lenguajes de desarrollo, en particular cuando el lenguaje a seleccionar es para grandes proyectos de software con fases de desarrollo y producción relativamente largas. Estos factores incluyen la productividad de los programadores, el tiempo necesario para acabar el desarrollo, escalabilidad, facilidad de mantenimiento, portabilidad y robustez, entre otros.

La Computación de Altas Prestaciones o HPC es un campo en rápida evolución en términos de hardware, software, y en la magnitud de los problemas que se abordan. Los entornos HPC de memoria distribuida suelen ser los más comunes, y normalmente están formados por muchos nodos de computación conectados con una red de alto rendimiento. Típicamente todos los nodos tienen acceso a un dispositivo de almacenamiento compartido, además de su propio almacenamiento local. Entre las tendencias más significativas en HPC en los últimos años se encuentran el aumento del número de núcleos por nodo de computación, y el aumento del tamaño del conjunto de datos que debe ser procesado. Uno de los desafíos significativos en HPC consiste en garantizar que los datos estarán disponibles en un nodo cuando un núcleo esté listo para procesarlos, evitando *tiempos muertos* del núcleo y proporcionando así un alto rendimiento. Además, uno de los peligros para obtener un buen rendimiento general de una aplicación es la disminución en el rendimiento de los dispositivos de almacenamiento compartido cuando se aumenta el número de procesos concurrentes accediendo a ellos. En otras palabras,

problemas de escalabilidad de Entrada/Salida (E/S) pueden convertirse en un cuello de botella en el procesado de datos. En esta tesis presentamos una infraestructura, *DpcbTools*, para el lanzamiento, monitorización y gestión de aplicaciones Java en entornos HPC, agrupando los nodos de forma jerárquica y permitiendo una distribución eficiente y robusta de los trabajos.

El intercambio eficiente de datos es muy importante para la gran mayoría de aplicaciones en entornos HPC. Muchos entornos HPC ofrecen redes de alta velocidad y baja latencia. Sin embargo, para sacar el máximo provecho a estas redes se requiere el uso de librerías nativas. Éste es un tema en el que Java se ha quedado atrás respecto a los lenguajes tradicionalmente utilizados en HPC desde hace muchos años. En esta tesis presentamos una investigación de las opciones actuales para proporcionar una comunicaci'on eficiente de datos en aplicaciones Java para entornos HPC. El paso de mensajes (o *Message Passing Interface*, MPI) es el modelo más comúnmente utilizado para implementar la comunicación entre procesos paralelos en HPC, especialmente en entornos de memoria distribuida. En Java, el paso de mensajes se conoce como *Message Passing in Java* (MPJ). En esta tesis también investigamos algunas implementaciones de MPJ y comparamos su rendimiento.

Aparte de lo mencionado anteriormente, una de las principales contribuciones de esta tesis es un entorno *middleware* de comunicaciones para sistemas Java, al cual hemos llamado *MPJ-Cache*. Este middleware hace uso de una implementación de MPJ para obtener un alto rendimiento de la aplicación mediante el uso de las redes anteriormente mencionadas, y añade funcionalidades de precarga de datos, almacenamiento temporal (*caché*) de datos, y división y recombinación de archivos grandes. Proporciona al programador una interfaz de programación (*Application Programming Interface*, API) de alto nivel, facilitando conseguir un alto rendimiento de la aplicación y una alta productividad entre los programadores. En esta tesis comparamos el rendimiento total que se puede obtener usando nuestro middleware, contra el que se puede obtener accediendo a un dispositivo de almacenamiento compartido (GPFS) directamente durante la distribución de datos entre los nodos de un clúster. Como mostraremos más adelante, el uso de MPJ-cache proporciona un rendimiento total de hasta unos 100 Gbps.

Las aplicaciones Java se ejecutan en una JVM, que es básicamente un *entorno de ejecución administrado*. La ejecución de aplicaciones dentro de este tipo de entornos es muy diferente de la ejecución directa de código nativo, compilado previamente a su ejecución. El entorno de ejecución Java dispone de varios componentes sofisticados, incluyendo el núcleo del sistema de ejecución, un *Garbage Collector* (responsable de liberar la memoria no usada) y un compilador *Just-In-Time* (JIT). A pesar de que estos componentes añaden complejidad al entorno de ejecución, también conllevan grandes beneficios. La funcionalidad *Garbage Collection* (GC) libera a los programadores de la responsabilidad de manejar la memoria explícitamente en el código, permitiendo un código más limpio, más seguro y unas aplicaciones más robustas, reduciendo además el tiempo de desarrollo. El compilador JIT permite una amplia gama de optimizaciones, incluyendo algunas que usan información obtenida después de cierto tiempo de ejecución de la aplicación, no disponibles para aplicaciones que hayan sido compiladas antes de la ejecución.

Las JVM modernas intentan proporcionar un alto rendimiento sin necesidad de que el usuario deba especificar parámetros concretos de configuración (o *tuning*). Sin embargo, en algunas situaciones, los usuarios podrían necesitar afinar algunos de estos parámetros de la JVM para satisfacer ciertas características y requisitos particulares de una aplicación o de un entorno de ejecución. La obtención de un perfil de la aplicación durante su ejecución ayuda enormemente a ese fin. En el caso del *tuning* del GC, un elemento de particular interés es el patrón de uso de memoria de la aplicación. Para el caso del compilador JIT, saber el número de veces que

se ejecuta cada método es muy útil. Para todo esto se requiere una alta comprensión de los efectos que las distintas opciones de *tuning* disponibles tienen sobre la aplicación, así como el modo en que dichas opciones pueden complementarse o interferir unas con otras. En esta tesis describimos una metodología que puede ser utilizada para explorar el espacio de posibles opciones útiles de *tuning* que puedan mejorar el rendimiento de nuestra aplicación, incluyendo un método que trata la aplicación como una caja negra, y otro que usa un patrón del uso de memoria que lleva a cabo la aplicación. En este ámbito presentamos una nueva herramienta llamada *Memory Usage Model* (Modelo de Uso de Memoria, MUM), que puede ser utilizada para modelar el patrón de uso de memoria de cualquier aplicación y así facilitar al usuario la selección de las distintas opciones de ejecución de la JVM.

Otro tema tratado en esta tesis es el *profiling* o análisis detallado del rendimiento de nuestra aplicación, lo que proporciona una comprensión del patrón de uso de la memoria, los métodos que utilizan más tiempo de procesador, o los tiempos muertos causados por E/S o por el propio Garbage Collector. El conocimiento adquirido puede ser utilizado para optimizar el código de la aplicación o para refinar (con el *tuning* previamente mencionado) el entorno de ejecución administrado de Java, el *garbage collector* y el compilador JIT. Las JVM más recientes ponen a disposición del usuario una enorme cantidad de información para su inspección. Así mismo existen muchos programas de *profiling* de alta calidad que permiten presentar esta información visualmente. En esta tesis presentamos una evaluación de la información disponible y de las utilidades de *profiling* para Java.

Los entornos HPC están formados por recursos muy costosos que conllevan unos gastos de funcionamiento elevados. Los usuarios y administradores requieren la capacidad de controlar la ejecución de aplicaciones para detectar posibles fallos. Monitorizar la ejecución de una aplicación en un clúster es más complejo que monitorizar una ejecución en una máquina local: aparte de la gran cantidad de recursos a monitorizar, en muchos casos los usuarios no pueden conectarse directamente a los nodos de computación donde se ejecutan las aplicaciones, lo que limita su capacidad para monitorizarlas. En esta tesis presentamos una herramienta denominada *Java-Mon*, para monitorizar la ejecución de aplicaciones Java en entornos HPC. Este sistema, que hace uso de las extensiones de gestión de Java (*Java Management Extensions*, JMX) permite monitorizar y controlar propiedades de la JVM y de aplicaciones instrumentadas, y ofrece una visión general de la ejecución de una aplicación en un entorno HPC.

Por último, las evaluaciones de rendimiento o *benchmarking* se realizan frecuentemente en entornos HPC, tanto en hardware como en software. En esta tesis se ha evaluado la capacidad de computación disponible en algunos de los sistemas a utilizar en el procesado de datos de la misión Gaia, además del rendimiento de algunas aplicaciones. Presentamos nuestras campañas de *benchmarking*, y también discutimos las precauciones que deben ser tomadas durante la evaluación de aplicaciones Java, con el fin de obtener resultados fiables.

En resumen, en esta tesis investigamos el uso de Java para el desarrollo de aplicaciones científicas en entornos HPC. La conclusión principal es que Java está totalmente cualificado para esta tarea. Los principales puntos a favor son que la JVM proporciona un rendimiento muy alto, se dispone de numerosas herramientas dentro del ámbito de la plataforma Java, y se consiguen diferentes ventajas inherentes a Java, como la portabilidad, la facilidad de mantenimiento y la robustez. Además se dispone de muchas librerías y herramientas de alto rendimiento disponibles para ayudar al desarrollo y ejecución de aplicaciones Java en HPC. A todo esto hay que sumar las distintas herramientas y librerías creadas en el ámbito de esta tesis, así como las recomendaciones obtenidas gracias a nuestras investigaciones tomando como caso práctico el procesado de datos para la misión Gaia.

Abstract

Java is a very commonly used computer programming language, although its use amongst the scientific and High Performance Computing (HPC) communities remains relatively low. In this thesis, we investigate the use of Java for scientific applications in HPC environments, taking into account modern Java Virtual Machines (JVMs), the tools and libraries available for Java applications and the current trends in HPC. As the size of the datasets that must be processed and the number of available processors increase, the ability of being able to efficiently distribute *chunks* of data amongst many processing components is growing in its importance. We present our middleware application, MPJ-Cache, that has been designed for the efficient distribution of data within a distributed-memory HPC environment, maintaining high throughput, and avoiding bottlenecks and deadtime. MPJ-Cache builds on top of an implementation of Message Passing in Java (MPJ) and adds prefetching, caching, and file-splitting functionality. The use of this middleware for the distribution of data amongst the nodes of a cluster, over a Myrinet-2000 network, has produced very high aggregate data rates. We also present a framework for the launching of applications in HPC environments, as well as a monitoring and management system, these are all contained within a toolbox of utilities called DpcbTools. The launching framework involves the hierarchical grouping of nodes into *NodeGroups*, and the distribution of tasks and data amongst the worker nodes. The monitoring and management component, named JavaMon, makes use of the Java Management Extensions (JMX) framework for accessing JVM and application status. The JVM is a sophisticated and highly configurable runtime environment. Such runtime environments offer many advantages, including portability, automatic garbage collection and optimisations that are not available to statically compiled code. In this thesis, we describe approaches for profiling, and later tuning, the execution of Java applications and of JVM components — especially the garbage collector and the JIT compiler. We present our Memory Usage Model (MUM) tool that can be used to simulate the memory usage behaviour of other applications. All of this work has been performed within the context of the preparation of the data reduction pipeline for the Gaia space mission, which serves as a case-study of a Java-based large scientific software project running in HPC environments.

Contents

1 Introduction **1**
1.1 Introduction and motivation of the topics covered . . . 3
1.1.1 HPC environments . . . 3
1.1.2 Execution frameworks . . . 4
1.1.3 Data communication . . . 4
1.1.4 Profiling Java applications . . . 5
1.1.5 Garbage collection . . . 6
1.1.6 Just-In-Time compiler . . . 6
1.1.7 Monitoring of Java applications . . . 7
1.1.8 Benchmarking . . . 8
1.1.9 Java numerical processing . . . 8
1.2 Gaia data reduction software . . . 9
1.3 Main contributions of this thesis . . . 9
1.3.1 Assessment of the current status of Java and HPC . . . 10
1.3.2 Execution framework . . . 10
1.3.3 MPJ-Cache . . . 10
1.3.4 Guide to tuning the JVM garbage collector and JIT compiler . . . 11
1.3.5 Memory Usage Model . . . 11
1.3.6 Monitoring system . . . 11
1.4 Structure of this thesis . . . 12

2 Background **14**
2.1 High performance computing . . . 14
2.1.1 Sustaining Moore's law . . . 14
2.1.2 Multi-core and many-core . . . 15
2.1.3 Graphics processing units . . . 16
2.1.4 HPC architecture classification . . . 18
2.1.5 HPC programming models . . . 19
2.1.6 Languages used in HPC . . . 20
2.1.7 Trends in HPC . . . 22
2.2 The Java platform . . . 24
2.2.1 The origins of Java . . . 25
2.2.2 The main components of the Java platform . . . 27
2.2.3 Version history of Java . . . 30
2.2.4 The Java ecosystem . . . 31
2.2.5 Control of Java . . . 36
2.2.6 Extensions to Java for HPC . . . 37
2.3 The Gaia mission . . . 38
2.3.1 Gaia data reduction . . . 39
2.3.2 Data Processing and Analysis Consortium (DPAC) . . . 39
2.3.3 DPCB and MareNostrum . . . 41

2.3.4 DPAC systems 42

3 Java for Scientific Applications in HPC 45
3.1 Background 45
3.2 Relevant characteristics of Java 46
3.2.1 Easy to learn language syntax 46
3.2.2 Development and maintenance costs 47
3.2.3 Multithreaded 47
3.2.4 Performance 48
3.2.5 Monitoring capabilities 49
3.2.6 Problem detection 50
3.2.7 Efficient I/O 50
3.2.8 Portability 51
3.2.9 Tools available to assist development and testing 51
3.3 Java numerical processing 52
3.3.1 Java Grande Forum (JGF) 53
3.3.2 Fundamental aspects of Java affecting numerical computing 53
3.3.3 Numerical libraries 54
3.4 Java for Gaia data processing - a case study 55
3.5 Conclusions 57

4 Execution Framework 58
4.1 Background 59
4.1.1 DPAC systems 59
4.1.2 HPC infrastructure and considerations 62
4.2 Requirements 62
4.2.1 Launching framework 63
4.2.2 Data management 63
4.2.3 Data communication 64
4.2.4 Monitoring 64
4.3 Related work 65
4.4 DpcbTools design 67
4.4.1 Launching framework 67
4.4.2 Data communication 70
4.4.3 Monitoring 71
4.5 Conclusions 71

5 Data Communication 73
5.1 Background 74
5.1.1 Networks for HPC 75
5.1.2 OpenMP 76
5.1.3 MPI 76
5.1.4 Client-server model 77
5.2 Related work 78
5.2.1 Approaches for implementing HPC communication in Java 78
5.2.2 MPJ implementations 82
5.3 Communication middleware requirements 84
5.3.1 DPAC systems communication requirements at the DPCB 85
5.3.2 Scalability tests of initial IDU approach with GPFS 85
5.4 Comparison of MPJ middleware 90
5.5 MPJ-Cache 90
5.5.1 The communication layer 91

5.5.2 The data caches . . . 97
5.5.3 The execution of applications with MPJ-Cache . . . 100
5.5.4 MPJCacheTestClient . . . 100
5.5.5 Instrumentation of MPJ-Cache . . . 100
5.6 The tests . . . 100
5.7 Results . . . 101
5.7.1 Direct GPFS access vs. MPJ-Cache with 1 NG . . . 101
5.7.2 Direct GPFS access vs. MPJ-Cache with multiple NGs . . . 102
5.8 Conclusions . . . 103

6 Profiling 105
6.1 Background . . . 105
6.2 Related Work . . . 106
6.3 Profiling tools . . . 107
6.3.1 JVMTI . . . 107
6.3.2 Hprof . . . 107
6.3.3 PerfAnal . . . 108
6.3.4 IBM Support Assistant Workbench . . . 108
6.3.5 JRockit Mission Control / Java Mission Control . . . 108
6.3.6 VisualVM . . . 108
6.4 Profiling for GC . . . 109

7 Garbage Collection 110
7.1 Background . . . 111
7.1.1 GC policies . . . 113
7.1.2 JVM GC policies . . . 115
7.1.3 Allocation process . . . 117
7.1.4 Selecting a GC policy . . . 117
7.1.5 GC tuning . . . 117
7.1.6 Selecting heap options . . . 118
7.1.7 Ergonomics . . . 119
7.2 Related work . . . 119
7.3 Initial GC test campaigns at the DPCB . . . 120
7.3.1 GC tests 1.1 . . . 120
7.3.2 GC tests 1.2 . . . 121
7.4 GC tests 2.1 — extensive GC test campaign . . . 122
7.4.1 Results . . . 125
7.4.2 Analysis . . . 133
7.5 Profiling for GC . . . 133
7.6 Memory Usage Model (MUM) . . . 136
7.6.1 Design . . . 136
7.6.2 Output . . . 137
7.6.3 Application characterisation . . . 138
7.6.4 Uses . . . 138
7.7 Conclusions . . . 140

8 Just-In-Time Compiler 141
8.1 Background . . . 141
8.1.1 Profiling method *hotness* . . . 143
8.1.2 Overview of compilation and optimisation processes . . . 144
8.1.3 JIT compiler optimisations . . . 144
8.1.4 JVM JIT compilers . . . 145

8.2 Related work 147
8.3 JIT compiler tuning to improve GASS performance 148
8.3.1 JIT tests 1 148
8.3.2 JIT tests 2 149
8.3.3 Analysis of results 149
8.4 Conclusions 150

9 Monitoring **151**
9.1 Background 152
9.1.1 Java Management Extensions (JMX) 153
9.1.2 Simple Network Monitoring Protocol (SNMP) 154
9.2 Initial performance tests of JMX 155
9.3 DpcbTools monitoring component - JavaMon 157
9.3.1 Properties to be monitored 157
9.3.2 Possible configurations 157
9.3.3 JavaMon 158
9.3.4 Application instrumentation 158
9.4 Conclusions 159

10 Benchmarking **161**
10.1 Background 161
10.1.1 Considerations while benchmarking hardware 162
10.1.2 Considerations while benchmarking Java applications 162
10.1.3 Statistical analysis 163
10.1.4 Benchmarks 163
10.2 Related work 165
10.3 Benchmarking hardware at the DPCB 166
10.3.1 Objectives of DPCB benchmarking 166
10.3.2 Execution of benchmarks 166
10.3.3 Benchmarks results 169
10.3.4 Analysis of results 174
10.4 Conclusions 177

11 Conclusions **178**
11.1 Future work 180

Bibliography **182**

Glossary **191**

Acronyms **201**

List of Figures **207**

Appendix A - Machine Specifications **209**

Appendix B - Communication Tests **213**

Appendix C - Selected JVM Tuning Options **220**

Appendix D - Configuration Files **223**

1

Introduction

It's OK to figure out murder mysteries, but you shouldn't need to figure out code. You should be able to read it

– Steve McConnell

Java is a very commonly used, general purpose, computer programming language. It is widely used in a number of fields, and Java applications are executed on a range of hardware environments — from smartphones to supercomputers. Very soon after its release, Java gained widespread popularity, and it has consistently been ranked as one of the most commonly used computer programming languages over the last ten years [Tiobe, 2012]. Java was ranked as the second most commonly used language in November 2012, behind the C programming language. The popularity of Java is due to a number of appealing features of the language itself, as well as the inherent advantages in the way in which Java applications may be compiled and executed. The appealing features of Java include: the clean and simple syntax of the language, the high level of maintainability of Java code, portability, type-safety, automatic memory management, its multithreading and networking capabilities, and the wealth of libraries and tools which are available to assist the development and execution of Java applications.

Typically, scientific applications are resource intensive, and they often push computer hardware close to its full potential, in terms of Central Processing Unit (CPU) and memory usage. Scientists often have to limit the size of the problems which are tackled by their applications, according to the computing resources which are available to them. There are two ways in which these problem sizes can be increased: increase the capabilities of the hardware — possibly by buying faster or bigger computers, or increase the ability of the software to make best use of the available hardware. More efficient software will require less hardware resources, while less efficient software will require more hardware resources.

Soon after its initial release, there was much interest in Java from the scientific communities, and many experiments were carried out to try to determine the level of performance offered by Java, and its suitability for developing scientific software. The Java Grande Forum (JGF) [Philippsen et al., 2001] was an initiative to encourage interest in Java amongst the scientific and HPC communities, and it was one of the groups which attempted to benchmark the performance offered by Java. In these early experiments, Java performed quite poorly, when compared against the performance offered by the languages traditionally used for writing scientific and HPC applications, such as C/C++ and Fortran, and interest in Java by those seeking high performance waned. In a number of places in this thesis, references are made to the "traditionally used HPC languages", these references refer to C/C++ and Fortran.

Since its first release, Java has been evolving relatively quickly compared with other languages. The Java language has been expanded upon, and many new features have been added over the last fifteen years. Changes to Java may be proposed, and are managed, by the Java community itself, through the Java Community Process (JCP). However, the changes to the Java language pale in significance to the progress which has been made to improve the performance of the Java runtime environment. Amongst the most significant improvements have been in the areas of code compilation and optimisation, as well as changes to the garbage collector, and Input/Output (I/O) performance. These changes have put the performance offered by Java on a par with the performance offered by the traditionally used HPC languages. In fact, in some cases, the performance offered by Java surpasses that of other languages. Despite these improvements, Java has found it difficult to shake off its initial reputation of providing relatively poor performance.

While performance is a very important factor when selecting a development language, it is certainly not the only one, and in some cases, it should not be the most important one. In the case of large software projects, where software may be in the development and production stages for many years, there are a number of considerations which should be taken into account when selecting a language. These include maintainability, scalability, portability, as well as development time and cost. Long-term software projects typically involve many different developers accessing, and modifying code during the lifetime of the project. The degree to which developers can easily understand and maintain code written by others is very important. Java lends itself extremely well to the development of modular, extendible, easily comprehensible and maintainable code

It is hard to over-emphasise the advantages that Java's portability offers to software projects. These are especially significant for projects that involve the execution of software over a number of hardware platforms, and that will involve the migration from one hardware platform to another.

This thesis was written in the context of the preparations of the software that will form the Gaia data reduction pipeline. Gaia [Lindegren et al., 2008] is a European Space Agency (ESA) astrometric mission that involves the launching of a satellite that will transmit data to Earth every day over a five year period. Preparations have been under way for almost ten years to put in place the data reduction pipeline that will process the Gaia data. The final output of this data reduction pipeline will be an extremely accurate catalogue of around one thousand million stars in our Galaxy. This catalogue will be immensely valuable to science and is expected to greatly contribute to our understanding of the structure and evolution of our Galaxy. All of the software of the Gaia data reduction pipeline is written in Java, and as such, it serves as an example of a Java-based large scientific software project that will run in HPC environments. Additionally, the Gaia project possesses the hindsight gained from using Java on a large scientific software development project over many years, and its experience could be useful for other large projects.

The main motivation behind this thesis was to investigate the use of Java in HPC, in the hope of benefiting the Gaia data reduction pipeline. When this thesis was started, there were a number of topics related to the use of Java in HPC that required investigation. These included: how to take maximum advantage of high-performance networks using Java applications, how to determine if a Java application is performing well, and approaches that might be followed to improve performance. Although its main motivation came from the Gaia data reduction project, much of this thesis is quite general, and as such, it serves as a general investigation into the use of Java in HPC.

To summarise, Java remains one of the most widely used computer programming languages,

it possesses a rich set of features, and offers many attractive advantages. However, despite its popularity in general computing, it is not used to the extent that one might expect by the scientific and HPC communities. In this thesis, the current status of Java for HPC is investigated. Firstly, an overview is given of the Java platform, and it is discussed in terms of its compatibility with the requirements of scientific applications in HPC environments. A number of topics important for the execution of applications in HPC environments are identified, and the extent to which Java handles these topics is examined. Building on existing technology, new tools to assist the execution of Java applications in HPC environments are presented. We present MPJ-Cache, a new Java library for efficient inter-node communication and data distribution. We present a framework for the launching, monitoring and management of applications running in HPC environments. We describe a number of studies into profiling, benchmarking and optimisation of Java applications, and the Java Virtual Machine (JVM). We present Memory Usage Model (MUM), a synthetic application that can be used for simulating the memory usage patterns of other applications.

1.1 Introduction and motivation of the topics covered

1.1.1 HPC environments

A typical distributed-memory High Performance Computing (HPC) environment includes many computing nodes, with each computing node containing one or more processors, and each processor containing one or more cores. Each computing node is typically connected to every other computing node over some high-speed, low-latency network, while each computing node will typically also be connected to some central storage devices, which are often shared by many, or all, of the computing nodes.

One of the most significant trends in HPC over the last few years has been the move towards an ever increasing number of cores per processor, and an increase in the number of computing nodes per computer cluster. This trend increases the importance of developing scalable parallel software that can take advantage of the available cores. Typically, this is done by running several threads or several processes within the same computing node.

Another trend of particular importance in HPC is the increase in the size of the datasets that must be processed, this trend is commonly referred to as the *big data* [IBM, 2011] trend. With more threads (or processes) accessing larger datasets, the requirement for efficient data access and data communication becomes more important. The accessing of shared data by many concurrent processes has the potential to become a bottleneck in processing.

It is often the case in HPC environments that a number of users share the same hardware resources, such as network infrastructure and storage devices. If there is a high demand for these resources, and therefore they are normally being used, then the performance achieved by one users application may be affected by the activity of the applications of other users. This influence should be minimised by ensuring that shared resources are used optimally and minimally.

Despite the huge size of the Java community, there has been a lack of a coordinated effort to develop tools to assist the use of Java in HPC environments. Although there have been numerous projects with this objective, they have generally involved small teams working independently from each other, and in many cases, these projects have been short-lived and are no longer being actively supported. Many useful tools and libraries have been developed to support the

use of Java in HPC. However, finding up-to-date information on these projects can be difficult and time-consuming.

1.1.2 Execution frameworks

Executing an application in an HPC environment is usually more complicated than simply launching an application on a local computer. HPC users may have a large amount of computing resources available to them, and these resources could potentially be used in a number of different configurations. One choice that must be made is how many processes to run per computing node, per processor and per core. For example, if an application is to be executed on a cluster where each computing node contains two quad-core processors, it would be possible to run two processes per computing node (and attempt to make use of the four cores inside of each process), or to run a single process per core. In the case of Java applications, it must be decided how many JVMs to execute per computing node. Running fewer JVMs results in less overall overhead, however each individual JVM must perform extra work — possibly so much extra work that performance decreases.

The management of data is important for any execution framework, and is it even more important when executing applications that must handle large volumes of data. Any required input data must be made available to the processes running in the nodes, and any output data must be stored in an adequate location. The caching of data, that might be needed multiple times, close to the core where that data might be needed, can be used to minimise accessing of shared storage devices and will likely improve performance.

A key issue in HPC is ensuring that the computing resources are being used as close to their full potential as possible. Normally jobs in an HPC environment are managed by some kind of job management system. The sophistication of these systems varies, with some offering very basic capabilities, while others offer quite intelligent job scheduling and management features. One desirable capability in a job management system would be the ability to decompose large computing tasks into smaller tasks, and the intelligent distribution of these tasks amongst the available computing resources, taking data locality and load balancing into account, while allocating computing resources to execute tasks.

1.1.3 Data communication

Efficient communication is a crucial aspect of HPC. Applications running in a computer cluster of multi-core nodes may require the ability to perform communication between computing nodes, known as *inter-node* communication; as well as communication between processes running within the same node, known as *intra-node* communication. Poorly performing communication software can act as a bottleneck, leading to an increase in the time required for the application to complete its work. The importance of I/O efficiency increases with the number of parallel processes, and the problem is further amplified if the data is accessed repeatedly.

Message Passing Interface (MPI) [Forum, 1994] continues as the leading approach for implementing inter-process communication in distributed-memory environments, offering point-to-point as well as collective functions. Typically, HPC environments offer some high-performance network technology to allow for low-latency high-bandwidth communication between computing nodes, as well as allowing for fast communication with any shared storage devices. In order to make best use of such high-performance networks, communication libraries must be

available which can natively communicate with these networks. The MPI libraries available for the traditionally-used HPC languages, such as MPICH2 and OpenMPI, contain native support for a number of high-performance networks, thus allowing applications using these libraries to make best use of high-performance networks.

MPI for Java is normally known as Message Passing in Java (MPJ). Although there is no official MPI binding for Java, there have been many projects which have implemented MPJ. One of the challenges which these projects have faced is that Java cannot communicate natively with many high-performance networks, such as Infiniband or Myrinet. In some implementations of MPJ, communication over high-performance networks is achieved through the use of Transmission Control Protocol/Internet Protocol (TCP/IP) emulation, this approach fails to fully take advantage of the capabilities of high-performance networks.

Despite the strengths and maturity of MPI, using it to provide data communication functionality to parallel applications can be a challenging task, especially for applications that involve many concurrent processes (or threads). Writing parallel applications is not as intuitive to human beings as writing sequential applications. There are a number of issues which can potentially cause unexpected scenarios to arise, which do not exist for sequential applications, such as race conditions between threads. Therefore, writing parallel applications requires a careful design and programming effort, and even then, it is difficult to foresee all scenarios and prevent bugs.

It is generally accepted that the availability of high-level programming constructs leads to higher levels of productivity amongst developers. The ideal communication middleware would provide both high-level and low-level programming constructs in an extendible framework, while also being able to take maximum advantage of any high-speed, low-latency network infrastructure.

1.1.4 Profiling Java applications

The execution of a Java application involves a complex interaction between a number of sophisticated components, including the application bytecode, the JVM core runtime system, the Just-In-Time (JIT) compiler, and the garbage collector. Profiling the execution of a Java application provides an insight into how this interaction plays out. Amongst the properties which might be of interest during profiling are the consumption of computing resources; such as CPU, memory and I/O by the application; additionally, the activity of the garbage collector and the JIT compiler might provide useful information.

Profiling is a worthwhile activity as the information gained from it can be used to improve application performance. For example, the discovery, and subsequent elimination of bottlenecks in processing, will generally lead to improved application throughput. Information gained through profiling can also be used to tune the activity of the garbage collector and the JIT compiler, which are introduced in the next sections.

The JVM exposes a lot of information that may be queried for profiling purposes. This information may be accessed in a number of ways. The Java Virtual Machine Tools Interface (JVMTI) is the main interface through which JVM information can be accessed. The Java Management Extensions (JMX) framework may be used to access this information directly, alternatively, there exists a large number of high-quality profiling tools for the Java platform, that can retrieve and visualise profiling information. The Java Development Kit (JDK) itself comes with a number of profiling tools. In addition to JVM profiling, it may be necessary to obtain system information directly from the OS. Interpreting profiling information can be

difficult, as it is often not clear which information is the most significant, and therefore it may not be clear what follow-up steps should be taken following a profiling study.

1.1.5 Garbage collection

Java, like many other languages which execute within a managed runtime environment, relies on an automatic memory management system known as Garbage Collection (GC). GC is a process which runs, when its service is required, during the execution of an application, and which safely releases memory at some point in time, after the data which was stored in that memory space is no longer reachable. The existence of the GC process means that application developers generally do not need to concern themselves with memory management, as this task is taken care of by the garbage collector. This allows developers to concentrate on the design of their software. Research has shown that the removal of explicit memory management code from applications, allows for the writing of cleaner and less bug-prone code.

Although GC is designed to be able to work without any instructions whatsoever from the user, it is in fact, highly configurable. The widely used JVMs offer several GC policies, GC tuning options, and many other options which can also affect GC. For example, the HotSpot JVM (version 1.7) has over 600 options which can be specified by users at runtime, of which, at least 250 could have an effect on GC. The idea of selecting a particular set of GC options to match the particular characteristics, or needs, of a particular application is known as *application-specific garbage collection*. In order perform it, an understanding must be obtained of the memory usage characteristics of the target application while it is executing. Also, an understanding of the available tuning options must be gained. This includes understanding how these options can affect (interfere with) each other, and how they can affect the execution of an application.

Obtaining detailed information on the available JVM tuning options, including GC and JIT compiler tuning options, can be more difficult than one might expect, as discussed in the next section.

1.1.6 Just-In-Time compiler

Java application code is compiled into bytecode at compile-time. All modern JVMs contain a component called a JIT compiler which may compile the bytecode into native machine code at runtime. Executing native machine code requires much less instructions to be performed than interpreting bytecode does, resulting in much better performance. As well as compiling the bytecode, the JIT compiler may apply sophisticated optimisation techniques, such as dynamic compilation and adaptive optimisation to further improve performance. These techniques allow for a part of the code that has already been executing for sometime (either through interpretation or having already been compiled to native machine node) to be compiled (or recompiled) and optimised using insight that was obtained after the application had been executing for some time.

In common with GC, JVM implementations are free to implement JIT compilation in whatever way they wish, some JVMs follow a total-compilation strategy, while others include a Mixed Mode Interpreter (MMI) — which involves some code being interpreted and other code being compiled. Also in common with GC, the JIT compiler is highly configurable, although it can be surprisingly difficult to obtain detailed descriptions of all of the available tuning options. One of the reasons for this, seems to be that because some of these options are designated

as *non-standard* and *diagnostic* in some JVMs, and may change between JVM versions, JVM implementers are reluctant to encourage developers to use them, and very little information is published about the availability of these options. Additionally, in recent years, JVMs implementers have been placing greater emphasis on JVM *ergonomics*, which effectively refers to the ability of JVMs to tune themselves. This may also be a contributing factor as to why JVM implementers don't make more of an effort to inform developers of the details of the available tuning options.

The nature of some HPC environments, and the frameworks that are used for the execution of jobs in them, can inhibit the ability of JIT compilers to perform their full range of optimisations. This is due to the fact that some optimisations require that code is executed a large number of times before they can be applied. Some execution frameworks translate the calls to Java methods into individual jobs or *tasks*, and for each instance of these tasks, a transient JVM is created to execute that instance. These transient JVMs do not share profiling information, therefore no particular JVM is able to obtain a long-term profile of the application. Additionally, because the JVMs running in each computing node are generally independent from each other, each JVM must pay any compilation and optimisation costs.

1.1.7 Monitoring of Java applications

Monitoring tools are particularly important when executing applications in HPC environments such as on a computer cluster. Execution time in an HPC environment is a valuable resource, and often users must compete with each other for access to the computing resources. It is important, from the perspective of both system administrators and system users, to be able to monitor the status of HPC hardware and the software running on it. Administrators are keen to ensure that resources are not sitting idle and are being used close to their full potential, while system users are keen to ensure that their applications are running correctly. If a problem is detected, users may wish to be made aware of this, allowing them to take some corrective action. The importance of monitoring increases when dealing with applications that use many computing resources, and for those that must execute for relatively long periods of time, such as days or weeks.

JMX is a technology built into the Java platform that allows for the monitoring and management of Java systems, including those systems running in a distributed environment. JMX makes use of the concept of managed beans (known as MBeans), with an MBean associated with each JVM resource that is to be managed or monitored. All of the MBeans are registered with a server process called the MBeanServer. Remote applications, such as monitoring applications, may connect to the MBeanServer and interact with the MBeans registered with the MBeanServer.

The JMX specification defines a number of MBeans which JVMs should make available for monitoring their internal state. For example, the *GarbageCollectorMXBean* which can be used to monitor GC activity. As well as making use of the provided MBeans, users may define their own MBeans, which may be registered with the JMX framework, and they can then be accessed in the same way as the standard MBeans. Such user-defined MBeans can be used to monitor specific properties of an application while it is executing, and also, this technology allows for the management of applications, as properties may be changed at runtime. The process of creating MBeans to allow for the monitoring or managing of application properties is known as *instrumenting* the application.

Although monitoring is extremely useful, it must be kept in mind that it does add an overhead to the system. The closer an application is monitored, the higher this overhead will be. The level of granularity of the monitoring must be weighed against the associated overhead.

There exists many tools for the monitoring of Java applications — the JDK itself includes a number of such tools. However, there are few monitoring tools specifically designed for monitoring Java applications in HPC environments. It is often the case in HPC environments that users of the system may not be able to directly access the computing nodes where their jobs are executing, this limits the ability of users to perform real-time monitoring of their applications.

1.1.8 Benchmarking

In computing, the term *benchmarking* is used to refer to the process of measuring the performance offered by some component. In the context of hardware benchmarking, the term FLoating-point Operations Per Second (FLOPS) is used to refer to the number of floating point calculations a given machine (or processor or core) can perform per second, while in the context of software benchmarking, the term FLoating-point OPerations count (FLOP count) normally refers to an estimate of the number of floating point operations required by a given application (or algorithm) to complete all of its calculations. If the FLOPS available on a particular machine is known, and the FLOP count of an application is also known, then it is possible to get a rough estimate of the execution time of that application on that machine.

The execution of Java applications within a Java Runtime Environment (JRE) is quite different from the execution of native machine code. It involves a complex interaction between several components, and there are many factors which can affect the execution, including compilation and optimisation by the JIT compiler, the activity of the garbage collector, the state of the heap, thread scheduling, class loading at start-up, and many more. For this reason, care must be taken when attempting to measure the performance of, or *benchmark*, the execution of a Java application.

When measuring application performance, it is important that a methodology is followed which takes in account a certain level of non-determinism which can exist when executing Java applications. For example, it is possible that the garbage collector may perform a minor or a full garbage collection while a benchmark is executing. Such events can be detected by monitoring the execution of the benchmark. Additionally, the application of statistical analysis techniques can improve the confidence that we can have in benchmarking results.

1.1.9 Java numerical processing

One of the factors which has militated against a wider use of Java amongst the scientific and HPC communities has been the belief that Java cannot deliver the same performance as the traditionally used programming languages such as Fortran, C and C++. The performance of floating-point numerical operations is typically very important for scientific applications, and despite the advances of the Java platform, there still exists a general perception that Java does not perform such operations very efficiently. In addition to such perceptions, there is also a general unawareness amongst the scientific and HPC communities of the strengths of Java, and of the high-performance libraries and tools which are available for the development and execution of Java applications.

1.2 Gaia data reduction software

Gaia [Lindegren et al., 2008] is a ESA mission, whose primary objective is to catalogue the position and movements of around one billion stars in our Galaxy. This mission involves the launching of a space satellite containing two optical telescopes and a large array of CDD detectors, allowing for very precise measurements to be taken of the sources observed by Gaia.

The Gaia data reduction pipeline is the set of software applications that will be executed on the Gaia data. This pipeline represents a large and complex computing challenge. It is a policy within the Gaia data processing project that all software should be written in Java. The selection of Java for this kind of large, scientific, data processing project is relatively uncommon. Therefore, the Gaia data reduction pipeline represents an opportunity to study the use of Java to implement a scientific data processing pipeline, and its execution in HPC environments.

It is expected that one of the most technically challenging aspects of executing the Gaia data reduction pipeline will be to maintain a smooth flow of data through the various processes, avoiding bottlenecks and deadtime. The volume of data itself is not unmanageably large, compared with other large scientific projects. However, due to the many relationships within the data, and the need to perform cross references during the processing of the data, the issues of efficient data access and data communication are extremely important.

The preparation of the Gaia data reduction pipeline was one of the main motivational factors behind the work presented in this thesis. In several places in this thesis, we use and discuss some of the Gaia data reduction software. However, the scope of this thesis covers the general use of Java for scientific applications in HPC, and is not limited to the Gaia data reduction software.

1.3 Main contributions of this thesis

The main contributions of this thesis are listed below, and they are described in more detail in the subsequent subsections.

- A broad investigation into the current status of the Java platform, and in particular, its usefulness for the development of HPC applications.
- A framework for the utilisation of computing nodes, using a hierarchical grouping system, as well as the decomposition of large processing tasks into smaller tasks, and the launching of these tasks in the *worker* nodes
- MPJ-Cache — a middleware for the distribution of data in HPC environments, which makes use of an underlying implementation of MPJ, and adds higher-level features including data prefetching, file splitting and data caching.
- A description, and investigation into the use of the tuning options available for JVMs, in particular, the GC and JIT compiler options for the HotSpot and J9 (IBM) JVMs.
- MUM — a synthetic application that can be used to simulate the memory usage behaviour of other applications.
- JavaMon — a JMX-based system that can be used to monitor and manage the execution of Java applications in HPC environments.
- A number of profiling and benchmarking campaigns.

1.3.1 Assessment of the current status of Java and HPC

This thesis includes in-depth investigations, and analysis, of the key developments in Java and HPC over recent years, both in academia and in industry. This includes a description and assessment of the libraries and tools which are available to assist the development and execution of Java applications for HPC. The capabilities and performance of Java, in a number of fields important in HPC, are described. These include resource utilisation, data communication, memory management, compilation and optimisation, monitoring, benchmarking and numerical processing. The capabilities and performance of Java are compared with those of the languages traditionally used in HPC.

1.3.2 Execution framework

We present a framework for the efficient distribution of work amongst the computing resources of an HPC environment. This framework involves the initial decomposition of the entire computing challenge into smaller chunks of work. Each of these chunks is defined by the application (or part of an application, such as a method) that must be executed, as well as the input data that this application requires. The available computing nodes are divided following a hierarchical system, into groups of nodes known as Node Groups (NGs), and within each of these NGs, a single node is designated as the Node Group Manager (NGM), which coordinates the activity of the other nodes within its NG, by sending them instructions of what needs to be done, as well as input data.

An efficient data communication middleware called MPJ-Cache (introduced in the next section) is used to ensure the efficient use of any available high-performance networks, and also allows for the caching of data. A particular data communication pattern for distributing data amongst the nodes in each node group will be defined which minimises the number of concurrent processes which need to access shared storage devices.

1.3.3 MPJ-Cache

The work of this thesis includes the development and documentation of an application that we call MPJ-Cache, designed to provide a high-level, intuitive, flexible, and efficient set of communication functionality to Java applications in HPC environments. MPJ-Cache generally follows the well-known client-server model, with one server process communicating with many client processes. MPJ-Cache can be configured to operate in either a *push* or a *pull* mode, with the server process pushing data and instructions to client processes, or the clients requesting data from the server. MPJ-Cache was designed to work in collaboration with the framework described in the previous section, however it can also work on its own. When running in a computer cluster that has been divided into NGs, as described above, then the server process would be executed in the NGM node, while client processes would be executed in the other nodes.

MPJ-Cache provides intelligent caching of data with the goal of minimising the number of requests to shared storage devices. Data may be cached in memory, or on a local disk, by the server process. Additionally, if the server is aware of the full list of data that it will need to send to the clients, it will prefetch data to its local cache during periods of time when it would otherwise be idle.

The communication component of MPJ-Cache makes use of an underlying implementation of MPJ. MPJ-Cache can work with any implementation of MPJ which implements the `mpiJava` 1.2 [Carpenter et al., 1999] specification. It has been tested with FastMPJ and MPJ Express. MPJ-Cache provides an Application Programming Interface (API) to application developers which is of a higher level of abstraction than MPJ (which is at the same level of abstraction as MPI). This allows for the development of simpler, and probably less bug-prone application code.

1.3.4 Guide to tuning the JVM garbage collector and JIT compiler

Two of the most sophisticated components of the JRE are the garbage collector and the JIT compiler. These components contribute greatly to the performance obtained by Java applications. It is possible to achieve very high performance for most applications without performing any tuning of these components. However, in some occasions, it may be possible to improve performance by tuning these components to suit the particular characteristics of the target application. Such information may be found through profiling the execution of the application.

This thesis describes a number of approaches and tools that can be followed for the profiling of Java applications. Studies are presented of the available GC and JIT compiler options, and recommendations are made regarding how to select a particular set of options to match the particular characteristics of an application.

1.3.5 Memory Usage Model

MUM is a modelling application, that can be used to model the memory usage behaviour of other applications. The basic idea behind MUM is that, if an application has been profiled, and its memory usage has been characterised in terms of a relatively small set of basic characteristics, then an instance of MUM can be executed which simulates the memory usage behaviour of the characterised application.

Performing GC tuning often involves the execution of instances of an application. This can be a costly activity in an HPC environment, especially when the execution of the applications involves the consumption of many resources, such as CPU, I/O and network bandwidth. The availability of a model application, which simulates the memory usage pattern of the target application to a high degree could be a valuable resource. As well as its use as a tool for exploring the memory usage behaviour of applications, other possible uses of MUM are discussed.

1.3.6 Monitoring system

This thesis presents a JMX-based monitoring system that permits the monitoring of JVMs, and the applications running in them, in an HPC environment. We discuss the difficulties associated with monitoring the execution of an application across many nodes of a distributed-memory HPC environment. Our monitoring system allows users to monitor the status of an individual computing node, as well as providing users with an aggregated overview of the entire execution. We also instrumented our applications so that they can also be managed using the JMX framework *on the fly*. This allows for reconfiguration of applications, based on monitoring information.

1.4 Structure of this thesis

In Chapter 2, background information relevant to this thesis is given. This chapter first introduces HPC environments, the languages and technologies that are commonly used in HPC, and also the current trends in the field. Later, a brief history of the Java language, including a description of the changes and improvements which have been made to Java with the release of each new version. The key components of Java technology, including the JVM, the garbage collector and the JIT compiler are described, and their roles are explained. The Java *ecosystem* is described, including the companies and groups which have contributed to the development of the Java platform. Finally, the Gaia space mission, and most importantly, the applications which form the Gaia data reduction pipeline are introduced. The efficient execution of these applications was one of the main motivational forces behind this thesis, and they serve as test applications in several parts of this thesis.

In Chapter 3, an assessment is given of the current status of Java for the development of scientific applications in HPC environments. A number of features relevant to such applications are discussed, and the extent to which the Java platform provides these features is described. The capabilities and performance of Java is compared with those of the languages traditionally used by the scientific and HPC communities. The experience, and some lessons learned by using Java in the Gaia project are given.

The execution of applications in an HPC environment (such as in a distributed-memory computer cluster) is more complex than doing so in a local computer. It requires a framework to manage the launching of the software across the entire HPC environment, making sure that the available computing resources are being used efficiently, that I/O data is managed correctly, and that bottlenecks and deadtime are avoided as much as possible. Several projects have produced frameworks for the launching of Java applications in HPC environments. In Chapter 4 we discuss these frameworks. The requirements of data-intensive scientific applications are described, and a new framework for the launching and controlling of applications is described.

Efficient data communication options for Java applications in HPC environments are discussed in Chapter 5. A number of projects which have attempted to provide efficient data communication for Java in HPC are described, and the approaches taken are discussed in terms of their associated advantages and disadvantages. MPJ — Java's equivalent of MPI — is introduced, and a number of implementations of it are described, including MPJ Express and FastMPJ. A set of tests to compare the performance of these is described. In particular, their ability to provide scalable and efficient communication over high-performance networks is investigated. Our new middleware application, that we call MPJ-Cache is introduced. This middleware makes use of an underlying implementation of MPJ, and provides an intuitive, high-level, and efficient set of communication methods for Java applications, following a client-server model. The use of this middleware for distributing data amongst the nodes of a computer cluster is described.

The profiling and characterisation of Java applications is discussed in Chapter 6. Profiling an application provides an insight into what is happening while that application is executing. This insight can later be used for a number of purposes, including code optimisation, and GC and JIT compiler tuning. An investigation into the tools available for profiling Java applications is given. A selection of these tools are examined and compared.

In Chapter 7, the topic of GC is discussed. Firstly, a brief introduction to GC is given, followed by a description of how GC has been implemented in Java. This includes describing some of the vast array of tuning options which are available in each of the commonly-used JVMs. A

number of approaches for attempting to tune GC are discussed, and an extensive series of tests are described. Also, an application that we call MUM is presented. MUM can be configured to exhibit the memory usage behaviour of other applications, and we investigate if instances of MUM can be used when searching for a better performing set of GC options for particular applications. Other uses of MUM are also suggested.

The JIT compiler is discussed in Chapter 8. The sophisticated optimisations provided by the JIT compiler at runtime are described. The options available for controlling the activity of the JIT compiler are discussed, and attempts to improve the performance of some applications through JIT compiler tuning are presented. HPC environments can place constraints on the ability of the JIT compiler to perform optimisations, due to their architecture and the way in which applications are launched. This issue is discussed, and some possible solutions are suggested.

The monitoring of Java applications in HPC environments is discussed in Chapter 9. Monitoring applications is extremely important in HPC. Executions often last for long periods of time, and the computing hardware being used are valuable resources which are typically shared amongst many users. If a problem arises during the execution of an application, it is important that the user can detect this, and possibly take some corrective action. We present a JMX-based monitoring system for the monitoring of applications running across many computing nodes.

The issue of benchmarking is discussed in Chapter 10. This includes the topic of estimating application execution time on a particular machine. The key issues which must be considered when benchmarking Java applications are discussed, and the most widely used Java benchmarks are compared.

Finally, in Chapter 11, a summary of the contribution of this thesis is given, some conclusions that were reached during this work are presented, and suggestions are made regarding how this work could be furthered in the future.

2

Background

As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realised that a large part of my life from then on was going to be spent in finding mistakes in my own programs

– Maurice Wilkes — inventor of the first computer with an internally stored program

This chapter provides background information on a number of topics addressed in this thesis. This background information is grouped into three main sections, namely: HPC, the Java platform and the Gaia data processing pipeline.

2.1 High performance computing

The term HPC is generally used to refer to the use of powerful computing resources to solve challenging computing tasks. The adjectives *powerful* and *challenging* are, of course, relative. The capabilities of HPC environments, as well as those of standard computers, have been increasing at least as fast as Moore's law (see Section 2.1.1) for more than half a century. Hand-held devices of today possess more computing power than the supercomputers of a few decades ago. This continuous increase in computing capability allows for ever larger computing challenges to be undertaken.

HPC environments are often used to solve *large* scientific problems in fields such as mathematics, physics, chemistry and astronomy. Typically, software developers creating applications for scientific fields want to take maximum advantage of the available hardware. Therefore, the performance of the software is very important. Parallel programming models, including multi-threading and message passing, are widely used for the development of software that can take advantage of the massive parallelism available in modern HPC environments. Traditionally, languages such as Fortran, C and C++ have been used by the scientific and HPC communities (see Section 2.1.6). However, there are several alternative programming options emerging, including the use of Java.

2.1.1 Sustaining Moore's law

Moore's law [Moore, 1965] predicts that the number of transistors which can be placed on an integrated circuit of a given size, doubles, on average, every two years. This increase in the

number of transistors leads to an increase in the computing capability of processors. The ability of the processor industry to maintain Moore's law over several decades allowed each generation of uniprocessor computers to simply get faster without significant changes to their architecture. Single-threaded applications running on such systems could run faster without any change to their design. Predictions of the imminent ending of Moore's law have been made for several decades, yet the trend has continued.

Frequency scaling refers to the increasing of processor frequency in order to increase the computing capability. However, these increases in frequency must be *paid for* by an increase in the power used, which also leads to an increase in the heat generated, in turn requiring extra cooling. Frequency scaling was used as a complimentary technique to placing higher densities of transistors within processors for many years, until around 2004, when the industry began to move away from this technique as further increases in frequency would have required unacceptable power increases.

In addition to the problems of increased power requirements and heat generation, processor designers also have to deal with *current leakage*. The term *off-state* is used to refer to a particular leakage problem that occurs when conventional transistors are reduced below a certain size. As transistors get smaller, the gates which control the flow of electricity (either on or off) get thinner and less able to block the flow of electrons. Small transistors tend to use electricity all the time, even when they are not switching.

Despite the challenges that it is facing, the processor industry has continued to produce new innovations which have allowed it to sustain Moore's law. For example, Intel's 22 nanometre processor, known as Ivy Bridge [Damaraju et al., 2012] and released in 2012, is the first commercially available processor to use tri-gate (or 3D) technology. This technology greatly reduces leakage of current, allows for higher densities of transistors, and also requires less power. In addition, it allows for increases in transistor density within a single core. One the other hand, AMD's Trinity processor, also released in 2012, is the first commercially available processor to make use of Resonant Clock Mesh Technology to deliver higher performance with reduced power consumption.

Although the industry has so far been able to uphold Moore's law, the average clock rate of processors is no longer increasing in tandem with the number of transistors. By 2004, processors with clock rates of 3 gigahertz (GHz) were commercially available, but clock rates have not advanced very far in the intervening years. This has been due to a shift in the processor industry away from absolute clock rates towards other technologies to further improve performance, such as the development of multi-core processors, and processors which support simultaneous multithreading. The shift to multi-core has been one of the most important trends that computing has ever experienced, it is discussed in Section 2.1.2 below.

2.1.2 Multi-core and many-core

For the last number of years, the microprocessor industry has been putting increased emphasis on the development of multi-core processors as an efficient approach for increasing processor computing capability. A multi-core processor can be thought of as a single composite component, but with two or more processing units (or cores) within it. The term *many-core* is used to refer to processors with a number of cores so large that approaches used in a multi-core processors are not efficient, and some other computing model should be followed in order to take advantage of the large number of cores, for example, utilising Network on Chip (NoC)

technology [Atienza et al., 2008].

The trend towards multi-core processors can be seen in the typical CPU architecture used in both HPC machines as well as in personal computers and laptops. This trend has created both an opportunity and a challenge for software developers. It is important that software executed on multi-core processors can make use of the available cores — it is of little benefit having a quad-core processor if only one of the cores is being used. In the case of personal computers, the importance of utilising all of the available cores is normally not so great, as users will rarely require the full processing power of their computer. However, in HPC, the ability of software to take full advantage of the available hardware is of great importance.

Unlike other hardware changes that have happened, the move to multi-core processors has massive implications for software development. Although multi-core processors have been commonplace for a number of years, many applications that will be executed in a multi-core environment continue to be written in a serial manner, meaning that they often fail to take full advantage of the available multiple cores. Software developers must take the multi-core processor architecture into account when designing software, if they wish their software to be able to take full advantage of the computing capability available to it [Dongarra et al., 2007]. The difficulty is that parallel multi-threaded applications are more difficult to write than single-threaded applications. There exists a large set of software bugs which do not arise with single-threaded applications, but can easily be introduced into multi-threaded applications, if great care is not taken to prevent them. These bugs are typically related to race conditions existing between threads, and locking problems.

The ability of software developers to design efficient, robust, scalable and bug-free software that can take advantage of the now ubiquitous multi-core processors is one of the biggest challenges facing HPC computing, as well as general computing, presently. If software is generally not able to take advantage of multi-core processors, then a gap will grow between the theoretical performance of computer hardware and the actual performance obtained.

2.1.3 Graphics processing units

Graphics Processing Units (GPUs) are a type of processor specifically designed for performing a large number of floating-point operations in parallel. GPUs can be thought of as a single processor with a large number (hundreds or even thousands) of small cores. GPUs were originally designed for rendering graphics, the primary computing task in graphical computing being the transformations needed to render 3D shapes on a 2D display. This task is inherently parallel and well suited to parallel computing, as each core may render a particular part of any image independently of the other cores. Traditionally, GPUs were used along with a CPU — GPUs were viewed as a mechanism to reduce the workload of the CPUs. Sometimes GPUs are referred to as *graphics* accelerators as they accelerate the process of rendering graphics, and remove this burden from the CPU.

OpenGL [The Khronos Group Inc, 2012] and DirectX [Blythe, 2006] [1] are two graphics APIs, which were initially developed in the 1990s, that allow the rendering of 2D and 3D graphics using GPUs. GPU design, processing capability, and programmability has been going through a very rapid process of enhancement since the late 1990s. These enhancements have lead to

[1] DirectX is a group of APIs developed by Microsoft for processing a range of multimedia, including sound and graphics. Direct3D is the graphics processing API within the DirectX collection of APIs, however the term DirectX is often used to refer specifically to the Direct3D API

the scientific and HPC communities becoming interested in GPUs as a tool for executing their CPU-intensive applications — which typically require the parallel execution of large numbers of floating-point operations. The terms *General-Purpose Graphics Processing Unit (GPGPU)* or *GPU computing* are used to refer to the use of GPUs for tasks other than graphics processing In some places, the term GPGPU is only used to refer to the initial attempts to use GPUs through the use of graphics APIs, and the term *GPU Computing* is used for present work with GPUs, while in other places, the term *GPGPU* continues to be used. The term *Accelerated Processing Unit (APU)* is used to refer to a processing unit which includes a CPU as well as some additional processing capability, such as a GPU. Such units are becoming increasingly common.

Initially, software developers had to use a graphics API in order to make use of GPUs. This not only required a strong familiarity with the graphics API, but it also meant that developers had to express their problems in terms of graphical concepts, even if the problem was non-graphical. To increase GPU programmability, a number of models have been developed which ease the use of GPUs by non-graphical applications. Chiefly amongst these models are CUDA [Nickolls et al., 2008] and OpenCL [Khronos Group, 2010].

CUDA, released by NVIDIA in 2006, is a software and hardware architecture that allows software written in high-level languages to make use of GPUs. Initially CUDA was an acronym for *Compute Unified Device Architecture*, however NVIDIA now uses CUDA as an umbrella term encompassing all GPU computing products. The CUDA programming model is an extension of the C programming language that allows the execution of single-threaded parts of an application to run on a CPU (which is highly optimised for such processing), while multi-threaded parts of the applications may run on a GPU. OpenCL [Komatsu et al., 2010] is an open standard programming model for cross-platform parallel programming, supported by all GPU vendors (including AMD, NVIDIA and Intel). CUDA and OpenCL provide similar functionality, although there are some important differences in their design. The underlying hardware is hidden from OpenCL developers, which means that they cannot make use of features which are vendor specific. CUDA, on the other hand, is specific to NVIDIA GPUs, and therefore it can, in some cases, provide better performance than OpenCL. The performance offered by CUDA and OpenCL are compared in [Fang et al., 2011] through the execution of an extensive set of benchmarks. CUDA was shown to generally perform better than OpenCL (30% being the most significant difference), although the portability advantages offered by OpenCL seem to compensate for a lower level of performance.

It is worth noting that there is currently no *pure Java* mechanism for running Java code on GPUs. However, there have been a number of projects that allow Java applications to interact with an underlying framework such as OpenCL or CUDA using the Java Native Interface (JNI). JOCL [JOCL, 2012] is a Java binding for OpenCL, while JCuda [JCuda, 2012] and JCUDA [Yan et al., 2009] are both Java bindings for CUDA.

For certain types of problems, the use of GPUs provides impressive speed-ups, but GPUs are not suited for all types of problems. GPUs generally perform best with problems that can be broken up into many independent threads, which require little synchronisation or inter-thread communication. An interesting observation is that each generation of GPUs continues to provide more CPU-like programmability and flexibility, and as CPUs continue in the direction of multi-core processors, GPUs and CPUs are, in some ways, converging.

2.1.4 HPC architecture classification

HPC environments are often classified based on the relationships between their processors and memory. Due to the speed of change in HPC, the exact meaning of a term can sometimes change, and often more than one term exists to refer to the same concept. Systems where all the processors can access the same memory space are known as *shared-memory* systems, while systems that include memory that is only accessible to a subset of all of the processors comprising the system, are known as *distributed-memory* systems.

The term *Symmetric Multiprocessing (SMP)* is used to refer to a system comprising more than one identical processing unit, each of which having access to the same shared-memory[2]. The SMP architecture is typically followed within multi-core processors, whereby a number of identical cores have equal access to the same shared memory, in this context, the term Chip Multiprocessor (CMP) is sometimes used.

A typical computer cluster is sometimes referred to as a hybrid SMP cluster, as it contains several SMP systems (computing nodes) joined together using some network infrastructure. The majority of current computer clusters are composed of the same, relatively cheap, commodity parts that are found in personal computers.

Shared-memory systems can be further divided into those which provide Uniform Memory Access (UMA) and those which provide Non-Uniform Memory Access (NUMA). An UMA system, is one in which all the processors have equal access (and latency) to the shared memory. If the number of processors in an UMA system is increased beyond a certain point, scaling problems arise, caused by too many processors attempting to access the shared memory. In order to improve scalability, alternative processor-memory architectures such as NUMA are followed. NUMA systems include some memory that is *local* to a particular processor, or group of processors. Local memory provides better performance, and reduces the number of accesses to the shared memory. However, if a processor needs to access memory which is not located in its local memory, then the access time will depend on the location of the required memory relative to the processor.

A Distributed Shared Memory (DSM) system is one which includes distributed memory that may be addressed using a single global address space. Software-based DSM systems generally hide the complexity of managing distributed memory regions from application developers. Although DSM systems allow any processor to access any part of the global memory, the memory which is local to a particular processor will provide the best performance for that processor. The terms Partitioned Global Address Space (PGAS) and DSM are generally considered synonyms.

Another classification scheme for computer architectures is Flynn's taxonomy [Flynn, 1972]. It is based on the number of instruction streams that can executed, and on the number of data streams that are available for processing at any one time. According to this scheme, all computing systems fall into one of four possible categories: Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD) and Multiple Instruction Multiple Data (MIMD). Uniprocessor machines fall into the SISD category. SIMD are systems which can execute the same stream of instructions on multiple data streams in parallel — GPUs fall into this category. MISD systems may execute a multiple instruction stream on a single data stream, there are very few example of such systems. Finally, MIMD systems allow multiple streams of instructions to be executed on multiple data streams

[2]In some publications, SMP is used as an acronym for Shared-Memory Processor, however, in the vast majority of cases, and in this thesis, SMP is used to as an acronym for Symmetric Multiprocessor

in parallel. Computer clusters are generally considered as MIMD systems. MIMD systems are sometimes further subdivided into two subcategories: Single Program Multiple Data (SPMD) and Multiple Program Multiple Data (MPMD).

2.1.5 HPC programming models

A number of programming models have been developed to allow for the development of parallel software that can take advantage of HPC environments. These models have been shaped, to some extent, by the developments in HPC hardware. The SPMD programming model assumes that many instances of a program will be executing in parallel, and it passes certain responsibilities to the developer, such as segmenting data amongst each instance, and performing tasks specific to particular instances, such as handling inter-process communication and synchronisation. An alternative to the SPMD model is the global-view model, in which developers do not concern themselves with how their code will be parallelised. Instead, they write code in a serial manner, and parallelism is achieved through the use of certain constructs and the insertion of directives.

Within shared-memory environments, threads may be used to allow for the development of parallel software. In such systems, threads executing in parallel may communicate, and share data with each other, by accessing the same memory space. In distributed-memory environments, threads may be used along with sockets — although this is quite a low-level approach, and it can require a large and careful programming effort.

OpenMP is an API of functions that allows applications to make use of parallel computing capabilities in shared-memory systems. A single thread initially begins executing, and at certain points of the code, known as parallel regions, additional threads are created which run in parallel. Once the parallel threads have completed their work, they are joined back into the main thread. The creation of the parallel regions is controlled by the insertion of directives.

In the case of distributed-memory systems, which do not share any memory, the sharing of data amongst separate processes can be achieved through the use of message passing, such as MPI. Message passing is normally used in an SPMD manner, although this is not always the case, as MPI may be used to allow instances of distinct functions or programs to communicate with each other.

Typical computer clusters, are often described as hybrid systems, as they consist of both shared-memory as well as distributed-memory components. To make best use of such systems, multiple levels of parallelism must be exploited. Shared-memory and distributed-memory programming models are often used together. For example, OpenMP may be used to allow communication between processes within a single node, while MPI is used for communication between nodes, in effect, combining the message-passing model, with the multithreading model. More recently, concurrency tools have been developed which provide both approaches in a single implementation, such as OpenMPI. OpenMP and MPI are further discussed in Sections 5.1.2 and 5.1.3 respectively.

The PGAS programming model presents developers with an abstracted view of the global memory space, however it also allows developers to take advantage of data locality by allowing for data to be labelled as global or local. Although the specific implementations differ from language to language, the PGAS model is followed by a number of parallel programming languages, including Unified Parallel C (UPC), Co-array Fortran (CAF), Fortress, Chapel (see

Section 2.1.6), as well as by a number of extensions to Java, such as Titanium and X10 (see Section 2.2.6).

2.1.6 Languages used in HPC

In this section, a very brief summary is given of the languages which have been most widely used for the development of scientific applications, and those intended for use in HPC. Some recent developments in HPC languages are also given.

The Java language and platform are introduced later, in Section 2.2. After which, a number of extensions to the Java language, designed with the objective of enhancing its usability for the development of high-performance software for HPC environments, are described in Section 2.2.6.

2.1.6.1 Fortran

Fortran is a general purpose programming language, first developed by International Business Machines (IBM) in the 1950s. Previous to Fortran, applications were typically written in assembly language. Relative to assembly, Fortran is a much higher level language, therefore its adoption greatly improved programmer productivity. Fortran quickly gained popularity soon after its release, and within a few years, Fortran compilers were available for many hardware architectures.

Fortran became popular within the scientific community early on, and this popularity has continued to the present. Its popularity is due to a number of factors: the language syntax is well suited to performing numerical operations which are typical of scientific applications; it provides very high performance; many high-performance libraries have been developed for Fortran which are useful for developing scientific applications; due to its use over many decades, generations of scientists have used Fortran throughout their careers, and therefore a huge amount of experience has been gained with Fortran. Additionally, some natural inertia against change might have developed in some fields where Fortran has *always* been used.

There have been many official releases of Fortran, as well as many variants of the language released over the years. CAF [Numrich and Reid, 1998], initially known as F–, is an extension to Fortran which allows developers to write parallel code following the SPMD model. CAF introduced a number of parallel programming constructs to the Fortran specification, most importantly it introduced the *co-array* construct which allows processes, executing in parallel to share data. The co-array construct was incorporated into the mainstream branch of Fortran in *Fortran 2008*. High Performance Fortran (HPF) is an extension of *Fortran 90*, that provides developers with a global-view language, supporting distributed arrays and a single logical thread of computation. A compiler for generating Java bytecode from Fortran source code is described in [Seymour and Dongarra, 2003]. There have been many other versions and extensions of the Fortran language. Interestingly, one of the earliest studies into the advantages of using an Adaptive Optimisation System (AOS) was implemented using a version of Fortran [Hansen, 1974].

2.1.6.2 C family of languages

The C language, developed in early 1970s at Bell Laboratories, has arguable been the most successful computer programming language ever created, and it has spawned many variations,

including Split-C, C++, Objective-C and C#. Many of its features were based on an earlier language called B. C has been commonly used by the scientific and HPC communities due to the high performance that it provides. UPC [Ghazawi et al., 2005] is an explicitly parallel extension of C, suited for the development of software that will run in PGAS environments.

A significant difference between C and Java is that the definition of the C language does not precisely specify the sizes of data types, nor does it define rules on expression evaluation. This freedom allows C compilers to generate high-performance applications, specific to particular hardware environments. There is a downside to this freedom however, and that is that there may be cases where applications generated from the same source code, but compiled on different machines, produce different results. Such discrepancies are typically known as *rounding differences*. The Java language definition, on the other hand, precisely defines data type sizes and expression evaluation steps, and Java applications produce the same results across all environments (this strict reproducibility feature of Java can however be relaxed, developers can decide between reproducibility and performance optimisations through the use of the *strictfp* keyword, see Section 3.3.2).

C++ is a general purpose programming language, derived from C, which supports a number of programming paradigms, including procedural, functional, as well as Object-Oriented (OO). The Java syntax was heavily influenced by C++, and they are often compared with each other. Differences between the languages include the availability of pointers in C++; and the requirement for automatic GC in Java, while C++ allows for the manual memory management — making possible a range of memory bugs such as dangling pointers. C++ follows a Write Once Compile Anywhere (WOCA) compilation approach, allowing the creation of system-dependent executables, while Java's portability allows for Write Once Run Anywhere (WORA). Java is generally considered safer and easier to master than C++, additionally, Java offers higher developer productivity, and reduced development costs (see Section 3.2.2). Fortran and C-based languages have been, by far, the mostly widely used by the scientific and HPC communities.

C# is another member of the C family of programming languages. It is arguably the language which has most in common with Java at the moment. Like Java, C# is usually compiled to a platform-neutral representation that is executed in a runtime environment, called the Common Language Runtime (CLR) [Meijer and Goug, 2000]. The CLR is the managed runtime environment intended for the execution of the Microsoft .NET family of languages, which also includes VB.NET. The CLR mirrors the JVM in many aspects, and it is probably true that the development of the CLR was encouraged by the success of the JVM. The CLR supports the execution of code in the Common Intermediate Language (CIL) format, which like Java bytecode, is a stack-based representation. Like the JVM, the CLR provides a range of runtime services to applications, including GC and an optimising JIT compiler.

As in Java, objects are the fundamental building blocks of C# programs. C#, like Java, provides an extensive set of classes that may be used for storing and processing collections of objects, including *arraylists*, and *hashtables*. Both languages also support *reflection*, which allows for the type of an object to be determined. Both provide a very similar exception handling mechanism, although checked exceptions (see Section 3.2.6) are supported in Java and not in C#. Neither Java nor C# support multiple inheritance, although both allow classes to implement multiple interfaces. Unlike Java, C# supports the use of memory address pointers, although they are not considered part of the core C# language, are not widely used, and code which uses them must be marked as *unsafe*. The use of these pointers can potentially lead to memory problems, such as dangling pointers. Unlike most JVMs, the CLR follows a total-compilation strategy, therefore no code is ever interpreted.

2.1.6.3 HPC specific languages

In 2002, the Defense Advanced Research Projects Agency (DARPA) launched a research and development programme called *High Productivity Computing Systems* which offered funding for the development of new technologies to increase productivity and performance — including in the HPC domain [Weiland, 2007]. Based on the premise that only a small percentage of the competent serial developers are also competent parallel developers, the project aimed to develop languages which would be familiar to serial developers, but also offer parallel programming constructs in an intuitive manner. Three new languages were developed during this programme: Chapel, Fortress and X10.

Any language attempting to meet the requirements of the scientific and HPC communities must possess a number of characteristics, and offer a range of features. [Chamberlain et al., 2007] identified ten such features, including: providing a global view of parallelism; support of both task and data parallelism; data abstractions; broad-market language features, such as allowing OO programming; offering high performance, portability and interoperability.

Chapel [Cray, 2011] developed by Cray, offers a global-view programming model, building upon the approach taken by HPF, but also offering greater support for general parallelism. Chapel attempts to allow developers to write code relatively quickly, thus providing high productivity, but also allowing the progressive tuning of code over a longer period of time. It provides developers with high-level abstractions of threads, thus relieving developers of the burden of low-level thread management, such as performing *forks* and *joins*. Chapel supports task parallelism, allowing for the launching of many tasks in parallel using the *cobegin* statement. Chapel allows developers to write code in an OO manner, or to follow a procedural style. Chapel is now an open source project managed by Cray, but accepting contributions from the community.

Fortress [Allen et al., 2007], developed by Sun Microsystems and later managed by Oracle, was designed with the idea of creating a modern, extendible, and secure language offering the mathematical processing abilities of Fortran. The syntax of Fortress was designed to closely resemble the mathematical notation used in mathematical formulas, allowing statements to be written in Fortress source code in an identical manner to how they would be written on paper.

Fortress is OO, supporting multiple-inheritance, as well as representing primitive types as objects. It supports parallelism at a number of levels, including at the level of expressions, loops, and parallel regions. By default, certain constructs will be executed in parallel, if the application is running in a multi-core environment.

In 2007 Fortress became an open source project, in order to encourage participation and contributions to the language by its community of users. Fortress is an interpreted language, and in recent years efforts have been centred on developing a compiler that would allow Fortress to run in a JVM. In July 2012, Oracle Labs announced that further development on Fortress would end, citing problems between the Fortress type system and VMs — including, but not limited to the JVM.

The third of the *DARPA HPC languages* — X10, is described in Section 2.2.6, along with other extensions to the Java language, specifically intended to aid its use in HPC.

2.1.7 Trends in HPC

HPC is a fast changing field. Novel systems designed today, if they are judged to provide good performance, can quickly become mainstream. Software developers must be conscious of

the current trends in HPC system design, if they wish their applications to be able to take maximum advantage of HPC resources. For a number of years, some of the most important trends in HPC have been the move to multi-core processors (see Section 2.1.2), and the growth of the use of GPUs for general computing (see Section 2.1.3). In the following sections, some other emerging trends are discussed.

2.1.7.1 Supercomputers trends

The TOP500 project [TOP500, 2012] is an initiative which ranks and describes the 500 most powerful computers in the world. It was started in 1993 and since then, it has published an up-to-date list twice per year — in June and November. This list serves as a useful resource for observing trends in HPC. Today, what we call supercomputers, are typically composed of thousands of multi-core computing nodes connected with some high performance interconnect network such as Infiniband. As of June 2012, 75% of the machines in the TOP500 list use processors with six or more cores. Two years earlier, in June 2010, this figure was just 5%. In June 2012, 10.2% of the machines make use of GPUs, while two years earlier, only 0.2% (just a single machine) used GPUs. These statistics highlight the move to larger numbers of cores per processor, and the increasing importance of GPUs.

2.1.7.2 Big data

The volume of data being generated annually in the world is following a trend which is increasing even faster than Moore's law. According to IBM, 90% of the data in the world today has been created in the last two years [IBM, 2011], and this is expected to remain true into the future. This increase in the volume of data being generated is known as the *big data* trend, and it presents opportunities and challenges. The opportunities derive from the knowledge and insight that can be extracted from the available data, while the challenges are due to the fact that the manipulation of data generally becomes more difficult as the volume of data increases. *Data analytics* refers to the application of data analysis techniques to datasets, and this field is gaining increasing importance due to the ever larger volumes of data available for analysis. The trends towards an ever increasing number of parallel processes, running in many cores across many computing nodes, combine to make the issue of data management a critical issue.

2.1.7.3 Cloud computing

The term *cloud computing* is used to refer to the provision of computing resources as a service. Over the last number of years, this term has gone from being unheard of, to becoming one of the most commonly encountered *buzzwords* in computing. Typically, the computing resources offered by cloud computing providers are physically located in a data centre which is remote from its users. Users purchase the use of a certain amount of these resources for a period of time. One advantage to users of the service is that they can avoid the cost of purchasing and running computing resources themselves, and they only pay while they are actually using the resources. Critics of cloud computing point to the fact that users are expected to entrust their private data to, and can become dependent on, the service provider.

Hadoop [Borthakur, 2007] is a Java-based, open source data processing framework, managed by the Apache Software Foundation, that is often used in cloud computer when attempting to process large datasets using many computing nodes. It provides a scalable and secure framework

allowing data-intensive applications to make use of many distributed computing resources. Hadoop defines its own filesystem known as Hadoop Distributed File System (HDFS), and its uses an implementation of the MapReduce programming model to allow for the parallel processing of data.

2.1.7.4 Green computing

The term *green computing* is used to refer to attempts to reduce the effect that computing has on the environment. The increasing importance of green computing is being driven by two factors: challenging economic conditions leading to a desire to minimise power consumption, and an increased awareness of the effect of computing on the environment — CO2 emissions from the Information and Communication Technologies (ICT) industry already equals that of the airline industry. The carbon footprint of large data processing centres and supercomputers is very significant. The cost of operating a large computing facility for a year, such as a supercomputer or large data processing centre, can exceed the initial cost of purchasing and installing the hardware, and can be the equivalent of providing power to several thousand typical homes. An increasingly important performance metric used in HPC is performance per watt, which is used to evaluate and compare the computing power provided by a system against the power consumed. Comparing the no. 1 rated supercomputer in the June 2012 version of the Top500 list against the no.1 rated supercomputer from November 2011 — *Sequoia* from IBM and the *K computer* from Fujitsu respectively — there has been a 78% increase in the theoretical peak performance, while there has been a 37% drop in the power required, resulting in a much improved performance per watt score. This dramatic improvement is testament to the importance currently being placed on improving performance per watt. The *Green500 List* is a list, modelled on the TOP500 list, which ranks the 500 most energy efficient supercomputers in the world.

2.2 The Java platform

Java is one of the most widely used computer programming languages in the world [Tiobe, 2012]. It is a general purpose, OO, high-level language, with a syntax quite similar to C++. It is generally considered to be easier to program with, and to achieve fluency in Java than in C++. One of the reasons for this is the fact that Java developers do not have to concern themselves with low-level memory management, thanks to the existence of Garbage Collection (GC) in Java. The absence of pointers from the Java language also simplifies Java code, and removes a common source of bugs in some other languages. Java is used for creating software for all kinds of uses, and it runs on a vast range of hardware environments from hand-held devices to supercomputers. Today, the specification of the Java programming language is controlled by the Java community itself, through an organisation called the JCP. Anyone can become a member of the JCP, and any member can propose changes to the specification. The reference implementation of the Java platform is now the open source OpenJDK implementation, and Java is now generally considered to be an open source software project [3].

[3] Oracle, as the current owner of Java, still maintains considerable control over the Java platform through its licensing model, see Section 2.2.5

2.2.1 The origins of Java

The origins of Java lie in a small project which began within Sun in 1991, known as *Project Green*. The objective of this project was to develop a programming language that could be used for writing software that could run on consumer devices, such as PDAs, VCRs and set-top boxes. These devices could be characterised as possessing a range of processor architectures and limited amounts of processing power and memory. Therefore, the new programming language would need to be able to write software that could be run on a variety of hardware platforms. This portability requirement posed a technical challenge to the engineers of Project Green, and the approach that they followed would lead to one of the core strengths of Java.

Initially, the engineers working on Project Green considered extending the C++ compiler to meet their needs. However, eventually they came to the conclusion that starting from scratch would be their best option. They decided that the new language would not be compiled to native machine code at compile-time. Instead, they followed a compilation and execution approach that had previously been used by some implementations of the Pascal programming language. The approach that had been used with most programming languages involves application code being translated (or compiled) to native machine code at compile-time, which is normally referred to as *static* or *ahead-of-time* compilation. Each hardware architecture supports a particular instruction set, therefore applications must be compiled for a particular hardware architecture. An executable which is compiled for a particular architecture should only make use of instructions which are supported by that architecture. This means that application code must be compiled for each particular hardware architecture, possibly resulting in many executables running on different machines. In contrast, the compilation and execution processes followed by the Project Green engineers involves the compilation of application code into an intermediate code — which is known as bytecode. The instructions in the bytecode are not specific to a particular hardware platform. Instead, they are generic in nature, and are intended to be executed on a hypothetical machine — or a Virtual Machine (VM). The same bytecode can be executed on any machine, as long as that machine possesses a VM which can interpret the instructions in the bytecode. This approach, which is illustrated in Figure 2.1, offers a number of benefits, chiefly amongst them is the fact that application code only needs to be compiled once, as opposed to being compiled for each hardware environment. This is known as the WORA advantage. Project Green resulted in an OO, platform-neutral language that was initially called *Oak*. Credit for much of the initial design of the new language is generally given to James Gosling — one of the engineers working on Project Green.

Sun established a subsidiary company, called *First Person Inc*, to develop software for small consumer devices used in the cable television market, using Oak as the development language. This company did not enjoy much success in its attempts to win contracts and it was later wound down by Sun in 1994. However, around this time the Internet — which provided a mechanism for sharing content amongst a massive audience of users, regardless of the hardware platform of the individual users — was beginning its meteoric rise in popularity. The team behind the new language realised that many of the features that they had included in their language could be very useful in this new age of content sharing amongst heterogeneous hardware platforms. Typically, users access content on the Internet using a web browser, Sun realised that by using a browser in cooperation with a JVM, users could download, and then execute the same bytecode regardless of their hardware platform. It would not matter if one user was running Microsoft Windows, while another was running one of the many distributions of the Linux Operating System (OS). As long as both had a JVM installed on their machine, both could execute the same bytecode. The First Person engineers moved their focus to the Internet, and in 1994 they

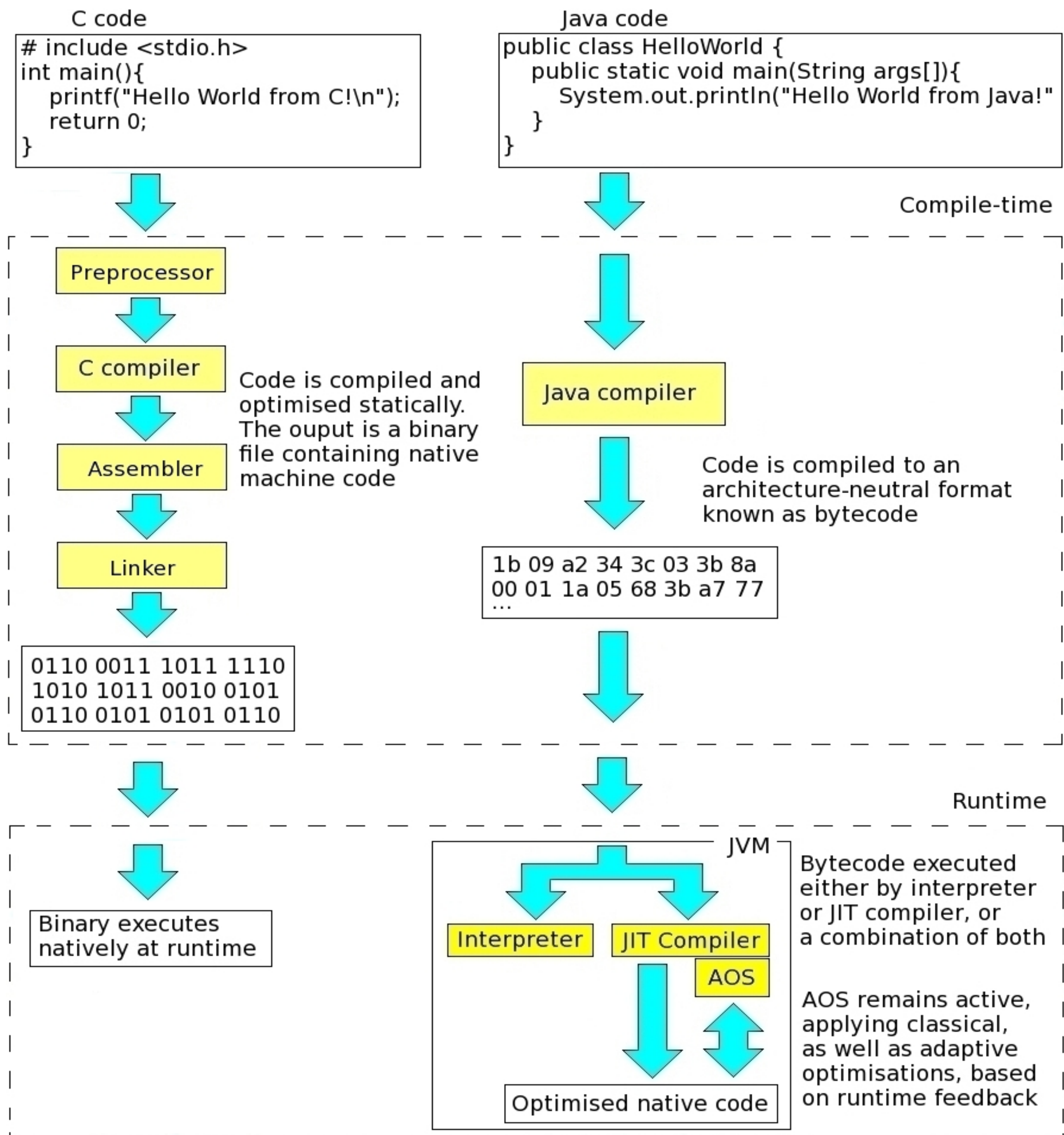

Figure 2.1: Simplified overview of Java's compilation and execution processes, compared with those of C. The most obvious difference is the simplification of the compilation process at compile-time for the Java platform, and its more sophisticated runtime environment.

developed a web browser prototype, which would later become the HotJava browser. This was the first browser to support the execution of downloaded bytecode. These applications running inside a browser were named applets, and soon all the widely used browsers supported the execution of Java applets. The usefulness and popularity of applets were part of the reason for the sudden rise in the usage of the new language. They demonstrated one of its fundamental strengths — its portability.

In 1995, Oak was renamed to Java, and the following year, the first public release of the Java platform was made. Since then, there have been regular major and minor releases, which have updated and expanded the specification of the Java language, as well as the components which make up the Java platform. Java has evolved from a language intended for creating applications for a particular market, into one of the most widely used programming languages in the world, with a vast community of software developers using Java on a diverse range of projects.

In the past, there has existed some confusion regarding the correct terms to use when referring to the components which make up the Java platform, as well as the correct name to use when referring to a particular version of Java. This confusion was primarily due to a tendency within Sun to change its own naming conventions. In the following sections, we describe the main components of the Java platform, as well a summary of the version history of Java, together with a description of how the naming conventions have changed over the years.

2.2.2 The main components of the Java platform

The term *platform* is widely used in computing to refer to a particular environment, often it is used to encompass an OS and a hardware architecture (normally the processor architecture). For example, a Windows OS running on an Intel processor, or a Linux OS running on an AMD processor. Typically, software compiled for one platform cannot run on a different platform. The term *Java platform* is used to refer to the software that allows for the development and execution of Java applications. The Java platform effectively sits on top of an underlying OS/architecture platform, and is composed of a number of components. The main components of the Java platform are described in the following sections.

2.2.2.1 Java development kit

A JDK contains a set of tools for developing and executing Java applications. Strictly speaking, the term JDK should only be used to refer to a Java development kit produced by Oracle (and formerly by Sun), however the term is commonly used to refer to the Java development kits from other implementers of the Java platform, for example the IBM JDK. The tools included in a JDK include the Java compiler, called *javac*, which compiles Java code into a format known as bytecode. A JDK also contains a range of tools, both command-line and graphical, which are useful for debugging and monitoring purposes, such the *jdb* debugger and VisualVM. The JDK contains within it a JRE, which in turn contains a JVM.

2.2.2.2 Java runtime environment

A JRE is a software component that can execute Java applications. A JRE may exist on its own, or it may form part of a JDK. A JRE consists of two main components: a JVM, and a

set of libraries which implement the Java API and are known as the Java Class Library. How these components relate to each other is illustrated in Figure 2.2.

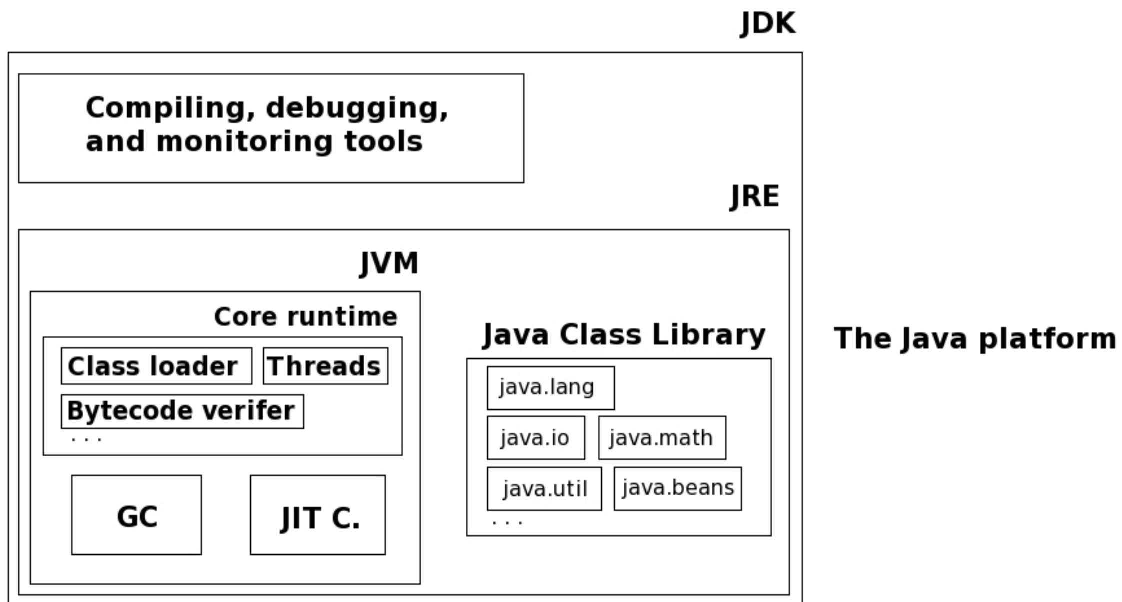

Figure 2.2: How the key components of the Java platform relate to each other.

2.2.2.3 Java virtual machine

A JVM is a software component capable of executing Java bytecode. Like the Java language itself, which has an official specification [Gosling et al., 2012], there is also an official specification for JVMs [Lindholm et al., 2012]. A JVM which adheres to this specification is known as a compliant JVM. Although the specification defines how a JVM should behave in certain circumstances, compliant JVMs still have a high degree of freedom in how they implement certain functionality, and in the options that they provide to users. For example, GC may be implemented in whatever way the JVM developers wish, as long as the JVM meets the memory management specifications.

Java bytecode is the format which is produced by the Java compiler — *javac*. It is stored in a file known as a Java class file, which are often packaged together into Java ARchive (JAR) files. Bytecode is not the native machine code for any CPU architecture. Instead, it makes use of the instruction set supported by a JVM. Formats such as Java bytecode, which can be interpreted by a VM, are sometimes referred to as Directly Interpretable Representation (DIR) formats, as opposed to native machine code which is referred to as a Directly Executable Representation (DER) format. Bytecode is a compact, stack-based format, consisting of a series of operations, with zero, one or multiple operands. Its operations are encoded using a single byte, therefore the maximum number of supported operations is 256. As well as the actual bytecode instructions, Java class files also contain a section known as the *constant pool*, which is used for storing pieces of data required by the code. References are included in the bytecode which link to items in the constant pool. It is the responsibility of JVMs to execute bytecode, they have generally followed two approaches to do this:

- JVMs may interpret bytecode. This involves the JVM emulating each bytecode instruction as a function of the JVM. In this case, the only native machine code which executes is the code comprising the JVM itself. The JVM was originally designed to interpret bytecode,

however the performance obtainable using this approach is much lower than that which can be obtained by executing native machine code.

- The JVM may compile parts (or all) of the bytecode to native machine code, and then call this native machine code.

Some modern JVMs follow a total-compilation approach, this involves all of the bytecode of an application being compiled to native machine code before execution. Other JVMs use a combination of interpreting and compiling, in a strategy known as MMI. See Chapter 8 for more on the approaches that JVMs may take while executing code.

The JVM is composed of a number of components. These include the core runtime system, the garbage collector and the JIT compiler. The core runtime system itself contains a number of components, and it performs many tasks including class loading, bytecode verification and thread management. The Classloader has the responsibility of loading Java class files. Classes are usually first loaded when they are actually required. Classes may also be unloaded in order to free-up memory space. The bytecode verifier performs a number of checks on all bytecode before attempting to run it, to ensure the safety and stability of the system. No operations should be permitted which could potentially cause the JVM to crash.

GC is the name given to an automatic memory management system. Java, like many other languages which execute in a managed runtime environment, relies on a GC service to manage memory while applications are executing. See Chapter 7 for a detailed description of the topic. The JIT compiler is a component within the JVM which can compile bytecode into native machine code, and apply optimisations to the code. The JIT compiler allows for much better performance than simply interpreting bytecode. The optimisations that may be performed at runtime include some which are not available to statically compiled languages, such as the aggressive inlining across virtual method invocations. In some cases, these allow Java to outperform other languages.

Although the JVM was initially designed for the execution of Java applications, any language which can be compiled to bytecode can be executed using a JVM, and there exists bytecode compilers for a number of languages, including C and Python. Together, a JVM and the Java Class Library — which implements the Java API, form a JRE.

2.2.2.4 Java class library

The Java Class Library is a large set of dynamically loadable libraries that Java applications may call at runtime. As the Java platform is independent of any particular hardware environment or OS, it is not possible to call system libraries from Java applications, as is possible in some other languages. Instead, the Java Class Library provides the functionality that might be included in system libraries, such as file I/O in *java.io* and *java.nio*, and multithreading capabilities in *java.lang.thread*. Because the underlying OS is *abstracted away* behind the Java Class Library, Java applications are not system dependent.

In addition to providing a means for interacting with the OS, the Java Class Library also provides a huge library of useful and commonly used functionality to applications, such as mathematical operations (in *java.math*). The Java Class Library has grown in size with each successive major release of Java from a few hundred classes in the initial release of Java to over 3000 in JDK 7.

2.2.3 Version history of Java

Firstly, it should be noted that the version number that is used to identify a particular version of Java is the version number of the Java Development Kit (JDK) — not the version of the Java Virtual Machine (JVM). Java 1.0 was released in 1996, and it was orientated for creating web applications, however it lacked many of the basic capabilities that programmers had come to expect from a programming language. It even lacked the ability to perform a simple *print* statement. The following year, Java version 1.1 added many capabilities to the Java platform, including Remote Method Invocation (RMI), support for database access through JDBC, as well JavaBeans, and support for inner classes and reflection. Java was beginning to look like a general purpose programming language.

In 1998 Sun released “Java 2 Standard Edition Software Development Kit Version 1.2”. Renaming the entire platform to “Java 2” to represent the fact that this version of Java was greatly updated from the previous version. Later, the “1.2” part of the name was dropped, and it simply became known as Java 2 Standard Edition (J2SE). J2SE included the JIT compiler, giving the option of compiling code instead of interpreting it, and greatly increasing the level of performance that could be obtained from Java applications.

In 1999 Sun released JDKs designed specifically for particular platforms, including Java 2 Enterprise Edition (J2EE) for enterprise applications, and Java 2 Micro Edition (J2ME) for mobile applications. J2SE 1.3, released in 2000 included the HotSpot JVM (see Section 2.2.4.1), and represented a significant step forward in terms of performance. J2SE 1.4, released in 2002 was the first version of Java developed under the JCP, with features such as the Java Web Start (JWS) framework, regular expression processing modelled on Perl, and support for exception chaining.

The next version of Java, released in September 2004, was initially due to be called J2SE 1.5, however Sun decided that as this version included a significant number of additions to Java, it merited the version number “5.0”. Therefore the full name was “Java 2 Standard Edition 5.0”. J2SE 5.0 added generics to the Java language, providing an extra level of compile-time type checks which in turn can prevent casting exceptions from occurring at runtime. A new set of utilities designed to assist in the development of multithreaded applications was introduced, known as the Java Concurrency Framework. It provides developers with a set of threading utilities at a higher-level of abstraction than was available previously. Developers no longer needed to implement multithreading functionality such as thread pools, leading to better developer productivity. The JMX framework was added to the Java platform, providing powerful monitoring and management capabilities to Java applications. The JVMTI was introduced, providing an API that allows for the inspection of the state of JVMs, and for the control of applications running in them. Other new features in this version of Java include autoboxing, enumerations, and a new *for loop*.

In December 2006 Sun released Java Standard Edition (SE) 6, on this occasion Sun renamed J2SE to Java SE — dropping the “2”, and also dropping the “.0” from the version number. Sun later released many updates to this version, which included several improvements to the performance of the JVM, including major improvements to the JIT compiler and GC. Java SE 6 also included a lightweight database called Derby.

Java SE 7 was released by Oracle in July 2011, Sun having been purchased by Oracle in 2010 (see Section 2.2.5 for more on the control of the Java platform). This version contained a large number of changes to the Java platform, including JVM support for the execution of dynam-

ically typed languages such as Ruby and Python. This version includes a new concurrency framework known as Fork/Join. Networking, I/O, and JMX MBeans monitoring and reporting capabilities were all also enhanced.

At the time of writing, Java SE 8 is currently expected to be released mid-2013. It will bring further modularisation of the JDK, as well as a number of further additions to the language specification. Java SE 8 will include *Lambda expressions*, sometimes known as *anonymous functions*, which may be passed to other functions as constant values, and which execute within the scope of where they appear, meaning that they may access variables that are defined within that scope. They are already supported in a number of other languages, including LISP and Python, and are considered a useful addition to Java. Amongst the features which Oracle has suggested will be included in Java SE 9 are better support for very large heaps, better native machine code integration, and more sophisticated JVM ergonomics.

Although the Java version naming convention does now seem to have settled into a predictable pattern, there is one issue which can still cause a little ambiguity. For any particular version of Java, such as Java SE 7, two numbering systems are used to refer to the same version. In this example, "JDK 7" (Java Development Kit 7) is used to refer to the development kit, while the version number "1.7" is used as the internal version number of the components that make up JDK 7, such as the JVM. This can lead to some confusion, and in many cases the numbers "7" and "1.7" are used interchangeable, although strictly speaking "JDK 1.7" does not exist, and neither does "JVM 7".

2.2.4 The Java ecosystem

Many companies and groups have been involved with the development of Java technology over the last two decades. The term *Java ecosystem* is used to refer to the broad spectrum of entities involved with the development of Java technology. As the creators of the Java platform, Sun naturally controlled its development for several years. As the usage of Java grew, the number of companies and software developers with an interest, and an opinion, on what would be best for the technology also grew. A movement for the development of an open source implementation of Java emerged, and there have been several projects with this objective, including the Kaffe JVM and the Harmony project from Apache.

Sun took steps quite early to involve the Java community in the development of the platform, through the establishment of the JCP, and some years later the OpenJDK project (see Sections 2.2.4.7 and 2.2.4.8 respectively). After its purchase of Sun, and thereby becoming the owner of Java technology, Oracle continued the commitment to these initiatives.

However, despite the very significant progress which has been made to *open up* Java technology, there are still concerns within the Java community about the direction of the technology. There are some concerns that Oracle dominates the JCP, and that it still retains too much control over the future of Java. Some have voiced concerns that decisions could be taken based on commercial reasons rather than in the interest of the wider Java community. Additionally, Oracle retains control over the Java trademark, and it owns various patents used in the Java platform. Oracle currently has license agreements with other implementers of the Java platform, and it may impose its own conditions on such implementations.

In the following subsections, the companies and groups which have played the most significant roles in the development of Java, as well as the main implementations of Java platform are described, and finally, the future outlook for Java is discussed.

2.2.4.1 HotSpot

HotSpot is a JVM maintained and distributed by Oracle. It includes a sophisticated JIT compiler and GC service. Its name is derived from the adaptive optimisation techniques used by the JVM to improve performance. As it executes an application, it continuously attempts to identify *hotspots* within the code. These are parts of the code which are executed frequently, and therefore, it is logical to concentrate optimisation efforts in those sections of code.

HotSpot was first developed by a small company called Anamorphic. The first public release was in 1999. Thanks to the impressive performance that it delivered, the HotSpot JVM was soon recognised as representing a significant improvement over previous JVMs. Sun decided to purchase the Anamorphic company together with the HotSpot JVM, subsequently the HotSpot JVM was included as the default JVM within the Sun JRE from Java version 1.3 onwards.

The HotSpot JVM actually contains two separate JVM runtime implementations known as *Client* and *Server*. Each one is optimised for use in a particular class of environments and for particular workloads. Client is optimised for use on relatively modest hardware environments, and for applications that require a fast startup time. Server is optimised for use on more powerful hardware, such as computer servers with many processors and relatively large amounts of memory. The Server runtime is configured by default to attempt to perform more aggressive optimisations, and it delays the application of these optimisations until an application has been running for some time. Unless a particular runtime is specified, the HotSpot JVM automatically checks certain characteristics of the environment in which it is executing and selects either the Client or the Server version accordingly.

2.2.4.2 JRockit

JRockit [Lagergren and Hirt, 2010] is a JVM with a reputation for high-performance, high-responsiveness and allowing for accurate profiling and monitoring without incurring much overhead. It was originally developed by a small company called Appeal Virtual Machines in the early 1990s. They had worked on JVMs for the Smalltalk and the Self programming languages, and made their first public release of the JRockit JVM in 2000. JRockit quickly gained a reputation as one of the best performing JVMs available. The company was purchased by BEA Systems in 2002, which resulted in extra funding for research and development efforts. Soon JRockit established its position in the market as one of the most widely used JVMs — together with the HotSpot and J9 JVMs. For a number of years, JRockit was at the forefront of developing adaptive code generation techniques, and speculative optimisations based on runtime feedback.

The JRockit JDK contains a powerful suite of profiling, monitoring, diagnostics and debugging tools called JRockit Mission Control (JRMC). These include the JRockit Management Console, for monitoring the state of JVMs; the JRockit Flight Recorder (JFR), based on the idea of an aircraft flight recorder, it can be used for logging the events that happen during the execution of an application; and the JRockit Memory Leak Detector. These tools have access to the inner workings of the JVM, and their execution results in very little overhead. Unlike many other diagnostic tools, they can be attached to an already running JVM. These tools are further described in Section 6.3.5, as part of our description of profiling tools for Java.

BEA Systems was purchased by Oracle in 2008, and two years later Oracle also purchased Sun, thereby becoming the owner of both the JRockit and the HotSpot JVMs. It did not make commercial sense for Oracle to continue to develop and release two competing JVMs using two

different teams of software engineers, so it decided to begin to merge these JVMs. The goal of this merger was to create a single JVM with the best features of both. The HotSpot JVM within JDK 8 is expected to be the first JVM from Oracle to contain JRockit features. Going forward, JRockit will no longer be released as a standalone JVM, however it will continue, at least for the moment, to be the JVM used by the JRockit family of products, which have now been integrated into the Oracle set of products. Oracle is also continuing the development of the powerful JRMC suite of tools, and it will continue be shipped with a number of Oracle packages that include JRockit technology. As the time of writing, Oracle recently made the first release of Java Mission Control — a port of JRMC for the HotSpot JVM. This system supports HotSpot versions in Java SE 7 update 4 and later. It is not clear if the JRMC will be totally replaced by Java Mission Control in the future. Additionally, Oracle has promised that it will contribute the best features from JRockit to the OpenJDK project.

2.2.4.3 J9 — IBM Java virtual machine

The J9 JVM [Grcevski et al., 2004], from IBM, is a high-performance compliant implementation of the official JVM specification. J9 forms the core of a number of highly successful IBM products, including the WebSphere brand of products. It was designed for execution on a range of hardware environments, from mobile devices to powerful server-class machines with large numbers of processor and large amounts of memory. J9 was designed in a modular manner, with many individual components, although only a small number of these are mandatory. For example, the *j9vm*, *j9jit* and *j9gc* components (core JVM functionality, JIT compiler, and GC respectively) must be used, while the *j9prf* and *j9bcv* (performance monitoring framework, and bytecode verifier respectively) are optional. J9 contains a number of monitoring and diagnostic capabilities which are proprietary to IBM. The TestaRossa (TR) JIT compiler is used with the J9 JVM. It offers a wide range of optimisations, including classical as well as adaptive optimisation techniques — which are speculative in nature. Like the HotSpot JVM, it performs application profiling to locate the *hot* methods of the executing application, and it then concentrates its optimisation efforts in these methods. The range of GC policies and tuning options are also proprietary, and are quite extensive. In this thesis, we use the terms *IBM JVM* and *J9 JVM* interchangeably.

2.2.4.4 Jikes research virtual machine

The Jikes Research Virtual Machine (RVM) [Alpern et al., 2005] is an open source and free software VM that is intended primarily for research and investigation purposes. Although Jikes is often referred to as a *JVM*, technically this term should not be used as it is a trademark of Oracle (formerly of Sun), and only those implementations which are certified as compliant can be referred to as a JVM. For this reason, Jikes is referred to as a RVM (also emphasising the fact that it is primarily intended for research purposes) or simply as a VM. The Jikes RVM began as an internal IBM research project called *Jalapeño*, with the objective of developing a flexible and extendible VM that could be used for developing and testing new functionality and techniques. As the team of researchers who were working with the Jalapeño VM began to publish their work, interest in this research VM grew amongst the academic community. This led to IBM making Jalapeño available to a number of universities through a licensing agreement. Eventually, it was decided to turn Jalapeño into an open source project, and it was made available under an open source license in October 2001. The project was renamed to Jikes due to copyright issues. Unlike most other JVMs, the Jikes RVM itself is written in Java,

and it is self-hosted — it can run its own code without requiring another JVM. Jikes can make use of the Java Class Library from the GNU Classpath or from the Apache Harmony project. The Jikes RVM has been used to investigate many VM topics, and in particular, it has been heavily used in the investigation of memory management topics such as GC. Jikes contains a component known as the Memory Manager Toolkit (MMTk) which allows for the development of novel GC techniques and their use with the rest of the VM.

The Jikes RVM does not include an interpreter, therefore it always compiles bytecode to native machine code before executing it. This initial compilation is usually performed as quickly as possible, without apply any optimisations. Later, as the *hot* parts of the application are identified, optimisations are applied to these.

2.2.4.5 The GNU project and Java

The GNU project is a long running free software initiative, founded by Richard Stallman in 1983. In this context, the term *free* is not used to mean *free of cost*. Instead it refers to the rights of anyone to freely execute, study, modify, and redistribute the software. The terms *free software* and *open source* are generally considered synonyms. Despite the fact that the term *open source* is now more widely used, the GNU project continues to use the term *free software* as it emphasises the principles of freedom that were the core reasons for its founding. Stallman also founded the Free Software Foundation (FSF) is 1985, to support the free software movement. Software deemed to be *free* by the FSF is said to be Free and Open Source Software (FOSS).

The GNU project created a license agreement known as the GNU General Public License (GPL). Open source software is often distributed under a GPL, which permits the free execution, inspection, modification, and further distribution of the software. The GPL license agreement, sometimes referred to as a *copyleft* license, states that software under that license (including modified versions of this software) may only be distributed under that same license terms. This prevents third parties from modifying free software, and releasing a version of the software (possibly a closed implementation) under some other license. A number of GPL versions have been released. One set of licenses, known as Lesser General Public License (LGPL) permits the associated software to be used by other applications, even if the other application is not under a GPL. This license is often associated with libraries, and allows non-GPL applications to link these libraries during building. This extra permission, that is present in the LGPL, is referred to as a linking exception.

Under the GNU project, an OS has been developed, as well as a range of libraries and applications. The GNU Classpath project is one such library, and it aims to provide a free and open source implementation of the entire Java Class Library. The GNU Classpath project merged with a number of other projects over the years, and it is now used by a number of JVM implementations including the Kaffe and Jikes VMs. Additionally, the GNU Compiler for Java (GCJ) [GCJ, 2012], has been developed under the GNU project. The GCJ can compile Java source code to native machine code, as well as bytecode to native machine code.

2.2.4.6 Dalvik

Dalvik is an open source VM developed by Google and used in the Android OS to run applications, commonly known as *apps*. Dalvik was designed with the restrictions which exist in mobile devices in mind, such as limited battery power and memory. Internally, the Dalvik VM

operates quite differently from the standard JVM, using a register-based data storage scheme instead of the stack-based scheme which is used in the JVM. Dalvik can execute files known as Dalvik executables, which are the result of performing some transformations on bytecode to reduce their memory footprint. Dalvik includes it own Java Class Library, based on the Apache Harmony implementation.

2.2.4.7 Java Community Process

Since 1998, changes to the Java platform have been managed by the JCP. The JCP was initially established by Sun with the objective of allowing the Java community to work together on the development of Java technology. The JCP is governed by an Executive Committee (EC), consisting of representatives of companies and groups with an interest in Java technology. As the current owner of Java, Oracle plays a leading role in the JCP. Participation in the JCP is governed by an agreement called the Java Specification Participation Agreement (JSPA), which is entered into between Oracle and the participating entity. Changes to Java can be proposed by any member of the JCP, through the creation of a Java Specification Request (JSR). Each JSR has an associated suite of tests, known as a Technology Compatibility Kit (TCK), and a reference implementation which are used to check if an implementation conforms to the specification of the JSR. An implementation of the Java platform must pass an extensive and rigorous suite of tests known as the Java Compatibility Kit (JCK) in order for it to be certified as compliant with the specification. Oracle retains licensing control over the running of the JCK. If Oracle does not allow an implementer to run the JCK then they cannot claim that their implementation of Java is compliant.

2.2.4.8 OpenJDK

OpenJDK is an open source and free software implementation of the Java platform which is available under the GPL. Specifically, OpenJDK is released under GPLv2 with a linking exception which allows parts of the Java Class Library to be used (or linked) by other applications, even if those other applications are not under the GPL. The project was started by Sun in 2006 when it made the majority of its implementation of the Java platform open source. Later, in May 2007, Sun released the complete source code of the Java Class Library under the GPL, except for a small number of exceptions due to some third parties not accepting the terms of the GPL. Over the course of the next few years, these encumbered components were either made open source, or were replaced by open source versions, leading to the complete Java Class Library being open source by 2011.

Over the past few years, the OpenJDK project has grown in importance, with the majority of the large software companies involved with the development of Java having agreed to become contributors to the project. In 2010 both IBM and Apple announced that they would join the project. Previously IBM had been one of the largest contributors to the Apache Harmony project (an alternative open source and free software implementation of the Java platform), while Apple had previously developed its own Java implementation. In 2011 SAP — a large software development company with its own JVM implementation, and involved in VM research — announced that it would also be joining the OpenJDK project. These announcements meant that a significant number of the largest software companies in the world — including Oracle, IBM, and Apple were now committed to the OpenJDK project.

As of Java SE 7, OpenJDK serves as the reference implementation of the Java platform. Oracle

uses the OpenJDK codebase, together with some additional non-open source add-ons, to build its closed JDK and JRE releases, which are then released under the Binary Code License (BCL).

IcedTea

Following the establishment of the OpenJDK project by Sun, it became possible to make a build of the JDK project, however not all of the third party libraries making up the Java Class Library were open source. In order to build the OpenJDK, these libraries had to be included as binary plugins. This represented a problem for some Linux distributions which only permit free software that does not reply on any non-free software.

The IcedTea project was started by Red Hat in 2007 with the objective of allowing for the building of the OpenJDK without the need for the encumbered libraries. Instead, IcedTea planned to use a set of libraries from the GNU Classpath project. IcedTea itself does not actually contain the OpenJDK code, instead its build process involves the downloading of the code from the OpenJDK repository. The first release of IcedTea was in November 2007. Although the binary plugins are no longer required to build an open source implementation of the OpenJDK, the IcedTea project has continued to be widely used amongst the Linux community as it provides some other add-ons not present in the OpenJDK.

2.2.5 Control of Java

After the establishment of the Java Community Process (JCP) and OpenJDK projects, Sun still retained ownership of the Java trademark, the right to execute various JCK (required by an implementation if it wished to claim that it is Java compliant), as well as patents on various technologies used in the Java platform. This meant that Sun, and now Oracle, retains strong legal rights over Java. Oracle is currently following a strategy with the Java platform which is known as an *open core model*. This strategy involves a product being made available in an open source manner, while an implementation of the product is also made available as a closed implementation, but with extra features included which are not open source. In the case of the Java platform, Oracle makes the OpenJDK implementation open source. However it also makes its closed implementation, known as the Oracle JDK, available with extra features. Therefore, Oracle is currently involved in two implementations of the Java platform:

- OpenJDK - which is an open source implementation. Oracle has said that it will contribute almost all of its HotSpot developments to the OpenJDK codebase. This implementation is made available under GPL.
- Oracle JDK - which is a closed implementation. This implementation contains the HotSpot JVM. The codebase of this implementation is based on the OpenJDK codebase, but with some added components. Binary releases of this implementation are made available under the BCL.

Given that the Java platform source code is available in the OpenJDK project, one might ask the question "what is stopping some entity from using that code to start a new fork of Java?" — which would be outside of the control of Oracle. The answer is that Oracle still retains ownership of the Java trademark and various patents in the Java specification. An implementation of the Java platform cannot be called Java, or obtain the Java compliant status, unless permitted to do so by Oracle. And few developers would be interested in using an implementation of the Java platform which was not recognised as adhering to the standard.

Oracle is following a dual licensing strategy for implementations of the Java platform. It offers one license to the non-profit implementations of Java platform which are based on the OpenJDK project. This license includes a Field Of Use (FOU) restriction which limits where the software may be executed. In particular, it prevents their use on embedded devices. Oracle enters into licensing agreements with commercial companies wishing to obtain the Java compliant certification for their implementations. Through these licensing agreements, Oracle effectively retains control over Java, while still working together with the Java community to advance the platform.

2.2.6 Extensions to Java for HPC

Although Java is relatively well equipped for the development of parallel software — with its built-in multithreading, networking and synchronisation features, there have been several proposals, and implementations, of extensions to the Java language, in order to enhance its usefulness and its productivity in this regard. In this section we describe some of these projects.

Titanium [Yelick et al., 1998] was a project that attempted to develop a new High-Performance language for the development of parallel scientific software. It includes a language and a runtime environment. The Titanium project selected Java as its base language due to the appealing features of Java, including its clean and OO syntax. Amongst the features that it added to the language include: immutable classes, *truly* multidimensional arrays, an explicitly parallel SPMD model of computation with a global address space, and zone-based memory management. Titanium supports shared-memory as well as distributed-memory environments. It can make use of several network interconnects, including Infiniband, Ethernet and Myrinet, using the GASNet language-independent networking layer.

X10 [Charles et al., 2005] developed by IBM, is based on Java, but it adds a number of new concepts and programming constructs. X10 attempts to deliver high performance and high productivity parallel programming, also allowing developers to take advantage of the typical designs of HPC environments. In particular, X10 is designed to make best use of the NUMA characteristics present in many HPC environments — which they refer to as Non-Uniform Cluster Computing (NUCC).

X10 contains a number of new constructs to take advantage of NUCC, which are not present in Java, including *places*, *regions*, and *distributions*. X10 also introduces the concept of *activities* which are similar to threads, but at a slightly higher level of abstraction. A *place* is a domain in which a certain number of activities and data reside, and it could, for example, be mapped to a single processor, or to a single computing node. A number of constructs are provided to allow for the synchronisation of *activities*, including *clocks* (similar to barriers) and *atomic sections*.

In addition to the language itself, IBM has developed an Integrated Development Environment (IDE) called X10DT (X10 Development Toolkit). It is based on the Eclipse IDE which is very widely used amongst the Java development community. X10DT includes the type of development tools that Java developers have become accustomed to, and the availability of such an IDE greatly increases the attractiveness of the X10 language.

Building on the definition of the X10 language, Habanero-Java (HJ) [Cave et al., 2011], developed by researchers at Rice University, is a further extension of the Java language intended for the development of HPC applications. Its primary objective is to allow developers to more easily take advantage of homogeneous and heterogeneous multi-core environments by providing high-level, safe and efficient parallel constructs. Its includes a HJ compiler which generates

standard Java classes, but with calls to HJ libraries. It also includes a HJ runtime that provides support for HJ constructs, manages the creation and execution of HJ tasks, and ensures that multi-core processors are utilised appropriately. The HJ extensions to the Java language add support for complex numbers, as well as multidimensional views of single dimensional arrays.

The addition of language extensions to facilitate the automatic compiler parallelisation of code, including the extensions supported in X10, are further discussed in [Shirako et al., 2008]. They proposed the addition of a number of additional enforceable declarations and annotations that not only ease the task of automatic parallelisation, but also enable more aggressive optimisations of sequential applications. They differ from the directives used in OpenMP, which are exclusively intended for the parallelisation of code, and which is typically achieved through the creation of parallel regions which involve the execution of multiple threads in parallel. Some of the features which have contributed to the success and high-productivity of Java; such as dynamic memory allocation, object encapsulation and error handling; make the automatic parallelisation of code more challenging than it is for traditionally used HPC languages. This is due to the fact that they make it is more difficult for parallelising compilers to determine data dependencies and flow control. The proposed features include the use of OO multidimensional arrays (as already supported in Titanium and X10), the introduction of *array views*, *points* and *regions*, as well as the *disjoint* and *retained* modifiers for method declarations. They performed a number of tests comparing the performance of ten benchmarks from the JGF suite of benchmarks. They compared four implementations of the same benchmarks: sequential Java, parallel Java, sequential X10 and manually parallelised X10 versions — which represented what could be produced by a parallelising compiler if their proposed declarations and annotations were supported. These tests showed that sequential applications enjoyed 20% speed-ups on average, while the performance of the manually parallelised version matched the performance of the parallel Java version.

2.3 The Gaia mission

Gaia [Lindegren et al., 2008] is a ESA mission whose primary objective is to produce a detailed and precise catalogue of around one billion stars in our Galaxy. The information contained in this catalogue will detail the positions, distances, motions, brightnesses and colours of the observed stars. The number of stars observed, and the precision of the measurements that Gaia will take are of a much greater scale than any previous mission of this type.

The information gained through the Gaia mission will greatly contribute to our understanding of the structure, history, and evolution of our Galaxy, and there is huge interest and anticipation amongst the astronomy community for the Gaia catalogue. In addition to stars, the Gaia data will also allow for the detection of many thousands of extra-solar planets, as well as asteroids within our solar system. Each observed object (whether it is a star, binary star system, or any other object) is generally referred to as a source.

The mission involves the launching of a space satellite which contains a number of extremely sophisticated instruments, including two astrometric telescopes, separated by 106.5 degrees. Light from these telescopes is focused onto an extremely sensitive array of Charge-Coupled Devices (CCDs), which detects the light falling on it from the sources observed. At the time of writing, the Gaia satellite is expected to be launched in September 2013, and to commence normal operations around three months later. Gaia will be placed in an orbit around the Sun at the L2 Lagrangian point — an area of space 1.5 million km beyond the orbit of the Earth

where the gravitational force of the Sun and the Earth act to hold objects in stable orbits. L2 will provide Gaia with a quasi-stable thermal environment, as well as a view of the Galaxy unobstructed by the Earth, or the Sun.

Gaia is expected to be operational for five years. Each day, Gaia will transmit the data that it has collected back to Earth. This daily download is expected to be of the order of 25 GBs (compressed at a ratio of 2.5). Therefore, over the course of its nominal five year mission, Gaia is expected to transmit more than 100 Terabytes (TBs) of data.

2.3.1 Gaia data reduction

The data which will be returned to Earth by Gaia is quite raw in its nature, and the process of constructing a star catalogue from this data is an extremely complicated task, and is known as the *Gaia data reduction pipeline*. Over the course of its five year mission, as Gaia's view rotates around the sky, it will observe each source, on average, 70 times, therefore the Gaia data will contain many relationships within the data that must be taken in account when building up the data associated with each source.

There are many processes which must be executed on the Gaia data, and these processes must be run in an iterative manner. Although the Gaia data is not overwhelming in its size, the complex relationships within the data, and the iterative nature of the required processing mean that the Gaia data reduction is by today's standards, a large data processing challenge.

The instruments of the Gaia satellite are very precise, and current technology has been pushed to its limits to produce them. One of the challenges that the Gaia mission must face is damage to these highly precise instruments due to radiation from the Sun. Predicting the level of radiation that Gaia will be exposed to is difficult, however, it is certain that the accuracy of the Gaia instruments will decrease over time, due to radiation damage. This process has been studied, and it is quite well understood. The effects of radiation damage are taken into account in the Gaia data reduction software, adding to the processing challenge.

2.3.2 Data Processing and Analysis Consortium (DPAC)

The Data Analysis and Processing Consortium (DPAC) [Mignard et al., 2007] is the organisation with the responsibility to prepare and execute the Gaia data reduction pipeline. This includes the design, testing, and implementation of the required software, providing the hardware and communication infrastructure that will be used during processing, as well as managing the actual execution of the Gaia data reduction pipeline. Finally, DPAC will also create the Gaia catalogue, and make the catalogue publicly available. In this thesis, the term *DPAC systems* is used to refer to the software systems produced by DPAC.

DPAC is a multidisciplinary organisation, with around 400 members, consisting of scientists and engineers from several fields, based at many locations across Europe. The majority of DPAC members work in academia, although it also includes members from ESA and the space industry. DPAC work is organised into nine Coordination Units (CUs), each CU having the responsibility of preparing the software applications necessary for some part of the processing, as shown in Table 2.3. CU1 acts an the overall architecture designer and coordinator for the other CUs.

DPAC contains six Data Processing Centres (DPCs) as illustrated in Figure 2.4. These DPCs

are the data processing centres where the DPAC systems will be executed. The data processing is shared amongst these DPCs, although the amount of processing that will be performed at each DPC is different.

One of the DPCs within DPAC is the Data Processing Centre of Barcelona (DPCB), which includes computing facilities at the Barcelona Supercomputing Center (BSC) [BSC, 2012], and at *Centre de Serveis Científics i Acadèmics de Catalunya* (CESCA) [CESCA, 2012]. This thesis was written in the context of the preparations of the Gaia data reduction pipeline at the DPCB. More details on the DPCB environment, and on the DPAC systems that are executed there, are given in Section 2.3.3.

The Data Processing Centre of ESAC (DPCE), located close to Madrid, Spain, will play a coordinating role during data processing. Data will initially arrive at the DPCE from the ground station. The DPCE will then act as central data hub, making the data available to the other DPCs. The data will be stored in a central database at the DPCE, known within DPAC as the Main DataBase (MDB). Other DPC will retrieve data from the MDB, perform the required processing and then write the results back to the MDB. The execution of the entire data reduction chain (known as a Data Reduction Cycle) will occur once every six months, and a new version of the MDB will be made available (within DPAC) at the end of each cycle. The Gaia satellite is expected to operate until around 2019, and it is expected that at least another year will be needed to process all the data and to prepare the data to be made available, therefore the final Gaia catalogue is expected around 2021.

It is a policy within DPAC that all DPAC systems should be written in Java. In this thesis, DPAC systems and the DPCB environment are used as case studies during the investigations of the topics treated here, and the Gaia data reduction pipeline is discussed as an example of a large Java-based scientific data processing project running in HPC environments.

CU1: System Architecture
CU2: Data Simulations
CU3: Core Processing
CU4: Object Processing
CU5: Photometric Processing
CU6: Spectroscopic Processing
CU7: Variability Processing
CU8: Astrophysical Parameters
CU9: Catalogue Access

Figure 2.3: DPAC Coordination Units (CUs).

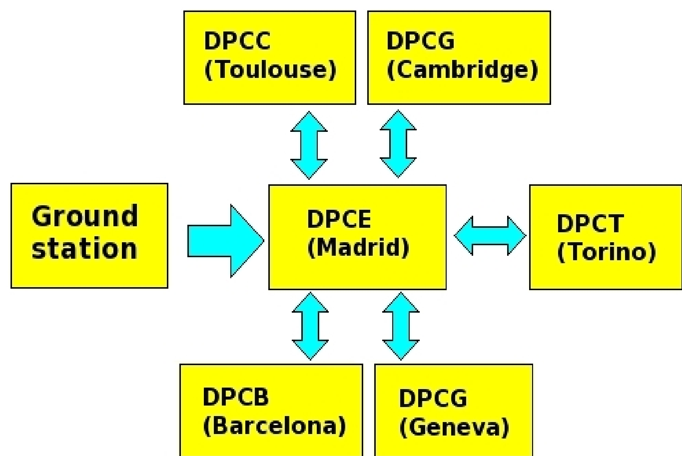

Figure 2.4: The flow of data between DPCs.

The software development model adopted by DPAC is based on six month cycles, whereby a new release of each application is released at the end of each cycle, typically including a number of enhancements and added features. This has allowed the DPAC systems to be developed in parallel with each other, and in an iterative manner.

2.3.3 DPCB and MareNostrum

The DPCB has access to computing resources at the BSC, and at CESCA. For a number of years, the most powerful computing resource available to the DPCB was the MareNostrum supercomputer located at the BSC. In this thesis, this machine is known as *MareNostrum II*, it was decommissioned in October 2012. At the time of writing (November 2012), a new machine is currently being built. This new machine is known as *MareNostrum III* (or simply MareNostrum, but we add the version numbers to differentiate these machines from each other). Many of the tests mentioned in this thesis were performed on MareNostrum II. The technical specifications of all the machines referenced in this thesis, including MareNostrum II, are given in Appendix A. In this section, a brief description is given of the key features of MareNostrum II which are most relevant to this thesis.

As the data processing at the DPCB may continue until around the year 2020, the hardware at the DPCB will go through a number of major upgrades, and may completely change before the mission is complete. The other DPCs will also go through several changes to their hardware, and at the time of writing, it is impossible to know the hardware architecture that will be available in each of the DPCs at the end of the mission. For this reason, it is very important that DPAC systems are developed to be portable, and not dependent on any particular hardware features.

Simulation applications have played an extremely important role during the development of the Gaia data reduction pipeline. Simulators have been used to generate the telemetry stream that will be returned by the Gaia satellite, as well as simulating data at various stages of the processing. The data generated by these simulators has been used to test other components of the data reduction pipeline. A number of simulators have been developed including GAia System Simulator (GASS) (see Sections 2.3.4.2) and Gaia Object Generator (GOG) — which can be used to generate Gaia catalogue as well as MDB data. The GASS simulator has been running at the DPCB for many years, with more features added during each software development cycle. The most data-intensive application of the Gaia data reduction pipeline is an application called Intermediate Data Updating (IDU) (see Section 2.3.4.4), and this application will be executed at the DPCB.

The MareNostrum II supercomputer consisted of 2560 JS21 blade computing nodes, each with two dual-core IBM 64-bit PowerPC 970MP processors running at 2.3 GHz, which totalled 10240 cores (9.2 GFlops and 8 GB memory per computing node). Each computing node contained a local disk that provided 36 GBs storage capacity, which was intended for temporarily storing data belonging to jobs running in that node. The system included a high-performance storage device from IBM called General Parallel File System (GPFS), which is designed to provide fast, and reliable, simultaneous access from many processes (running in many of the computing nodes). For long-term storage, and the *backup* of data, the system also included over 3 Petabytes (PBs) of storage capacity in a tape-based system called Hierarchical Storage Management (HSM).

MareNostrum II computing nodes were interconnected through a Myrinet 2000 network (12 switches), as well as through a Gigabit Ethernet (GbE) network which connected each computing node to the GPFS device (see Section 5.1.1 for a detailed description of these network technologies). The OS was SuSE Linux Enterprise Server 9. Two versions of the IBM J9 JVM were available: 1.6 and 1.7, both were ppc64-64. The MPI implementation was MPICH-MX 1.2.7.4, while the MX driver version was 1.2.7-64.

It is important to note that the machines at the BSC are not under exclusive DPAC control.

Instead, DPAC systems running at the BSC share the computing resources with other users. To make use of one of the machines, users must log into one of the *login nodes*, from where they can submit jobs to the job management system. Users cannot not directly access the nodes where their jobs are actually executed.

2.3.4 DPAC systems

In the following sections, a brief description is given of some of the DPAC systems.

2.3.4.1 AGIS

The Gaia satellite is a totally self-calibrating system, that is to say that it does not rely on information from any external source. This means that Gaia will construct its own reference system, measurements of observed sources can then be made against this reference system. Only later are the observations of Gaia mapped to the International Celestial Reference System (ICRS). Astrometic Global Iterative Solution (AGIS) is the complex system which performs the astrometric data reduction, using a least-squares fitting scheme. AGIS has been in development since 2006, and it has been continually enhanced in an iterative manner, while being scaled to larger executions involving more sources. The theory of AGIS is described in detail in [Lindegren et al., 2011], while the implementation of this application in Java is described in [O'Mullane et al., 2011a].

2.3.4.2 GASS

GASS is a simulator application, that can generate the telemetry stream that will be returned by the Gaia satellite to Earth. This includes *house keeping* data, as well as the scientific data. GASS is a very mature application, as it has already been running successfully, on a number of different machines at the DPCB since 2006. Data generated by GASS has been a crucial element in the preparation of the Gaia data reduction pipeline, as GASS data may be feed into the pipeline, simulating data from the satellite.

During some large simulations of the telemetry stream, GASS has to process over 1 billion sources. These simulations require the use of many worker processes (thus far, simulations involving over 3000 cores working simultaneously have been executed involving over 5 TBs of data). These worker processes need to repeatedly access a set of shared data-files during the execution of GASS (such as spectra and instrument calibration files). If all of the executing worker processes simply access these files directly on the GPFS, then its performance decreases to unacceptable I/O response times.

Internally, the GASS application must generate large numbers of relatively short-lived objects while performing simulations. For this reason, memory must be managed carefully to avoid problems such as Out Of Memory (OOM) errors.

2.3.4.3 IDT

Intermediate Data Treatment (IDT) is the main data processing application that will be executed on the newly received scientific telemetry from the Gaia satellite each day. During

normal operations of the mission, IDT will be executed each day at the DPCE, and will process 24 hours of telemetry data. Initially, raw telemetry packages received from the satellite are written to a database by another system which then notifies IDT of the availability of these packages. IDT threads then retrieve these packages, perform a number operations on the data, and write their output back to the database. The operations performed by IDT include attitude determination, source identification, and image parameter extraction.

2.3.4.4 IDU

IDU is one of a number of applications that will be executed on the Gaia data at regular intervals (once every 6 - 12 months) over the duration of the mission. IDU and AGIS together form an iterative chain of processes that must be executed regularly. Each time that IDU is executed, it will process all of the data amassed at that point, so as the mission continues, the volume of data that will need to be processed will increase, reaching roughly 100TB at the end of the mission. Although the actual volume of data may not be overwhelming, the challenging aspect of the processing is the relationships within the data, and the reorganising and movement of the data, which must be carried out between tasks.

IDU itself is composed of several independent tasks, which run in a particular sequence and which are effectively independent sequential applications. However, some of these tasks have dependencies on the output from other tasks, and the input for one task may come from the output of another task, which might have been executed in a different computing node. Therefore, data transfers are required between the execution of each task in order to deliver the correct input data to the correct process in the correct computing node. See Section 4.1.1.1 for a more detailed description of IDU.

2.3.4.5 DpcbTools

DpcbTools [Fries et al., 2011a] is the name given to a toolbox of software tools that have been developed to assist the execution of DPAC systems at the DPCB. These tools have been designed to be as generic as possible, and could be used by other applications in other environments. However, certain parts of DpcbTools are specific to the applications that will run at the DPCB.

DocbTools contains a framework designed to facilitate the execution of data processing applications which are composed of a number of sequential applications which must be executed in a particular order, but containing data dependencies, as is case in the IDU application. Given a large input dataset, this framework allows for the division of the processing work into smaller more manageable tasks. These tasks can then be distributed amongst the available computing resources. It involves the organisation of the available computing nodes following a hierarchical pattern, and the controlling of each nodes and the processes running in these nodes.

DpcbTools includes monitoring and management systems. It includes a Python-based monitoring system for monitoring system information such as CPU, memory, and network usage. It also contains a JMX-based monitoring and management system, specifically designed for operating in a cluster environment (see Chapter 9). High-performance file access functionality is provided in DpcbTools, including support for Hierarchical Data Format (HDF5).

Also included in DpcbTools is MPJ-Cache — a data communication and distribution system designed for execution in HPC environments (see Section 5.5). It builds on the functionality provided by an underlying implementation of MPJ, and provides application developers a high-level API of data communication functions, as well as providing data prefetching and caching features. It is also instrumented to allow its monitoring and management through the JMX framework.

3

Java for Scientific Applications in HPC

Computing science is no more about computers than astronomy is about telescopes

– Edsger W. Dijkstra

For scientific applications, and in particular those running in HPC environments, the raw processing performance offered by a language (more specifically, the performance offered by the compilation and execution options available for that language) are of primary importance. Such applications often involve the execution of many floating-point operations, therefore the ability of a platform to efficiently perform such operations is one of the criteria upon which developers select a development language. Additionally, such applications often require the availability of efficient libraries offering a wide range of numerical processing functionality.

Java has been an extremely popular programming language since its release in 1996. The reasons for its success are numerous and include its object-oriented design, its multithreading and its portability. However, despite its well-documented advantages, and its popularity in general computing, Java has had limited success in terms of its adoption by the scientific and HPC communities. One of the factors militating against a wider use of Java in HPC is the continuing belief that Java applications cannot obtain the same performance from HPC resources as the traditionally used HPC languages, such as Fortran and C. Additionally, there may be a belief that there are a limited number of libraries available which support the execution of scientific applications, particularly numerical libraries.

In this chapter, the selection of Java as the development language for scientific applications in HPC environments is discussed. Firstly, the characteristics of Java are reviewed, with the requirements of scientific applications in mind. In particular, the handling of floating-point numbers by Java is described and discussed. Later, the availability and performance of numerical libraries for Java is described. Finally, in the last section of this chapter, the use of Java for the development of the Gaia data reduction software is discussed as a case study.

3.1 Background

Scientific-based applications, such as those used in large-scale modelling applications, often push computing hardware close to its full processing capability. It is therefore important that the language used for the development of such applications can efficiently make use of the available hardware in order to maximise the outcome of the executions.

Fortran, C and C++ are amongst the programming languages which were traditionally used for the development of scientific applications, and these continue to be widely used in scientific computing. The reasons for their popularity include the very good performance that they offer; high-quality compilers; the availability of libraries useful for scientific and HPC computing, such as numerical and message-passing libraries; as well as the fact that these languages have an established reputation within the scientific communities, therefore they tend to be selected by default.

Java was initially largely used for the development of relatively simple, Internet-based applications. Gradually, over the years, as the performance of Java improved, and as more libraries and tools became available that could support the development and execution of Java applications, it began to be used for the development of more diverse applications. Java became widely used for the development of server-type applications, where performance and reliability are of utmost importance. Despite the continuing popularity of Java in general computing, and the many independent projects which have developed extensions and libraries for Java applications, the use of Java by the scientific and HPC communities remains relatively low.

It is almost certainly the case that part of the reason for the slow adoption of Java in HPC is simply due to the inertia of moving from the languages which have traditionally been used in the HPC and scientific computing communities. It is also due to the initial reputation that Java acquired as providing poor performance due to it being an interpreted language. Finally, part of the problem may also have been the lack of reliable, and actively supported, high-performance Java libraries; in areas such as data communication (as discussed in Chapter 5, this is no longer the case).

3.2 Relevant characteristics of Java

The advantageous characteristics of Java for general computing have been described on many occasions. The significance of these characteristics for scientific applications running in HPC environments is much less often discussed. In the following sections, some of these characteristics are discussed.

3.2.1 Easy to learn language syntax

The Java language was specifically designed to have a clean and easy to learn syntax. Its syntax was largely based on C++, which in turn is an extension of C, therefore those familiar with C or C++ can become comfortable with the Java syntax quite quickly. It is possible to become a proficient Java developer in a relatively short period of time, this permits those unfamiliar with Java to quickly begin working in it. One of the reasons for this, is that Java does not include explicit pointers, or any other low-level memory manipulation functions. The absence of explicit pointers also allows Java developers to avoid one of the most bug-prone areas of computer programming (see Section 3.2.6 for more on the benefits of not having explicit pointers).

Java was designed to allow developers to operate as productively as possible, and it continues to be enhanced with this objective in mind. For example, Java SE 7 included a number of small language changes, collectively known as *Project Coin*, and described in *JSR 000334*, which ease some commonly performed programming tasks. These enhancements include simpler handling

of binary literals, and allow simplified creation of generic types. Such enhancements of commonly performed programming tasks improve developer productivity and reduce development time. Using a language with a simple syntax leads to a much higher level of maintainability, discussed in the next section.

3.2.2 Development and maintenance costs

Large software applications are often in the development, testing, and production stages for long periods of time, often many years. Over the course of the lifetime of a piece of software, it might be examined and modified by many developers. The importance of designing and writing code that can be easily understood and modified by others is extremely important for large software projects. Java lends itself very well to the development of easily understandable and maintainable code.

Speaking from experience, some long-lived pieces of software, written in older programming languages, reach the point that developers are reluctant to modify complex sections of the code as they are unsure of how the code actually works, and are afraid of breaking it. The root cause of such situations is that the software was not designed and written with maintainability in mind.

In [Butters, 2007], the results of a survey of the costs of developing and maintaining applications in Java against C and C++ are presented. The survey involved querying the opinions and experience of over 350 developers from the Java, C and C++ communities, taking many factors into consideration, including development time, maintainability, availability of developers with the required skills, and licensing costs. The survey revealed that the development time for C and C++ projects is on average 50% longer than would be required if Java was used. In addition, the survey highlighted the more extensive set of tools that are available for the development and maintenance of Java software (see Section 3.2.9 for more on the tools available to assist Java development).

The existence of GC in Java (see Chapter 7) frees developers from having to concern themselves with the allocation and release of memory — often a source of bugs in some languages. Developers can instead concentrate on the design and logic of their applications, leading to safer applications being developed in a shorter period of time.

3.2.3 Multithreaded

The trend towards an ever increasing number of cores per processor, and processors per computing node, in typical HPC environments, has increased greatly the importance of multithreading. Java, with its built-in and convenient-to-use multithreading capabilities, is perfectly equipped for developing multithreaded applications that can take advantage of multi-core environments.

As of J2SE 5.0, the Java Class Library has included a package called the Java Concurrency Framework [Lea, 2005]. This package includes a series of utilities to assist in the development of multithreaded applications. It includes a task scheduling framework, concurrent collections, atomic variables, locks and synchronisers, and more precise timing methods.

Prior to the inclusion of the Java Concurrency Framework, the synchronisation primitives available in Java were of a low level of granularity, requiring application developers to build on top of these low-level synchronisation primitives in order to implement larger, more complex

synchronisation structures. As many applications require the same types of synchronisation functionality, application developers often ended up duplicating this work in different projects. The Java Concurrency Framework provides a range of reusable, and scalable concurrency functionality, at a higher-level of abstraction, leading to better developer productivity, and reduced maintenance effort.

The concurrency functionality provided in the Java SE was further enhanced in Java SE 7, with the addition of the Fork/Join framework [Lea, 2000]. This framework allows for processing problems to be recursively decomposed into smaller *subtasks* that are executed in parallel threads, and once all of the *subtasks* have been completed, the results are combined and the threads join back into the main thread.

3.2.4 Performance

For many years, the relatively-low raw processing power of early JVMs was cited as a reason to avoid the use of Java for the development of scientific applications, and for those intended for execution in HPC environments. The processing power of Java has been increasing steadily over the last fifteen years, and also, significantly for performance, the sophistication of the optimisations that are available in Java has increased beyond those that are available for statically compiled languages.

Statically compiled applications consist of native machine code instructions that can be executed natively on a particular processor architecture. The execution of Java bytecode inside a JVM is much more complicated, is very configurable, and there are wider range of optimisations available to JVMs that are not available when executing statically compiled code.

Managed runtime environments, such as the JVM, can apply a range of optimisations to the code that they are compiling. Firstly, they can apply the same optimisations which are available to statically compiled and optimised code, such as standard inlining; but they can also apply some optimisations that are only available when executing code within a managed runtime environment. They can make assumptions about the execution of code, based on *runtime feedback*, which will probably remain true for the entire execution of the application. Following this approach, JVMs can produce more highly-optimised code. If however, some of these assumptions later are invalidated, the JVM can revert back to a less optimised version of the code. Because of the fact that their executions are performed using very different approaches, it is difficult to compare the execution of statically compiled applications with that of Java applications. However, when recent direct comparisons have been made, the performance obtained by Java applications, through the use of compilation and aggressive optimisations, is of the same level as performance offered by optimised statically compiled applications. In fact, in some cases, the performance offered by Java can exceed that offered by statically compiled code. In the following paragraphs, we gave a summary of two of the many published studies which have compared the performance of Java with that of other languages.

In [Chen, 2010] C, C++, C# and Java are compared using five benchmarks, which test their performance in the areas of integer arithmetic, double arithmetic, long arithmetic, trigonometric functions, and I/O functions. The C compiler was *gcc* version 4.4.1, the C++ compiler was *g++* version 4.4.1, the .NET framework was 4.0.30319.1, the Java version was 1.6.0_20-b02. These tests measured the execution time of each benchmark, therefore, shorter times indicate better performance. All tests were executed ten times, and average execution times were obtained. A summary of their results are given in Table 3.1. The shortest execution times are highlighted

in bold. As shown, Java performed best in the first 3 benchmarks. In the I/O benchmark Java was the slowest, although all four languages delivered similar performance, and Java was less than 10% slower than the fastest I/O performer, which was C++. Java performed quite poorly in the trigonometric benchmark, indicating that the built-in trigonometric functions in the Java class library provide quite poor performance, relative to the other languages tested. Most Java-based scientific applications use the Commons Math library from Apache (see Section 3.3.3), which provides much better performance for such functions.

	Integer arithmetic	**Double arithmetic**	**Long arithmetic**	**Trigonometric functions**	**I/O functions**
C	14273.5 ms	18718.6 ms	33110.9 ms	13626.8 ms	6052.1 ms
C++	14275.3 ms	18659.0 ms	31781.9 ms	13492.9 ms	5563.0 ms
C#	12601.3 ms	17920.8 ms	37974.6 ms	**5308.4 ms**	**4260.0 ms**
Java	**8916.9 ms**	**10322.7 ms**	**27716.4 ms**	67401.5 ms	6098.1 ms

Table 3.1: Execution times of five benchmarks using C, C++, C# and Java

In [Sestoft, 2010], the numerical performance of C, C# and Java are compared. They investigated four particular types of calculations: matrix multiplication, a division-intensive loop, polynomial evaluation, and a distribution function. The C compiler was *gcc* 4.2.1, the .NET runtime environment was 4.0, and the Java version was Hotspot 64-bit Server VM 1.6.0-17. They found that C generally performed slightly better than Java, and generally C# came third. However, the three languages are variously fastest and slowest across the different types of calculations. In particular, in regards to the performance of Java, they express their surprise at how well the Java platform performs, given its less efficient array representation.

3.2.5 Monitoring capabilities

The monitoring of executions is particularly important in HPC, as they involve the use of valuable resources with high running costs. If a problem develops, then it is important that users have some way of detecting this. The Java platform includes a number of lightweight built-in features designed to allow for its monitoring, which add little overhead to the JVM. The Java Platform Debugger Architecture (JPDA) defines a collection of APIs which allow for the inspection of the JVM, it includes the Java Debugger Interface (JDI) and the JVMTI. The JDI exposes debugging information and is intended to allow for the development of remote Java debuggers. JVMTI is a framework that allows for the inspection and controlling of applications running in a JVM, at a low level. The information provided allows for the development of low level debuggers and profilers.

JMX is a built-in framework that allows for the monitoring and management of JVMs and the applications running in them. Using the JMX framework, the information which JVMs expose through the JVMTI may be retrieved remotely. It defines a number of MBeans that allow for the monitoring of a range of JVM properties, and it is also highly extendible. Application developers may instrument their own applications to allow for their monitoring and management using JMX. (See Chapter 9 for more on monitoring, including our JMX-based monitoring system).

3.2.6 Problem detection

Traditionally used HPC languages, such as Fortran and C, provide limited runtime error checking and debugging information. This can lead to applications terminating without giving many clues as to the cause of the problem. Java however, offers a high degree of runtime error checking, and if an application does terminate unexpectedly, at the very least it provides a stack trace that can be used to locate the cause of the problem. As Java runs in a managed runtime environment, the JVM can verify the safety of code as it is loaded. The bytecode verifier performs a range of checks and analysis on bytecode before it is executed.

Exception handling is the name given to a system that handles exceptional situations during the execution of an application. When such a situation occurs, an exception is said to be *thrown*, and should be *caught* by some exception handling code. Many languages support exception handling, including C++, C# and Java. The benefits of built-in exception handling include the separation of error handling code from the rest of the application code, leading to cleaner, more understandable code.

In Java, there are three types of exceptions: checked exceptions, errors, and runtime exceptions. Checked exceptions include situations such as being unable to find a required file. Errors are triggered by some external events, such as a resource being unresponsive. Runtime exceptions are caused by some internal application problem, such as attempting to make use of uninitialised object instance, or a failed cast. At compile-time, the compiler checks that all code which could potentially lead to a checked exception is either enclosed inside a *try-catch* block, or that the code declares that an exception could be thrown (in which case the exception must be handled by the calling code), otherwise the code will not compile. For this reason, they are called *checked* exceptions. This compile-time checking in Java leads to many processing errors being caught at compile-time. Amongst the errors that this checking may catch are array bounds checking.

Explicit pointers, which are heavily used in some languages, including C, lead to a high danger that same data held in memory will be unintentionally modified, which can lead to very difficult to find bugs. The strong error checking in Java leads to more robust software as more problems are detected at an earlier point. This greatly reduces the number of runtime errors, and therefore the amount of time spent investigating runtime errors.

3.2.7 Efficient I/O

The Java platform offers two built-in I/O packages. The original package, called Java I/O, is mainly stream-oriented, and offers blocking I/O functions. As of JDK 1.4, Java developers have also had the Java New Input/Output (NIO) package available to them. This package, which is buffer-orientated, added support for high-performance non-blocking I/O functions. The addition of non-blocking I/O functions allows a single thread to perform multiple operations concurrently. In other words, non-blocking functions allow a single thread to initiate a non-blocking I/O function, and while it waits for that function to return, it may initiate another I/O function. This allows applications to avoid the deadtime spent in a blocking state if only blocking functions are available. Java NIO allows for the development of more scalable software, which is a major concern for HPC applications. In the NIO package, file and socket I/O is passed through a buffer class, called *ByteBuffer*, which implements a byte array, and it allows the data to be located outside of the JVM heap. NIO was later extended again in Java SE 7, with the addition of a new file management API called NIO2.

3.2.8 Portability

A fundamental strength of the Java platform is its portability. Java applications are compiled to a format —bytecode — that is designed to be executed on a JVM. Any machine with a JVM installed can execute exactly the same bytecode (assuming that the correct version of the JVM is available). This feature of the Java platform is commonly referred to as Write Once Run Anywhere (WORA), as it means that Java source code only needs to be compiled once and then can be executed anywhere. This has huge advantages when distributing software amongst a group of users who may be using different operating systems or different hardware environments. Java applications are not dependent on any system libraries, instead they make use of the Java Class Library for interacting with the underlying hardware environment. This hides any particular system features away from developers, and prevents the creation of Java applications which are, in any way, system dependent (the use of JNI can obviously break this system independence).

Given the speed of developments in computing hardware, there is a high probability that an application with a relatively long lifetime will be executed on successive generations of machines. Even if the application is only used by a single organisation, it is quite likely that that organisation will upgrade its hardware during the lifetime of the application. This will result in the need to migrate the applications from one hardware environment to another. In respect of some programming languages, one problem that may arise during such migrations is the realisation that an application is dependent on some library which is available for the old machine architecture, but not available for the new machine architecture. This can necessitate changes to the applications code, in order to utilise an alternative library that is available for the new machine. The portability features of Java eliminate the possibility of such problems arising during hardware migrations.

Finally, some languages specifications, including those of C and C++, do not define precisely how many bytes of space should be allocated to particular data types — the amount of space allocated depends on the compiler and the hardware environments. Another consideration is that some languages do not define the order in which the operations making up an expression are evaluated. Both these issues can lead to different results being produced on different machines. In Java, the language specification explicitly defines the size of each data type, which is completely independent of the hardware platform.

3.2.9 Tools available to assist development and testing

Java is one of the most widely used programming languages in the world, and there have been a huge number of tools developed to assist the design, implementation, debugging and testing of Java applications. In this section, we briefly mention some of the most commonly used of these tools, and describe how they collectively provide a convenient system for the management of large software projects.

The Java platform makes use of the JAR format to group together, and distribute, the classes that comprise Java applications, as well as any associated resource files. JAR files facilitate the clean and fast distribution of Java applications in a distributed environment such as in a computer cluster — all within a single file.

Eclipse is a FOSS IDE, mostly developed in Java, which is very widely used by the Java development community, although it also supports several other languages. Eclipse provides a

wide range of features in areas such as debugging, refactoring and tuning. Additionally, Eclipse is highly extendible, and it includes a framework that supports the execution of plugins, which add extra functionality. There are a wide range of plug-ins available, providing a very rich set of functionality in areas such as application design, profiling, testing and version control. Its refactoring features are extremely useful and provide a fast mechanisms for making changes to an existing piece of code, such as making code more modular. For example, it allows for the automatic extraction of a method from a section of code, including creating the interface for the new method, determining what arguments should be passed, and what should be returned.

JUnit is a FOSS Java implementation of the xUnit testing framework. It allows for the fast development of repeatable tests that can be used to verify the behaviour of a part of an application. Typically, JUnit tests are created to test the behaviour of each of the methods in a Java class. The JUnit framework is widely used within the Java development community, and has become the *de facto* standard testing mechanism for Java applications. As such, it allows Java testers to easily move between projects and between organisations. Support for developing and executing JUnits tests is included in several IDEs, most notably in Eclipse. Complementing the JUnit framework, there are a number of tools for determining the percentage of code in a Java project that is executed in at least one JUnit test — thus providing some assurance that code has been tested, and operates correctly. Cobertura is one such tool, which additionally can perform analysis of the code complexity of a project, and can generate a range of useful reports.

Large software projects are typically managed by a version control system such as Concurrent Versions System (CVS) or SubVersioN (SVN). Often many developers, possibly working at different locations, may be making changes to the code on a daily basis. In such cases, it is important that regular checks are carried out to ensure that the entire system is compiling, and that *breaking changes* have not been introduced. There are a number of continuous integration tools available for the Java platform that allow Java projects to be built on a regular basis to verify that the target software builds correctly. Jenkins, Hudson, and CruiseControl are all free software, open source, written in Java, and are widely used by the Java development community.

The use of JUnit tests together with a tool such as Cobertura greatly increases the robustness and reliability of applications. The regular execution of tests, over the entire lifetime of an application greatly reduces the chances that bugs will be introduced over time, thus helping to preserve the correctness and reliability of the system.

There exists a number of tools that facilitate the automatic building (compilation) of Java projects, such as Maven and Ant — both developed by Apache. Such tools are useful when working with large software projects that involve many resources, and when the build process involves several steps. These steps might involve downloading code from a version control server, downloading library files, compiling the code, running JUnit tests, and finally packaging the project into a single, easily deployable resource. The availability of tools that allow for the easy automation of these steps greatly reduces the effort required to develop, maintain and extend software projects.

3.3 Java numerical processing

Numerical processing is typically very important for scientific applications, especially for those executing in HPC environments. In this section the numerical processing capabilities of Java

are discussed.

3.3.1 Java Grande Forum (JGF)

The Java Grande Forum (JGF) [Philippsen et al., 2001] was an initiative to encourage the use of Java for creating software to tackle *grande* computing challenges. The JGF recognised the potential of even the very early versions of Java for this purpose. They stated their belief that "Java has the potential to be a better environment for developing grande applications than any previous programming language" [Thiruvathukal, 1998].

Although strong advocates of Java, the JGF highlighted the weaknesses of the early versions of Java, and it proposed changes and additions to the Java platform that would address these weaknesses. For example, the proposal from the JGF to introduce the *stritfp* keyword was adopted in J2SE 1.2 (see Section 3.3.2 below).

The JGF comprised a number of working groups, including the *Java Numerics Group* and the *Concurrency and Applications Group*. The JGF was active from 1998 until around 2003 when the group stopped publishing papers or updating its website. It is difficult to quantify exactly why the JGF stopped working. However it seems that the JGF simply reflects the changes in the level of interest in Java by the scientific community. In the early days of Java there was much interest in it from the scientific community, but after the execution of many benchmarks using early versions of the JVM, and comparing the performance of Java with the performance offered by the traditionally used languages, it was clear that Java was still behind these languages in terms of performance. For that reason, interest in Java from the scientific community began to fall away.

Despite the cessation of activity by the JGF, the performance and functionality offered by Java has steadily increased in the intervening years, and in the last few years, the scientific and HPC communities have began to reassess Java as a development language. This is especially true in the case of astronomy. This is demonstrated by the fact that, in addition to the Gaia project, several other ESA astronomy missions, including Herschel and Planck, also selected Java for the development of some of their data reduction software. The Centre National d'Etudes Spatiales (CNES) has also selected Java for the development of a number of large scientific software projects, including STELA (started in 2008) and SIRIUS (started in 2010). See Section 3.4 for more information on the experience of using Java by the Gaia project.

3.3.2 Fundamental aspects of Java affecting numerical computing

Java contains eight primitive data types: four are integer types, two are floating-point types, one is boolean, and the final data type is char. In addition, the Java Class Library contains the classes BigInteger and BigDecimal that allow for the use of arbitrarily large numbers. Java includes the constants *POSITIVE_INFINITY*, *DOUBLE.POSITIVE_INFINITY*, and *NaN (Not a Number)*.

The Institute of Electrical and Electronics Engineers (IEEE) 754 standard defines how 32-bit and 64-bit floating-point numbers should be represented, how floating-point operations should be performed, rounding rules, as well as defining exceptions. The handling of floating-point operations in Java is largely based on the IEEE 754 standard. Java does not however support all of the exceptions defined in this standard. For example, Java does not provide a built-in method for trapping calculations that produce a *NaN* result. Effectively, Java assumes that all

numerical operations lead to a valid result. This can lead to invalid values propagating through a series of calculations, and can result in bugs that are difficult to locate [O'Mullane et al., 2011b].

One of the goals of the initial Java language specification was to ensure strict reproducibility across all environments. The traditionally used HPC languages are typically compiled in such a way that particular hardware capabilities, such as support for 3-operand instructions can be utilised. Making use of such features, that are not universally supported by all environments, can result in rounding differences between the results produced on different environments.

During the performance of floating-point operations, intermediate results are calculated. Until Java 1.2, these intermediate results were stored in Java by either single or double precision representations. An alternative approach is to represent these intermediate results more precisely — using larger data types. Using more precise representations for these intermediate values, such as 80-bit representations, results in more precise final results. Since Java 1.2, intermediate results may be stored using more precise representations, if they supported by the hardware environment. Use of the *strictfp* keyword however, ensures that the associated code strictly uses single or double precision for all intermediate results through truncation. This guarantees reproducibility across all platforms.

The JGF [Smith and Bull, 2000], and others [Kahan, 1998] highlighted several weaknesses with the numerical processing characteristics of the early versions of Java. Java does not support overloading, nor does it include support for complex numbers. Complex numbers may of course be implemented using classes, and there are a number of high-quality libraries which provide complex number support, such as the Apache Commons library (see Section 3.3.3). However, the lack of built-in support does mean that an extra level of abstraction must be put in place, leading to an additional overhead.

The Java Class Library includes the Math and StrictMath classes which provide basic numerical functions, such as square root, and the natural logarithm functions, as well as defining some commonly required constants. These classes provide the same set of methods as each other. However, StrictMath ensures consistency across platforms, while the Math class may make use of hardware specific routines in order to provide the best performance. In many cases, the methods in the Math class simply call the corresponding method in the StrictMath class.

Scientific applications are heavily dependent on arrays [Moreira et al., 2000]. Multi-dimensional arrays in Java are implemented as arrays of arrays. Java does not provide any guarantee that the elements in a row of a multi-dimensional array will be stored contiguously in memory. This reduces the ability of caching mechanisms to improve performance. The issue of how arrays are implemented in Java is one of issues that has affected the reputation of Java amongst the scientific community. A number of alternative implementations have been proposed and implemented, including in the X10 language, where see Section 2.2.6.

3.3.3 Numerical libraries

Libraries offering a wide range of efficient numerical functions are extremely important for the development of scientific applications. They serve as extensions to the core functionality available in a language, offering sets of numerical processing functions that are typically designed to deliver high-performance.

Within the scientific and HPC communities, the Basic Linear Algebra Subprograms (BLAS) API serves as the standard API for basic linear algebra functions, such as vector and matrix multiplication. There are many implementations of the BLAS API, the majority of these

are implemented in Fortran or C. Some of these implementations exploit machine-dependent, high-performance operations supported by some architectures, resulting in some BLAS implementations being able to reach the theoretical peak performance of some machines. Linear Algebra PACKage (LAPACK), written in Fortran, is a numerical library which builds on top of the BLAS API, and provides functions for solving least squares problems, matrix factorisations and several other commonly required functions. An early implementation of LAPACK in Java, known as JLAPACK, is described in [Doolin et al., 1999]. This library was created by translating Fortran source code into Java bytecode, using a compiler called as F2J [Seymour and Dongarra, 2003].

There have been several projects which have developed numerical libraries for Java. A consideration for these libraries has been maintaining the portability features of Java against taking advantage of the machine-dependent processing capabilities available in some environments. An alternative to the implementation of purely Java-based libraries is the use of native libraries through the use of JNI.

The Commons Math library is a mathematical and statistical library, developed by Apache. It has been, for a number years, the most widely used mathematical library for Java, and can be regarded as the *de facto* standard maths library for Java. It contains support for a wide range of mathematical functions, including in the areas of statistics, linear algebra, interpolation, complex numbers, geometry, and differential equation solvers. It continues to be actively developed and enhanced by the Apache community, and new functionality is added with each release.

Flanagan [Flanagan, 2012] is a scientific library for Java, developed by Michael Thomas Flanagan, offering a very wide range of functionality, useful in many scientific and engineering fields. It is a live project, that continues to be updated. *ojAlgo* [ojAlgo, 2012] is another numerical library for Java, mainly concerned with linear algebra. It includes implementations of the most commonly used matrix decompositions, including bidiagonal, cholesky and eigenvalue.

A set of Java-based high-performance numerical libraries, for the development of scientific applications, are presented in [Heimsund, 2005]. They compare the performance of their libraries with some of the most widely used numerical libraries, including BLAS and LAPACK. In some cases, their libraries actually call these native libraries through JNI, and in other cases, a pure Java implementation is used. Their tests show that their libraries generally match the performance of native libraries, implemented in Fortran and C++, while still enjoying the benefits of the use of the Java language. They go on to describe how parts of their libraries have been extended, through the addition of a message-passing library, that they call *Java Message Passing Package* (MPP), in order to exploit the parallel processing capability of HPC environments, such as computer clusters.

3.4 Java for Gaia data processing - a case study

In [O'Mullane et al., 2011b] a discussion is given on the experience of using Java for the development of the DPAC systems — which constitute the data reduction software pipeline for the Gaia mission (see Section 2.3 for more on the DPAC systems). Here, we describe that study, and we elaborate a little further on some of their comments. The authors of that study considered several factors in their assessment, including the performance offered by the Java platform, portability, maintainability, and costs — including the cost of developers, the cost of purchasing hardware, and energy costs. They mention the fact that collectively they possess

decades of programming experience with C and C++, and as such, feel that they can make some informed comparisons between developing software in Java and in other languages.

The DPAC development approach, based on new software releases being made at the end of each six month development cycle, has worked very well with Java. The clear and OO nature of Java code generally leads to the development of modular software that can be easily extended and refactored repeatedly. They allows new features to be easily added to applications in each cycle.

The authors describe the history of the evolution of the AGIS application from 2005 until 2011 as a specific example. During that period of time, the features and complexity of the application increased, as did the scale of the largest executions of the application (from 1.6×10^7 to 4×10^9 sources). They describe how bottlenecks in the processing were continuously identified and removed through the regular use of profiling — for which many tools are available in Java. They believe that certain parts of the application could be made to run faster if rewritten in another language — this is not surprising, given that for any collection of benchmarks, typically some will run faster in one language while others may run faster in another language. However, when it comes to the developerpower[1] required to develop and maintain an application, they feel that Java is the best option.

The development of DPAC systems, like most software projects, is constrained by a limited amount of available resources, most importantly, a limited amount of developerpower for the development and maintenance of the applications. It is expected that roughly 2000 *developer years* will be required to develop and maintain the DPAC systems over the course of the mission. It is expected that roughly half of these resources will be consumed pre-launch — for the development of the software, and the other half will be consumed during normal operations — for the execution and maintenance of the applications. The authors believe that the developerpower required to maintain and update a large software project in Java is much less than would be the case for a C-based project. This agrees with other studies which have suggested that the cost of ownership of a Java application is less than the cost of ownership of applications written in the traditionally used HPC languages [Butters, 2007].

The authors of the study point out that, in most large ICT projects, the total cost of human resources normally ends up costing more than the hardware used. As Java runs well on the most standard hardware architectures, the project will not need to purchase any specialised hardware. The project will therefore be free to choose whatever hardware is offering the best value at the time of purchase — allowing hardware costs to be kept relatively low. Furthermore, the portability of Java means that even if the software migrates over several hardware architectures over the course of its lifetime, this will not necessitate modifications to the software. For other languages, such as C or Fortran, the issue of migrating software to a different architecture is normally a considerable concern.

The upgrade of the MareNostrum supercomputer (underway at the time of writing) at the BSC, is an example of the smooth hardware upgrade that is possible for the Java platform. The *old* MareNostrum (MareNostrum II) possessed PPC64 processors, while the *new* MareNostrum (MareNostrum III) possess Xeon processors, however the same version of DPAC systems can run on both these machines without recompilation, or any problems at all.

The authors suggest that it would probably be possible to improve the performance of an application such as AGIS, by customising the software to be able to take advantage of some hardware optimisations available on certain hardware. However, one must keep in mind that to do this,

[1]We use the term *developerpower* as a synonym for *manpower*

considerable developerpower must be spent, as well as the cost of purchasing the specialised hardware. Additionally, following this approach would discard the portability advantages of Java. Making an application run faster may reduce energy costs, however this reduction is unlikely to equal to the cost of making this reduction possible, or the cost of maintaining specialised software.

3.5 Conclusions

Java has emerged as an attractive option for the development of scientific applications for HPC. The performance offered by Java has been improved continuously since its first release, and it is now on a par with that offered by the traditionally used HPC languages. Tests have shown that in some case, Java outperforms these languages.

It would however be unwise to focus entirely on raw performance when selecting a development language, even for scientific applications in HPC. We have described a number of considerations that should be taken into account when selecting a development language, including development time; maintainability; portability across hardware platforms; the availability of tools and libraries; and costs in terms of human resources and hardware. If this range of characteristics are taken into account, Java must be view as a very appealing language for the development of scientific and HPC applications, especially for large projects such as the Gaia DPAC software.

4

Execution Framework

> *People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones*
>
> – Donald Knuth

Generally speaking, the execution of applications in an HPC environment is a more complex procedure than that which may be followed when using a local workstation. HPC environments may follow various architectures, and may offer a range of computing resources to users. Instead of simply executing their applications manually, it is often the case that users must use a resources management system to submit their jobs for execution.

Execution time in an HPC environment is a valuable resource, often, the jobs of several users will be competing to gain control of some resources, such as network infrastructure and storage devices. If there is a high demand for these resources, and therefore they are normally in use, then the performance achieved by one users application may be affected by the activity of other users applications. This influence should be minimised by ensuring that shared resources are used optimally.

The applications that constitute the data reduction pipeline for the Gaia space satellite are a real example of a large Java-based data processing system that will run in an HPC environment. In this chapter, we focus on the execution framework that is being developed to facilitate the execution of DPAC systems at the DPCB — in particular the IDU application. Related to the issue of how applications are executed, are the issues of data management, and application monitoring and management, we briefly mention them here, but are discussed in detail in Chapters 5 and 9.

In order to facilitate the efficient processing of DPAC systems, and to ensure that the available HPC resources are utilised to their full potential, an execution framework is required. This framework should offer the capability of launching, monitoring and managing the data processing applications that must be executed, while efficiently exploiting the available computing resources. A critical feature of this system is that it should allow the smooth flow of data through the processing pipeline, avoiding bottlenecks and deadtime. In this chapter, we describe our investigations into such frameworks, and we discuss why we developed our own framework.

In Section 4.1, we give some additional background information on some of the DPAC systems that will run at the DPCB. The requirements that the framework within DpcbTools should

fulfil are described in Section 4.2, including the launching and monitoring of applications. An investigation into related work is described in Section 4.3. The design of DpcbTools, including a description of its individual components is given in Section 4.4. Finally, in Section 4.5, we explain the current state of the development of DpcbTools, we mention further work that we plan to carry out, and we give our conclusions.

4.1 Background

The scheduling of jobs in most HPC environments is handled by one or more work management systems. In most cases, a *resource manager* is used to manage the available resources, these systems are normally queue-based. Typically, users submit their jobs to a queue, and depending on a number of factors, including: the availability of resources, the number of requested resources, and the user's privileges within the system; the resource manager will decide if, when, and where to execute the job. Commonly used resource managers include Portable Batch System (PBS) [Henderson, 1995], Load Sharing Facility (LSF) [Dell, 2002] and Simple Linux Utility for Resource Management (SLURM) [Jette et al., 2002].

Often, an additional system, known as a *meta-scheduler*[1], is used in conjunction with a *resource manager*, to help efficiently manage HPC resources. Although users submit their jobs to the resource manager, the meta-schedulers can instruct the resource manager when and where to execute the jobs in order to make best use of the resources, and to follow a predefined set of rules governing the use of the resources. It could be said that a meta-scheduler adds a higher degree of intelligence to the system. Moab [Moab, 2012] is a commonly used meta-scheduler.

4.1.1 DPAC systems

DPAC is the organisation with responsibility to design, implement and execute the Gaia data reduction pipeline (see Section 2.3.2). DPAC systems have been running at the DPCB for many years, they include simulation software, namely GASS and GOG, as well as real data processing systems. During normal mission operations, the IDU application, which forms part of the Gaia data reduction pipeline will be executed at the DPCB.

During the design stage of IDU, it was decided that an execution framework should be developed that could handle the launching of the application (launching of the separate tasks that form IDU) in the assigned computing nodes, as well as the management of the input and output data. This decision was largely based on the nature of the IDU application. This application is quite data-intensive, in fact, it is most data-intensive of all the DPAC systems. It is composed of a number of tasks, which are effectively separate applications, but with data input dependencies from other tasks. A particular data distribution pattern is required for the distribution of data amongst the processes running in the assigned computing nodes. Despite the fact that a particular data distribution pattern is required, and in keeping with standard software development practices, this framework should be developed in a generic manner, to allow for its possible use by other applications. As well as handling the launching of the application and the management of data, it should allow Java-based applications to be launched, monitored and managed in an HPC environment, making efficient use of the available resources.

[1]The terminology used can differ, for example, a *meta-scheduler* may be known as a *workload manager*. Sometimes the distinction between a *resource manager* and a *meta-scheduler* is blurred as the range of features supported can overlap

DpcbTools is being developed to assist the execution of the aforementioned DPAC applications at the DPCB. In some cases, DpcbTools will simply act as a library, providing an API of functions useful in an HPC environment. However, DpcbTools also includes the functionality to act as an overall execution framework, allowing for the launching, monitoring and management of applications. DpcbTools will be key for the execution of IDU, as it will control the overall execution of the application.

4.1.1.1 IDU

IDU is one of a number of applications, developed by DPAC, which collectively form the *Gaia data reduction pipeline* — the name given to the chain of applications that will process the Gaia data. The other applications will run at other DPCs and we are not concerned with them here. IDU itself is composed of seven sub-processes, which we call *tasks*, that perform various operations on the Gaia data. The framework must permit the launching of the tasks, the management of I/O data of each tasks, and the monitoring and overall management of the application. Issues of efficiency are especially important for this work, such as providing efficient data access, and ensuring the efficient movement of data between the hardware elements of the system while the application is being executed.

IDU could be described as a batch processing system, as opposed to a parallel application. However, due to data dependencies between the tasks, their execution must be coordinated, and considerable data transfers and arrangement are required. The individual IDU tasks are sequential applications that process data obtained as output from previous tasks, therefore, there are data dependencies between some of these tasks, and the tasks must be executed in a particular order. These dependencies, and the order of the execution of each of the IDU tasks is illustrated are Figure 4.1. One complete execution of IDU can be broken into five steps which must be executed consecutively, these steps are described below:

- Step #0 — the preparation of the input data, and the preparation of the launching of the tasks.
- Step #1 — the execution of three IDU tasks, these could be executed in parallel as there are no data dependencies amongst them.
- Step #2 — the execution of one task, which requires input data from two tasks executed in the previous step.
- Step #3 — the execution of two tasks, which have data dependencies on all of the previously executed tasks.
- Step #4 — the execution of a single task, which has data dependencies on all of the previously executed tasks.

One additional consideration that must be taken into account is that some data rearranging will be required during the period between some of the tasks. This is due to the fact that some of the tasks require the data to be arranged by time, while some other process expect data to be arranged following some spatial criteria. To facilitate this rearranging of data, all of the data must be brought together in one repository, before being distributed again amongst the computing nodes for processing.

IDU will be executed once every six months over a five year period. Each time that IDU is executed, it will process all of the data amassed at that point, so as the mission continues, the volume of data will increase reaching, roughly 100TB at the end of the mission. Although the

actual volume of data will not be overwhelming, the challenging aspect of the processing is due to the relationships within the data, and the rearranging and movement of the data, which must be carried out between tasks.

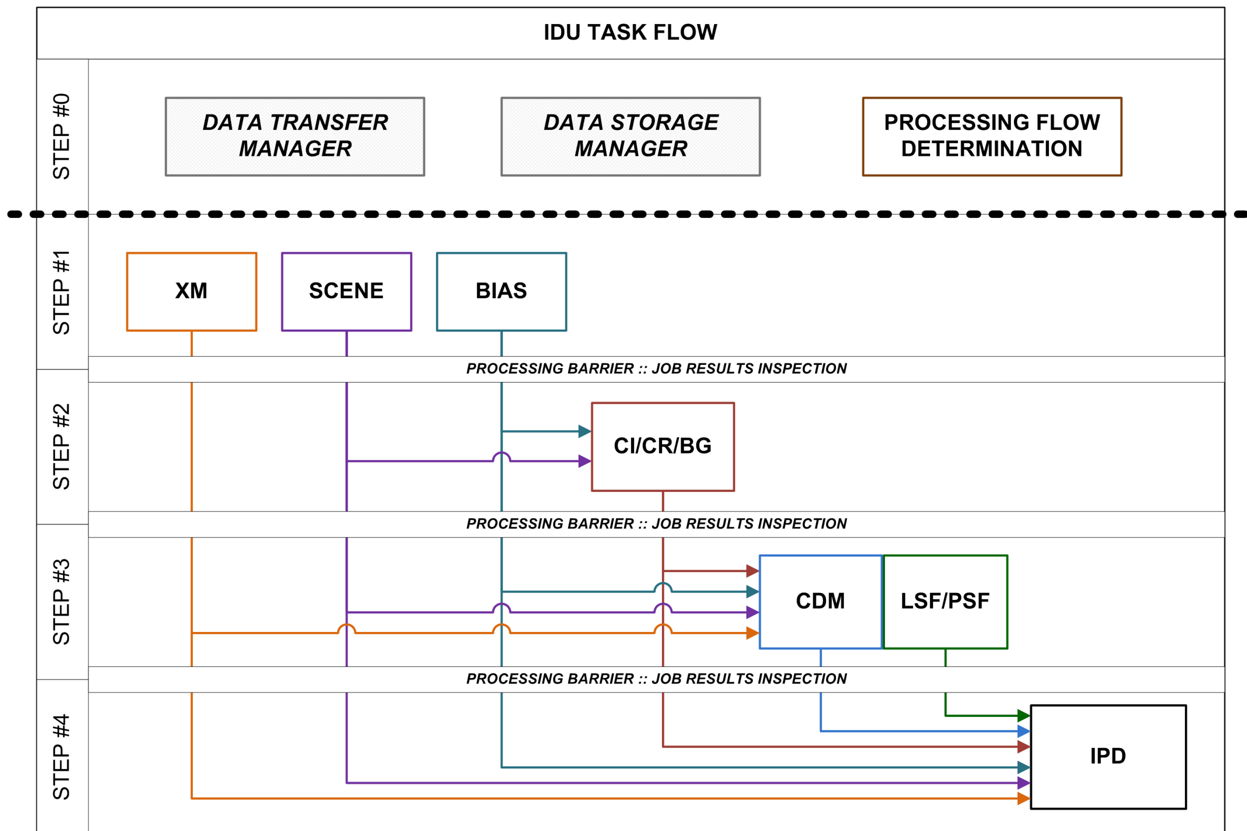

Figure 4.1: IDU Task flow — showing the constituent tasks which comprise IDU, the order in which these tasks must be executed and the data dependencies between these tasks. Figure taken from *IDU Software Design Description* document.

4.1.1.2 Data simulation applications

Prior to the execution of the actual data processing software (IDU), simulation applications are being executed at the DPCB. These simulation applications play a very important role in the preparations for the processing of the Gaia data. The primary simulation applications are GASS and GOG. GASS simulates the raw telemetry stream that will be generated by the Gaia satellite, while GOG simulates data statistically equivalent to the final mission data.

Although these applications are very sophisticated simulators, their structure and execution is much simpler than that of IDU. These applications are composed of a single sequential task, and there are no data dependencies between instances of the task. However, each instance of the task does need to access some common resources files. For the simulators to achieve their data generation goals, the access to these resources must be very fast. These resource files cannot be cached in the scope of the task memory to avoid faulty program termination. Therefore, some other caching mechanism in each node, or located in some dedicated cache nodes would be advantageous.

4.1.2 HPC infrastructure and considerations

A typical distributed-memory HPC environment includes many computing nodes, each computing node containing one or more processors, and each processor containing one or more cores. Each computing node will typically be connected to every other computing node over some network infrastructure, and it is often the case that all of the nodes share some storage device(s).

For the past number of years, the primary computing resource available to the DPCB was the MareNostrum II supercomputer, located at the BSC, see Appendix A for the full specification of MareNostrum II. At the time of writing (November 2012), MareNostrum II has recently been decommissioned, and a new machine is being built, which will also be known as MareNostrum (or MareNostrum III). Although MareNostrum III is a very different machine from MareNostrum II — different processors, more cores and memory per node, and a different network interconnect — the overall design of these machines is quite similar. Therefore, although we mention MareNostrum II in the context of designing our launching, monitoring and management systems, these systems are equally suited to MareNostrum III, or any similar distributed-memory HPC environment.

MareNostrum II was a distributed-memory computer, consisting of over 2500 computing nodes, with each node containing 4 computing cores. Each node was linked to every other node via a Myrinet network, in addition, there was also a Gigabit Ethernet network. Communications over Myrinet add much lower overhead than Gigabit Ethernet, and provide higher-performance, in particular, it provides low-latency communication. The need for significant amounts of data from the users of MareNostrum has motivated the design of a hierarchical storage solution that allows handling an overall capacity of several PBs. Thus, there are three data repositories: a local repository for each node (36GB local hard disk), one global data repository accessible to all the nodes, backed by the high-performance IBM GPFS (280TB), and finally a tape-based backup repository.

The scheduling of jobs in MareNostrum II was managed by the SLURM resource management system, in conjunction with the Moab meta-scheduler. Users connect to login nodes to submit their jobs to the SLURM system, and they are then executed in the computing nodes. An important point to note is that users cannot connect directly to computing nodes, nor can they start their applications themselves. To allow for the transmission of Gaia data, to and from the MareNostrum environment, an interface server has been put in place. This server acts as a gateway between the MareNostrum environment and other DPAC DPCs.

4.2 Requirements

The requirements that DpcbTools must fulfil are based on the requirements of the DPAC systems that will run at the DPCB . They also take into account the particular characteristics of the DPCB execution environment, as DpcbTools should be able to take maximum advantage of these resources, while still remaining flexible enough that it can be used in other hardware environments. In the following sections we describe the requirements that DpcbTools should meet. Although we have broken the main requirements into particular categories, these requirements are all closely linked to each other.

DPCB forms part of the overall Gaia data reduction pipeline, which will be executed across six DPCs. Before and after the execution of IDU, data will flow between the DPCB and the

DPCE. Although, strictly speaking, this functionality is not part of the execution framework, it must be implemented, and can be viewed as a preliminary step in the processing of data at the DPCB.

4.2.1 Launching framework

We require an execution framework which will be able to launch DPAC systems at the DPCB. These applications can be quite complex, IDU in particular. In fact, a full execution of IDU involves running each of the IDU tasks, using many computing nodes, with quite a lot of data transfers and rearranging of data during the period between the execution of each task. The DpcbTools framework will need to act as a node and task manager, launching the execution of particular tasks in particular nodes, and ensuring that the correct data is available for the IDU tasks executing in each node.

The execution environment has particular characteristics that must be taken in account, such as the presence of the Myrinet network, and the relatively small amount of memory and the small hard disks per computing node. An acceptable framework must be able to take advantage of the available resources, as well as ensuring that these resources are being utilised to their potential, for example, ensuring that the four cores in each of the assigned computing node are actually busy most of the time.

Although the execution framework must be able to take advantage of the particular characteristics of the current execution environment, it should still be independent from the hardware, and it should be flexible enough that it can adapt to future hardware changes. The execution framework should be robust, handling localised failures, and it should be able to take corrective steps in order to assure a successful execution of the applications.

4.2.2 Data management

Ever present trends in HPC are the continuous increase in the volume of data being processed, as well as the increase in the processing capabilities of HPC hardware. However, while the processing capabilities available in computing nodes, and the speeds of the networks linking them have been increasing rapidly; the issue of accessing data held in shared storage devices is increasingly becoming a concern, and it has the potential to be a bottleneck during the processing of data. Ensuring that the correct data is available to particular processes, running in separate computing nodes, when that data is required, therefore avoiding deadtime is becoming more challenging, and requires intelligent and efficient data management systems.

One particular concern which must be addressed by an execution framework is the controlling (minimising) of the number of concurrent accesses to shared storage devices — GPFS in the case of MareNostrum. We require a means to distribute data amongst a group of nodes without all of the nodes accessing the GPFS. Although the GPFS is a high performance storage device, designed to allow many processes access the device concurrently, the performance of the GPFS does decline if a large number of processes running in separate computing nodes attempt to access it at the same time. See Section 5.3.2 for an investigation into the scalability of the GPFS.

IDU is a data-intensive application, and the volume of data involved in its execution (by today's standards at least) is quite large, and the data dependencies between the tasks that form IDU require substantial data transfers between nodes. Therefore, the management of data is crucial

to the efficient execution of the application. The input data for IDU will be received by the interface server during a set period prior to the execution of IDU. Initially, this data will need to be stored in the local disk of the interface server, and will then be arranged in a particular order and stored in the GPFS in preparation for its processing by IDU at a later date. During the execution of IDU, data will need to be migrated from the GPFS to the computing nodes for processing. During the period between the execution of the IDU tasks, output data will need to be written to the GPFS, and finally, once IDU has completed its work, the output data will be exported from the MareNostrum environment via the interface server.

There are some common files which will be required by each execution of IDU. These files are relatively small, and could be stored in each node, thereby increasing their access speeds. Therefore, the execution framework should allow for certain files to be stored in a local cache.

Data Access Layers (DALs) will be required to allow transparent access to the various data storage repositories within the execution environment, in particular a *node DAL* and a *GPFS DAL* should be developed. Applications running in one particular node may need to access data held in another node, therefore the node DAL should permit the accessing of data in remote nodes. Data access routines should be independent from the storage mechanisms making it possible to easily change one without affecting the other. Some kind of data caches should be maintained in the nodes to minimise the number of accesses made to the shared disk.

4.2.3 Data communication

Given the distributed memory architecture of most current HPC environments, and the challenge of data management described in the previous section, it is clear that the system for transferring data between the components of an HPC environment is a crucially important element of an efficient HPC data processing system.

In the case of the MareNostrum environment, in order to efficiently exploit the available resources, the Myrinet network (or any other future high-performance network interconnect) should be utilised to provide high-performance, low-latency inter-node communication. Taking advantage of the capabilities of the Myrinet network using a Java-based system is not a straightforward task, as standard Java libraries do not provide support for such networks.

4.2.4 Monitoring

Execution time in an HPC environment is a valuable resource. Users and administrators are keen to ensure that the available computing resources are being used as close to their full potential as possible. This requires some mechanism for monitoring the state of the system. Often, the applications which are executed in HPC environments require relatively long periods of time to complete their work. If something goes wrong during their execution, the owners of such applications need to be made aware of that fact, to allow them to take some corrective action. The requirements of a monitoring system for HPC environments are discussed in more detail in Chapter 9.

In the case of the DPAC systems running at the DPCB, it was decided that three levels of monitoring should be supported: OS level monitoring, JVM monitoring, and finally, monitoring of application properties. The OS level monitoring should allow for the monitoring of basic system statistics, such as CPU and memory usage. The JVM monitoring should allow for the internal state of the JVM to be monitored. Pertinent JVM information might include the

current amount of free space available in the heap, and the rate at which the garbage collections are being performed. The third element of the monitoring system — application properties monitoring — should allow for certain internal application properties to be monitored, and possibly modified, when the application is running. This would allow an application to be tuned *on the fly*, based on how the execution is progressing.

The monitoring framework should be independent of the actual application being monitored, as it will be used to monitor several distinct applications. Given the fact that DPAC software will be using a large number of computing nodes, it would be useful to aggregate all of the monitoring information from all of the computing nodes into one unified overview of the status of the execution. One final consideration is that the monitoring framework should be as *lightweight* as possible, incurring little overhead, thus causing the minimum effect on the performance of the application being monitored.

4.3 Related work

Taking the requirements of the DPAC systems discussed in the previous section into account, and the particular characteristics of the DPCB environment, we investigated existing execution and management systems for Java applications in HPC environments. We were interested in the approaches taken, and wished to determine if these systems met our needs. Given the fact that our requirements cover a wide range of functionality, we were aware that it was unlikely that we would find one solution which satisfied all of our needs. Therefore, we investigated our main requirements separately, and we looked at applications and libraries which provide solutions for any of our requirements, and not necessarily meeting all of our needs. In this section we discuss related projects providing execution frameworks for Java applications in HPC.

Research into the use of Java in HPC can be traced back to the JGF (see Section 3.3.1). This initiative, to encourage interest in Java amongst the scientific and HPC communities and to investigate possible additions to the Java language, began around 1999. After a few years however, it lost much of its impetus after initial investigations showed relatively poor performance by early versions of the JVM. This initial set of disappointing results contributed to the poor reputation of Java amongst the HPC community. However, in the intervening years, considerable work has been carried out improving the performance of JVMs, and it has improved vastly.

ProActive [Baduel et al., 2006] is an extensive, open source, middleware framework, released under the LGPL, that attempts to support the development, integration, deployment and execution of Java applications in a grid environment. ProActive continues as an ongoing project, and there is a strong community of researchers and developers supporting it. It takes an object and component-based approach, and a key concept is that of *active objects*. These are Java objects, but each one has its own associated thread of execution. Active objects can be created on any of the JVMs involved in an execution (which may be executing on different host machines), and in fact, they can migrate from one JVM to another. Method calls to *active objects* are processed through a queue system, and they are free to decide in what order methods should be executed. These objects communicate with each other via asynchronous requests, which are similar to the messages passed between MPI processes. For example, the MPI function `Sendrecv()` has a similar `exchange()` function in ProActive.

Proactive provides software designers with a general purpose programming model allowing for the design of applications using a hierarchical component framework called the Grid Compo-

nent Model (GCM). It advocates that applications should be abstractly described in terms of the activities necessary to complete their execution, including, for example, a description of which components of the applications should be executed in parallel. Such an abstract description eases the deployment of applications to various environments, and helps to eliminate any application dependence on particular execution environments.

ProActive, by default, uses RMI as its transport layer, however it has been designed in such a way as to allow its transport layer to be easily changed without requiring any changes to the above layers (the data communication layer of ProActive is discussed in Section 5.2.1.4).

COMP Superscalar (COMPSs) [Tejedor and Badia, 2008] is an execution framework that eases the development and execution of applications on GRID environments, such as a typical computer cluster. Building on the functionality provided by the earlier GRID Superscalar (GRIDSs) [Badia et al., 2003], it offers a component-based runtime environment, following the GCM model. It can automatically exploit any parallelism within applications, thereby allowing such applications to make efficient use of the available computing resources. It supports Java as well as C and C++.

The basic idea behind COMPSs is the decomposition of applications into smaller units of work, known as *tasks*, which can be executed in parallel, and the execution of these tasks by transient worker processes across the distributed resources of the system. In the case of Java, users define which methods of their application should be used as the tasks in an interface, along with some additional metadata using Java annotations. Users do not need to modify their original applications at all. COMPSs performs some analysis of applications that involves determining the dependencies between the tasks; generating a task graph; and where possible, executing tasks in parallel. COMPSs also manages the distribution of input data and the collection of output for each task.

COMPSs is a very powerful tool, allowing seemingly serial applications to become parallel, and taking advantage of the available HPC resources. However, it does not deal with the potential bottleneck associated with many processes accessing shared storage devices. Although COMPSs removes the issue from the concern of user applications, COMPSs itself may encounter I/O performance issues if it tries to distribute some data to a large number of nodes concurrently. Also, there are circumstances when application developers may prefer to maintain control of the actual flow of data around the available hardware.

Ibis [Ibis, 2012] is an extensive, open source project, that aims to develop programming models, libraries and middleware that make the development of distributed Java applications easier. The Ibis project encompasses several individual projects, these include: JavaGAT (Java Grid Application Toolkit), which provides a number of APIs useful to applications running in a Grid environment; an implementation of the RMI protocol which boosts performance; the Ibis Portability Layer (IPL), which acts as an interface to a number of underlying programming models and pluggable components; Satin, a parallel programming environment that facilitates a divide-and-conquer approach for decomposing tasks into smaller tasks that may be executed in parallel; and an implementation of MPJ known as MPJ/Ibis (discussed in Section 5.2.2).

DiSK [Szeliga et al., 2009] is a C-based, distributed shared disk cache for HPC environments. Although this work is not related to Java, it is of interest to this thesis as it identifies the challenge of migrating data from back-end storage devices to computing nodes as a potential bottleneck within the processing of data in HPC environments. DiSK allows for the distribution of requested files across the nodes involved in an execution, thereby reducing the dependence on shared back-end storage devices.

Our investigations of related work revealed many projects with the aim of facilitating the execution of Java applications in HPC environments. These systems provide a wide range of functionality, and the needs of the majority of Java applications could be met by some of them. However, none of these system dealt with all our requirements. We were unable to find, for example, any existing monitoring system which presents an aggregated overview of the execution of a Java application across many computing nodes. Neither were we unable to find any data communication libraries offering efficient and sufficiently high-level data communication functions for Java applications in HPC environments. There is an additional concern with the use of the investigated launching frameworks. They involve the creation of transient JVMs in the worker nodes in order to execute instances of the tasks. The use of transient JVMs limits the ability of the JIT compiler to perform optimisations based on long executions. This issue is further discussed in Section 8.2.

4.4 DpcbTools design

Here we present the design of the main components of DpcbTools. Having investigated the currently available execution frameworks supporting Java applications in distributed-memory environments, and considering the specific data communication pattern of IDU, as well as the fact that we would like to be able to control and manage the creation of JVMs in the computing nodes, it was decided that a bespoke job launching and data management framework should be developed which would manage the launching of the IDU tasks. This will allow us to directly control the execution of each task, and the management of data — which is the most challenging aspect of the design.

The launching framework was designed with an *IDU-like* application (where the processing can be broken up into small independent units of work, that can be launched in a batch system) in mind. However, where possible, it has been designed to support any type of application, regardless of the applications design. In particular, the data communication system and the monitoring system can be used by any application, regardless of whether or not the DpcbTools launching framework is used.

4.4.1 Launching framework

The framework involves the initial division of the total processing challenge into smaller *chunks* of work, and the subsequent distribution of these chucks amongst the available computing resources. The available computing nodes are grouped together into NGs, one computing node within each NG is designated as a NGM. It is the responsibility of the NGM to distribute chunks of work to the computing nodes within its NG. The management of the assigned computing nodes is described in Section 4.4.1.1, and job management is described in Section 4.4.1.2.

4.4.1.1 Node management

We have designed a scheme which involves the grouping together of the available computing nodes into NGs. Within each NG, one computing node is designated as the NGM, as proposed [Fries et al., 2010], and illustrated in Figure 4.2. It should be noted that this grouping of computing nodes into NGs is purely conceptual. And in fact, the nodes within a single NG have no more connection with each other than any two computing nodes.

Only the NGMs access the shared-memory storage device, while all of the nodes within a NG communicate with each other over the Myrinet network. Limiting the number of processes accessing the shared storage device ensures that its performance does not decrease. Following this scheme, each NG is therefore largely isolated from each other, and they can be seen as separate clusters, any failures which occur within a single NG do not affect the processing in the other NGs.

A server process is executed in the NGM, which creates and maintains a cache of data that user application processes can query. We refer to user application processes simply as client processes. The inter-node communication — between clients and servers, as well as amongst clients — makes use of the MPJ-Cache middleware (see Section 5.5 for a detailed description). Taking advantage of any high-speed network support provided by an underlying MPJ implementation.

There are of course, a number of possible NG configurations that a given set of computing nodes can be grouped into. For example, 64 nodes could be grouped into 4 groups of 16, or into 8 groups of 8 nodes, or 32 groups of 2 nodes. However, larger numbers of NGs implies larger numbers of nodes that will attempt to access the shared storage device, as one node within each NG may access the shared storage device. Tests were carried out to understand how the performance of the shared storage device reduces as the number of concurrent accesses increases. These test are described in Section 5.3.2.

The following software components were designed for the purposes of node management:

- NodeCoordinator — one instance would be executed in each node, with responsibility for the execution of jobs in its node.
- NodeGroupCoordinator — one instance would be executed in the NGM of each NG, with responsibility for the management of the nodes within its NG. The NodeGroupCoordinator will be aware of which data has already been processed by each NodeCoordinator, and can take this information into account when distributing jobs to NodeGroupCoordinators in order to minimise the number of data transfers required.
- NodeTaskManager — one instance of would be executed for the entire application, with responsibility to initialise the entire application, monitor the execution of the system, and to take action to correct any failures which might occur.

These components were designed and implemented in a phased development. This allowed for initial implementations of IDU to be executed through the "manual" launching of the individual NodeCoordinators as separate SLURM jobs. Later, when the NodeGroupCoordinators were added, the system was executed through the launching of the individual NodeGroupCoordinators (which subsequently launched the NodeCoordinators) as separate SLURM jobs. And finally, once the NodeTaskManager is in place, the entire system will be launched through the submission of a single SLURM job.

4.4.1.2 Job management

The following job management scheme was designed with IDU in mind, however it could be used for any application of a similar structure. It was decided to break the entire processing challenge into small units of work, of similar sizes. The information defining each of these units of work is encapsulated in an Extensible Markup Language (XML)-based file, which we call a *task file*. These task files define which IDU task should be executed, where the input data can be found, and where the output data should be written, as well as configuration properties.

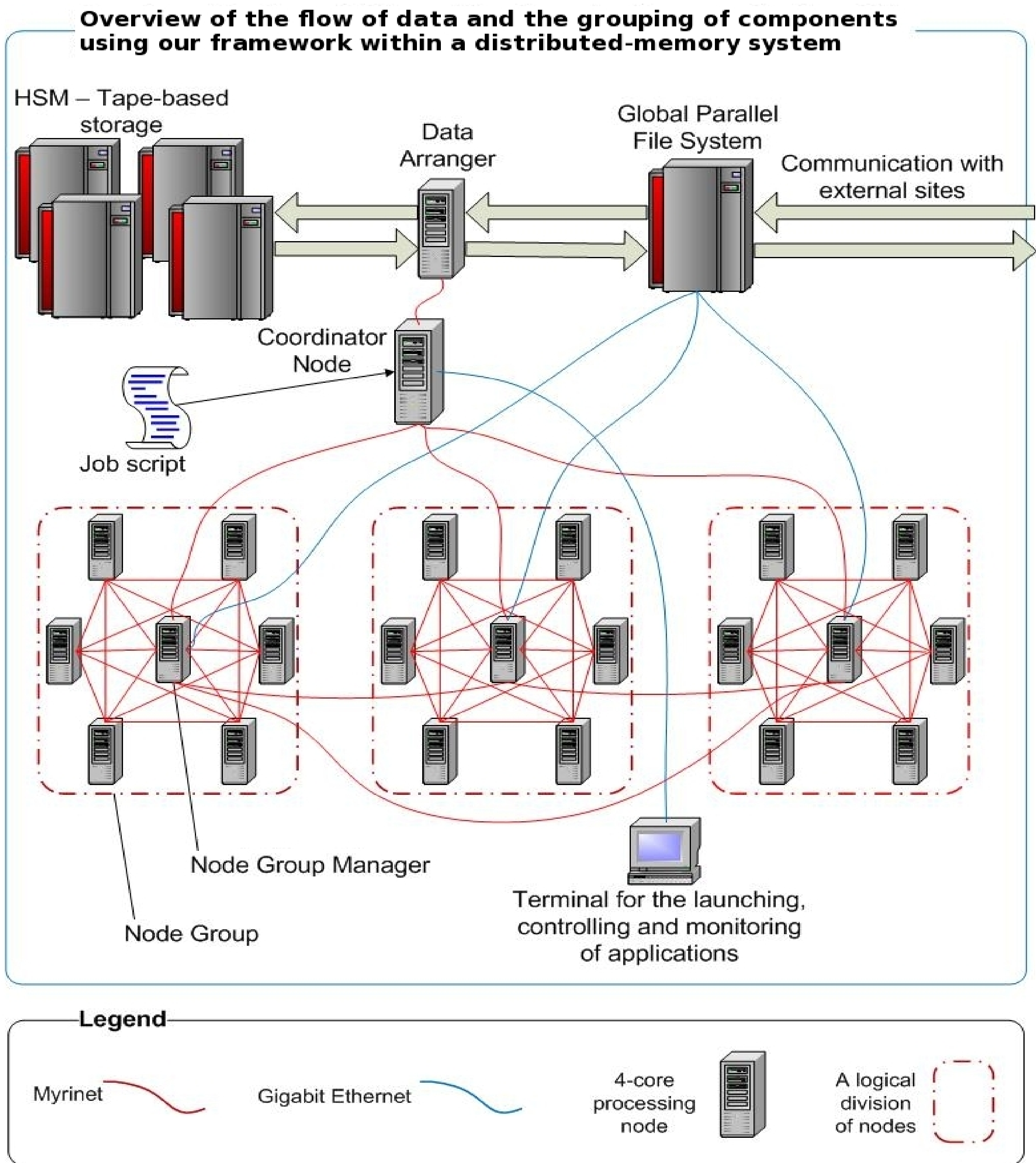

Figure 4.2: Overview of how the nodes in a distributed-memory system, such as MareNostrum, can be grouped using our node management framework.

The creation of the task files is an automatic process which is carried out by an application called *createTaskFileFromTemplate*. This application generates all of the task files that are needed to be processed in order to complete an execution of IDU. Once the entire stack of task files has been created, we must then create *job files* that can be submitted to the job scheduler system. The relationship between *task files* and *job files* can be one-to-one or many-to-one, depending on how we wish to launch the application. The creation of the job files is handled by an application called *createTaskLaunchScript*. Based on the arguments specified, this application can generate the following possible job files:

- An individual job script for each task file.
- A single job script with the individual launch command for each task, in other words a script which will run each task sequentially.
- A single job script with a single process — the NodeCoordinator, which will take care of the concurrent execution of the given task files.

The job management scheme that will be used during the actual processing of the Gaia data will involve the NodeTaskManager first distributing task files amongst all of the NodeGroupCoordinators, and later, the NodeGroupCoordinators distributing its task files amongst the NodeCoordinators running in each of the nodes in its NG. Before the actual processing of data commences, each NodeGroupCoordinator will have a stack of task files ready to be distributed. Once the NodeCoordinators are initialised, they will begin asking for task files from their NodeGroupCoordinator, following a client-server model. Due to the grouping of the computing nodes into NGs, any failures will limited to a single NG, and any NG can be relaunched, without affecting other NGs.

4.4.2 Data communication

As stated previously, it is crucial that any available high-performance communication networks are exploited to their full potential, if maximum performance is to be obtained. After investigating the options for exploiting such networks in Java, it was decided to build our inter-node communication system on top of an implementation of MPJ. This approach has been reported to provide the highest performance, and best scalability, to HPC applications on high-performance, low-latency networks [Taboada et al., 2009], such as Infiniband and Myrinet. The use of a message passing library gives access to the convenient, rank-based system used within message-passing libraries to uniquely identify processes. Communication between processes is relatively easy to program, given the extensive point-to-point and collective functions supported by these libraries. MPJ Express and FastMPJ were both tested to act as the MPJ layer, tests comparing these applications are described in Chapter 5.

MPJ-Cache is the data communication component within DpcbTools, it sits on top of an implementation of MPJ (such as MPJ Express or FastMPJ) and has two main components within it, namely the communication layer and the data cache. A brief description of these components is given in the following sections, and MPJ-Cache is explained in greater detail in Section 5.5.

4.4.2.1 MPJ-Cache - the communication layer

The objective of the MPJ-Cache communication layer is to provide user applications with a high-level API of data access methods useful in HPC environments, while making best use of

any high-performance, low-latency networks which might be present. MPJ-Cache makes use of an implementation of MPJ to perform inter-process communication. Implementations of MPJ, such as MPJ Express and FastMPJ implement an API of methods, such as `MPI_Sendrecv()`. MPJ-Cache builds upon the MPJ API, and offers its own API of methods which are at a higher level of abstraction. For example, the MPJ-Cache API includes the static method `retrieveSingleFileAsByteArray()`, which allows clients to request a file as an array of bytes. The implementation of this method involves two calls to the MPJ method `MPI_Sendrecv()`. Firstly, a request is made to find the size of the file. Then, once the size of the file is known, a buffer can be allocated in order to store the data and a second call to `MPI_Sendrecv()` is made requesting the actual file. However, from the perspective of the client application, it must only make one call to `retrieveSingleFileAsByteArray()`.

The communication layer also includes file splitting and recombining functionality, which allows accessing files with sizes that exceed the maximum data size permitted by the underlying implementation of MPJ. Such large files are split into smaller chunks by the server, which are then sent in separate messages, and finally, once all of the chunks have been received at the client side, they are recombined and passed to the client application.

4.4.2.2 MPJ-Cache - the cache

In the context of MPJ-Cache, the *cache* refers to the part of the server application that maintains the actual cache of data. The server can be configured to either store the most frequently used data in memory, on the local disk of the node that it is running on, or to simply act as a gateway, retrieving data from a remote location as requests are received. Indeed, the cache can be spread over these three locations. The server maintains a list of all of the files that it is aware of: those it has in memory, those that it has on its local disk and those which might be in a remote location. The server initializes its cache at start-up, based on its configuration, but it can update the cache during the execution of the application, depending on its configuration. A number of policies to control the maintenance of the cache including First In First Out (FIFO) and Least Recently Used (LRU) are available.

4.4.3 Monitoring

The DpcbTools monitoring component is designed to monitor three distinctive categories of information: OS information, JVM information and application specific information. The OS information is monitored using a Python-based application. JVM state information as well as internal application properties are monitored using a JMX-based application. In addition to monitoring, this framework allows for management of applications through an instrumentation process, whereby certain application properties are exposed to the JMX framework. These monitoring and management systems are described in Chapter 9.

4.5 Conclusions

We have designed an execution framework that allows for the efficient launching, monitoring and management of Java applications in HPC environments. DpcbTools is still under development, however, already DpcbTools has produced some excellent results, in particular the results from the MPJ-Cache tests. In addition, the node management scheme, which includes the division

of the assigned computing nodes into NGs, as well as the automatic creation of task files and job files, and the subsequent launching and management of the job files is proving an efficient and reliable solution.

5

Data Communication

Knecht's Law: software development is harder than you expect, even when you take into account Knecht's Law

– A corollary of Hofstadter's law — Joseph Knecht

Efficient data communication is extremely important in HPC. Typical distributed-memory HPC environments, such as computer clusters, include many computing nodes — each possessing its own processors and memory. The execution of parallel applications in such an environment typically involves running many parallel processes in many of the computing nodes. These processes may require the ability to communicate with each other during processing. A communication infrastructure permitting fast communication between processes running in the same computing node, as well as communication between processes running in different computing nodes is essential if scalable high-performance is to be obtained. Efficient parallelism is often dependent on efficient communication between the distributed components of the system.

Apart from the requirement to provide fast communication for parallel processes, communication infrastructure is also very important for managing data distribution within HPC environments. Often, all of the computing nodes in a distributed-memory HPC environment may share a central large storage device. In the case of many applications, each of the processes running in the computing nodes may need to concurrently access some data held in the shared storage device. Additionally, processes running in one computing node may need to access data that is currently held in another computing node. In I/O-intensive applications, the accessing of data not held in the local computing node can become a bottleneck, especially in cases where the remotely-stored data is accessed repeatedly, and when accessing data from VMs such as in Java.

MPI [MPI, 2012] continues as the leading approach for providing communication functionality to applications running in distributed-memory HPC environments, such as in computer clusters. There have been a number of implementations of MPI for Java, these are known as MPJ. In this chapter, the current status of data communication for Java applications in HPC environments is described, including a description of the approaches that may be followed to provide communication functions to Java-based applications in HPC. Additionally, a summary of a number of projects that have provided communication systems to Java applications in HPC is given, this includes several implementations of MPJ.

Building on MPJ capabilities, a new middleware called MPJ-Cache [Fries et al., 2011b] is presented. This middleware offers a high-level API of communication functionality useful to Java

applications running in HPC environments. Conceptually, MPJ-Cache sits on top of an underlying implementation of MPJ, making use of the MPJ environment and the functions provided, while also taking advantage of any underlying high-speed network support provided. MPJ-Cache offers an API of functions at a higher level of abstraction than is available by only using an implementation of MPJ, and it is more intuitive for application developers — who may have little experience with message-passing. MPJ-Cache includes intelligent caching and prefetching features, which can further improve application performance.

In this chapter, we describe the existing options for providing high-performance data communication to Java applications in HPC environments, and we then present our MPJ-Cache middleware.

5.1 Background

A typical distributed-memory HPC environment includes many computing nodes, each one containing one or more processors and each processor containing one or more cores. Each computing node may be connected to every other computing node over some high-speed, low-latency network, while each computing node may also be connected to a central storage device (see Section 2.1.4 for a description of common HPC architectures). In order for high performance to be achieved, any available high-performance network infrastructure linking the distributed components of the system must be taken advantage of.

The current dominant trends in HPC include an ever increasing number of cores per processor and the increasing sizes of the datasets that must be handled — commonly referred to as "Big Data" (see Section 2.1 for a description of the current trends in HPC). Together, these trends generally lead to a larger number of processes running on a larger number of computing nodes, and these processes are accessing larger datasets. These trends combine to make the issues of efficient data access and data communication increasingly important. The importance of efficient I/O grows with the number of parallel processes, and if the data is accessed repeatedly. In the case of I/O-intensive applications, I/O can become a significant factor affecting the overall performance of the application, and has the potential to become a bottleneck in processing, if not implemented correctly.

Scalability is a primary concern while attempting to run large-scale executions in HPC environments. Poor scalability results in performance not increasing in proportion to the computing resources used. Although the computational performance of Java is now on a par with the traditionally used HPC languages, Java-based applications often do not scale as well as other applications. One of the factors for this has been poor support for high-speed networks. Additionally, scalability can be negatively affected by bottlenecks in processing occurring, one cause of these can be multiple processes (or threads) attempting to use a shared resource at the same time, resulting in some processes having to wait. Scalability can be improved by taking maximum advantage of any available high-performance computing resources, and by designing systems that include as few potential bottlenecks as possible.

Several approaches have been followed for providing communication functionality to Java applications in HPC. Some approaches are best suited to shared-memory systems, while others are better suited to distributed-memory systems. In the case of shared-memory systems, parallelism can be achieved through the use of threads, or some higher-level framework such as OpenMP. In the case of distributed-memory systems, RMI or an implementation of MPJ may be used.

5.1.1 Networks for HPC

Latency is a measure of the time delay that occurs within a system. In the context of network communications, latency is typically used to measure the delay between some data being sent at some point, and being received at another. Bandwidth refers to the amount of data that can be transmitted from one point to another in a given period of time, often expressed as the number of bits per second (bps), such as 2 Gbps (Gigabits per second).

Ethernet is a family of frame-based computer networking technologies, Gigabit Ethernet (GbE) and 10 Gigabit Ethernet (10GbE) are popular network technologies that allow for the transmission of Ethernet frames at rates of 1 and 10 Gigabits of data per second respectively. The TCP/IP protocol dates back to the 1970s, and was designed to operate over low-speed, unreliable, and physically extensive network infrastructures. TCP/IP is normally used over Ethernet, however it can also be used over other networks. TCP/IP includes a relativity large amount of error checking, which adds overhead to its performance, and it cannot full take advantage of the performance offered by high-performance networks. Therefore, the use of TCP/IP emulation over high-performance networks delivers relatively poor performance, compared with native libraries for such networks

Myrinet [Qian et al., 2004] is a high-performance network developed by Myricom. It has much lower protocol overhead than Ethernet, and provides better throughput, less interference, and lower latency. The protocol employed by the network is much simpler than, for example, TCP/IP, thus the protocol itself adds little overhead to communication performance. Although Myrinet remains a widely used technology, its popularity amongst supercomputers has been dropping steadily for a number of years According to the TOP500 list [TOP500, 2012], in June 2005, 141 of the top 500 supercomputers used Myrinet, representing 28%, but by June 2012 this had fallen to just 4, representing 0.8%. The latest Myrinet version, Myri-10G, is physically compatible with 10 Gigabit Ethernet (GbE).

Myrinet Express (MX) is a high-performance, low-level, message-passing software interface tailored for Myrinet. MX exploits the processing capabilities embedded in the Myrinet Network Interface Card (NIC) to provide exceptional performance for modern middleware such as MPI, and places very little processing burden on the CPU. OpenMX [Goglin, 2011] is a port of the MX protocol that can be run on top of generic networks such as 10GbE. OpenMX provides the same functionality as MX, and they are interoperable with each other.

Infiniband [Mellanox Technologies, 2007] is a high-speed low-latency network technology, based on switched fabric topology. Infiniband has established itself as one of the most widely used network technologies in HPC. According to the Top500 supercomputer list [TOP500, 2012] published in June 2012, 208 of the top 500 supercomputers in the world use Infiniband, representing 42%, and this percentage is increasing. The Infiniband specification does not define a particular API, however the *de facto* standard is the OpenFabrics Enterprise Distribution (OFED) from the Open Fabrics Alliance.

In January 2012, Infiniband technology was purchased by Intel, thereby broadening the range of HPC products offered by Intel, and enabling it to compliment its many-core processors, with a high-performance technology that can link distributed components.

Remote Direct Memory Access (RDMA) is a mechanism that allows for very efficient data transfers in a distributed-memory environment, without the need of going through the OS, thus avoiding unnecessary copying and buffering of data. Additionally, RDMA is performed without requiring processing by the CPU, thereby freeing the CPU to perform other tasks.

5.1.2 OpenMP

OpenMP is an API that allows for the development of parallel software, for shared-memory environments. Like MPI, it is defined by a group of leading hardware and software producers, including Intel and IBM, known as the OpenMP Architecture Review Board (ARB). OpenMP supports Fortran, C and C++, although there are also a number of OpenMP-like implementations available for other languages, including Java. The OpenMP API is designed to be both hardware and language neutral. It specifies a number of directives, known as *pragmas*, which can be added to sequential code to define which sections of the code should be executed in parallel. These *pragmas* can be added progressively, introducing more and more parallelism in steps, unlike some other approaches which require parallelism to be taken into account at the design stage.

The use of OpenMP offers a high-level of abstraction, and better programmer productivity than the use of threads. OpenMP may be used in combination with some other parallel programming model, such as MPI, in order to implement parallelism in distributed-memory environments, such as computer clusters. In such cases, OpenMP may be used for enabling parallelism within a computing node, while MPI is used for inter-node communication.

5.1.3 MPI

MPI [MPI, 2012] is the leading approach for implementing inter-process communication in distributed-memory HPC environments, such as computer clusters. The reasons for the success of MPI include its scalability to large numbers of processes, and its portability to a wide range of hardware architectures. MPI is built on the concept of message-passing — the passing of messages between processes which are running in parallel, thus allowing these processes to communicate with each other. It may be used to provide communication between processes running in a shared-memory, a distributed-memory or a hybrid system. The MPI approach follows the SPMD programming model.

The message-passing paradigm had been widely used for many years before its standardisation in the MPI specification. Previous to the release of the MPI specification, several organisations had been using their own message-passing libraries, including IBM and Intel. A standard set of communication routines would allow hardware vendors to provide efficient support for these routines, leading to increased performance, scalability and portability. A number of organisations began working together on the standardisation of the paradigm in 1992, leading to the first MPI API specification in 1994 [Forum, 1994].

The MPI API specification is a language-independent specification of communication functions that can be used to facilitate inter-process communication within parallel applications. It includes two main categories of functions: point-to-point functions for allowing communication between two MPI processes, and collective functions for allowing communication between many processes simultaneously. Point-to-point functions, or *primitives* as they are sometimes known, follow the *sender and receiver* model, with one process sending a message and another process receiving the message. There are a range of MPI collective primitives specified, including Broadcast, Scatter, and Gather. They provide developers with a relatively high-level set of functionality, leading to improved developer productivity. For applications that use collective primitives, the scalability of these primitives is an important consideration, poorly scaling primitives can act as a bottleneck in processing. One of the mechanisms employed by MPI

collective primitive algorithms to ensure scalability is to perform many communications in parallel.

There are a number of specifications of MPI, and they can be grouped into two main branches: *MPI-1* and *MPI-2*. MPI-1 was the original branch of the specification, it is only concerned with message-passing and involves a static runtime. The most recent version of MPI-1 is MPI-1.3. The MPI-2 branch was first released in 1997 and it is largely a superset of MPI-1. However, it also includes a number of extensions to the specification including the ability of processes to spawn new processes, one-sided communication functions, and parallel I/O [Forum, 1997]. At the time of writing, the next version of MPI — *MPI-3*, has already been in development for several years, and is expected to be released very soon. Amongst the major improvements in this version, is much improved support for Remote Memory Access (RMA).

The term *thread-safety*, in the context of MPI, is used to refer to the support for multithreading within a single MPI process. Providing such support adds an additional overhead and can negatively affect the performance of MPI implementations. The MPI-2 specification introduced four levels of thread-safety: *MPI_THREAD_SINGLE* provides support for single threaded applications only; *MPI_THREAD_SERIALIZED* supports multithreading, but only the thread that initialised MPI may make MPI calls; *MPI_THREAD_SERIALIZED* supports multithreading and each thread may make MPI calls, but not concurrently; while *MPI_THREAD_MULTIPLE* offers full multithreading support, allowing any thread to make MPI calls at any time. MPI implementations are not actually required to be thread-safe, thus thread-safety is not guaranteed with MPI.

MPI implementations are available for a number of languages, and MPI is commonly used by application developers as a means of implementing parallelism, when programming with the traditionally used HPC languages of Fortran, C and C++. MPICH, developed from the Argonne National Laboratory (ANL), is one of the most widely used implementations of MPI. Like the the MPI specifications itself, there are a number of versions of MPICH. MPICH-2, which supports MPI-2.2 is possibly the most widely used implementation of MPI, at the time of writing. MPICH-2 offers a high level of thread-safety. Message-passing in Java is commonly referred to as MPJ, and there have been many implementations of MPJ. These are discussed in detail in Section 5.2.

Despite the strengths and maturity of MPI, writing applications that can take full advantage of the resources of distributed-memory HPC environments, such as computer clusters, remains a challenging task. In the case of complex applications, where there may be strong dependencies between processes running in different computing nodes, the programming effort required to implement the communication can be considerable and is often bug-prone. The task of debugging parallel applications, especially those which use MPI, is considered a difficult task, compared with the debugging of sequential applications.

5.1.4 Client-server model

The client-server model is a long established and intuitive approach for making resources or services available. The basic idea behind this model is that one or more processes, referred to as *servers*, make a resource available to other processes, referred to as *clients*. Traditionally, such systems followed a *pull* model whereby the clients request, or *pull*, a resource from the servers, although they may also follow a *push* model, if the servers are aware of which resources that should be sent to each client. In such cases, the servers may send, or *push*, the correct

resource without having received a request. The client-server model is pervasive throughout computing. Most network technologies, including Hypertext Transfer Protocol (HTTP), and File Transfer Protocol (FTP) follow this model.

5.2 Related work

In [Taboada et al., 2009], and more recently in [Taboada et al., 2011a], up-to-date assessments of the current status of Java for HPC are given. In particular, they have focused on the options for implementing data communication for Java in HPC. They highlight the fact that, although the performance of JVMs has improved vastly since the early JVM versions, the Java community has lacked a coordinated effort to implement efficient and scalable communication middleware. The performance of serial Java applications is now very similar to the performance offered by the traditionally used HPC languages. However, the performance that can be achieved by parallel Java applications continues to lag behind that of the traditionally used HPC languages. The primary reason for this, is that parallel Java applications often rely on inefficient communication middleware, which cannot take full advantage of high-performance network infrastructure.

5.2.1 Approaches for implementing HPC communication in Java

There exists several programming options for implementing parallelism and permitting data communication for Java applications in HPC. In the following sections we describe some of the leading implementations of each of these approaches.

5.2.1.1 DSM systems

Jackal [Veldema et al., 2001] is a software-based DSM system that supports the execution of parallel Java applications in distributed-memory environments, such as clusters. It includes a compiler and runtime system. It supports the use of fine-grained distributed memory regions. Data is distributed based on the object level, large objects may occupy several memory regions. Developers may write applications without any concern of the possible distributed nature of execution environment. During compilation, the Jackal compiler inserts *access checks* into the code each time that an object is accessed. These checks are used at runtime to determine if an object is available in the local node, or whether a copy of it needs to be retrieved from another node. All objects are assigned a single *home node*, but copies of objects may be cached at other nodes. Changes to such copies may be made, and these must be flushed back to the home node at certain synchronisation points. The challenge of maintaining consistency, while also providing high performance is considerable, however Jackal utilises several sophisticated optimisations to achieve this goal [Veldema et al., 2002].

There exist several execution frameworks which automatically manage the distribution of input data to the computing nodes where instances of an application are to be executed. COMPSs [Tejedor and Badia, 2008], for example, not only supports the automatic parallelisation of sequential applications, it also manages the distribution of input data amongst computing nodes. Many systems perform this task automatically, meaning that application developers and operators does not need to concern themselves with the distribution of input data amongst computing nodes. Such functionality is a useful feature for many applications. Such systems transfer the required input data before instances of the application are actually launched in

the computing nodes. They generally do not concern themselves with the potential bottleneck associated with many concurrent processes accessing shared storage devices, therefore their data distribution service can suffer from scalability problems.

5.2.1.2 Shared-memory programming

Applications designed for running in shared-memory environments may make use of Java threads for implementing parallelism. However, solely relying on Java threads results in an application that is not transferable to distributed-memory environments — the much more commonly-encountered system architecture. Although Java now offers a very rich set of threading utilities, it still involves quite a low-level programming effort. In terms of complexity and programmer productivity, the use of threads for programming parallelism, especially for large applications, requires a significant and careful programming effort. Thread contention, deadlock, and race condition bugs can easily be introduced (see Section 3.2.3 for a description of Java threads).

Parallel Java [Kaminsky, 2007] is a concurrency library providing the functionality of both OpenMP and MPI in a unified, OO API, and is totally written in Java. In fact, Parallel Java was designed to provide the functionality needed for implementing parallelism in hybrid SMP clusters, which typically involve multiple shared-memory components connected through some network infrastructure. Parallel Java builds on top of the multithreading utilities provided in Java, and adds further high-level programming constructs, including *ParallelRegion* and *ParallelTeam*. It includes an execution framework that, in the case of a computer cluster environment, involves the launching of a *Job Scheduler Daemon* on a single frontend node, and *Job Launcher Daemons* on the other nodes. Parallel Java is an impressive tool, providing a rich set of functionality to parallel application developers. Although Parallel Java provides MPI-like and OpenMP-like functionality, it does not actually support the MPI or OpenMP specification, which might be limiting its adoption by the development community. However, it is suggested that the API offered by Parallel Java provides the same level of functionality as MPI and OpenMP, but through the use of a simpler API, thus improving developer productivity.

There have been several implementation of OpenMP for Java, JOMP [Kambites and Bull, 2001] is an OpenMP-like Java API. In common with OpenMP, it involves application developers inserting directives, or *pragmas*, into their code which dictate how the program should be parallelised. This code is then run through a precompilation process which produces a new version of the Java source code, containing calls to the JOMP runtime library, this code is then compiled and executed as a normal Java application. Although JOMP is still available, it appears that it has not been updated by its creators for several years, although JOMP was used by an early version of MPJ Express to achieve nested parallelism (see Section 5.2.2.1).

JaMP [Klemm et al., 2007] is another Java-based OpenMP-like system. It is built on top of Jackal framework, and it complements the DSM support provided by Jackal (see Section 5.2.1.1). JaMP supports a large subset of the OpenMP specification, and the JaMP directives closely match those of OpenMP. The authors evaluated the performance of JaMP using a set of microbenchmarks and an implementation of the Lattice-Boltzmann Method (LBM). They found a speedup of 6.1 for 8 nodes, achieving 83% of the performance of a C-based implementation that used MPI for communication. JaMP does however suffer from scalability problems with larger numbers of nodes, these problem seem to be inherent in the Jackal framework which JaMP is dependant upon.

5.2.1.3 Java sockets

A socket API specifies a relatively low-level set of communication functions which allow applications to make use of network sockets, thus enabling processes to communicate across a network. Most computer programming languages provide a socket API. For Java, the package *java.net* package contains a number of classes that can be used to utilise sockets, including *java.net.Socket* and *java.net.ServerSocket* that represent client and server processes respectively. These classes act as an interface to underlying platform-dependent socket implementations, and thus allow developers to write socket applications in a platform-neutral manner

Once a socket connection has been established, processes may use I/O methods to read and write data, to and from sockets. In Java, developers may use the *Java IO* or the *Java NIO* packages. Java IO is the original built-in library providing standard IO functions. Java NIO, which was added in J2SE 1.4, added support for a number of additional features, including the channels abstraction, and non-blocking I/O functions. IO in Java was further enhanced with the addition of NIO2 in Java SE 7.

None of the standard Java libraries support the utilisation of high-performance networks such as Infiniband, Myrinet or Scalable Coherent Interface (SCI). It is possible to run standard Java sockets over these networks through the use of TCP/IP emulation libraries such as IPoMX (IP over the MX library for Myrinet) and IPoIB (IP over Infiniband), although the performance does not match the use of native communication libraries, which are designed to take maximum advantage of high-speed networks.

Direct implementations of socket libraries on top of high-performance communication libraries avoid the overhead associated with TCP/IP emulation. Socket-MX is a low-latency sockets implementation on top of MX, which provides high performance on the Myrinet network.

In [Taboada et al., 2008], an alternative Java socket library called Java Fast Sockets (JFS) is presented. This library provides high-performance socket communication on a number of networks. They first analysed the steps involved in sending and receiving data using standard Java sockets, and they identified a number of possible optimisations, which they then implemented in JFS. Amongst the inefficiencies identified with the use of standard Java sockets were: the overhead associated with the serialisation process, and the number of copy operations which occur as data is prepared to be sent, and after it has been received. JFS manages to eliminate the copy operations by allowing native libraries to access the data by a pointer, through the use of JNI. Additionally, JFS can take advantage of shared-memory environments, through the use of non-TCP/IP sockets and direct memory transfers, avoiding the costly TCP/IP protocol used by standard Java sockets even in shared-memory environments. JFS provides support for Ethernet, Infiniband, Myrinet and SCI networks. On the latter three, JFS makes use of native socket libraries, rather than relying on TCP/IP emulation. As JFS implements the standard socket API, it is a relatively effortless task to switch from standard Java sockets to JFS. JFS can also be used as the sockets layer, underneath a higher-level middleware communication system, such as an RMI implementation or a message-passing library.

5.2.1.4 Remote Method Invocation

RMI is a framework built into the Java platform that allows for methods in Java objects, which exist in remote JVMs, to be called. RMI may be used in combination with JNI to interface with non-Java legacy systems. A number of communication middleware systems have been built on top of the functionality provided by RMI.

In [Taboada et al., 2007], an improved RMI implementation is described. The solution presents users with the same API as the stand RMI implementation, thus existing applications which use the standard RMI implementation could easily switch to this implementation. Its improvements include the use of a high-performance sockets layer, called JFS (described in the previous subsection), sending much less versioning information, and making use of native array serialisation. This implementation does assume that the target environment does not involve different JVM versions, as well as the presence of a shared filesystem.

ProActive [Amedro et al., 2009] is a Java-based GRID middleware that attempts to meet many of the needs of modern parallel applications (see Section 4.3 for a general description of ProActive). Its default communication layer is built on top of the RMI protocol. An analysis of the performance of ProActive, together with the results of some benchmarks are given in [Amedro et al., 2009]. This study found that, in respect of data-intensive applications, and when using RMI as the transport layer, the speed and scalability obtainable with Proactive was lower than MPI. However this work did find that substituting the default sockets used by the RMI implementation with a higher-performing communication sockets implementation, such as JFS, the performance of ProActive could be improved significantly.

5.2.1.5 Message passing in Java

In this work, it was decided to focus efforts on the MPJ approach as it has been reported to provide the highest performance for HPC applications on low-latency networks, such as Infiniband and Myrinet [Taboada et al., 2011a]. MPJ may be implemented in a number of ways, including through the use of RMI, using JNI to call an underlying native message-passing library, or through the use of Java sockets. Each approach has advantages and disadvantages in the areas of efficiency, portability and complexity. The use of RMI results in a totally Java-based solution, thus ensuring portability. However, it may not be the most efficient solution in the presence of any high-performance communication networks, as RMI is unable to take advantage of such infrastructure. The use of JNI allows for the efficient use of high-performance networks using native libraries, but it introduces portability issues. Finally, the implementation of MPI using Java sockets, involves a *from the ground up* development of a large set of functionality, therefore it requires a considerable development effort, however the performance is normally better than a RMI-based system.

The highest priority for application developers and users may vary from one case to another. For example, in some cases, the highest priority might be performance, while in other cases the highest priority might be portability. Additionally, when high performance is important, in some cases, bandwidth might be more important than latency, but in other cases, low-latency might be of primary importance. When designing or selecting a data communication system, it would be beneficial to first understand what are the primary needs of the application.

There are two main specifications of MPJ: the mpiJava API [Carpenter et al., 1999] and the MPJ API [Carpenter et al., 2000]. Both specifications were defined by largely the same small group of researchers working within the JGF [Thiruvathukal, 1998], and are quite similar. The differences between them are mainly related to different naming conventions. The mpiJava specification (which is based on the MPI 1.1 specification), is followed by all except one of the currently available MPJ implementations, and is effectively the *de facto* standard MPJ specification.

Several message-passing libraries have been developed following a modular component design [Baker et al., 2005] [Taboada et al., 2011b]. Following this approach, the entire communi-

cation system is composed of a number of layers, at each layer there might be several alternative components available for use, these components are generally known as *devices*. This design allows users to select particular devices at particular layers in order to meet their specific needs. For example, there might be a *pure Java* device, and a device that accesses a native communication library through JNI, available at a particular layer. One offering better portability, and the other offering better performance. Native MPI implementations such as MPICH and OpenMPI, as well as MPJ implementations such as FastMPJ and MPJ Express all contain a low-level communication device layer, where new communication devices may be *plugged in* as support for new networks is added or improved.

Typically, low-level communication devices, which *sit* directly on top of an underlying native communication library offer a simple API to higher-level devices within the system. Such low-level APIs typically contain simple *send* and *receive* functions. Higher-level devices within the system can then build on these relatively low-level functions to implement higher-levels of functionality such as MPI collective functions. This approach, of providing a limited set of simple and extendible functions in low-level devices allows for new low-level devices to be implemented with relatively little effort, and higher-level communication functions such as MPI collective functions can instantly make use of new devices — presuming that new devices implement the API that is expected by high-level devices.

5.2.2 MPJ implementations

There have been many projects which have implemented some form of MPJ, eleven projects are cited in [Taboada et al., 2003], and seven more are cited in [Taboada et al., 2011a]. Many of these projects were quite short-lived, and are no longer being developed, or supported. The number of implementations of MPJ, and the lack of a coordinated effort by the community, has perhaps contributed to the limited adoption of the use of Java in HPC.

The mpiJava library [Baker et al., 1999] supports the mpiJava specification. It was developed by the same team as the specification itself, and for several years, it was the most widely used MPJ implementation. It is implemented as a relatively thin wrapper layer, on top of an underlying native MPI library, through the use of JNI. It offers high performance, although its reliance on JNI introduces portability issues. Additionally, mpiJava is not thread-safe, thus is unable to take advantage of multi-core systems through multithreading. Several of those involved in mpiJava were later involved in the development of the MPJ Express implementation (see Section 5.2.2.1).

MPJ/Ibis [Van Nieuwpoort et al., 2005] is a message-passing implementation, supporting the JGF MPJ specification, and built on top of the Ibis framework (see Section 4.3). MPJ/Ibis aims to achieve portability and high-performance by being totally Java-based, thus ensuring that it maintains the portability of Java, but also giving applications the choice of using optimised but non-standard solutions, for increased performance in special cases. For example, MPJ/Ibis allows for the efficient use of the high-performance Myrinet network — just as MPJ Express does with its *mxdev device*. There are two low-level communication devices available: TCPIbis, based on Java IO sockets; and NIOIbis, based on Java NIO sockets. MPJ/Ibis does not provide a device specifically for shared-memory environments, thus it cannot take maximum advantage of typical multi-core, shared-memory computing nodes. Also, the MPJ/Ibis implementation is not thread-safe, and its Myrinet support is based on the GM library, which provides poorer performance than the MX library.

In this work, we concentrated our investigations in the MPJ Express and FastMPJ projects. These are two of the most recently developed message-passing libraries, and both are being actively developed and supported. Additionally, both have enjoyed a high degree of usage amongst the development community. They both provide good performance and portability, also both implement the same specification of MPJ — `mpiJava` 1.2 [Carpenter et al., 1999] — making it easy to swap between these implementations. These two implementations: MPJ Express and FastMPJ are briefly described in the next two sections.

5.2.2.1 MPJ Express

MPJ Express [Baker et al., 2006] is an open source, free software, message-passing implementation, that follows a pluggable device design and is totally written in Java. At the lowest level in the MPJ Express stack of layers is the *xdev* device. Currently, MPJ Express includes three implementations of the *xdev* device: *niodev* over Java NIO sockets; *mxdev* which uses JNI to communicate with the MX library, thereby supporting the Myrinet network; and *smpdev* — for use in shared-memory environments. MPJ Express is thread-safe, supporting the highest level of multithreading support defined in the MPI specification — *MPI_THREAD_MULTIPLE.*

MPJ Express includes an intermediate buffering layer between low-level data transfer routines, and high-level message-passing functions. This buffering level contains an extensible device known as *mpjbuf* [Baker et al., 2007]. Higher-level devices access *mpjbuf* through an interface, and therefore there is a separation between implementation details and the functionality provided. The *mpjbuf* device makes use of the *direct byte buffer* classes available in the Java NIO package (see Section 3.2.7 for a description of Java NIO). One of the features of these classes is that they allow data to be stored outside of the Java heap. This can be exploited to provide fast data transfers, as high-performance network libraries can be instructed to send the data stored in such buffers without having to go through the JVM to access the data. In this way, data in such buffers may be sent using a mechanism similar to Direct Memory Access (DMA) transfers. Using this technique avoids one of the main sources of overhead associated with the use of JNI — the copying of data between the JVM and the OS. This buffering layer also allows MPJ Express to support derived data types, they may be packed and unpacked at this level. MPJ Express also supports the communication of serialised Java objects. Their tests on a Myrinet network showed that combining the use of direct byte buffers with JNI to transfer data incurs virtually no overhead, albeit, the packing and unpacking of data into the buffers does lead to an overhead.

In [Shafi et al., 2009], a description is given of how MPI and OpenMP-like functionality may be combined within MPJ Express. They define *nested parallelism* to mean the use of multithreading within a single MPI process, and describe how Java threads, or an OpenMP-like implementation in Java such as JOMP (see Section 5.2.1.2), may be used in combination with MPJ Express. They found that MPJ Express and JOMP could be integrated with relatively little effort, although they had to update some of the JOMP code to make use of the Java Concurrency Framework — which was not available when JOMP had been first developed. Their tests showed that in a typical cluster, composed of multi-core computing nodes, the combination of MPJ Express MPI processes with JOMP leads to significant improvements over a system purely based on MPJ Express.

Two implementations of the *xdev* device, intended for use in multi-core shared-memory environments, are presented in [Shafi and Manzoor, 2009]: one is built using Java threads, while the other one uses the UNIX System V (SysV) IPC library through JNI. The authors compared the

performance of these two devices against the shared-memory device of mpiJava. They found that for small message sizes (less than 2 kBs), mpiJava provided the best performance, but for larger messages, the device based on Java threads performs best.

5.2.2.2 FastMPJ

FastMPJ [Expósito et al., 2012] is a high-performance, and scalable message-passing implementation. A prototype version of FastMPJ was known as F-MPJ, however, as of 2012, the system has been known as FastMPJ. FastMPJ supports several high-speed networks, including GbE, Infiniband and Myrinet. FastMPJ supports the widely adopted mpiJava API specification for message-passing in Java.

FastMPJ follows the pluggable device design, with a number of devices available at different levels. Most significantly, FastMPJ includes the *xxdev* device layer. Devices in this layer must implement the *xxdev* API, which is an extension of the *xdev* API present in MPJ Express (see Section 5.2.2.1). The *xxdev* API, unlike the *xdev* API, supports the direct transmission of any serialisable objects, without the use of buffering. The *xxdev* API has been designed to be simple and concise. It supports both blocking and non-blocking point-to-point communication. Higher-level functions, such as collective communication functions, are implemented at higher levels within the FastMPJ stack. The concise nature of the *xxdev* API means that new *xxdex* devices can be implemented with relatively little effort, and plugged into the application without necessitating other changes within the application.

Currently, there are six *xxdev* devices available: *iodev* and *niodev* both use TCP/IP to utilise the Ethernet network, using the Java IO and Java NIO libraries respectively; *mxdev* uses native libraries (either MX or OpenMX) to fully utilise the Myrinet network; both *ibvdev* and *psmdev* support the Infiniband network, they make use of the native libraries *Infiniband Verbs* and *InfiniPath PSM* receptively, through the use of JNI; finally, *smdev* is built on top of the Java thread library and provides high-performance on shared-memory environments. Where possible, these protocols follow a zero-copy approach, thereby avoiding the overhead associated with data buffering.

FastMPJ provides scalable collective functions, in fact, it actually provides a number of alternative algorithms for each collective primitive. FastMPJ can automatically select the algorithm likely to give the best performance at runtime, based on message size and the characteristics of the HPC environment. Its primitives are therefore said to be *topology aware*.

They present an extensive set of tests which involved comparing the performance of FastMPJ with that of a number of MPI libraries, on a number of HPC environments, over a number of high-speed networks, as well as using shared-memory systems. Both point-to-point and collective primitives were tested. These revealed that FastMPJ rivals the performance of MPI libraries; and in come cases, especially point-to-point primitives involving *smpdev*; outperforms them.

5.3 Communication middleware requirements

The availability of high-performance communication middleware is becoming increasingly important in HPC, given the trends of increasing numbers of cores per computing nodes, and the increasing size of the datasets that must be handled (see Section 2.1 for a description of the current trends in HPC). Despite the availability of high-performance storage devices and

networks, the accessing of data held in shared storage devices can act as a bottleneck in the processing of data, especially if there are a large number of processes accessing the data and if it is accessed repeatedly.

The objective of this work on data communication was to identify (or implement or extend) a communication middleware that provides high-performance and portability between environments, and also enabling high productivity amongst application developers, through the provision of a high-level API. In the following subsections we describe the data communications requirements of applications at the DPCB, including some scalability tests of initial data management approaches.

5.3.1 DPAC systems communication requirements at the DPCB

A particular objective of this work was to understand the communication requirements of DPAC systems running at the DPCB (see Section 2.3.4 for a description of DPAC systems at the DPCB), to understand the primary bottlenecks affecting the flow of data through the system, and to devise a solution that maximises performance. A concern for these systems was that data access and data communication could potentially become a bottleneck in the overall processing, as these systems (particularly IDU) involve a relatively large amount of data being processed by many processes running in many computing nodes. Therefore, performance of the communication middleware is a high priority for these systems, as is devising approaches for minimising the number of concurrent accesses of the shared storage device.

Given that the DPCB will likely go through a number of upgrades during the lifetime of the DPAC systems, it is important that the communication middleware system would not be tied to any particular hardware environment, or be dependant on libraries that are only available for some environments.

5.3.2 Scalability tests of initial IDU approach with GPFS

An initial version of the IDU application involved all processes reading their input data from the GPFS shared storage device. In order to determine how well this approach would scale to a large number of worker processes, running in a large number of computing nodes, a number of scalability tests were performed. These tests were executed on the MareNostrum II supercomputer, the specification of which is given in Section 2.3.3. The input files for these tests were of equal size and represented a certain amount of processing, the IDU version used for these tests was IDU 6.0.

The dataset used during these tests covers two hours starting on day eleven of the Gaia mission (that is, 2012.0 + 11.0). The AstroObservation files, which are the most relevant input to the IDU application, amount to 1.23 GB. They were arranged in twenty files, each covering 100 to 900 seconds of mission data, holding 140,000 to 375,000 observations, and occupying 46 MB to 91 MB in the Gbin format. The total number of observations was 4,802,990, thus leading to an average rate of about 57 million observations per day — roughly the expected average during the real mission.

The tables of results given below show the parameters that were varied between each test. *Job files* are simply a file describing a small *chunk* of IDU processing work. An execution of IDU is composed of many job files (see Section 4.1.1.1 for a description of IDU job files). We also give the number of nodes that were used in each test, as well as the number of tasks that were

executed per node. In this context, the term *task* is a synonym for process (*task* is the term used in a SLURM job file when specifying the number of process that should be executed). In all tests, the number of tasks per node was set to 4.

5.3.2.1 Increasing computing nodes and job files

Test	No. job files	No. nodes	No. tasks	No. tasks per node	No. job files per node	Time (s)
1.1	20	1	4	4	20	1714
1.2	40	2	8	4	20	1708
1.3	80	4	16	4	20	1714
1.4	160	8	32	4	20	1786
1.5	320	16	64	4	20	1948
1.6	640	32	128	4	20	2174

Table 5.1: Scaling the number of computing nodes and job files.

In the first set of tests, the number of job files processed by any individual computing node was kept constant, but the number of computing nodes being utilised was increased. If the system was scaling perfectly, then each of these tests would of taken roughly the same length of time, since in each case, the maximum amount of processing that must be performed by any particular node did not increase, and all the processing was performed concurrently.

As we can see in the results given in Table 5.1 and illustrated in Figure 5.1, this approach scales quite well up to 8 computing nodes (32 cores). Beyond that, as the number of computing nodes and job files increases, the time taken also increases. Taking the shortest time as a reference result (1708 seconds), the increase with 8 nodes is a modest (and acceptable) 4.5%, while for 16 nodes it rises to 14% and for 32 computing nodes it reaches 27%. We can thus conclude that IDU 6.0, with a straightforward deployment, scales acceptably for up to 8 nodes (32 cores), although 16 computing nodes can also be used with a modest penalty — largely compensated by the reduction in the processing time, as we will see hereafter.

5.3.2.2 Increasing No. job files, constant No. nodes

In the second set of tests, the number of nodes was kept constant — 4 computing nodes were used in each test, but the number of job files that were processed in each node was increased. The results of these tests are given in Table 5.2 and illustrated in Figure 5.2. As is shown, the time taken for each test increases linearly up to 240 job files. The time taken to process any individual job file is not dependent on the total number of job files, and the average time required to process a single job file remains in the range 20 — 22 seconds. Thus, the approach taken in these tests can be considered to provide good scalability. Following this approach — keeping the number of computing nodes used constant — allows for accurate estimations of the execution time required to process a given number of job files.

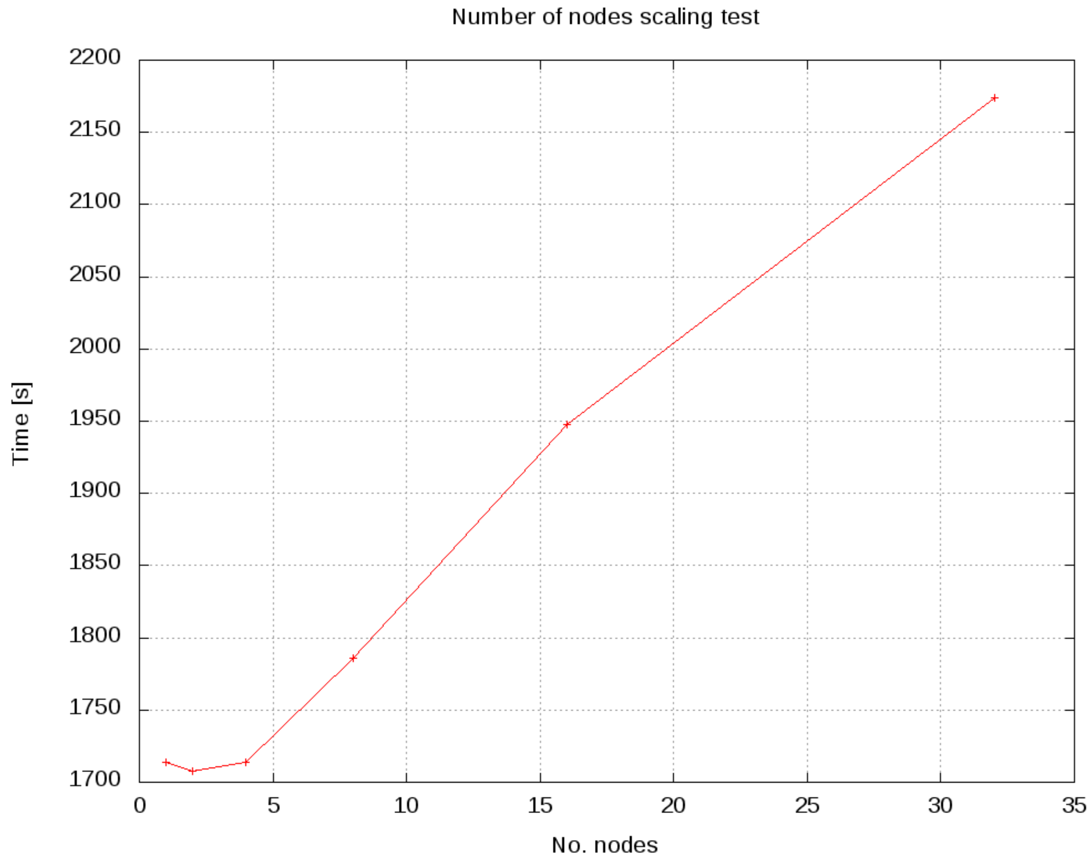

Figure 5.1: Scaling the number of computing nodes and job files.

Test	No. job files	No. nodes	No. tasks	No. tasks per node	No. job files per node	Time (s)
2.1	20	4	16	4	5	554
2.2	40	4	16	4	10	914
2.3	60	4	16	4	15	1350
2.4	80	4	16	4	20	1696
2.5	100	4	16	4	25	2128
2.6	120	4	16	4	30	2516
2.7	140	4	16	4	35	2941
2.8	160	4	16	4	40	3294
2.9	180	4	16	4	45	3902
2.10	200	4	16	4	50	4129
2.11	220	4	16	4	55	4540
2.12	240	4	16	4	60	4871

Table 5.2: Scaling the number of job files (constant number of computing nodes in all tests).

5.3.2.3 Increasing nodes, constant job files

In the third set of scalability tests, the number of job files was kept constant, but the number of computing nodes sharing the burden of processing these job files was increased. These tests provide a good insight into the actual scalability of this early version of IDU. As shown in

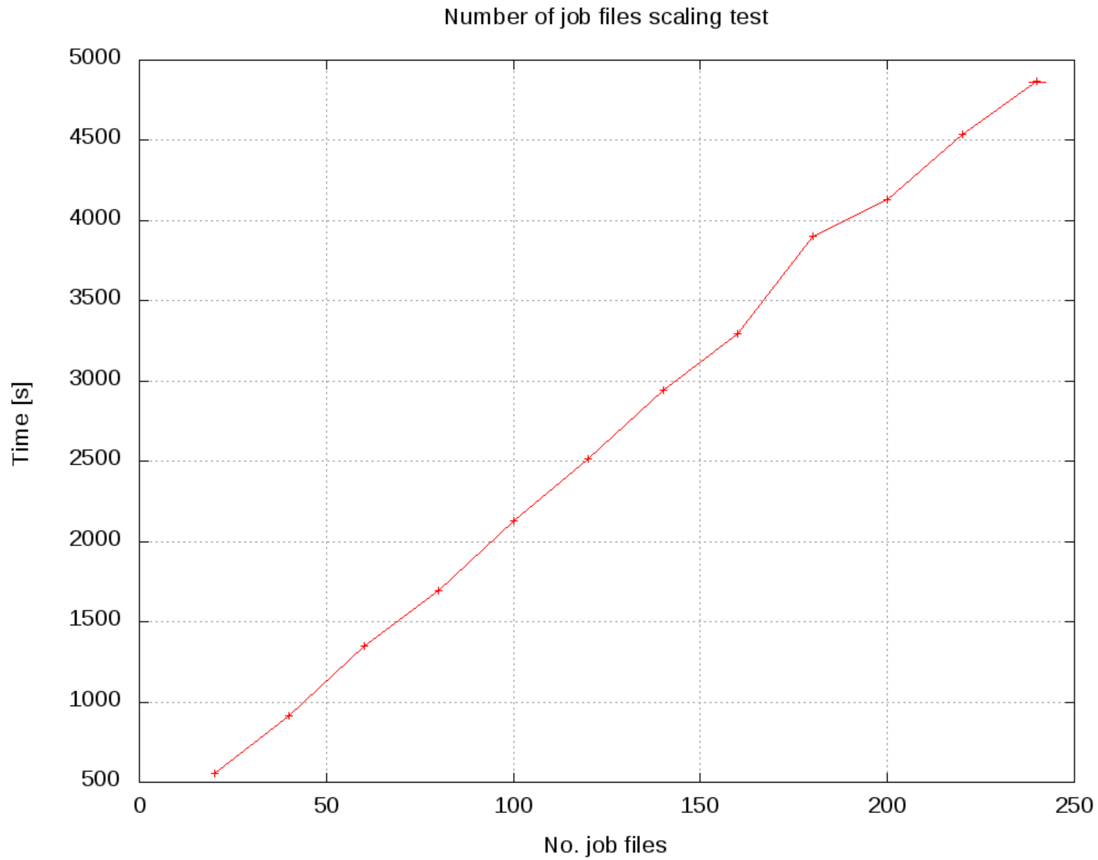

Figure 5.2: Scaling the number of job files.

Table 5.3 and in Figure 5.3, the system scales fairly well up to about eight nodes — as already seen in the first set of tests. The speed-up factors are quite good (7.4 for 8 nodes). When moving to 16 nodes the scalability is not as good, the speed-up factor being 13.0.

These tests revealed that it is possible to safely use up to eight concurrent processes accessing the IDU input data stored in the GPFS, without incurring a significant drop in I/O performance. This observation is useful when selecting an optimal configuration for the data processing framework that will be used with the IDU application, when executing it in an environment like the one used for these tests

Given that the NG framework will be used for managing the execution of IDU (see Chapter 4 for a description of the proposed execution framework), the scalability of multiple processes accessing the GPFS observed in these tests suggests that the creation of eight NGs would result in an optimal NG configuration. One computing node within each of these NGs would be designated as the NGM, and would be responsible for retrieving data from the GPFS. The required amount of worker nodes could be assigned to each NG, and data would be transferred from the NGM, to the worker nodes in its group, over the Myrinet network. For example, 16 worker nodes could be assigned to each NG, thus efficiently delivering the power of 128 computing nodes (512 cores) without any significant I/O bottleneck.

Test	No. job files	No. nodes	No. tasks	No. tasks per node	No. job files per node	Time (s)	Speedup factor
3.1	240	1	4	4	240	19084	N/A
3.2	240	2	8	4	120	9504	2.0
3.3	240	4	16	4	60	4932	3.9
3.4	240	8	32	4	30	2588	7.4
3.5	240	16	64	4	15	1465	13.0

Table 5.3: Scaling the number of nodes (constant number of job files).

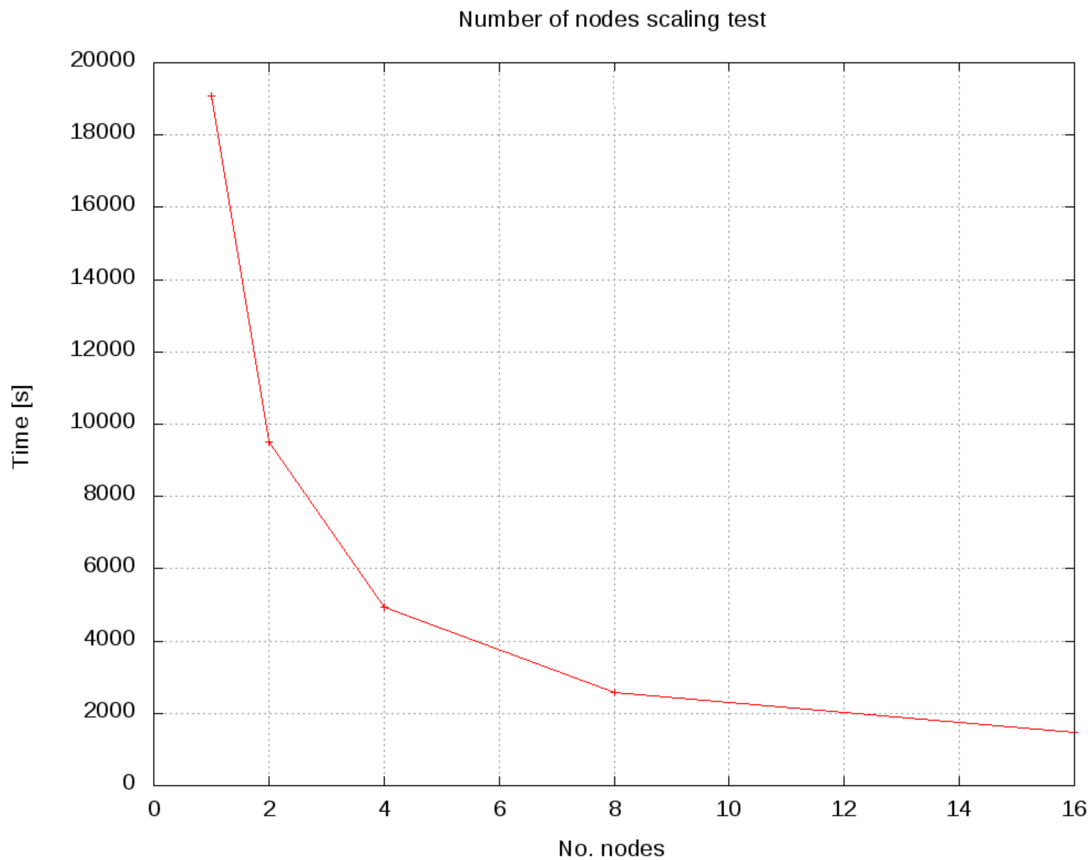

Figure 5.3: Processing time of an IDU prototype over a fixed amount of data, when using 1 to 16 nodes, and accessing the GPFS disk directly.

We must, however, consider the network capacities before taking any assumption on the possible architecture. Eight nodes have been able to process 240 AstroObservation files (14.8 GB) in 2588 seconds. That means a reading throughput of about 5.8 MB/s — plus a similar write throughput, as the AstroElementaries have similar sizes than AstroObservations. Theoretically, the GPFS, if adequately used, could deliver up to 100 MB/s or even more. Thus, using about 10% of the GPFS just for IDU seems a reasonable and safe option, considering the large amount of other users and projects running at MareNostrum. We could even consider using 16 nodes accessing the GPFS, but not more than that. Thus, the IDU architecture should be based on eight (at most 16) NGs.

5.4 Comparison of MPJ middleware

Amongst the available MPJ implementations, MPJ Express and FastMPJ were, from the perspective of this work, the most attractive options. These projects are both actively being developed and supported, and they offer good performance, and portability. They both support the Ethernet, Myrinet and Infiniband networks. A number of tests were performed to assess and compare the performance of MPJ Express and FastMPJ in the DPCB environment. Additionally, and for the purposes of comparison, MX (see Section 5.1.1) and Intel MPI Benchmark (IMB) were also tested. IMB is a C-based set of benchmarks, developed by Intel, for measuring the performance of standard MPI functions.

Therefore, the four tested systems were:

- MPJ Express (Version 0.36)
- FastMPJ (Version beta — prototype version of FastMPJ)
- MX utilities (Version 1.2.7)
- IMB (uses MPICH-MX) (Version 3.2)

These applications were tested using a Pingpong and a Broadcast benchmark. Pingpong is a simple point-to-point communication test. Broadcast is an MPI collective primitive that allows one process to send a message to a group of processes. These tests were also executed on the MareNostrum II supercomputer (see Section 2.3.3).

A summary of the results of these tests are given in Tables 5.4 and 5.5, while the full results are given in Appendix B. These showed that FastMPJ performs very well in the target environment, in particular, it offers very low latency for short messages and good bandwidth with longer messages. The results of FastMPJ, MX and IMB are almost identical in the Pingpong test. MPJ Express however shows roughly 35% higher latency and 25% less bandwidth.

Application	**Latency**	**Bandwidth**
MPJ Express	23.193 μms	180.841 (MB/s)
FastMPJ	17.038 μms	246.162 (MB/s)
MX utilities	16.980 μms	247.012 (MB/s)
IMB (MPICH-MX)	16.975 μms	247.00 (MB/s)

Table 5.4: Pingpong tests (point-to-point communications).

Data	**MPJ Express**		**FastMPJ**		**IMB (MPICH-MX)**	
	Latency	**BW**	**Latency**	**BW**	**Latency**	**BW**
2MB	43.6 μms	48.1 MB/s	25.8 μms	81.3 MB/s	21.9 μms	96.0 MB/s
4MB	87.5 μms	47.9 MB/s	52.4 μms	80.0 MB/s	42.7 μms	98.2 MB/s

Table 5.5: Broadcast test - 8 nodes, 1 process per node.

5.5 MPJ-Cache

In this section we describe MPJ-Cache, our new communication middleware application. Amongst the desirable features of a communication middleware are high performance, scalability, high-productivity, portability and extendibility. The extensive work on message-passing described in

the previous sections has resulted in high performance, and scalable communication libraries. However, the use of message-passing does require a careful programming effort, and for developers unfamiliar with the message-passing approach, it can be quite challenging.

To address these issues, a new high-level API of communication functionality was designed that enables high productivity amongst developers, and requires no previous message-passing experience. This system encapsulates sophisticated and commonly required communication functionality within single methods, for example, allowing for the seamless distribution of groups of input files amongst a group of computing nodes.

In addition to the communication functionality, MPJ-Cache possesses intelligent caching and prefetching features. If the order in which input data must be processed is known, then the next set of input data may be prepared for its distribution to a worker node before it is actually required. In MPJ-Cache, a reduced set of computing nodes, known as the server nodes, retrieve data from the shared storage device, while the majority of the available nodes, known as client nodes, are busy processing data. Server nodes can be configured to attempt to always have the next set of input data ready, stored in its memory cache for best performance.

Using a reduced set of nodes to access shared storage device reduces communication bottlenecks, which can be created if many concurrent processes attempt to access such resources. Also this approach reduces idle time amongst the data storage devices, as the server processes can be continuously preparing its cache of data, retrieving the next set of data, and optimising the manner in which data has been stored, in preparation for the next set of requests.

MPJ-Cache can be used not only to distribute data, but also to distribute instructions to client nodes. These instructions can define, for example, which class should be used to process the data, and any arguments and configuration parameters that should be used.

Conceptually, MPJ-Cache has two relatively distinct components within it, namely the communication layer and the data cache. These components are described in the following sections, and are illustrated in Figure 5.4. How MPJ-Cache may be integrated into an overall execution framework is described in Section 4.4.

5.5.1 The communication layer

5.5.1.1 Message types

A number of message types are supported, Table 5.6 lists these message types and gives the meaning of each message type. These messages are used as the building blocks of the communication protocols defined in the following section. These protocols define the sequences of messages that are exchanged between clients and servers while operating in particular modes of operation. It should be noted that additional communication protocols could be constructed using the defined messages.

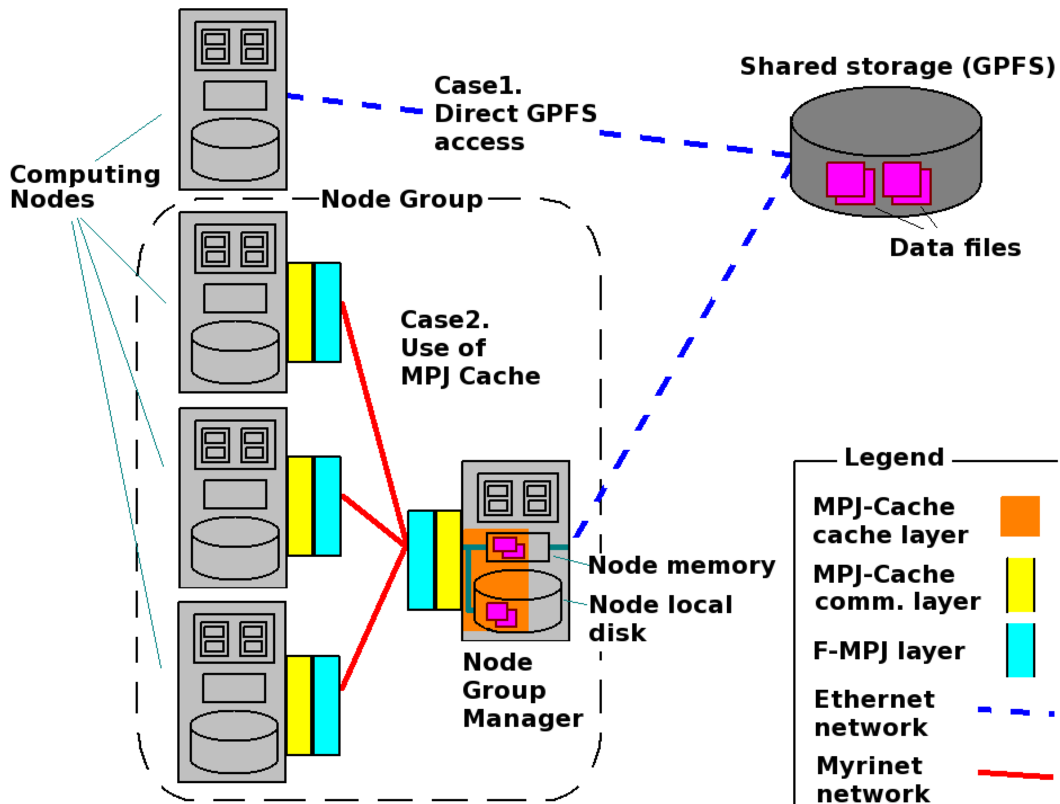

Figure 5.4: MPJ-Cache components and test setup.

Message type	Message Meaning
1	Sent by clients to servers to ask what is the size of a data file
2	Sent by clients to servers to request a data file
3	Sent by clients to servers to ask for the number of files that are available
4	Sent by clients to servers to request a list of the available files
5	Sent by clients to servers to tell the server that the client is currently idle
6	Sent by servers to clients in response to a *message type 5*, to tell the client that there is still work to be done, and that the server is going to send the client a single data file, this message contains a field indicating the size of the file that will be sent. Once a client receives a *message type 6*, it expects that the next message from the server will contain the name of the file that will be sent
7	Sent by servers to clients to inform the clients that all of the work is done, and that the server will not be sending any more data, therefore the clients can stop waiting for messages from the server
8	Sent by servers to clients to inform them that the server is going to send the client a FileGroup. The message also contains a field which indicates the number of files in the FileGroup. This allows the client to prepare to receive the FileGroup
9	Sent by servers to clients to inform the client of the size of each of the files in the FileGroup that the server is about to send to the client
10	Sent by servers to clients to inform them of the names of each of the files in the FileGroup that the server is about to send to the client

Table 5.6: MPJ-Cache message types.

5.5.1.2 Communications protocols

Two basic distribution modes are supported by MPJ-Cache, these are the *push* mode and the *pull* mode. When operating in these modes, clients and servers follow particular communication protocols, which involve the sending of particular messages in a particular order.

Basic pull protocol

The *basic pull protocol*, as illustrated in Figure 5.5, follows the typical client-server model, where clients make requests from servers. Following this mode, servers typically do not know which files will by requested by the clients, or how many requests they will receive, and they must operate indefinitely until terminated or until a given number of requests have been processed.

Basic push protocol

The *basic push protocol*, as illustrated in Figure 5.6, involves servers sending, or *pushing*, data to clients. In this mode, the servers must either know in advance the order in which files must be sent to clients, or they must decide the order. In either case, the servers could be described as *intelligent*, as they are pushing particular sets of data to clients, while the clients could be described as *dumb*, as they simply accept whatever is pushed to them.

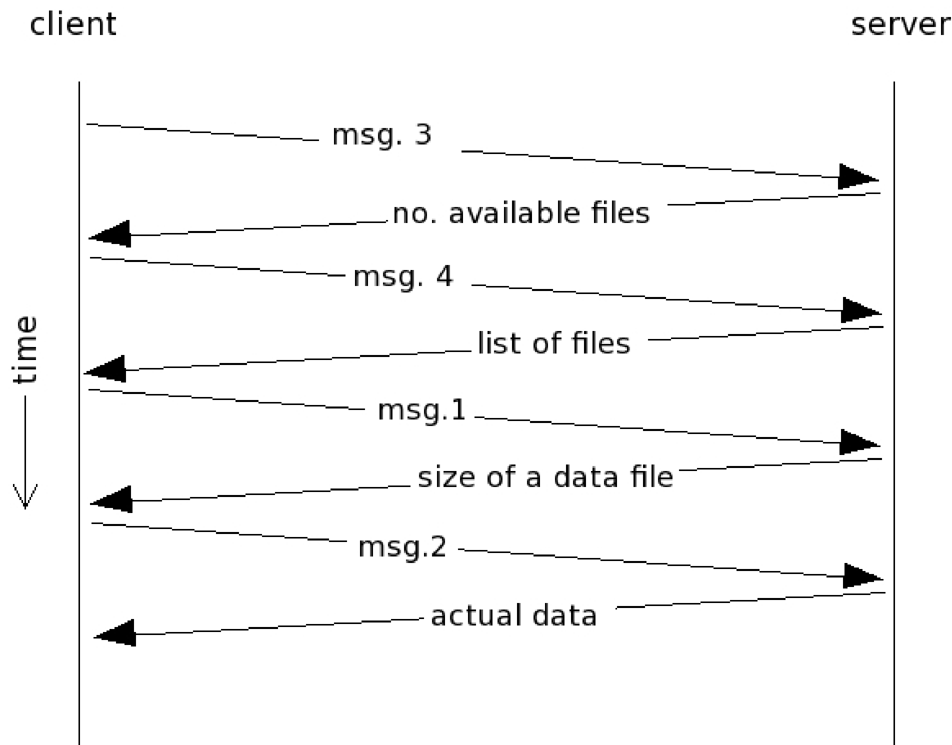

Figure 5.5: Client-server communication protocol following the basic *pull* protocol.

5.5.1.3 Distribution of work

These protocols can be used for the distribution of work amongst clients, as well as the distribution of data, through the use of *job files*. Job files encapsulate a *chunk* of work that needs to be executed. They define the application be that should be executed, any application arguments,

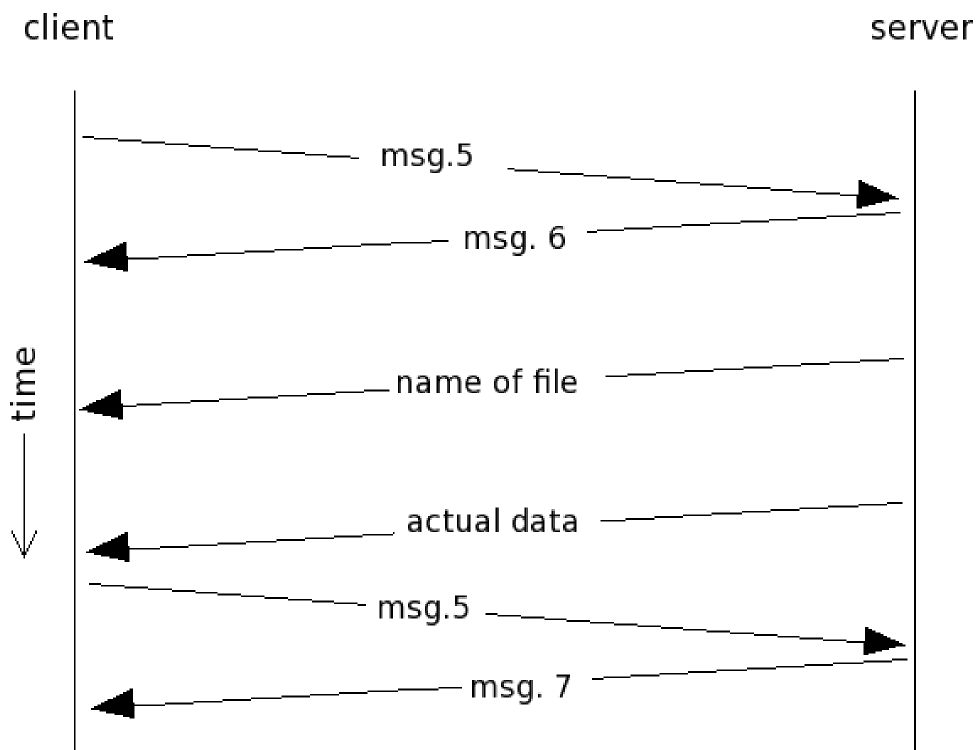

Figure 5.6: Client-server communication protocol following the basic *push* protocol.

as well as defining the input data needed for the execution. The distribution of work using *job files* is discussed further in the Section 5.5.1.5 where the concept of FileGroups is introduced.

5.5.1.4 Client states

From the perspective of MPJ-Cache, when operating in the *push* mode, client applications may be in one of two possible states: busy or idle. When a client is idle, this means that the client is currently not performing any processing and that it is willing to receive data (or *job files* specifying work to be executed) from a server. Servers maintain a list of the currently idle clients, and while there are still job files to be distributed, they will attempt to distribute these amongst the idle clients.

Initially when an application starts, all the clients are in the idle state. Once a server has sent data (or a *job file*) to an idle client, it will remove that client from its list of idle clients, and it will only add that client to the idle list again once it receives an idle message from that client.

5.5.1.5 FileGroups

Often applications require a number of input files. When such applications are running in an HPC environment such as a computer cluster, all of the input files must be distributed to each of the computing nodes where the application is running. If there are several instances of the application running in a single node, then several sets of input files must be distributed to each node. The distribution of sets of data files amongst groups of nodes is a commonly required piece of functionality.

MPJ-Cache supports the grouping of files into a construct called a *FileGroup*. FileGroups can be specified in an XML-based configuration file, an example is shown in Figure 5.8. This file

defines a number of FileGroups, and their location on a storage device. Each FileGroup may contain an arbitrary number of files. This file is parsed by MPJ-Cache servers during startup, and the servers take the responsibility of distributing each FileGroups to clients.

Servers attempt to distribute FileGroups to the idle clients until all the FileGroups have been distributed. If there are still FileGroups to be distributed, but there are no idle clients, then servers will attempt to perform some cache maintenance, and prefetching of data, in anticipation of the distribution of the remaining FileGroups. The communication protocol used, when operating in the *push* mode of operations, and using FileGroups, is shown in Figure 5.7.

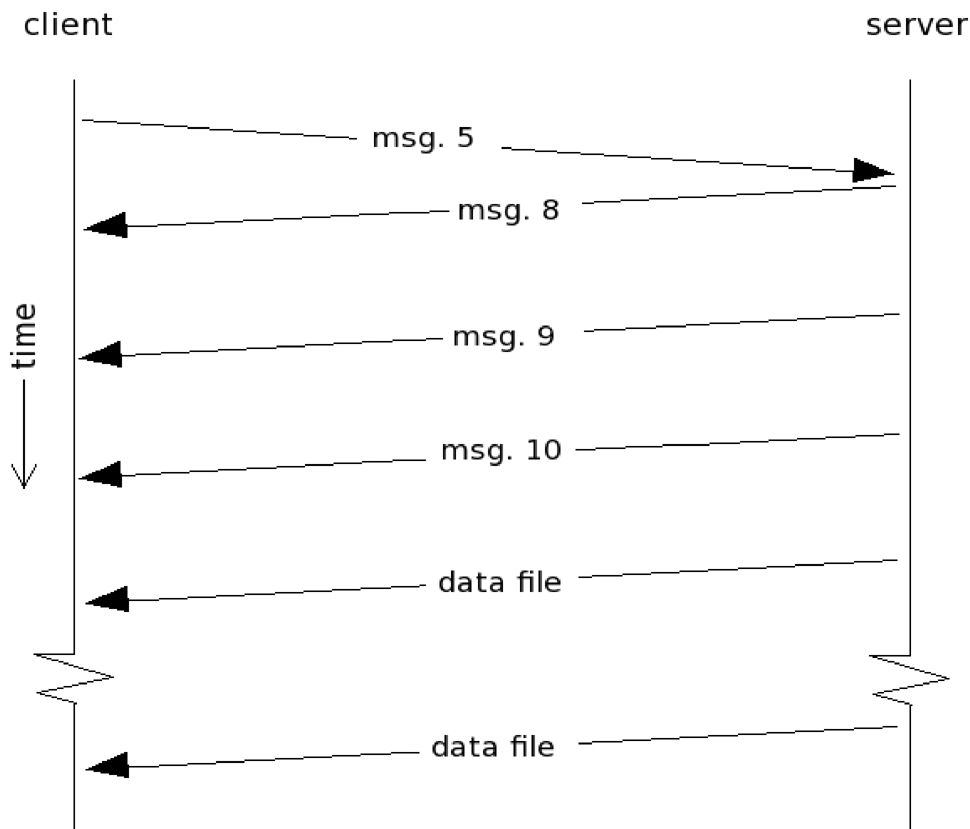

Figure 5.7: The protocol used when operating in the *push* mode of operation and using FileGroups.

5.5.1.6 MPJ-Cache API

The objective of the MPJ-Cache communication layer is to provide developers (who may be unfamiliar with message-passing) with a high-level API of data communication methods useful in HPC environments, while making best use of the available communication resources such as any high-speed, low-latency networks which might be present.

MPJ-Cache builds upon an underlying implementation of MPJ to perform inter-process communication. Implementations of MPJ, such as MPJ Express and FastMPJ implement an API of methods, such as `MPI_Sendrecv()`. MPJ-Cache builds upon that API, and offers its own API of methods which are at a higher level of abstraction.

The class *gaia.dpcb.tools.infra.mpjcache.Client* exposes the API given in Table 5.7.

```xml
<fileGroups>
    <fileGroupsRootDirectory>/common/dds/global/in</fileGroupsRootDirectory>
    <fileGroup>
        <file>ao/ao.0990.43212-0991.43200.R1.gbin</file>
        <file>tv/tv.0990.43212-0991.43200.R1.gbin</file>
        <file>isrc/E0/isrc.E00222.gbin</file>
        <file>isrc/E0/isrc.E02000.gbin</file>
        <file>isrc/E0/isrc.E02001.gbin</file>
        <file>isrc/E0/isrc.E02002.gbin</file>
    </fileGroup>
    <fileGroup>
        <file>ao/ao.0990.43212-0991.43200.R2.gbin</file>
        <file>tv/tv.0990.43212-0991.43200.R2.gbin</file>
        <file>isrc/E0/isrc.E00222.gbin</file>
        <file>isrc/E0/isrc.E02000.gbin</file>
        <file>isrc/E0/isrc.E02001.gbin</file>
        <file>isrc/E0/isrc.E02002.gbin</file>
        <file>isrc/E0/isrc.E02003.gbin</file>
        <file>isrc/E0/isrc.E02012.gbin</file>
        <file>isrc/E0/isrc.E02020.gbin</file>
        <file>isrc/E0/isrc.E02021.gbin</file>
    </fileGroup>
    <fileGroup>
        <file>ao/ao.0990.43212-0991.43200.R3.gbin</file>
        <file>tv/tv.0990.43212-0991.43200.R3.gbin</file>
        <file>isrc/E0/isrc.E00222.gbin</file>
        <file>isrc/E0/isrc.E02000.gbin</file>
    </fileGroup>
</fileGroups>
```

Figure 5.8: FileGroups configuration file — configured to distribute three FileGroups, consisting of different numbers of files.

public static String[] getListOfAvailableFilesFromServer()
public static int[] sendIdleMsgAndGetReply()
public static byte[] retrieveSingleFileAsByteArray()
public static int getNumOfAvailableFilesFromServer()

Table 5.7: API exposed by Client application.

The method `retrieveSingleFileAsByteArray()`, which allows Clients to request a file as an array of bytes involves two calls to the MPJ method `MPI_Sendrecv()`. Firstly, a request is made to find the size of the specified file, then, once the size of the file is known, a buffer can be allocated that will contain the data and a second call to `MPI_Sendrecv()` is made requesting the actual file. However, from the perspective of any client applications, it must only make one call to `retrieveSingleFileAsByteArray()`.

5.5.1.7 File splitting

There are limits on the maximum message size supported by the underlying message-passing framework. To cope with this limitation, MPJ-Cache includes file splitting and recombining functionality, which allows for large data files to be broken into smaller chucks, and these chunks are then sent in separate messages, and finally, once all of the chunks have been received at the client side, they are recombined and passed to the client application. This process is transparent to the client application, and it allows MPJ-Cache to handle data files of an arbitrary size.

5.5.2 The data caches

In the context of MPJ-Cache, the term *cache* refers to a repository of data. MPJ-Cache server applications maintain two data caches: a memory cache, and a local disk cache. Both of these caches are highly configurable. In addition, the server application can be configured to simply act as a gateway, retrieving data from a remote location as requests are received, this is effectively a *fallback* functionality, which would be used if the required data is not available in either of the local caches. Therefore, the server is aware of three data repositories, they are the following:

1. The memory cache
2. The local disk cache
3. The remote repository

Generally speaking, the response times of these repositories follows the order shown above. This is due to the simple fact that local memory provides the best performance, followed by a local disk, and lastly, the accessing of remote disks generally provides the slowest response times. If a required file is available in more than one of these repositories, the server will attempt to retrieve it from the repository highest in this list, in order to offer the best performance.

The server maintains a list of all of the files that it is aware of, that is, all of the files which are available to it in any of these three repositories. The server initializes its caches at start-up, based on its configuration, but it can also update the caches during the execution of the application, depending on its configuration.

5.5.2.1 Cache size limits

There are a number of parameters which affect the capacity of a cache. Firstly, let us consider the memory cache. The memory cache is stored within the JVM heap, therefore its size cannot exceed the maximum heap size. However, we must also keep in mind that the server application will also require a portion of the JVM heap to operate, for example, in order to allocate data to buffers, while sending and receiving messages. Therefore, it is necessary that a certain percentage of the heap is left free for the application to operate, while another percentage of the heap should be allocated for caching purposes. For this reason, MPJ-Cache offers a configuration parameter called *maxPercentageOfHeapUsedByCache* which limits the maximum size that the memory cache can grow to, relative to the total heap available. In addition, there is another configuration parameter, called *MemoryCacheByteSize* which limits the absolute maximum size of the memory cache. The size of the memory cache cannot exceed the smaller of these two limits.

Let us consider an example memory cache configuration. If the server is running in a JVM with a maximum heap size set to 2GB (using -Xmx2g), and if the value of *maxPercentageOfHeapUsedByCache* is set to 25%, then the memory cache would not grow beyond 512MB. However, if *MemoryCacheByteSize* was set to 1,048,576, then the memory cache would not grow beyond 1MB. The reason for including two mechanisms to set the cache limit size was that we found having both of these methods available to be useful. In some cases, users may wish to make use of the majority of the heap for a memory cache, regardless of what the actual heap size is. However, in such cases, it is still necessary to have some mechanism to ensure that a certain percentage of the heap remains available for other purposes.

The size of the disk cache is managed in a similar way to the memory cache. Its maximum size is also kept below the lower of two limits: a configurable percentage of the actual amount of disk space called *maxPercentageOfDiskUsedByCache*, and a configuration parameter called *maxDiskCacheByteSize* which specifies an absolute limit.

5.5.2.2 Data prefetching

When operating in the *push* mode, servers will be aware of the files that will need to be sent to clients, and they will be aware of the order in which files should be sent. This knowledge allows servers to perform some maintenance on the contents of their caches during periods when they are not busy. This cache maintenance includes the prefetching of data from the remote repository in anticipation of the future distribution of data to clients. Initially, servers populate their caches with as many files as possible, within the limits discussed in the previous section. If a server later wishes to add more files to its caches, it will need to remove some existing files. A number of policies have been implementing for deciding which files should be removed from the cache, in order to make space for the next set of files, these policies include FIFO and LRU.

The status of a cache after its initialisation is represented by Figure 5.9. This could could be a memory cache or a disk cache, as the same maintenance techniques are applied to both. All of the files that the server will need to distribute are represented by the horizontal bar. Individual files are represented by the blocks within this horizontal bar. Those files which are available in the cache are enclosed within the outer rectangle (in this configuration, there are seven files within the cache at any one time). As the server is configured to use the push mode, it knows the order in which files must be distributed, so it was able to populate its cache accordingly. However, we can see that in this example, only about 25% of the total number of files to be distributed, were actually available in the cache at the start of the distribution.

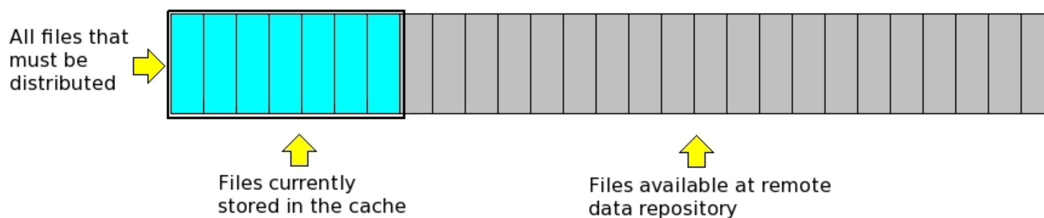

Figure 5.9: Showing the status before any files have been distributed, the cache has been populated according to the order in which files will be distributed.

The status of the same cache at a later point in time is represented by Figure 5.10. We can see that some files have been removed from the cache, and some new files have been added to the cache. Conceptually, the cache is sliding to the right, across the full set of all files to be distributed.

Over the subsequent figures: Figure 5.11 and Figure 5.12, the percentage of files in the cache that have already been sent increases. Then in Figure 5.13 we can see that the server has performed some maintenance. It has removed some files from the cache, and it has retrieved some new files. The server performs such maintenance during periods when all of the clients are busy, and therefore the server would otherwise be idle.

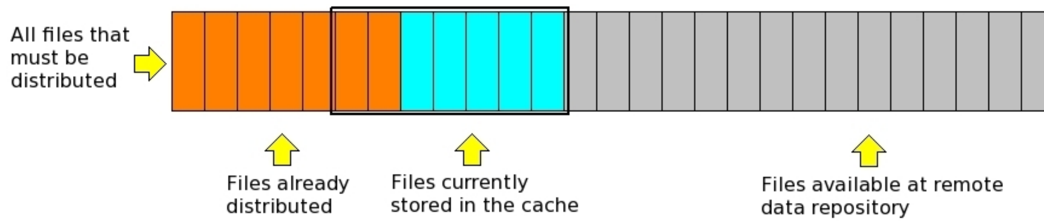

Figure 5.10: During the distribution of files, the cache contains some files which have already been sent to clients, and also some files which are yet to be sent.

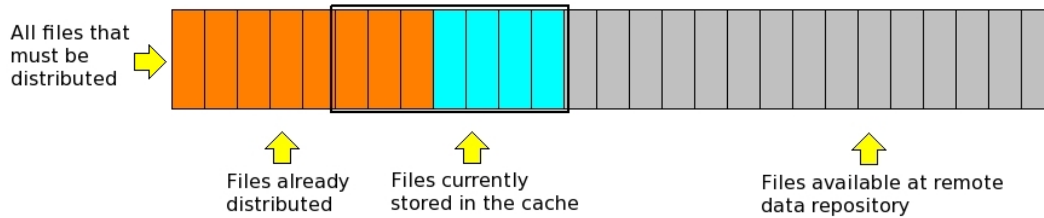

Figure 5.11: Following directly after the previous figure, the server has sent another file to a client, but it has not yet removed any files from the cache or retrieved new files.

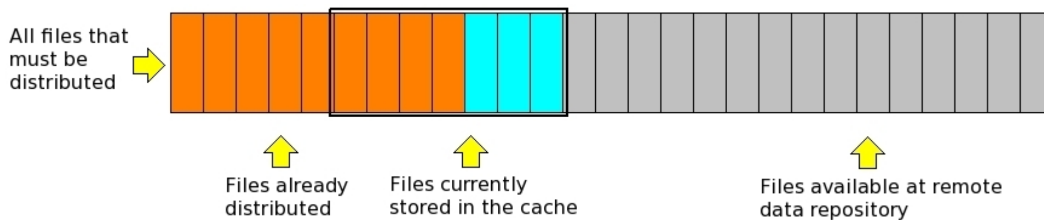

Figure 5.12: Following directly after the previous figure, the server has sent another file to a client, but it has still not removed any files from the cache or retrieved new files, as it has been busy all the time.

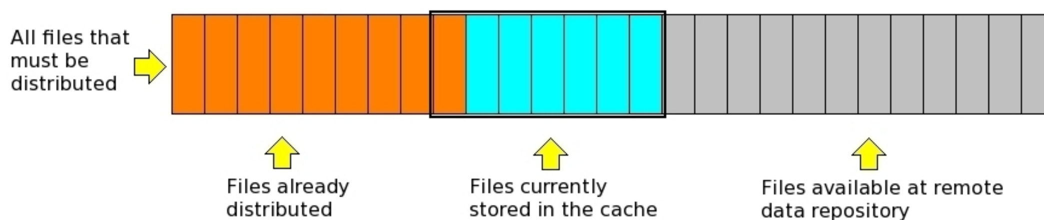

Figure 5.13: The server has experienced a period when there were no idle clients, therefore, it had an opportunity to perform some maintenance on the cache. It removed some files that were already sent to clients, and it retrieved some new files.

5.5.2.3 MPJ-Cache - configurations

MPJ-Cache is configured using an XML-based configuration file. The full list of configuration parameters for MPJ-Cache is given in Appendix D. These parameters include the cache maintenance policy, the cache size limit parameters, and the mode of operation.

5.5.3 The execution of applications with MPJ-Cache

One consideration that users of MPJ-Cache must take into account is that applications wishing to make use of the system must be executed within an MPJ environment (such as MPJ Express or FastMPJ). This has an effect on how the user application needs be launched. MPJ-Cache contains a class called *LaunchApp* which is used for the initialisation of the system. Initially, an instance of this class is executed by all of the MPJ processes. Then, within each of these process, a decision is made whether to launch a server process or a client process, depending on the rank of MPJ process — the default behaviour is that a server process is created within the MPJ process with a rank of 0, while client processes are created in the other MPJ processes. The name of the client application to be executed is passed as an argument to *LaunchApp*, and an instance of this class is then created at runtime using the reflection mechanism available in Java. In order to create several NGs, with each NG containing a server process and a number of client processes, it is necessary to launch several jobs, each one starting an instance of the MPJ environment.

5.5.4 MPJCacheTestClient

A test application called *MPJCacheTestClient* was developed in order to test MPJ-Cache. This application is highly configurable and was designed to simulate the behaviour of real data processing applications. It allows for the interleaving of sleeps between the retrieval of data, in order to simulate processing by a real client application. This application contains a number of timers within it, for the purposes of measuring the performance when it requests some data. For comparison purposes, it can be configured to request data from a server process or to retrieve data directly from a remote disk. The configuration parameters for MPJCacheTestClient are given in the Appendix D.

5.5.5 Instrumentation of MPJ-Cache

Instrumentation is a process whereby an application exposes some of its internal properties to an external systems. MPJ-Cache has been instrumented to make some of its properties available through the JMX framework. These properties can be accessed, and in some cases modified, through this framework. See Chapter 9 for more information on the instrumentation of MPJ-Cache and how it may be monitored and managed through JMX.

5.6 The tests

A number of test campaigns were carried out to directly compare the performance of MPJ-Cache against direct access to the GPFS shared disk, and to test specific functionality of the system. These tests were first executed on a local workstation, and then on MareNostrum. The test client application, *MPJCacheTestClient*, was used to request data from server applications.

These tests can be grouped into two main categories. In the first case, the client processes retrieved data directly from the GPFS, while in the second case, the client processes retrieved the same data using MPJ-Cache. These two cases are illustrated in Figure 5.4. Tests involving the use of MPJ-Cache can be further categorised into those involving one NG and those involving multiple NGs.

In order to be representative of real applications, the data communication within the tests was interleaved with processing by the client applications, this processing was simulated by the insertion of sleeps into the client processes, between each request for data. Tests involving sleeps of varying lengths were carried out. Each client also performs an initial sleep during its initialisation to ensure that not all of the clients begin requesting data at the same time.

In total, 96 tests were executed in this test campaign. In all cases, the tests involved clients retrieving twenty different files. There are many test parameters which were varied, including the size of the data files (1MB, 10MB, 100MB), the number of clients (16, 32, 64 or 128), the number of NGs (1, 4 or 8), and the lengths of the sleeps between requests. In the case of the MPJ-Cache tests, the servers were configured to store all of the data in a cache in memory.

The Bandwidth (BW) referred to in these results has been calculated using the time a request takes to be processed from the clients perspective. Therefore, it includes any delay which might occur at the server side.

5.7 Results

5.7.1 Direct GPFS access vs. MPJ-Cache with 1 NG

MPJ-Cache generally out performs direct access of GPFS in those tests with smaller files sizes and a short sleep between requests, as shown in Table 5.8, and illustrated in Figure 5.14.

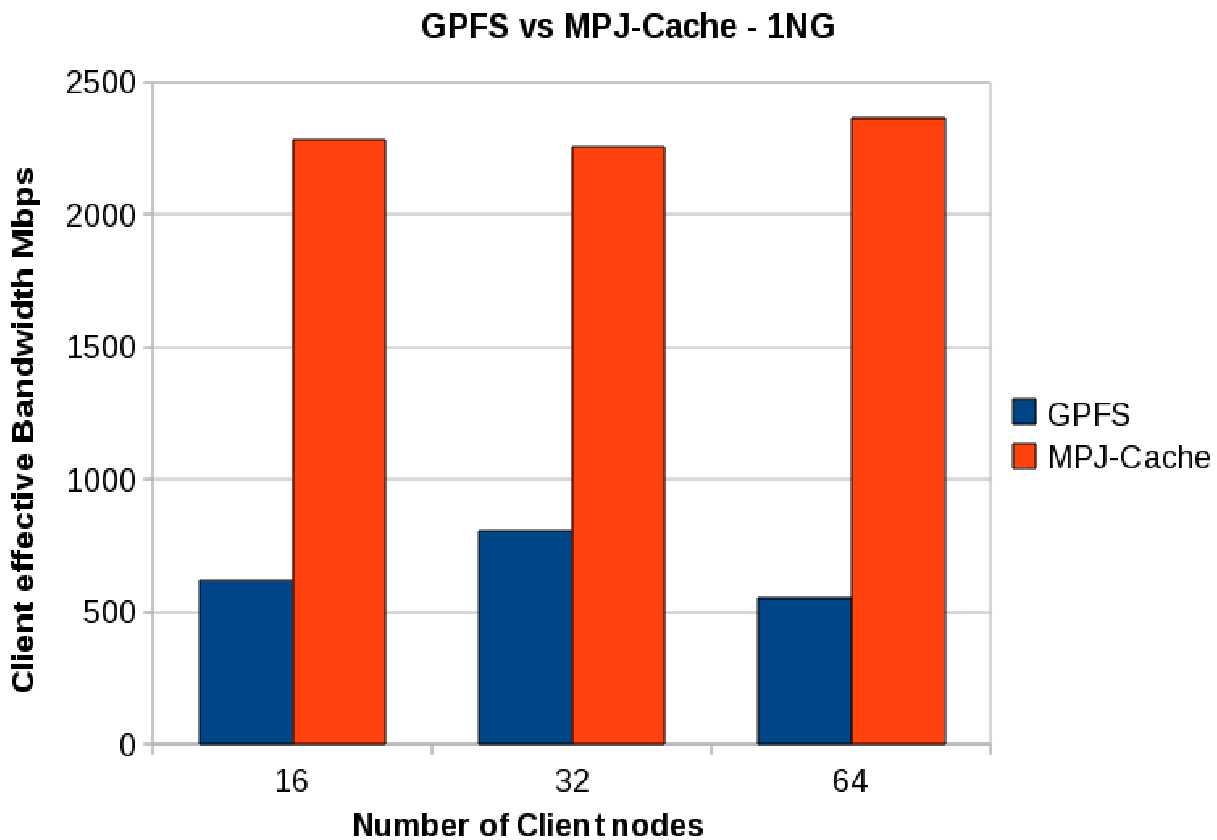

Figure 5.14: Comparison of GPFS against MPJ-Cache, when using 1 NG, small files (1MB) and frequent requests (1ms).

MPJ-Cache also outperformed direct access of GPFS in tests with large file sizes and a long sleep period, as shown in Table 5.9, the one exception being the case of 128 nodes, where

Clients	Data	Sleep (ms)	GPFS BW (Mbps)	Cache BW (Mbps)	Speed-up
16	1MB	1	616	2284	3.7
32	1MB	1	806	2256	2.8
64	1MB	1	553	2365	4.3

Table 5.8: MPJ-Cache Results — small files, small sleep, 1NG.

Clients	Data	Sleep (ms)	GPFS BW (Mbps)	Cache BW (Mbps)	Speed-up
16	100MB	20000	1026	1639	1.6
32	100MB	20000	1090	2164	2.0
64	100MB	20000	265	295	1.1
128	100MB	20000	796	83	0.1

Table 5.9: MPJ-Cache Results — large files, large sleep, 1NG.

GPFS performed best. The explanation for the poor performance of MPJ-Cache in that test is that the single server process that was running was overloaded. In other words, the server was receiving requests faster than it was able to handle them, therefore, a queue of requests built up.

In fact, in most of the cases where the GPFS outperformed MPJ-Cache, the difference was explained by an overloading of the server. We confirmed this by examining the client log files in those cases where the MPJ-Cache performed poorly. We noted that the initial requests received by the server were processed quickly, and therefore the initial bandwidth experienced by the clients was relatively good. However, as more clients began to request data from the server, it became overloaded with requests, and the average reply time experienced by the clients increased — hence the decrease in the calculated bandwidth.

5.7.2 Direct GPFS access vs. MPJ-Cache with multiple NGs

With this set of tests we intended to find the optimal configuration for accessing data from a given set of nodes. Effectively, we wanted to maximise the total amount of data that could be transferred around the system during a given period of time. We call this rate the *aggregate data rate*. We must note that this data rate contains within it, the initial sleep, as well as the inter-request sleeps performed by the clients in order to simulate real applications.

The highest aggregate data rate achieved by direct access of GPFS was 13.7Gbps, achieved through the use of 128 client nodes, requesting 100MB files with a sleep of 200 milliseconds between requests. Interestingly, if the sleep was reduced to 10 milliseconds, the data rate fell to 9.1Gbps.

The highest aggregate data rate achieved through the use of MPJ-Cache was 103Gbps. This was also achieved using 128 client nodes, but arranged in 8 NGs. Therefore, 136 nodes in total were used (including 8 MPJ-Cache servers). The files used were 100MB each, while the sleep was 1 millisecond.

The results of this test campaign showed, that given a certain number of nodes, MPJ-Cache allows for a much higher aggregate data rate than direct GPFS access, as illustrated in Figure 5.15

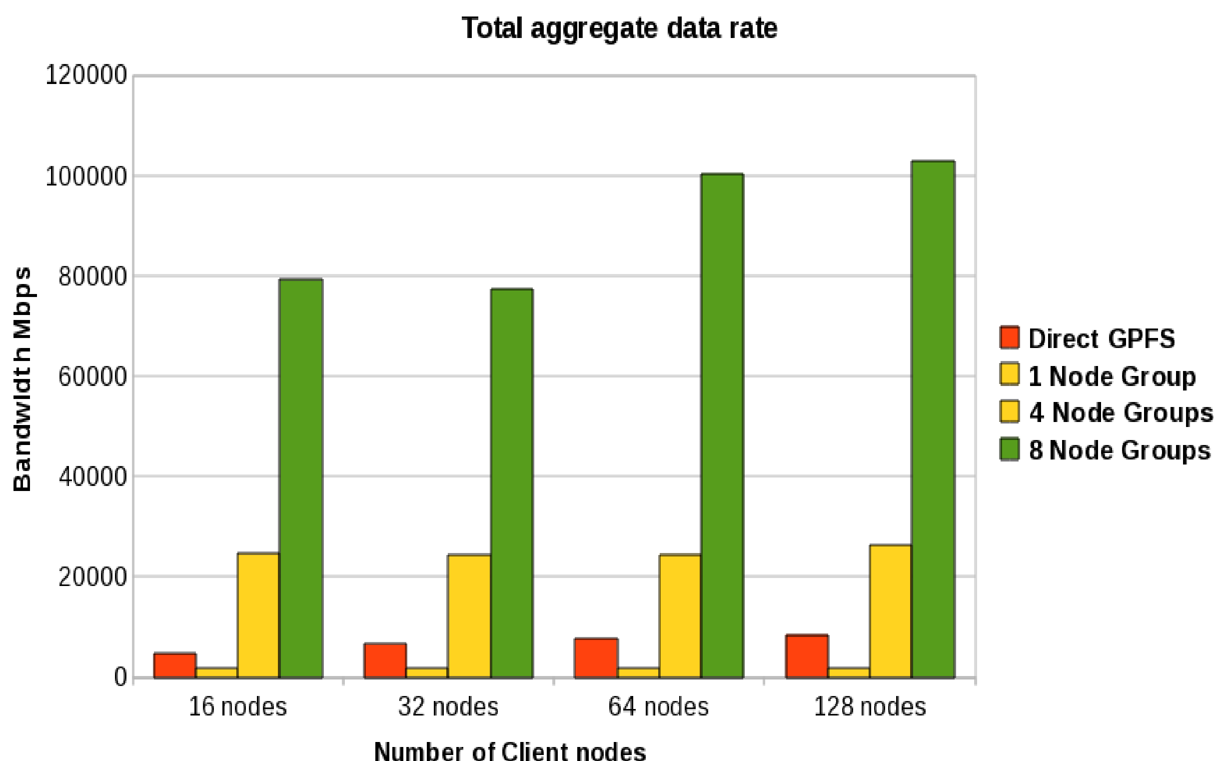

Figure 5.15: Total aggregate data rate for GPFS and MPJ-Cache, using different Node Group configurations. 100MB files have been used in this case.

5.8 Conclusions

For many years, Java applications lacked high-performance and scalable communication options for HPC environments. In particular, they could not take full advantage of high-performance networks. Our initial tests of the MPJ applications: MPJ Express and FastMPJ, showed the impressive performance offered by this systems.

MPJ-Cache builds on an underlying implementation of MPJ, taking advantage of its support for high-performance networks, and it adds a number of features useful for data intensive applications in HPC environments, including data caching, file splitting an prefetching.

Our tests showed that MPJ-Cache, configured to create a single NG, offers better performance than GPFS for small numbers of client nodes, and especially when working with small files and frequent requests. We observed in our test campaign that the available bandwidth on the Myrinet network was optimally being used, and the performance of MPJ-Cache only decreases when the server process is overloaded with requests. Furthermore, when we move to the situation of multiple NGs, the total aggregate data rate obtainable through the use of MPJ-Cache is much higher than that obtainable for the same number of nodes directly accessing GPFS.

Although our data cache system is a relatively thin layer, sitting on top of an implementation of MPJ, we believe that there are a range of applications which could benefit from its use, not just the applications described in this paper. The creation of data caches amongst the available computing nodes can avoid the I/O bottleneck which can occur when many processes are accessing a central storage device, while the use of FastMPJ allows for the best performance to be extracted from the available network.

In the case of HPC environments which are shared by many users, one possible issue is how the applications of one user may affect the applications of other users. One of the benefits of

using MPJ-Cache is that, due to the caching, there will be a reduced number of accesses of the shared disk, which should improve the disk I/O performance experienced by other users.

We intend to improve the functionality of MPJ-Cache by extending the API that it offers, and adding additional communication protocols. Finally, we intend to test MPJ-Cache in other HPC environments, such as its use with the other storage systems, such as LUSTRE [Wang et al., 2009] and Panasas [Welch et al., 2008] file systems, and by testing it on other high-performance networks.

At the time of writing, an upgrade of the MareNostrum supercomputer (from MareNostrum II to MareNostrum III) is underway. MareNostrum III features an Infiniband network, and MPJ-Cache will be tested on this machine once it is operational. MPJ-Cache will benefit from the fact that it uses an implementation of MPJ (FastMPJ) that provides Infiniband support. This should enable the execution of MPJ-Cache on MareNostrum III without any changes.

6

Profiling

The inside of a computer is as dumb as hell but it goes like mad!

– Richard Feynman

Software profiling is a broad topic, and many different techniques may be followed in order to perform this task. *Dynamic profiling* refers to the profiling of the actual execution of an application. It is generally performed to measure how some properties of the execution behave over the course of the execution. The properties of interest are often CPU or memory usage. In the case of the Java platform, profiling is often performed to better understand the interaction between the application and the runtime environment, including the garbage collector and the JIT compiler.

Profiling can be performed in a number of ways, some of which involve the modification of the target application, while other non-intrusive approaches simply involve observing the level of resource consumption during the execution. Like monitoring, the act of profiling an application adds an overhead, furthermore, the execution of a profiler on the same machine as the target application can have an influence on the profile obtained for that application.

The information gained from profiling may be used to improve performance of the profiled application, through modifying the application itself or through runtime tuning. Profiling can reveal problems in processing, such as a poorly performing GC policy, or an excessive amount of time being spent within particular methods. Once an application has been profiled, it may be *characterised* in terms of the measured characteristics. For example, an application could be characterised as being CPU-intensive, or as creating a relatively large number of long-lived objects.

The JVM has been designed to expose a huge level of profiling information, and there exist many tools for profiling the execution of Java applications. In this chapter we discuss some of the options available for profiling the execution of Java applications.

6.1 Background

The objective of profiling is to measure some properties of the execution of an application. The execution of an application within a runtime environment can involve several interacting components. As is the case with benchmarking, in order to perform useful profiling, care must be taken to ensure that the obtained profile has not been influenced by hidden, or unexpected

factors. It must be kept in mind that the very acting of running a profiler, can influence the target application.

When a JVM is created to execute an application, it must perform a number of initialisation steps, these include class loading. Additionally, when most applications begin executing, they must perform their own initialisations. Apart from these initialisations, many applications must execute for a certain period of time before they reach a state where their performance remains relatively stable. An application is said to be in a *steady state* once all initialisations are performed and its performance (including its responsiveness and throughput) are relatively stable. Profiling should not be performed before an application has reached a steady state, otherwise an inaccurate profile of an application can be obtained.

Another consideration when profiling the execution of an application, is the fact that the application may perform quite differently when executed on different hardware environments. Although the same Java bytecode can execute in a JVM on any machine, the observed performance may be quite different, in different environments. The observed performance depends on the JVM implementation, and on how well the JVM takes advantage of the available hardware. For this reason, it is important that various hardware environments are used when profiling applications, in order to avoid obtaining a machine-dependant profile of an application.

Most Java profilers operate by sampling the state of the JVM at regular intervals. Sampling involves pausing the execution of the application and recording certain properties of the current state of the JVM, such as the currently executing method. The HotSpot JVM makes use of an extremely lightweight sampling profiler called *Xprof* to quickly determine the parts of applications which are executed most frequently — known as the hotspots. It then concentrates its optimisation efforts in these parts.

6.2 Related Work

The issues of performance analysis, profiling and tuning for Java applications, executing in the HotSpot JVM, are discussed in detail in [Hunt et al., 2011]. They identify two general profiling approaches: *top down* and *bottom up*. The *top down* approach involves initially analysing the application as a blackbox, but then repeatedly analysing smaller and smaller components. The bottom up approach however, analyses the application in terms of low-level statistics, such as the *path length*, as well as the number of caches misses which occur while executing the application under load. They provide guides for the use of the Oracle Solaris Studio Performance Analyzer and the NetBeans Profiler, both of which are powerful profilers and are commonly used by the Java development community.

In [Mytkowicz et al., 2010], the accuracies of four commonly used profilers: *hprof* [hprof, 2012], *xprof* [Xprof, 2012], *JProfiler* [Jprofiler, 2012] and *yourkit* [YourKit, 2012] are compared. They used a number of single-threaded benchmarks from the highly-respected DaCapo suite of benchmarks [Blackburn et al., 2006], and performed tests on both the HotSpot and IBM J9 JVMs. They found that these profilers often produce very different profiles, and startlingly, in some cases, they can produce totally inaccurate profiles.

They investigated these inaccuracies using causality analysis to evaluate the accuracy of the profilers, and found two primary factors contributing to the problem. Firstly, they found that these profilers are not truly sampling at random intervals. Instead, sampling is performed at particular points during the execution, known as *yield points*. These are check-points inserted into bytecode by the compiler to signify that it would be safe to perform GC at those points.

The decision of where to insert yield points can therefore bias the profiles generated by sampling profilers. Compilers may, for example, choose not to insert any yield points within a loop, if it is sure that no allocations are performed within that loop, and if it will run for a predefined number of iterations. In such a scenario, a sampling profiler would never sample the state of the execution, while that loop is executing, even if that loop constitutes a very significant part of the execution. In fact, this problem can lead profilers to produce totally inaccurate profiles, if the majority of processing time is spent within sections of the code that do not contain any yield points.

Additionally, they found that the act of actually running a profiler in order to profile an application can have a significant effect on the target application, and therefore, the profile that is obtained for that application may not be accurate. This effect, which is often referred to as the *observer effect*, is mainly due to how the execution of profilers affect the dynamic optimisations that are applied to the target application at runtime. They found that different profilers affect target applications to different degrees. Finally, they introduced a new proof-of-concept profiler, called *tprof*, which does guarantee random sampling. Profiles obtained using this profiler were used for the successful optimisation of a number of benchmarks.

6.3 Profiling tools

Here we gave an account of a number of tools available for both monitoring and profiling JVMs and the Java applications running in them.

6.3.1 JVMTI

JVMTI provides an API, that is used by many profiling and monitoring tools, through which JVM state information can be retrieved. JVMTI forms the lowest level within the JPDA — a set of APIs provided by the Java platform for profiling, monitoring and debugging Java applications. It allows for profiling and monitoring *agents*, which are executed within the same process as the target JVM, to interact natively with the JVM. A number of profiling tools make use of the JVMTI, including Hprof and JProfiler.

6.3.2 Hprof

Hprof is an open source profiling tool which is included with a number of JVMs, including the HotSpot and the J9 JVMs. It may be enabled at runtime using a command-line option. If enabled, Hprof *instruments* the bytecode of class files before they are loaded by the JVM. This instrumentation processes, known as ByteCode Injection (BCI), involves inserting extra instructions into the class files. These extra instructions do not have any effect on the flow of control, or on the outcome of the application, but they allow for performance metrics to be obtained. The amount of BCI performed depends on the level of profiling requested by the user using Hprof command-line options. The overhead that Hprof adds to performance is proportional to the amount of information retrieved, however it is relatively low. The ouptput from Hprof may be obtained in American Standard Code for Information Interchange (ASCII) or a binary format. Hprof does not include a graphical tool for viewing these output files, instead they may be viewed using some graphical profiling tool. One such tool is PerfAnal, as described in the next section.

6.3.3 PerfAnal

PerfAnal [PerfAnal, 2000] is a Graphical User Interface (GUI)-based, free profiling tool for inspecting the output from Hprof. It interprets the measurements in the Hprof ASCII output file, and presents them in four simple interactive views of the execution. These views show the amount of time, and instructions, consumed by each method of the application, from a number of perspectives. Possibly the most useful of these views, is the bottom right view, which shows the amount of time spent within each method (exclusive of methods called from within the method). PerfAnal was used in the profiling activity described in Section 8.3

6.3.4 IBM Support Assistant Workbench

The IBM Support Assistant Workbench is a framework from IBM, which allows many tools (or plugins) to be installed. These tools allow for the profiling, analysis, and detection of problems when executing applications within IBM JVMs. It may be used for the submission of diagnostic information to IBM support, and it can also provide suggestions on how to resolve issues. A large number of tools can be installed within the IBM Support Assistant Workbench framework. These include: HeapAnalyzer, that can be used for detecting memory leaks; Garbage Collection Memory Visualizer (GCMV), that can parse and plot a number of output files, such as a verbose GC log, it also provides GC tuning recommendations; HealthCenter, a lightweight monitoring tool, that can also provide tuning suggestions; and Memory Analyzer, that allows for the analysis of heaps for the purposes of detecting memory leaks. GCMV is used for the purposes of GC tuning in Section 7.5. Also from IBM, the Pattern Modeling and Analysis Tool (PMAT), allows for GC trace files (from both the IBM J9 and HotSpot JVMs) to be parsed, plotted and analysed.

6.3.5 JRockit Mission Control / Java Mission Control

JRMC [Lagergren and Hirt, 2010] is a powerful suite of diagnostic tools for the JRockit JVM. It was originally developed by the JRockit development team for their own internal use, and was only later made public. JRockit Mission Control is characterised as adding little overhead to the profiled execution. JRockit, including JRMC, was later purchased by Oracle. Java Mission Control is a port of JRMC for the HotSpot JVM. At the time of writing, JRockit mission control is being totally migrated to Java Mission Control, and in the future, JRMC will not exist.

6.3.6 VisualVM

VisualVM, which also comes as part of the JDK, builds on top of the functionality provided by the earlier JConsole, and adds a number of additional lightweight profiling and monitoring tools. In addition, it makes use of an extendible plugins architecture, that allows new tools to be added. VisualVM allows for snapshots to be taken of a monitored JVM, these snapshots record the state of the JVM at a point in time and can be analysed at a later point.

VisualGC is a plugin available for VisualVM that provides a detailed graphical view of GC activity. A screen-shot of the VisualGC plugin is shown in Figure 6.1. This shows the occupancy levels, in real-time, of each of the generational spaces within the Java heap. When performing

profiling for the purposes of GC tuning, this information is very useful, as it shows which spaces are filling up, and what is causing garbage collections to occur. Another useful plugin is *MBeans* which allows VisualVM to monitor MBeans exposed through the JMX framework. VisualVM (and JConsole) are both integrated with the JMX framework, and it is possible in inspect, and control, instrumented applications using these tools. As is done with MPJ-Cache, and described in Section 9.3.4.

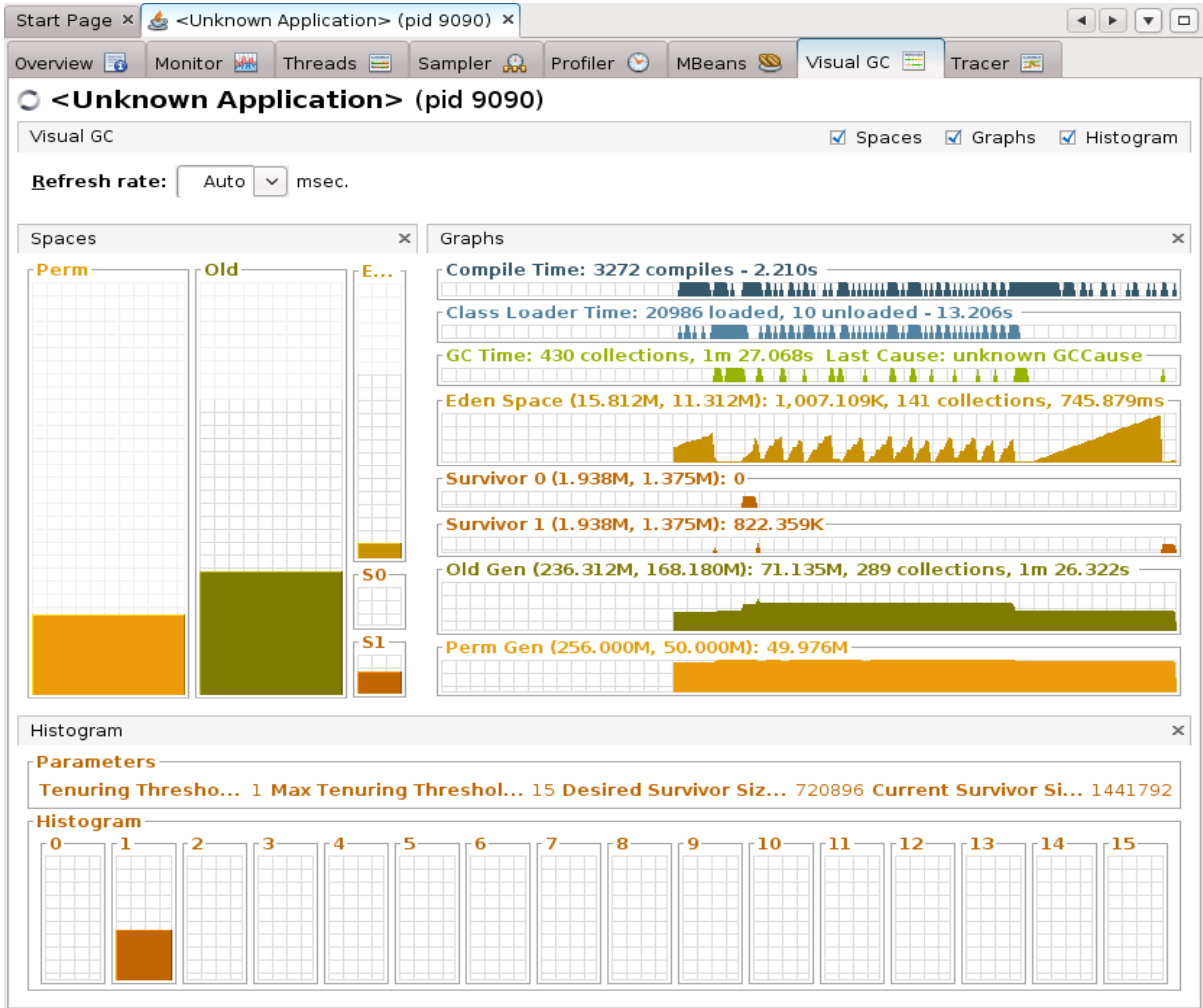

Figure 6.1: The *VisualGC* tab within VisualVM, showing the occupancy levels of each of the generational spaces within the Java heap, while a particular application is executing.

6.4 Profiling for GC

Profiling should be performed before attempting to tune the activity of the garbage collector or the JIT compiler. Such profiling can reveal the memory usage behaviour of the target application, as well as showing how well a particular GC policy (and set of tuning options) works with that application. In Section 7.5, the use of profiling for tuning GC collection is discussed. A number of command line options that can be used to profile GC and JIT compiler activity, while executing an application, are given in Appendix C.

7

Garbage Collection

The first step in fixing a broken program is getting it to fail repeatably

– Tom Duff

GC is an automatic memory management system, provided by a runtime environment, that takes care of releasing memory that is no longer referenced by an application. The vast majority of computer programming languages developed in recent decades execute in managed runtime environments, such as within the JVM or the CLR. The existence of automatic GC removes the responsibility of *freeing up* memory from software developers, and it is generally accepted that it leads to cleaner, more modular, and more robust applications, while also reducing development time.

JVMs provide GC as a service, and it is considered one of the strengths of the Java platform. The GC service provided by most JVMs is highly configurable. The widely used JVMs offer several GC policies, GC tuning options, and many other options which can affect GC. For example, the HotSpot JVM (version 1.7) has over 600 options which can be specified by users at runtime, of which, at least 250 could have an effect on GC. In this work, when we refer to *GC options* we are referring to the combination of a GC policy and additional memory management options.

The implementation of GC, and therefore the GC policies and tuning options that are provided by JVMs differ from vendor to vendor. For example, the HotSpot JVM offers a completely different set of GC policies from the J9 JVM (from IBM). Different GC policies and tuning options are best suited to different applications and workloads. The idea of selecting a particular set of GC options to match the particular characteristics, or needs, of a particular application is known as *application-specific garbage collection*.

In this chapter, we investigate some approaches that can be followed while searching for a well performing set of GC options for a particular application, and a number of test campaigns are described. Firstly, we describe a searching technique that does not make use of any *a priori* knowledge of the application to be executed, instead it simply treats the application as a blackbox and measures its performance when executed with various GC options. Later, we describe how the information gained through the profiling of applications can be used to perform a profile-directed search of the GC options. We discuss these approaches in terms of the cost of performing each one, against the performance gains that we found. We use real scientific applications in our study — three DPAC systems. We also introduce our MUM application, that can be used to simulate the memory usage behaviour of real applications. We discuss the possible uses of MUM.

The rest of this chapter is organised as follows. In Section 7.1, we provide background information relevant to this work, including an overview of the main categories of GC policies, and a description of the GC services provided by the HotSpot and IBM JVMs. In Section 7.2, we discuss some related work, including other studies which have proposed and implemented systems capable of performing automatic application-specific GC. A number of initial investigative tests were performed to gain an understanding of the effects on the performance of some DPAC systems if certain GC options are selected, these tests are described in Section 7.3. A more extensive and thorough sets of tests of DPAC systems are described in Section 7.4. Profiling for the purposes of GC tuning is discussed in Section 7.5. Our MUM application is presented in Section 7.6. Finally, in Section 7.7, we gave our conclusions and mention some possible further work.

7.1 Background

The topic of GC has existed for decades and has been the subject of extensive study. The term *policy* is used to refer to a strategy for managing GC. A GC policy defines a number of distinct activities, including the identification of sections of memory that can be released, the procedure for actually releasing memory, as well as a mechanism for deciding when garbage collection should be performed. The terms *policy* and *algorithm* are often used interchangeably in the context of GC. The term *mutator* is often used to describe the application that is running, because it mutates the state of the system. Primarily, it mutates the data stored in memory, and the references to that data. The term *collector* is often used to refer to the GC process (or policy), because it *collects* objects that are no longer referenced by the application, and frees the space that these objects had been occupying.

GC brings a number of benefits. Chiefly amongst these is the fact that it eliminates the possibility of a whole set of memory management bugs from arising, including memory leaks, and dangling pointers. These issues are a considerable concern when developing applications using many other languages, such as C and Fortran. Additionally, it simplifies code, as developers do not need to include code for releasing memory. It is also considered to lead to the development of more modular, and reusable software, and it reduces development time.

It is sometimes suggested that GC has a significant negative effect on performance, and that software developed using explicit memory management always outperforms software relying on automatic GC. It is true that the execution of an application with a set of GC options that do not match the memory usage behaviour of the application will result in quite poor performance, but it is also possible to obtain the same level of performance through the use of automatic GC as can be obtained through explicit memory management. In fact, in some cases, GC can provide slightly better performance. However, to achieve this level of performance, GC does require more memory space (up to five times the minimum memory required by an application), as described in [Hertz and Berger, 2005].

The JVM, like many other managed runtime environments, provides automatic GC. The official JVM specification [Lindholm et al., 2012] does not define exactly how JVMs should implement GC, instead the implementation details are left to individual JVM implementers. Therefore, different JVMs may offer quite different GC policies. Typically JVMs offer a number of GC policies as well as additional tuning options.

The process of allocating space to newly created objects, also provided by the JVM, is inextricably linked to GC. In some publications, the term *garbage collection* is used to refer to both

the allocation and the reclamation of memory. In JVMs, free space is allocated to new objects from a contiguous section of the heap. The performance of the allocation process is dependent on the current status of the heap. Depending on the GC policy, fragmentation of the heap can occur over time, as space is allocated and reclaimed, and if the heap is not regularly compacted. Fragmentation negatively affects the performance of allocation. Therefore, the choice of GC options affects application performance not only during the reclaiming of memory but also during the allocation of memory to newly created objects.

GC in modern JVMs is highly configurable — users may configure GC behaviour by adding command-line options at runtime. Despite this high-degree of configurability, one of the goals of GC designers is to allow GC to deliver high performance while requiring the minimum level of instructions from users. This has lead to the development of sophisticated GC policies that can adjust their own behaviour, taking into account the characteristics of the running application, and the available computing resources. In fact, in most cases, GC delivers very good performance without users having to specify any options at runtime. The ability of a JVM to deliver high performance, matching the characteristics of the running application and the execution environment, is known as ergonomics. One of the goals of ergonomics is the provision of garbage collectors that can effectively tune themselves (see Section 7.1.7 for more details on JVM ergonomics).

However, notwithstanding the intelligence of modern JVMs, many users may still wish to attempt to improve the performance of their applications by selecting a particular set of GC options. To perform this task, users must have an understanding of how their application behaves during its execution, this information can be found through profiling the execution of the application. Additionally, users need an understanding of the available GC options, and how these GC options can complement, or interfere with each other.

Amongst the GC options which can have the largest effect on application performance, are those related to the size and management of each of the spaces within the heap. Users can set the initial and maximum sizes of each of these spaces (within the limits of the available memory), they may choose to permit heap *resizing* — the expansion and shrinking of the heap by the JVM, based on heap occupancy. Heap resizing allows for scenarios where memory usage varies over time, as happens in many applications during their execution. However, the process of shrinking or expanding the heap introduces an additional overhead. Smaller heaps generally result in more frequent, but fast collections, while larger heaps generally result in less frequent, but more costly collections. See Section 7.1.6 for more on *heap sizing*.

Garbage Collections come in a number of forms, and can be composed of a number of phases. Parts of a GC may be performed concurrently with the threads of the running application, while other parts of the collection may require that all other threads must be paused, to grant the GC thread(s) exclusive access of the heap. The term Stop-The-World (STW) is used to refer to any part of a GC that requires all other threads to be paused. When a STW GC phase is being performed, the running application will not be making any progress, and will be totally unresponsive to external systems, such as users or client applications. Long STW periods can be unacceptable to applications that require a certain level of responsiveness at all times.

7.1.1 GC policies

7.1.1.1 Basic GC policies

GC policies can be grouped into four basic categories, these all date back to the 1960's. They are: mark-sweep, mark-compact, copying, and reference counting. There are a vast number of specific GC policies, however they can generally be described as a variation or a combination of these four main categories [Jones et al., 2010].

Mark-sweep involves two steps. Firstly, the algorithm obtains a list of all the known *root* objects. Root objects are a key concept within GC. They refer to those objects that are immediately available to the GC without having to go through other objects, they include objects referenced from anywhere in the call stack, as well as globally accessible variables. Using the root objects as a starting point, it then builds a graph of all reachable objects. This process is known as tracing. As it finds each object, it *marks* them, by setting an associated flag (either in the object itself or in a table). The second step involves the GC directly inspecting every object on the heap. Any object that was not marked during the marking phase must have no references to it, therefore is unreachable, and its space be can safely reclaimed. Mark-sweep does not involve the copying or moving of any objects, and due to its simplicity, it imposes little overhead. One negative aspect of mark-sweep is that because it does not perform any compacting of the heap, fragmentation will eventually occur. Mark-sweep is a STW policy, that is, all mutator threads are paused during collection.

Mark-compact also begins with a tracing and marking step to determine which objects are alive. However, a compacting step is then performed which attempts to move all live objects to one end of the heap, eliminating spaces between objects. Mark-compact does involve extra overhead, compared with mark-sweep, due to the cost of moving objects during the compaction step, however its allocation performance is stable over a long period of time, whereas the allocation performance offered by mark-sweep can decrease due to fragmentation.

Copying GC algorithms involve the partitioning of the heap into (at least) two equally-sized spaces, known as semispaces. These spaces are typically known as the *tospace* and the *fromspace*. At any one time, objects are stored in the fromspace, while the tospace is empty. During the collection process, all the live objects in the tospace are copied to the fromspace, and the roles of the two spaces are switched. The main negative feature of basic copying algorithms is that they involve half of the heap not be used at any one time. They do however result in the compaction of the heap during each collection, therefore allowing fast allocation. Copying algorithms generally perform poorly if objects live for a relatively long time, because they will be repeatedly copied from one space to another.

Finally, reference counting, in its most basic form, involves the GC policy maintaining a counter for the number of references to each object. The counter associated with each object is increased and decreased, as the number of references to that object changes. An object is considered alive if its counter is greater than zero, otherwise it is are dead, and its space may be reclaimed.

7.1.1.2 Generational policies

Generational GC is a variation of the copying GC policy. Studies have shown that for the majority of applications, most objects die young. This is known as the *weak generational hypothesis*. Building on this theory, and taking into account the fact that frequently copying,

or moving, long-lived objects is inefficient; generational GC policies attempt to process older objects less often than younger ones. They involve partitioning the available memory into a number of spaces, each of these spaces is then used for storing objects of a certain age. Age is usually measured by the number of garbage collections that an object has survived, or by the number of bytes of heap space that have been allocated since an object was created.

The weak generational hypothesis has been found to be true for a range of languages and applications, although the term *young* is obviously relative, and the average object lifetimes for any application depends on the characteristics of the application. One way of measuring the lifetime of an object is the number of bytes that are allocated (for any purpose within the application), after the object was created, and before the object becomes eligible for deletion. The developers of the widely respected DaCapo benchmarks showed that, on average, across all of their benchmarks, setting the young generational space to 4MB resulted in less than 10% of objects getting promoted to the old generation [Blackburn et al., 2006]. This means that 90% of objects created in their benchmarks become eligible for deletion before 4MB of bytes are allocated to new objects.

The generational GC policies commonly used within JVMs, generally involve partitioning the heap into a *young space*, an *old space*[1], and in the case of the HotSpot JVM, a third space called the *permanent generation*. The permanent generation is quite different from the other two spaces, in that admittance to it is not based on the age of the data, and instead of storing objects created in application code, it is used by the JVM for storing data that it needs, such as class metadata. In both the HotSpot and J9 JVMs, large objects may be allocated from a particular section of the old space known as the Large Object Area (LOA) which is reserved for objects over a certain size. The rest of the heap is sometimes known as the Small Object Area (SOA).

Generational GC polices have a number of benefits over simpler copying approaches — which involve all objects being repeatedly copied from one semispace to another. Many applications generate objects which are long-lived, and the repeatedly coping of all these objects adds avoidable overhead. In a GC policy, such long-lived objects get promoted to the old space relatively quickly, where they can remain without being moved again.

Garbage collections in the young space are generally referred to as *minor collections* as the young space is often smaller then the old space, and therefore requires less time or resources to perform. Collections in the old space (or collections of the entire heap) are referred to as *major collections* and often require a lot of time or resources to perform. Some GCs have a STW phase, where all other threads must be paused to allow the GC thread to have exclusive access of the heap. Generational policies try to minimise the number of major GC that need to be performed.

The young space is subdivided into an area where new objects are initially allocated space, called the *eden* space, and an area where objects that have survived a number of GC collections reside, called a survivor space. There may be more than one survivor space, as is the case in the HotSpot JVM. During a collection in the young space, live objects are copied from the eden space to the survivor space, while live objects in the survivor space that have reached a certain age get promoted to the old space.

Each of these spaces can be managed by a different GC policy, and collections in these spaces can be carried out independently from each other. These areas have different characteristics —

[1]The exact terminology used to refer to these spaces varies. Sometimes the old space is known as the *tenured space*, and sometimes the *eden space* is known as the allocation space

objects in the young space normally are short-lived while objects in the old area are generally long-lived, therefore the use of different GC policies, appropriate to their respective characteristics, can improve performance.

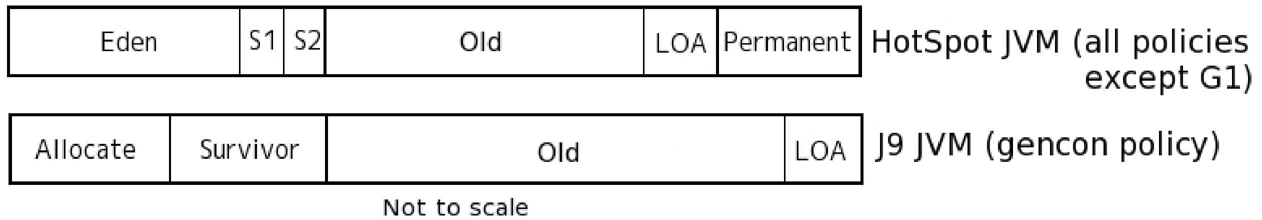

Figure 7.1: Main spaces in the generational heap layout of the HotSpot and J9 JVMs.

7.1.2 JVM GC policies

The Hotspot and J9 (IBM) JVMs are two of the most widely used. The HotSpot JVM always uses a generational policy, while the default policy in the J9 JVM, *gencon* (GENerational CONcurrent), is also a generational policy. The typical generational division of the heap by these two JVMs is illustrated in Figure 7.1. Although the *G1* policy in the HotSpot JVM is also a generational policy, its division of the heap into spaces is markedly different from the other HotSpot policies, as mentioned in the subsections below, where we briefly describe the GC policies offered by these JVMs.

7.1.2.1 HotSpot JVM GC

In the case of the HotSpot JVM, the young space contains an allocation space, generally known as the *eden space* and two survivor spaces, known as *S1* and *S2*, these are sometimes known as the *from space* and the *to space*. During a minor collection, surviving objects are copied from the eden space, and from the *from space* to the *to space*. Therefore after each minor collection, the eden space and the *from space* are empty. The survivor spaces then switch roles, i.e. the *to space* becomes the *from space* and vice versa. Objects in the young generation which reach a certain age get promoted to the old generation.

HotSpot allows for various policies to be used to manage each of these spaces, offering serial, parallel and concurrent policies for the young and old spaces. In the following paragraphs we summarise the main characteristics of the HotSpot GC policies.

The *Serial* policy is the simplest of the HotSpot policies. Both minor and major collections are performed in a STW manner — meaning that all application threads are paused during all GC activity. The young generation is managed by a copying GC policy, while the old generation is managed using a *mark-compact* policy. The Serial GC policy is advised for applications that do not have short pause requirements. It can however only make use of a single virtual processor, therefore, it cannot take full advantage of multi-core processors.

The *Parallel* policy, sometimes known as the *throughput* policy, is similar to the Serial policy with the exception that both minor and major collections may be performed in a parallel manner — meaning that multiple GC threads may be used to perform each step. An old version of the Parallel GC policy (enabled with *-XX:+UseParallelGC*) only performed minor collections in parallel, however a more recent version of the policy (enabled with *-XX:+UseParallelOldGC*) performs both major and minor collections in parallel, and is recommended over the older

version. Collections in the *Parallel* GC are also STW in nature, however their duration are shorter as multi-core processors may be fully utilised to perform collections. If the Parallel GC policy is used, then a technique called *adaptive sizing* is available, this technique automatically attempts to tune the size of the eden and survivor spaces, based on the rate of object allocation and survival. For many types of workloads, the *Parallel* policy provides the best throughput of any of the HotSpot GC policies.

The *Concurrent Mark-Sweep (CMS)* policy, sometimes known as the *mostly concurrent policy*, involves the old generation space being collected concurrently with application threads — only a relatively short part of the marking phase must be performed in a STW manner. The CMS policy therefore results in shorter maximum pauses due to GC activity, and applications can guarantee better responsiveness. It does however add more overhead than the other polices, as this concurrency requires extra management. For most application workloads, CMS provides slightly less throughput than the Parallel policy.

The *Garbage First (G1)* [Detlefs et al., 2004] policy was first introduced in HotSpot 1.6 update 14 as an experimental policy, and it is available as a standard policy in HotSpot 1.7. G1 is a generational policy — treating young objects and old objects differently, however its division of the heap is quite different from the other HotSpot policies. Instead of storing all the objects of a particular generation in a single contiguous space, it stores objects in small non-contiguous blocks, called *regions*. G1 attempts to behave like a real-time JVM, offering very short pauses, while still providing high throughput.

7.1.2.2 J9 JVM GC

The J9 JVM 1.6 offers four GC policies. The *optthruput* policy, which is the default, tries to provide high throughput, at the expense of occasional pauses. It is a mark-sweep-compact policy, that provides good throughput for the majority of workloads, but is not recommended for applications with high responsiveness requirements.

The *gencon* policy is a concurrent generational policy, that tries to keep pause lengths short, while also maintaining good throughput. It provides best performance for applications that create a relatively large number of short-lived objects. It also minimises heap fragmentation. It became the default policy in J9 JVM 1.7.

The *optavgpause* policy attempts to keep the average pause lengths to the minimum, and is recommended for large heaps. It can be used for applications with high responsiveness requirements. The *subpool* policy offers improved object allocation performance, it is specifically designed to offer fast object allocation on machines with sixteen or more processors. This policy was however deprecated in J9 JVM 1.7.

The J9 JVM 1.7 introduced two new policies: *balanced* and *metronome*. The *balanced* policy, like the *G1* policy for the HotSpot JVM, involves the heap being divided into a large number of spaces, known as *regions*. Regions are collected independently from each other, and the garbage collector decides which regions to collect based on a cost benefit analysis. At any given time, a particular set of regions act as the eden space, from which new objects are allocated. The *balanced* policy is only available in 64-bit JVMs, and is intended for use on heaps larger than 4GB.

The *metronome* policy breaks the task of GC into smaller interruptible steps, with the goal of removing some of the non-determinism in application behaviour that GC can introduce. Users can specify the minimum desired percentage of time that their application should be executing,

during an interval of time. The command-line options for enabling both the HotSpot and the J9 GC policies are given in Appendix C.

7.1.3 Allocation process

The process of allocating space to newly created objects, also provided by the JVM, is inextricably linked to GC (in some sources, the term GC is used to refer to both the allocation and reclamation of memory). In JVMs, free space is allocated to new objects from a contiguous section of the heap. The performance of the allocation process is dependent on the current status of the heap. Fragmentation is an effect that can occur, if space is allocated and reclaimed over a period of time, and the heap is not compacted. It results in spaces appearing between allocated sections of the heap, this negatively affects the performance of allocation. Allocation is faster when a compacting GC policy is used. Therefore the choice of GC options affects application performance not only during the reclaiming of space but also during the allocation of space to newly created objects.

The allocation of space to a new objects can require the allocating thread to obtain a lock on that part of the heap where new objects are created. However, the most widely used JVMs avoid this potential bottleneck by assigning an allocation area to each individual thread. In the IBM JVM, TLHs [Domani et al., 2002] are a small chuck of the heap which is reserved for each thread, that can be used for allocating space to small objects. Allocation in the TLH does not require the heap lock and therefore provides fast allocation for small objects. In the Hotspot JVM, a similar technique known as TLABs is used.

7.1.4 Selecting a GC policy

Before selecting a GC policy, thought must be given to the performance priorities of the application. In some cases, overall application throughput may have the highest priority, in other words, the priority might simply be to perform the maximum amount of processing within a period of time. This may mean that a small number of long pauses, caused by GC activity, would be preferable over a large number of shorter collections, if the overall throughput is higher. In other cases, particularly those which involve human interaction, minimising the maximum pause times due to GC activity might be preferred, even if this results in a reduced overall throughput. In some cases, minimising the memory footprint might be important, while in other cases, maximising startup time might be the priority. Often, increases in one of these properties leads to a decrease in some others.

As well as deciding on the performance priorities, the memory usage pattern of the application should be profiled and understood, as different GC options are best suited to different memory use patterns.

7.1.5 GC tuning

In addition to the selection of a GC policy, there are many tuning options and techniques that can be considered. An obvious GC tuning approach that can be taken is to attempt to minimise the amount of work that must be performed by GC. One way to do this is to reduce the amount of objects that are created during the execution of an application. A code review, and profiling exercise could be used to determine if some object creation could be avoided, thus leading to

reduced GC work, and better performance. One coding approach that may be taken is the use of *object pools*. These involve a fixed number of reusable (or recyclable) objects being created and stored in a pool. Objects may be taken from the pool, used to perform some task, before being returned to the pool for future use. Thus avoiding the necessity to create new objects each time one is required.

It should be kept in mind that reducing the amount of time spent in GC activity does not necessarily lead to an improvement in application performance. In most applications, the time spent in GC is quite short relative to the time spent in the actual application. Therefore, it might be beneficial to actually spend a little more time in GC, if that results in benefits for the application, for example, through providing faster allocation [Jones et al., 2010].

In respect of generational GC policies, varying the sizes of the various generational spaces, both their absolute sizes and their sizes relative to each other, can lead to very significant performance changes. Reducing the size of the young space will result in faster minor collections, however, as the minor collections would be more frequent, objects might not be given a sufficient period of time to die. This can lead to objects being promoted to the old space prematurely, leading to a costly major collection being required sooner. A general rule of thumb is that an application which generates a relatively large number of short-lived objects would benefit from a proportionately large young space.

7.1.6 Selecting heap options

The amount of memory available in computer systems — from iPads to large computer clusters — is continuing to increase. This allows applications to have larger memory footprints, and it lessens the concern, that was once very strong amongst software developers, of using the available memory as efficiently as possible. It is probably true that having more memory available than is actually required, generally leads to the development of less efficient software. This is especially true for languages which rely on GC for managing memory, as it is tempting for developers to adopt the attitude that limitless objects can be created and the garbage collector will still be able to manage memory efficiently. In the Java platform, *oom* errors are caused when an application attempts to create a new object, but there is insufficient space available to create it. In such situations, it is tempting to run the application with a larger heap. The application may genuinely require a larger heap to execute, therefore the use of a larger heap is justified. However, the *oom* may indicate a bug, or a design flaw in the application, and although a larger heap may allow the application to complete successfully, it may be running very inefficiently, for example, with more than 50% of its execution time being spent in GC. In such cases, it would be better to fully investigate the problem, and devise a solution, rather than simply *throwing* memory at it.

The expression *sizing the heap* is used to refer to the process of selecting the size of the heap, as well as selecting the management options that will control the heap during the execution of a particular application. Most JVMs offer a plethora of heap management options, these include selecting an initial and a maximum size of each of the main areas of the heap (the young space, the old space, and the permanent generation). If users set the initial and the maximum size of an area to the same value, then that area will have a fixed size during the entire execution of the application. If however, users set a maximum size larger than the initial size, then the size of that area may be expanded and shrank by the JVM during the execution. This process of expanding or shrinking the size of the heap is known as *heap resizing*. JVMs decide when to perform heap resizing based on heap occupancy levels. See Appendix C for a list of selected GC

options for setting the sizes, and a number of other heap management options, for the HotSpot and J9 JVMs. Heap resizing allows for scenarios where memory usage varies over time, as happens in many applications during their execution. Heap resizing does however introduce an additional overhead, as it requires a full garbage collection to be performed before resizing.

The default HotSpot and J9 JVM GC policies try to maintain a heap occupancy level of between 40% and 70%. When heap occupancy falls outside of this range, the JVM may either shrink or expand the heap accordingly. Lower levels of occupancy result in more frequent collections, while higher levels result in less frequent, but more costly collections. Users may specify the levels of heap occupancy that should trigger heap resizing, as well as by how much the heap should be resized at any one time. For a generational GC policy, the size of the young space in particular can have dramatic effects on the performance of an application.

7.1.7 Ergonomics

Ergonomics refers to the ability of JVMs to configure their own configuration and operation, based on the environment in which they are operating, the characteristics of the running application, as well taking into account some performance goals which may be specified by users. The objective of ergonomics is to provide high *out of the box* performance, without the need for users to tune the JVM manually. JVMs use ergonomics for selecting GC, JIT compiler and heap-size settings. As part of ergonomics, the HotSpot JVM detects system properties, including the number of processors and the amount of memory available, and determines whether it is running on a *server class* or a *non-server class* environment. Appropriate settings are selected based on this classification.

Adaptive heap sizing is a form of ergonomics used by the Hotspot JVM to automatically control the size of the young space when using the Parallel GC policy. This technique involves the garbage collector monitoring how long objects spend in the young space, and modifying its size to ensure that short-lived objects don't get promoted to the old space.

7.2 Related work

[Fitzgerald and Tarditi, 2000] made the case for profile-directed GC policy selection, their study, involving twenty benchmarks and several policies, found that for any given GC policy, there was at least one benchmark which would perform better with an alternative policy. In other words, there is no policy which performed best for all the benchmarks executed. [Soman et al., 2004] presents the implementation of a JVM which allows for the dynamic switching between a number of GC policies during application execution.

It has long been known that synthetic or toy applications, which exhibit simple memory usage behaviour are of little use when trying to determine how real applications will behave when using a particular set of GC options. [Wilson et al., 1995] discusses the reasons why simple synthetic applications tend not to model real application behaviour.

[Blackburn et al., 2006] presented the DaCapo suite of benchmarks, as an alternative to traditional benchmarks, which typically lack sufficiently rich behaviour to reproduce the complex interactions which occur during the execution of real Java applications. They strongly recommend to use a range of heap sizes when comparing the performance of GC policies because behaviour can change significantly based upon the heap size [Jones et al., 2010].

[Singer et al., 2007] presented an approach for the selection of application-specific GC options using a machine learning approach. They use the DaCapo benchmarks as their training set, and executed their tests on the Jikes JVM.

[Brecht et al., 2006] examined how the controlling of GC activity, and heap size management can affect the performance of Java applications. One of their findings being that the best GC configuration for a particular application is not only dependent on the memory usage behaviour of the application, but also on the size of the allocated heap. Analysis of the performance of a GC policy for a particular application should always be accompanied by a description of the initial and maximum heap sizes.

A detailed, and iterative approach for tuning the performance of GC for the HotSpot JVM is given in [Hunt et al., 2011]. They show how information obtained by adding HotSpot command-line options can be used to tune GC. Information of particular importance is the sizes and occupancy levels of each of the generational spaces of the Java heap, while the application is executing.

7.3 Initial GC test campaigns at the DPCB

In this study we used three DPAC systems, namely GASS, IDU-ipd and IDU-crb. See Sections 2.3.4.2 and 4.1.1.1 for descriptions of GASS and IDU respectively. As these applications are not interactive, occasional pauses are acceptable, and the objective of GC tuning for these applications is to increase throughput. The following test campaigns were executed.

7.3.1 GC tests 1.1

An initial set of tests, that we call *GC tests 1.1*, which involved the execution of the GASS application (release 11.0) on a local workstation, called Miranda (see Appendix A). Both the HotSpot and J9 JVMs were used. A configuration of GASS was selected that required two to three minutes on average to complete. This configuration was kept constant for each of the tests. A total of eleven test configurations were selected, the only parameters that were varied were the JVM used and the GC policy.

These test configurations, as well as the average execution times are given in Table 7.1. The initial, and maximum heap sizes were set as 1GB and 2GBs respectively, for all the tests. All tests were executed ten times and average execution times were calculated. Therefore, in total, 110 tests were executed in this test campaign. Care was taken to ensure that no other user processes were running on the machine at the same time as the tests.

Two scripts were created to facilitate the convenient execution of these tests: *runAllTests.sh* and *runGass.sh*. The script *runAllTests.sh* acts as the master script, and it is the script which is actually executed by the user. This script then calls *runGass.sh* for each individual test, passing the particular test configuration.

As can be seen in the *Average time* column in Table 7.1, the HotSpot JVM generally far outperformed the J9 JVM, while the shortest average execution time was obtained by using the parallel GC policy on the HotSpot JVM (the 1.6 and 1.7 versions offering roughly the same performance). Of the J9 GC policies, *gencon* performed best.

Test	JVM	Version	GC Policy	Average time (s)
1	J9	1.6	gencon	187.2
2	J9	1.6	optavgpause	193.4
3	J9	1.6	optthruput	203.5
4	J9	1.6	gencon	188.8
5	J9	1.7	gencon	183.8
6	J9	1.7	metronome	290.9
7	HotSpot	1.6	parallel	111.4
8	HotSpot	1.7	serial	121.6
9	HotSpot	1.7	parallel	112.3
10	HotSpot	1.7	CMS	115.2
11	HotSpot	1.7	G1	120.3

Table 7.1: GC tests 1.1 - test configurations and execution times.

7.3.2 GC tests 1.2

The next test campaign that we executed also involved the GASS application, but this test campaign, which we call *GC Tests 1.2*, was executed on MareNostrum II. The only JVM available on this machine was the J9 JVM from IBM, see Appendix A for the full specification of MareNostrum II. The primary objective of these tests was to observe how heap size affects the performance of GASS.

The characteristics of a GASS simulation are controlled by a configuration file. It is possible to configure the number and types of objects that are generated during an execution. Using this configuration file, two distinct configurations of the GASS application were selected, these were called *high density* and *low density*. The low density configuration involved the generation of relatively few objects during a period of time, while the high density configuration involved the generation of many objects.

The configurations of the tests in this campaign are given in Table 7.2. The *metronome* policy is not available for the PPC64 version of the J9 JVM, therefore it is not included in *GC Tests 1.2*. Similarly, the *balanced* policy is only available on the 64-bit J9 JVM, therefore it is included in *GC Tests 1.2*, but was not included in *GC Tests 1.1* as that test campaign was executed on Miranda, which is a 32-bit machine.

As we were already quite confident that the *gencon* policy provides the best performance for the GASS application, we executed many more tests with this policy than with the other policies. In total, 42 test configurations were selected, 20 using the 1.6 JVM, and 22 using the 1.7 JVM (there is 1 more GC policy available on the 1.7 JVM, and this results in 2 additional test configurations — one for low density and one for high density). Again, all test configurations were executed 10 times, and average values were calculated.

The results are also given in Table 7.2. *OOM* indicates that a test failed due to an OOM error. *N/A* indicates that a policy was not available. The observed execution times of the *gencon* tests, for both the low and high densities, are given in Figures 7.2 and 7.3 respectively.

The most striking feature from the low density, gencon tests, is the fact that the 1.6 JVM consistently outperformed the 1.7 JVM, and the fact that, although the execution times varied across different maximum heap sizes, the performance of the two JVMs relative to each other remained constant. This is clearly illustrated by the shape of the two lines in Figure 7.2. It is more difficult to draw conclusions from the high density gencon tests, which are plotted

in Figure 7.3. Overall, both JVMs end up with a slightly longer execution time with the largest *maximum heap size* than with the smallest *maximum heap size*. Although, the pattern of changes in the observed execution times, as the maximum heap size is increased, are very different between the two JVMs.

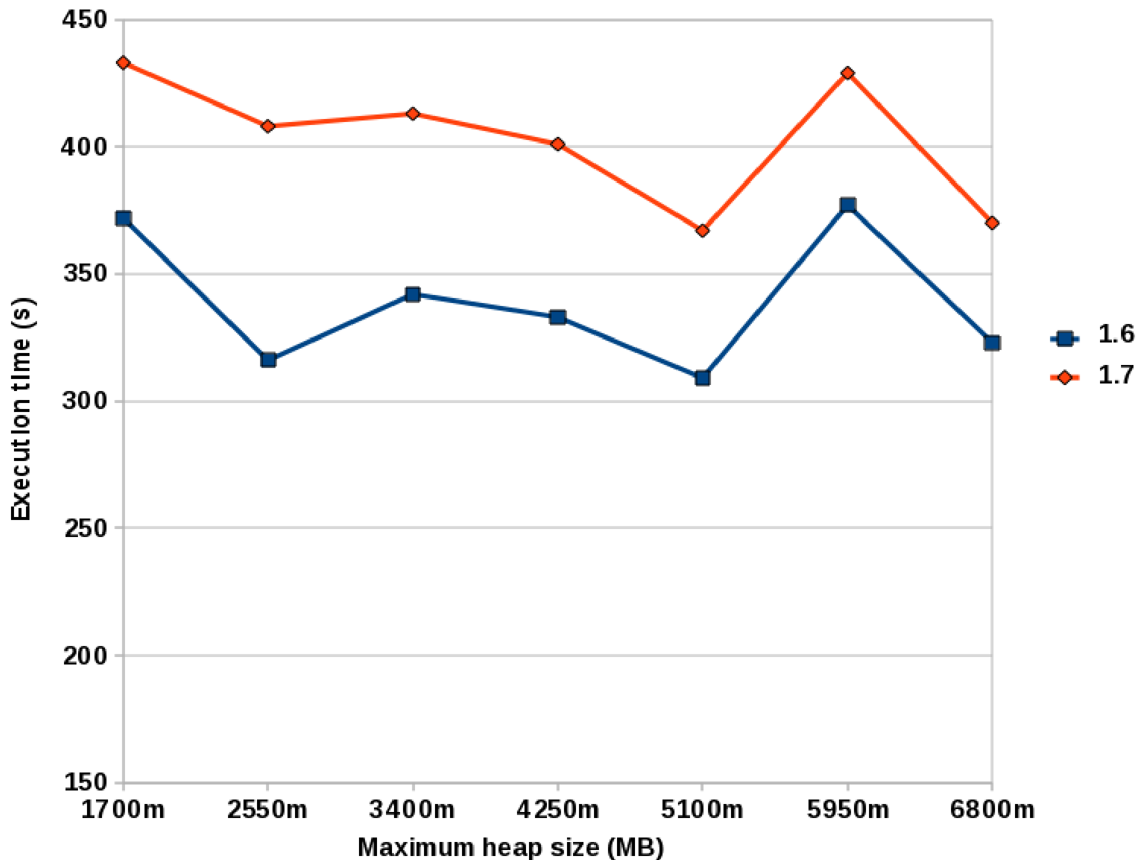

Figure 7.2: Execution time of low density configuration of GASS, with the gencon policy, on the J9 JVM 1.6 and 1.7.

7.4 GC tests 2.1 — extensive GC test campaign

As well as GC policies, JVMs offer a plethora of GC tuning options. In the next test campaign, which we call *GC tests 2.1*, we investigated the effect that setting a number of GC tuning options have on the performance of three DPAC systems: GASS, IDU-ipd and IDU-crb. For each of these applications, we first found a configuration which involved the application running for between one and two hours. We then determined the minimum heap size required for each application to execute successfully.

The next step was to investigate the available tuning options for each JVM and to select a subset of these options to be used in our tests. Care must be taken when selecting a combination of tuning options, as some options can interfere with each other. The number of GC options related to GC is large for both JVMs. Some of these options are specifically related to particular policies, for example, in the HotSpot JVM, the option *-XX:ParallelGCThread* has an effect only if a parallel policy is being used. We decided that we would include five GC policies and nine tuning options in our HotSpot tests, and four GC policies and nine tuning options in our J9 tests (in order to facilitate the direction comparison of both JVMs we selected options which were supported by both JVMs). The options that we selected are all related to the size and

Test	JVM	Version	GC Policy	Initial heap	Max. heap	Density	Average time (s)
1	J9	1.6	gencon	768m	1700m	Low	372
2	J9	1.6	gencon	768m	2550m	Low	316
3	J9	1.6	gencon	768m	3400m	Low	342
4	J9	1.6	gencon	768m	4250m	Low	333
5	J9	1.6	gencon	768m	5100m	Low	309
6	J9	1.6	gencon	768m	5950m	Low	377
7	J9	1.6	gencon	768m	6800m	Low	323
8	J9	1.6	optthruput	768m	6800m	Low	378
9	J9	1.6	optavgpause	768m	6800m	Low	370
10	J9	1.6	subpool	768m	6800m	Low	344
11	J9	1.6	gencon	768m	1700m	High	OOM
12	J9	1.6	gencon	768m	2550m	High	4240
13	J9	1.6	gencon	768m	3400m	High	4308
14	J9	1.6	gencon	768m	4250m	High	4233
15	J9	1.6	gencon	768m	5100m	High	4115
16	J9	1.6	gencon	768m	5950m	High	4148
17	J9	1.6	gencon	768m	6800m	High	4350
18	J9	1.6	optthruput	768m	6800m	High	4955
19	J9	1.6	optavgpause	768m	6800m	High	4890
20	J9	1.6	subpool	768m	6800m	High	5133
21	J9	1.7	gencon	768m	1700m	Low	433
22	J9	1.7	gencon	768m	2550m	Low	408
23	J9	1.7	gencon	768m	3400m	Low	413
24	J9	1.7	gencon	768m	4250m	Low	401
25	J9	1.7	gencon	768m	5100m	Low	367
26	J9	1.7	gencon	768m	5950m	Low	429
27	J9	1.7	gencon	768m	6800m	Low	370
28	J9	1.7	optthruput	768m	4250m	Low	551
29	J9	1.7	optavgpause	768m	5100m	Low	414
30	J9	1.7	balanced	768m	5950m	Low	495
31	J9	1.7	metronome	768m	6800m	Low	N/A
32	J9	1.7	gencon	768m	1700m	High	OOM
33	J9	1.7	gencon	768m	2550m	High	4167
34	J9	1.7	gencon	768m	3400m	High	4223
35	J9	1.7	gencon	768m	4250m	High	4306
36	J9	1.7	gencon	768m	5100m	High	4279
37	J9	1.7	gencon	768m	5950m	High	4561
38	J9	1.7	gencon	768m	6800m	High	4383
39	J9	1.7	optthruput	768m	4250m	High	4853
40	J9	1.7	optavgpause	768m	5100m	High	4903
41	J9	1.7	balanced	768m	5950m	High	4923
42	J9	1.7	metronome	768m	6800m	High	N/A

Table 7.2: GC tests 1.2 - test configurations and execution times.

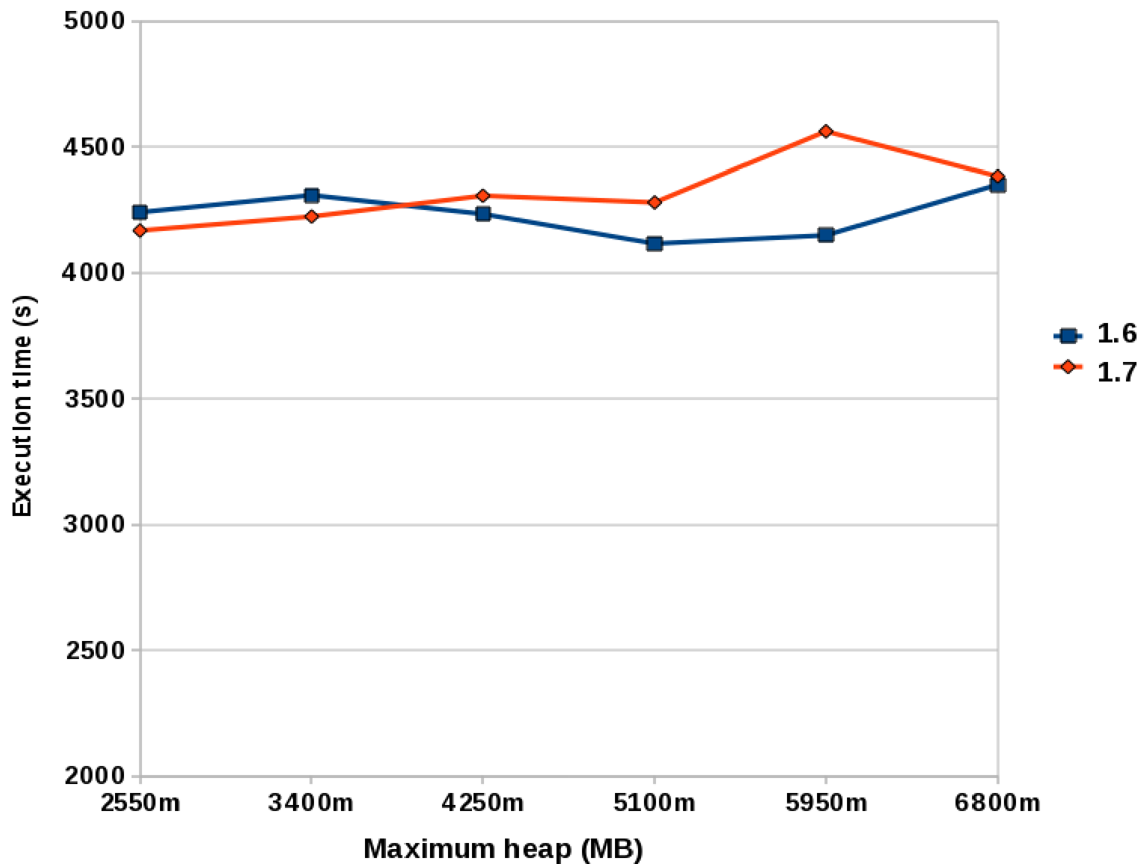

Figure 7.3: Execution time of high density configuration of GASS, with the gencon policy, on the J9 JVM 1.6 and 1.7.

management of the generational spaces within the Java heap, as these spaces play a crucial role in the GC performance. The GC policies and the tuning options that we used in this test campaign are given in Tables 11.5 and 11.6 in Appendix C.

This test campaign consisted of a large number of tests, in order to break the entire set of tests into smaller groups, we use the term *testset* to identify a group of similar tests. This test campaign involved four testsets, each of which are explained in the following list.

- Testset1 — we set the initial and maximum heap sizes to the minimum heap required to run the application successfully, and we ran tests using each of the GC policies without specifying any additional options. Testset1 can be considered as a reference test.
- Testset2 — we took Testset1 as the starting point, we then repeatedly increased the maximum heap size in a series of increments, until the maximum heap size was twice the minimum heap size.
- Testset3 — we set the initial heap size to the minimum heap required to run the application successfully, and we set the maximum heap size to double this value. We specified the heap occupancy levels that should trigger heap expansion and shrinking over a range of values, from 10% to 90%.
- Testset4 — we set the minimum heap size to the minimum heap required to run the application successfully, and we set the maximum heap size to 43% larger than the minimum heap required to run the application successfully. Setting the maximum heap size to this value ensures that, even when the application is using the maximum amount of memory that it will require during the its execution, the heap occupancy level does not exceed

70%, as is recommended by both the HotSpot and J9 JVMs. We set the relative sizes of the old and new spaces, using a range of ratios from 20:1 to 1:20.

All tests were executed ten times, and average times were calculated.

7.4.1 Results

In the following three subsections, we gave the results of performing the 4 testsets described in the previous section, on each of the three applications used in this study.

7.4.1.1 GASS

We found that the minimum heap size that is required to run this configuration of GASS is 2304 MBs. This figure is used as the initial heap for most of the GASS tests in this test campaign. As expected, the *gencon* policy provides the best throughput, as shown in Table 7.3. The reference execution time of GASS for this configuration, without using any tuning options, is 4487 seconds.

Test	**JVM**	**Version**	**Initial heap**	**Max. heap**	**GC Policy**	**Average time (s)**
1	J9	1.6	2304	2304	optthruput	5728
2	J9	1.6	2304	2304	gencon	4487
3	J9	1.6	2304	2304	optavgpause	6206
4	J9	1.6	2304	2304	subpool	5421

Table 7.3: GASS — Testset1.

Testset2, given in Table 7.4, revealed that setting the maximum heap size to be equal to the minimum heap size produces the best performance. Note that, this value is the actual minimum heap size that is necessary to allow the application to finish successfully. Therefore, the best performance is achieved using the absolute minimum amount of memory. The worst performance was from the test with the largest maximum heap size.

Test	**JVM**	**Version**	**Initial heap**	**Max. heap**	**GC Policy**	**Average time (s)**
1	J9	1.6	2304	2304	gencon	4487
2	J9	1.6	2304	2560	gencon	5870
3	J9	1.6	2304	2816	gencon	6031
4	J9	1.6	2304	3072	gencon	5873
5	J9	1.6	2304	3328	gencon	6042
6	J9	1.6	2304	3584	gencon	6057
7	J9	1.6	2304	3840	gencon	5863
8	J9	1.6	2304	4096	gencon	5482
9	J9	1.6	2304	4352	gencon	5650
10	J9	1.6	2304	4608	gencon	6495

Table 7.4: GASS — Testset2.

The option *-Xminf* specifies the minimum amount of heap space that should be free after a collection has been performed. If the actual amount of free space is less than this value,

then the JVM will attempt to expand the heap. Likewise, the option *-Xmaxf* specifies the maximum amount of heap space that should be free after a collection has been performed. If the actual amount of free space is more than this value, then the JVM will attempt to shrink the heap. The results of Testset3 for the GASS application, given in Table 7.5, showed that the best performance was achieved when *-Xmaxf* was set to 50% - 60%. This means that the JVM should shrink the heap size if the amount of free space after a collection exceeds 50% - 60% percent. The observed performance almost reaches the same performance as the reference performance.

Test	**JVM**	**Version**	**Initial heap**	**Max. heap**	**-Xminf**	**-Xmaxf**	**GC Policy**	**Average time (s)**
1	J9	1.6	2304	4608	10	-	gencon	5354
2	J9	1.6	2304	4608	20	-	gencon	5744
3	J9	1.6	2304	4608	30	-	gencon	6047
4	J9	1.6	2304	4608	40	-	gencon	5932
5	J9	1.6	2304	4608	50	-	gencon	5752
6	J9	1.6	2304	4608	60	-	gencon	N/A
7	J9	1.6	2304	4608	70	-	gencon	N/A
8	J9	1.6	2304	4608	80	-	gencon	N/A
9	J9	1.6	2304	4608	90	-	gencon	N/A
10	J9	1.6	2304	4608	-	10	gencon	N/A
11	J9	1.6	2304	4608	-	20	gencon	N/A
12	J9	1.6	2304	4608	-	30	gencon	N/A
13	J9	1.6	2304	4608	-	40	gencon	4788
14	J9	1.6	2304	4608	-	50	gencon	4553
15	J9	1.6	2304	4608	-	60	gencon	4574
16	J9	1.6	2304	4608	-	70	gencon	5080
17	J9	1.6	2304	4608	-	80	gencon	5577
18	J9	1.6	2304	4608	-	90	gencon	4972

Table 7.5: GASS - Testset3.

In Testset4, given in Table 7.6, we kept the size of the heap constant across all tests, and we varied the sizes of the young space and the old space relative to each other, using the *-Xmn* and *-Xmo* options. Ranging in sizes from a young space that constitutes 5% of the heap, to a young space that constitutes 95% of the heap, and *vice versa* for the old space. The results, plotted in Figure 7.4, show that for this configuration of GASS, the best performance is achieved with a young space and an old space of around 50% each. There is a clear and significant decrease in performance, as the sizes of the spaces are set to more extreme values. However, the decrease in performance is not as much as one might expect. The longest execution time, which involved a young space that constituted just 5% of the heap, was only 33% slower than the fastest execution, where the young space constituted 50% of the heap.

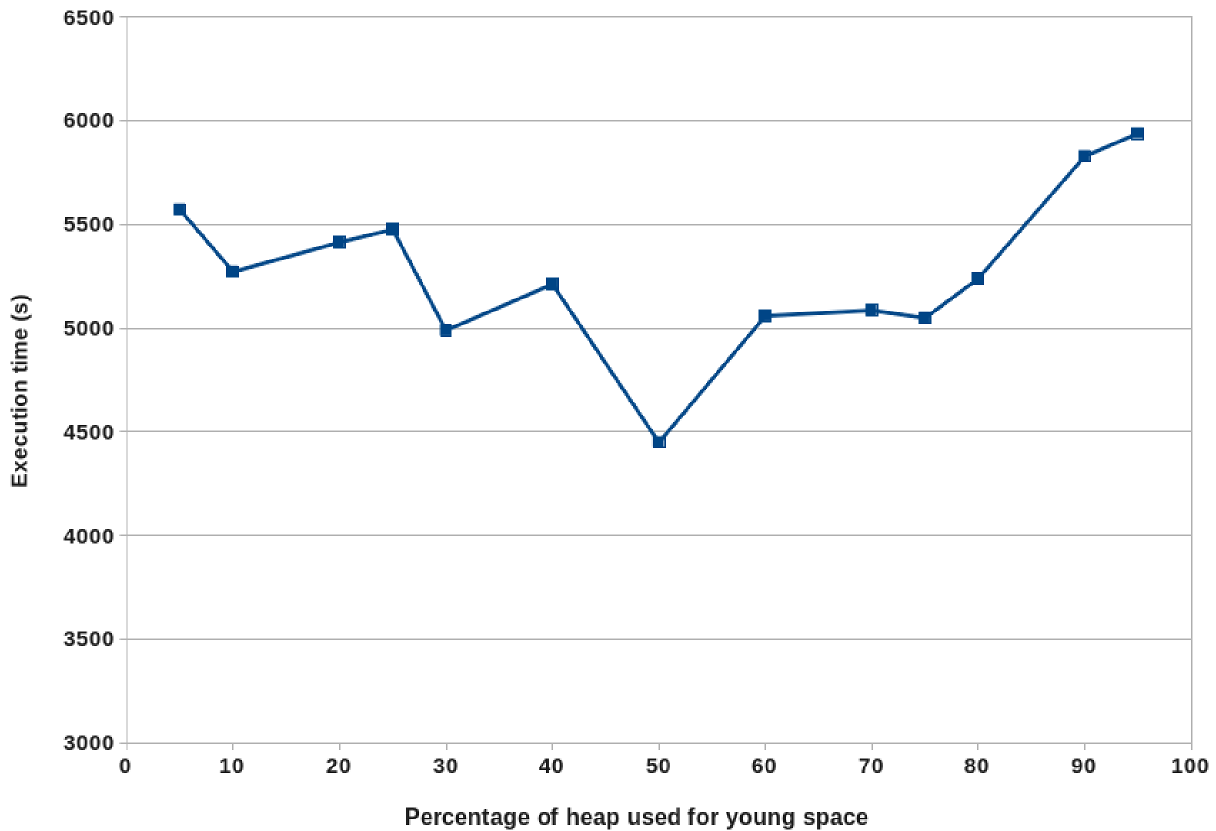

Figure 7.4: GASS - Testset4 - change in execution times as the percentage of heap used for the young space is varied.

Test	JVM	Version	Initial heap	Max. heap	-Xmn	-Xmo	GC Policy	Average time (s)
1	J9	1.6	3294	3294	165 (5%)	3129 (95%)	gencon	5573
2	J9	1.6	3294	3294	329 (10%)	2965 (90%)	gencon	5273
3	J9	1.6	3294	3294	659 (20%)	2635 (80%)	gencon	5415
4	J9	1.6	3294	3294	824 (25%)	2470 (75%)	gencon	5477
5	J9	1.6	3294	3294	988 (30%)	2306 (70%)	gencon	4990
6	J9	1.6	3294	3294	1318 (40%)	1976 (60%)	gencon	5213
7	J9	1.6	3294	3294	1647 (50%)	1647 (50%)	gencon	4450
8	J9	1.6	3294	3294	1976 (60%)	1318 (40%)	gencon	5060
9	J9	1.6	3294	3294	2306 (70%)	988 (30%)	gencon	5087
10	J9	1.6	3294	3294	2470 (75%)	824 (25%)	gencon	5050
11	J9	1.6	3294	3294	2635 (80%)	659 (20%)	gencon	5240
12	J9	1.6	3294	3294	2965 (90%)	329 (10%)	gencon	5830
13	J9	1.6	3294	3294	3129 (95%)	165 (5%)	gencon	5940

Table 7.6: GASS - Testset4.

7.4.1.2 IDU-ipd

The results of performing the four testsets on the IDU-ipd application are given in this subsection. Testset 1 for IDU-ipd is given in Table 7.7. As is the case for GASS, the best performing GC policy for this configuration of IDU-crb is *gencon*. Although, we found that it required more heap space than the other policies in order for the application to finish successfully.

Test	JVM	Version	Initial heap	Max. heap	GC Policy	Average time (s)
1	J9	1.6	512	512	optthruput	2697
2	J9	1.6	640	640	gencon	2467
3	J9	1.6	512	512	optavgpause	2981
4	J9	1.6	512	512	subpool	2800

Table 7.7: IDU-ipd - Testset1.

The striking aspect of the results from Testset2, Testset3 and Testset4 for the IDU-ipd application, is how little variation there is in the execution times, as shown in Tables 7.8, 7.9 and 7.10. Even in Testset4, when the young and the old spaces were set to extreme sizes relative to each other, the execution times did not change very much. Only when the young space was set to 90%, or more, of the heap, did the performance change significantly, as shown in Figure 7.5. The results suggest that, for this configuration of IDU-ipd, GC is playing a relatively minor role in determining the execution times.

Test	JVM	Version	Initial heap	Max. heap	GC Policy	Average time (s)
1	J9	1.6	640	640	gencon	2467
2	J9	1.6	640	704	gencon	2472
3	J9	1.6	640	768	gencon	2454
4	J9	1.6	640	832	gencon	2474
5	J9	1.6	640	896	gencon	2458
6	J9	1.6	640	960	gencon	2497
7	J9	1.6	640	1024	gencon	2490
8	J9	1.6	640	1088	gencon	2492
9	J9	1.6	640	1152	gencon	2488
10	J9	1.6	640	1216	gencon	2462
11	J9	1.6	640	1280	gencon	2438

Table 7.8: IDU-ipd - Testset2.

Test	JVM	Version	Initial heap	Max. heap	-Xminf	-Xmaxf	GC Policy	Average time (s)
1	J9	1.6	640	1280	10	-	gencon	2458
2	J9	1.6	640	1280	20	-	gencon	2639
3	J9	1.6	640	1280	30	-	gencon	2704
4	J9	1.6	640	1280	40	-	gencon	2716
5	J9	1.6	640	1280	50	-	gencon	2694
6	J9	1.6	640	1280	60	-	gencon	N/A
7	J9	1.6	640	1280	70	-	gencon	N/A
8	J9	1.6	640	1280	80	-	gencon	N/A
9	J9	1.6	640	1280	90	-	gencon	N/A
10	J9	1.6	640	1280	-	10	gencon	N/A
11	J9	1.6	640	1280	-	20	gencon	N/A
12	J9	1.6	640	1280	-	30	gencon	N/A
13	J9	1.6	640	1280	-	40	gencon	2467
14	J9	1.6	640	1280	-	50	gencon	2495
15	J9	1.6	640	1280	-	60	gencon	2496
16	J9	1.6	640	1280	-	70	gencon	2482
17	J9	1.6	640	1280	-	80	gencon	2444
18	J9	1.6	640	1280	-	90	gencon	2486

Table 7.9: IDU-ipd - Testset3.

Test	JVM	Version	Initial heap	Max. heap	-Xmn	-Xmo	GC Policy	Average time (s)
1	J9	1.6	915	915	46 (5%)	869 (95%)	gencon	2490
2	J9	1.6	915	915	92 (10%)	823 (90%)	gencon	2502
3	J9	1.6	915	915	183 (20%)	732 (80%)	gencon	2469
4	J9	1.6	915	915	229 (25%)	686 (75%)	gencon	2507
5	J9	1.6	915	915	275 (30%)	640 (70%)	gencon	2477
6	J9	1.6	915	915	366 (40%)	549 (60%)	gencon	2482
7	J9	1.6	915	915	458 (50%)	457 (50%)	gencon	2459
8	J9	1.6	915	915	549 (60%)	366 (40%)	gencon	2462
9	J9	1.6	915	915	641 (70%)	274 (30%)	gencon	2459
10	J9	1.6	915	915	686 (75%)	229 (25%)	gencon	2470
11	J9	1.6	915	915	732 (80%)	183 (20%)	gencon	2510
12	J9	1.6	915	915	824 (90%)	91 (10%)	gencon	2680
13	J9	1.6	915	915	869 (95%)	46 (5%)	gencon	2685

Table 7.10: IDU-ipd - Testset4.

7.4.1.3 IDU-crb

Testset1 for IDU-crb is given in Table 7.11. Again, the *gencon* policy performed best, although it did require more heap space in order to allow the application to finish successfully. There is a huge difference between the performance of each of the policies. The execution time when using *optavgpause* is 143% longer than when using *gencon*.

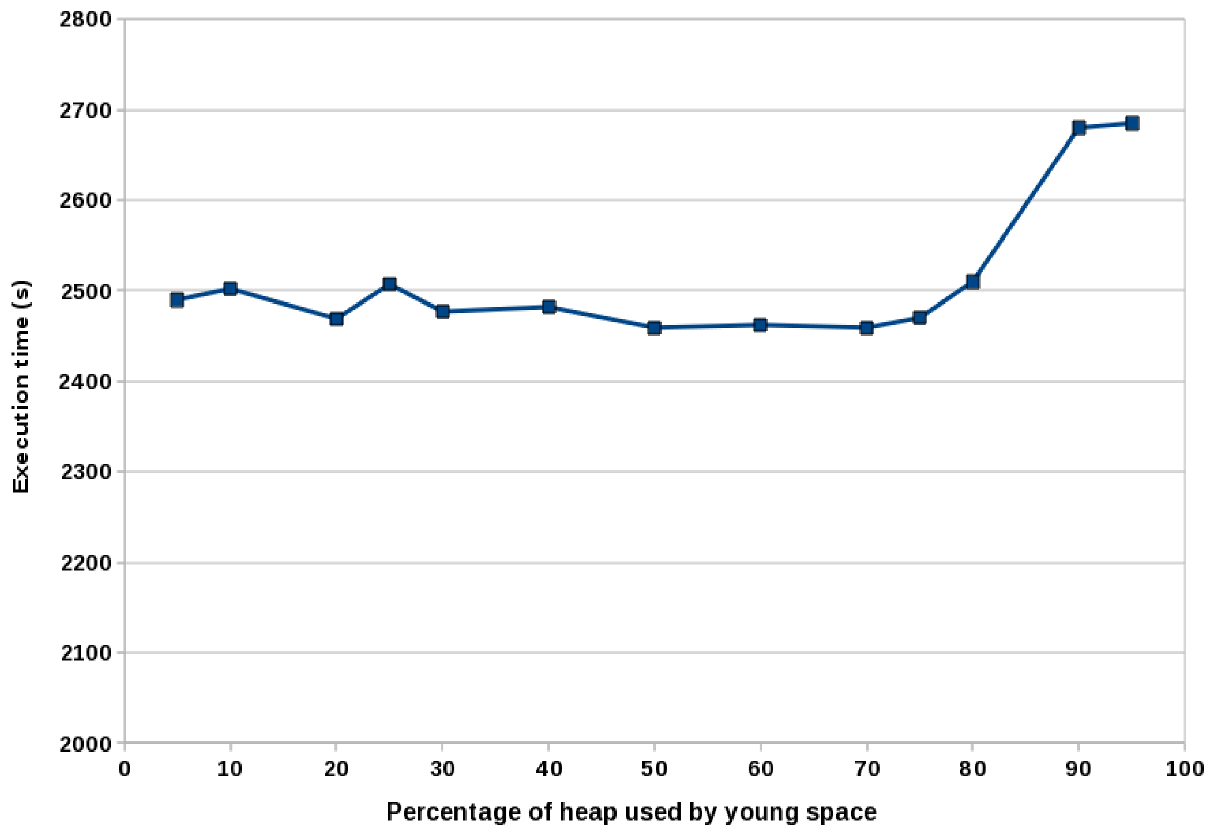

Figure 7.5: IDU-ipd - Testset4 - change in execution times as the percentage of the heap used for the young space is varied.

Test	JVM	Version	Initial heap	Max. heap	GC Policy	Average time (s)
1	J9	1.6	4352	4352	optthruput	1651
2	J9	1.6	5120	5120	gencon	1073
3	J9	1.6	4352	4352	optavgpause	2612
4	J9	1.6	4352	4352	subpool	1849

Table 7.11: IDU-crb - Testset1.

In Testset2, for IDU-crb, we tested two GC policies: *gencon* and *optthruput*. The results are given in Table 7.12, and are plotted in Figure 7.6. Unlike the other applications, IDU-crb benefits from a heap that is significantly larger than the minimum heap required to run the application successfully. This suggests that it is more memory intensive than the other applications. The best performance of any of the IDU-crb tests was obtained in Testset3, given in Table 7.13, when the maximum heap size was set to 7424MBs, and the *-Xminf* option was set to 50%.

Test	JVM	Version	Initial heap	Max. heap	GC Policy	Average time (s)
1	J9	1.6	4352	4352	optthruput	1651
2	J9	1.6	4352	4608	optthruput	1369
3	J9	1.6	4352	4864	optthruput	1236
4	J9	1.6	4352	5120	optthruput	1130
5	J9	1.6	4352	5376	optthruput	1098
6	J9	1.6	4352	5632	optthruput	1087
7	J9	1.6	4352	5888	optthruput	1045
8	J9	1.6	4352	6144	optthruput	1004
9	J9	1.6	4352	6400	optthruput	992
10	J9	1.6	4352	6656	optthruput	952
11	J9	1.6	4352	6912	optthruput	980
12	J9	1.6	4352	7168	optthruput	1003
13	J9	1.6	4352	7424	optthruput	1042
14	J9	1.6	5120	5120	gencon	1073
15	J9	1.6	5120	5376	gencon	877
16	J9	1.6	5120	5632	gencon	823
17	J9	1.6	5120	5888	gencon	821
18	J9	1.6	5120	6144	gencon	812
19	J9	1.6	5120	6400	gencon	809
20	J9	1.6	5120	6656	gencon	825
21	J9	1.6	5120	6912	gencon	801
22	J9	1.6	5120	7168	gencon	803
23	J9	1.6	5120	7424	gencon	798

Table 7.12: IDU-crb - Testset2.

Test	JVM	Version	Initial heap	Max. heap	-Xminf	-Xmaxf	GC Policy	Average time (s)
1	J9	1.6	5120	7424	10	-	gencon	808
2	J9	1.6	5120	7424	20	-	gencon	814
3	J9	1.6	5120	7424	30	-	gencon	807
4	J9	1.6	5120	7424	40	-	gencon	807
5	J9	1.6	5120	7424	50	-	gencon	779
6	J9	1.6	5120	7424	60	-	gencon	N/A
7	J9	1.6	5120	7424	70	-	gencon	N/A
8	J9	1.6	5120	7424	80	-	gencon	N/A
9	J9	1.6	5120	7424	90	-	gencon	N/A
10	J9	1.6	5120	7424	-	10	gencon	N/A
11	J9	1.6	5120	7424	-	20	gencon	N/A
12	J9	1.6	5120	7424	-	30	gencon	N/A
13	J9	1.6	5120	7424	-	40	gencon	802
14	J9	1.6	5120	7424	-	50	gencon	809
15	J9	1.6	5120	7424	-	60	gencon	853
16	J9	1.6	5120	7424	-	70	gencon	798
17	J9	1.6	5120	7424	-	80	gencon	814
18	J9	1.6	5120	7424	-	90	gencon	829

Table 7.13: IDU-crb - Testset3.

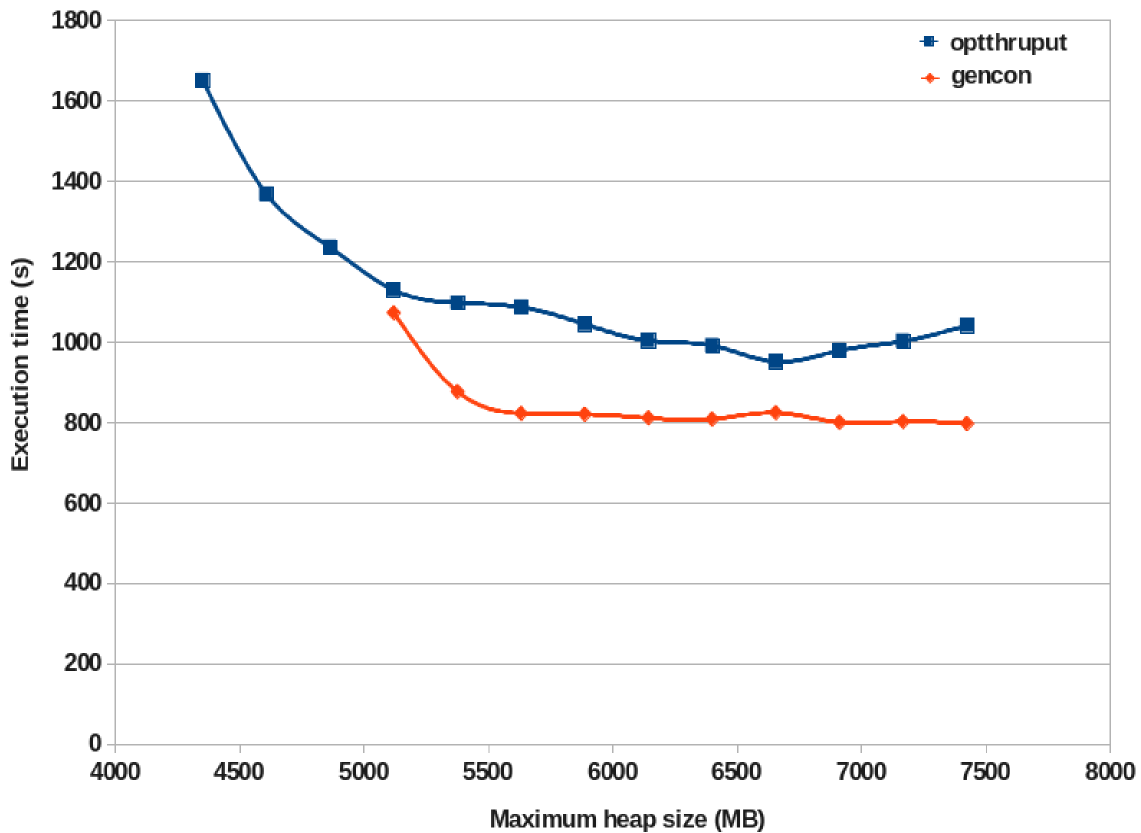

Figure 7.6: IDU-crb - Testset2 - change in execution times as the maximum heap size is increased, for two GC policies: *optthruput* and *gencon*.

Unlike the other two applications, the performance of IDU-crb is very sensitive to changes of (the setting of) the sizes of the young and the old generational spaces. When the size of the old space is set to a value less than 50% of the size of the heap, then the performance of the application changes dramatically, and in fact, results in OOM errors in some cases.

Test	JVM	Version	Initial heap	Max. heap	-Xmn	-Xmo	GC Policy	Average time (s)
1	J9	1.6	7322	7322	366 (5%)	6956 (95%)	gencon	856
2	J9	1.6	7322	7322	732 (10%)	6590 (90%)	gencon	834
3	J9	1.6	7322	7322	1464 (20%)	5858 (80%)	gencon	789
4	J9	1.6	7322	7322	1831 (25%)	5491 (75%)	gencon	779
5	J9	1.6	7322	7322	2197 (30%)	5125 (70%)	gencon	788
6	J9	1.6	7322	7322	2929 (40%)	4393 (60%)	gencon	784
7	J9	1.6	7322	7322	3661 (50%)	3661 (50%)	gencon	1404
8	J9	1.6	7322	7322	4393 (60%)	2929 (40%)	gencon	OOM
9	J9	1.6	7322	7322	5125 (70%)	2197 (30%)	gencon	1345
10	J9	1.6	7322	7322	5492 (75%)	1830 (25%)	gencon	2389
11	J9	1.6	7322	7322	5858 (80%)	1464 (20%)	gencon	2326
12	J9	1.6	7322	7322	6590 (90%)	732 (10%)	gencon	OOM
13	J9	1.6	7322	7322	6956 (95%)	366 (5%)	gencon	OOM

Table 7.14: IDU-crb - Testset4.

7.4.2 Analysis

It is clear that the IDU-crb application is much more sensitive to changes to the GC options used, than the other two applications. The performance of GASS also changed significantly due to the options used. However, the performance of IDU-ipd changed very little, regardless of the options used. For GASS and IDU-ipd, the best performance was obtained by not specifying any tuning options at all, highlighting the fact that modern garbage collectors provide very good *out-of-the-box* performance. For IDU-crb, the best performance was achieved by setting certain tuning options, and a reduction of up to 27% in execution time was achieved. This demonstrates the fact that, in some cases, it is possible to find a set of GC options that provide better performance than the garbage collector can provide on its own. In these tests, we were simply trying different options, in the *hope* that we would find a set of options that match the memory usage behaviour and needs of the application, and hence provide better performance. In the next section we discuss profile-directed searching for well performing GC options.

7.5 Profiling for GC

The search for a better performing set of GC options (policy and tuning options) can be described as a search of a multi-dimensional space. Due to the number of GC options, this search space is extremely large, and it is not practical to test every possible combination of options. Additionally, due to the fact that some options can interfere with each other, or can have contradictory effects, the approach of *blindly* trying different GC options is unlikely to uncover a set of GC options that perform well for all configurations of an application. However, if a profile of the memory usage behaviour of the target application is obtained, as well as an understanding of the how the GC options can effect an execution of an application, then a profile-directed search of the options may be performed. In this section, we discuss how a *memory usage profile* for an application may be obtained, and how that can then be used to select particular GC options.

All JVMs may be instructed, using command-line options, to produce an output file which logs the GC activity during an execution. In these tests, we used the J9 JVM, therefore the command-line option used was *-Xverbosegclog:<GC log filename>*. The same three DPAC systems that were used in *GC test 2.1*, namely GASS, IDU-ipd and IDU-crb, were executed on MareNostrum II, with this option added, and a GC log file was obtained. In all three cases, the minimum and maximum heap sizes were set at 1024MB 7168MB respectively, therefore, a large degree of heap resizing was permitted during these executions.

The GC log files produced by the J9 JVM may be viewed using a number of tools, we used the GCMV, which is part of the IBM Support Assistant Workbench (see Section 6.3.4). In Table 7.15, a brief summary of some of the characteristics of these executions are given. The length of these executions were very different, therefore it does not make sense to compare absolute values between the executions, however the mean and percentage values may be compared.

Description	GASS	IDU-ipd	IDU-crb
Length of execution (s)	4189	2486	442
Largest memory request (bytes)	4651432	5373976	43721064
Mean GC pause (ms)	828	90.4	2858
Mean heap unusable due to fragmentation (MB)	15.0	2.58	12.3
Mean interval between GCs (minutes)	0.19	0.06	0.28
No. collections	361	635	25
No. of collections triggered by allocation failure	359	635	25
Percentage of time spent in GC pauses (%)	7.18	2.35	17.57
Percentage of time spent unpaused (%)	92.82	97.65	82.43
Rate of GC (MB/minutes)	9693	13048	5812

Table 7.15: Profiling of GC activity.

Comparing the summary of each execution, we can see very significant differences between these applications. In particular, the mean GC pause times vary widely between the applications, the mean GC pause times for IDU-crb being 2.8 seconds. Such long pause times would be unacceptable for interactive applications, that require almost immediate responses in real-time. Not surprisingly, IDU-crb also spent the highest percentage of the overall execution time in GC pauses — 17.57%. This profile also shows that IDU-crb is creating some relatively large objects, the size of the largest memory request was 43MBs.

Figures 7.7 and 7.8, also generated by GCMV, plot four properties of the executions of GASS and IDU-crb respectively. The properties shown are: used heap (after collection), free heap (before collection), free heap (after collection) and the overall heap size. Firstly, these figures show the resizing of the heap that happened during the course of these executions (blue line). We can see that a significant amount of heap resizing was performed, as the heap size increases a lot in both cases, from the initial size. In the case of GASS, the JVM also reduces the heap on many occasions.

The used heap (after collection) (grey line), shows the total number of used space in the heap, after each collection. We can see that this value reaches a fairly stable size in IDU-crb after 10 - 12 minutes, and it reaches a stable size in GASS after 1 minute. This value is very important in terms of GC tuning, as it represents the amount of data that is surviving each GC collection. We could say that an application has reached a steady-state (in terms of GC), once the size of the used heap (after collection) is remaining relatively constant. We can see that, for these configurations of GASS and IDU-crb, this value is roughly 1.75GB and 4GB respectively.

If a generational GC policy is being used, one general rule of thumb to achieve good performance, is to set the size of the old space to the value of the *steady-state used bytes*. This is verified in the Testset4 for both GASS and IDU-crb, described in the previous section. Applications which create a relatively large number of short-lived objects benefit from having a larger young space, however applications that create a relatively large number of long-lived objects benefit from a larger old space.

As well as the options related to the size and management of the heap that we mentioned here, modern JVMs offer many other GC tuning options. An experience from executing DPAC systems at the DPCB highlights the importance of being aware of the default values that are being assigned to certain options. Recently, large scale executions of GASS were executed on Pirineus — a high-performance shared-memory machine with 6TB of memory, and 1344 cores (see Appendix A). The executions were not providing the expected level of performance. The

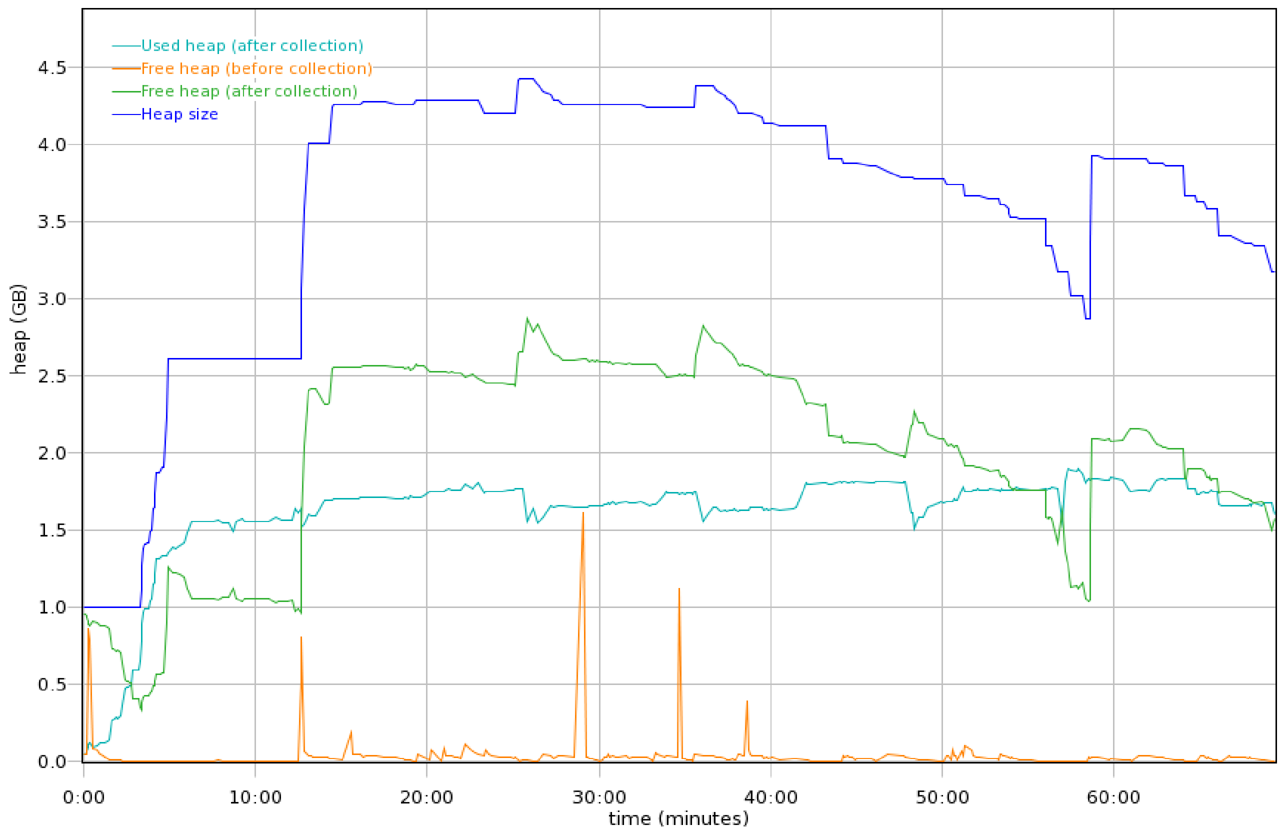

Figure 7.7: Heap size and the number of used bytes after each collection, during the execution of GASS.

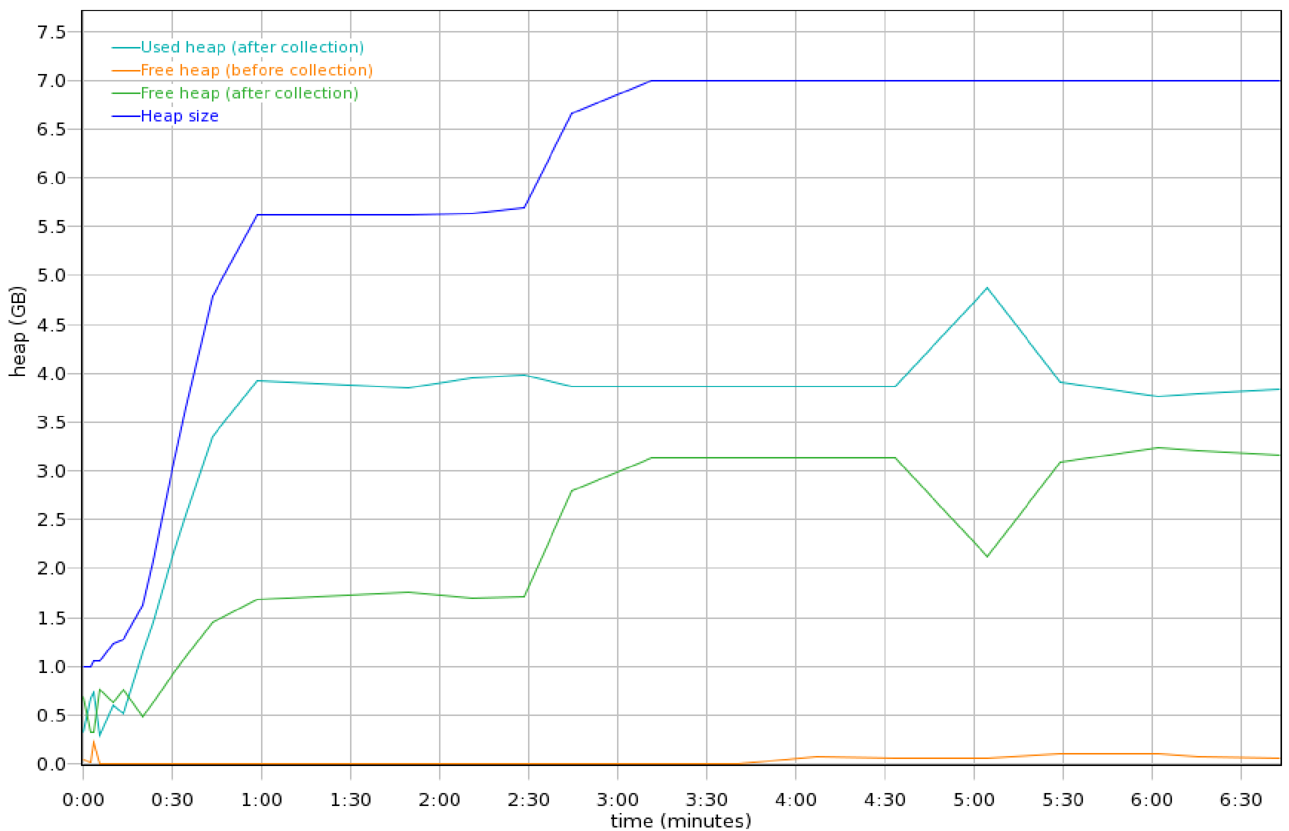

Figure 7.8: Heap size and the number of used bytes after each collection, during the execution of IDU-crb.

JVM that was being used was the HotSpot 1.7 JVM, and the GC policy was the parallel policy. An investigation revealed that the problem in performance was being caused by the default value of the number of parallel GC threads to be created. For environments with more than eight cores, the default number of parallel GC threads is calculated as 62.5% of the number return by the method *Runtime.available Processors()*. In the case of Pirineus, the value return from this method was 1344, therefore the number of GC threads created was 840. Obviously, the default value does not suit machines such as Pirineus, especially if the machine is shared with other users, and therefore any particular user is only using a subset of the total number of cores. Setting this value to 4, with the option *-XX:ParallelGCThreads*, resulted in roughly a 75% decrease in the execution times of GASS. Although this is an extreme example, it highlights the fact that the default GC options may not be the best.

7.6 Memory Usage Model (MUM)

As part of our work on understanding the memory usage patterns of DPAC systems, we developed an application, that we call MUM, which can be used to model the memory usage behaviour of other (real) applications, to some extent. As well as memory usage, MUM can also model the CPU usage and the threading behaviour of applications. Any model is of course only an approximation. The usefulness of the MUM application depends on how closely an instance of MUM reflects the modelled application, and on the modelling requirements.

7.6.1 Design

The primary objective of MUM is to simulate the memory usage behaviour of real applications, in particular, their creation of objects, and the lifecycle of those objects. During the initial design of MUM, two simple object characteristics were identified as being particularly important, these are:

1. Object size.
2. Object lifetime.

For both of these characteristics, three coarse-grained categories were defined. Therefore, using this approach, all objects may be characterised using the matrix shown in Table 7.16.

Short-lived, small size	Short-lived, medium size	Short-lived, large size
Medium-lived, small size	Medium-lived, medium size	Medium-lived, large size
Long-lived, small size	Long-lived, medium size	Long-lived, large size

Table 7.16: Object characterisation.

An instance of MUM is configured using a *characteristics file*. We use the term *characteristics file*, instead of the term *configuration file*, because this file specifies the characteristics that the execution should exhibit. A characteristics file corresponds to the memory usage profile of a real application, that could be obtain following the profiling of that application. An example of a characteristics file is given in Figure 7.10.

An execution of MUM consists of two main interleaved activities: object processing, and static processing. Object processing involves the creation, and updating of objects. Static processing

simply involves the execution of some *number crunching*, but does not involve processing the created objects. To prevent the JIT compiler from applying Dead-Code Elimination (DCE) optimisations to the static processing code, the static processing methods return a value which is later used.

The memory usage behaviour of real applications varies greatly from application to application. Studies has shown that executions are typically composed of distinct phases. Some phases involve the creation of many objects, while other phases might involve very little object creation, and instead may involve a large amount of computations or I/O activity.

MUM incorporates the idea of distinct phases into its design. Each phase is characterised by two properties: the length of the phase, and the percentage of the phase that involves object processing. MUM can be configured to perform any number of phases, and the percentage of each phase that is dedicated to static processing can be set, the percentage of the phase that is dedicated to object processing is determined from this value.

Although we set the number of object sizes and lifetimes to three, the actual values corresponding to each category is configurable. We can configure the number of bytes corresponding to each of the object size categories, and we can configure the *age* that corresponds to each of the possible object lifetimes. The ratio of the number of objects, from each of the nine categories that are created during an execution is configurable. The actual number of objects from each category that are created during an execution is determined by the length of the execution, and by the percentage of time performing object processing.

MUM is configured by a single XML-based characteristics file, an example of which is given in Figure 7.10. A description of all the fields in this file are given in Appendix D. In this characteristics file, four phases are defined. In the first phase, which consists of 1000000 iterations of the *phase loop*, 10% of the iterations are spent performing static processing, therefore 90% of these iterations are consumed by object processing. The third phase, which involves twice as many iterations as the first phase, mostly involves static processing. The ability of being able to define any number of phases, of varying memory usage behaviour, allows for the simulation of object creation phases, which are typical of many real applications.

7.6.2 Output

MUM uses the Simple Logging Facade for Java (SLF4J) logging framework to produce log files. The use of this framework brings the advantage of being able to easily increase or decrease the level of logging produced. MUM itself, contains a number of counters which are used to record the amount of time spent within each part of the execution. This information includes the length of time spent within each phase of the execution, and the total amount of time spent performing static processing, as well as object processing. An extract from a MUM log file is given in Figure 7.9. Amongst the information logged are: the time spent in each phase, the number of objects of each category that were created, and the total time spent in object processing as well as static processing.

```
11:44:45 INFO    MemoryUsageModel    - Object creation finished. Post-processsing results are:
11:44:45 INFO    MemoryUsageModel    - Phase information:
11:44:45 INFO    MemoryUsageModel    - Phase 0 - total time: 2199.562 milliseconds
11:44:45 INFO    MemoryUsageModel    - Phase 1 - total time: 4765.775 milliseconds
11:44:45 INFO    MemoryUsageModel    - Phase 2 - total time: 1100.568 milliseconds
11:44:45 INFO    MemoryUsageModel    - Phase 3 - total time: 4594.493 milliseconds
11:44:45 INFO    MemoryUsageModel    - -------------------------
11:44:45 INFO    MemoryUsageModel    - Number of iterations performing each type of processing:
11:44:45 INFO    MemoryUsageModel    - Total number static processing iterations: 3525000
11:44:45 INFO    MemoryUsageModel    - Total number object processing iterations: 1975000
11:44:45 INFO    MemoryUsageModel    - -------------------------
11:44:45 INFO    MemoryUsageModel    - Objects created information:
11:44:45 INFO    MemoryUsageModel    - Total number of objects created: 465160
11:44:45 INFO    MemoryUsageModel    - Number of objects created: [0,0]: 219465
11:44:45 INFO    MemoryUsageModel    - Number of objects created: [0,1]: 73135
11:44:45 INFO    MemoryUsageModel    - Number of objects created: [0,2]: 20875
11:44:45 INFO    MemoryUsageModel    - Number of objects created: [1,0]: 81935
11:44:45 INFO    MemoryUsageModel    - Number of objects created: [1,1]: 27305
11:44:45 INFO    MemoryUsageModel    - Number of objects created: [1,2]: 8325
11:44:45 INFO    MemoryUsageModel    - Number of objects created: [2,0]: 22220
11:44:45 INFO    MemoryUsageModel    - Number of objects created: [2,1]: 9255
11:44:45 INFO    MemoryUsageModel    - Number of objects created: [2,2]: 2645
11:44:45 INFO    MemoryUsageModel    - -------------------------
11:44:45 INFO    MemoryUsageModel    - Time spent in processing categories:
11:44:45 INFO    MemoryUsageModel    - Main Loop execution time: 12660.585 milliseconds
11:44:45 INFO    MemoryUsageModel    - Total time static processing: 6470.760 milliseconds
11:44:45 INFO    MemoryUsageModel    - Total time object processing: 3341.049 milliseconds
11:44:45 INFO    MemoryUsageModel    - Total time static modifying: 730.834 milliseconds
11:44:45 INFO    MemoryUsageModel    - Total time object creation: 447.531 milliseconds
11:44:45 INFO    MemoryUsageModel    - -------------------------
11:44:45 INFO    MemoryUsageModel    - Final value returned from static processing: 352855878
```

Figure 7.9: An extract from a log file generated by MUM.

7.6.3 Application characterisation

In order to create a MUM classification file that represents the memory usage behaviour of a particular application, then the application must be profiled in terms of the characteristics listed below:

1. CPU usage
2. Number of objects created
3. Lifetime of these objects
4. Size of objects

7.6.4 Uses

MUM should not be viewed as solely a tool for modelling memory usage behaviour, instead we propose that it could be useful for a range of purposes. One possible use for MUM is the creation of benchmarks consisting of distinct phases. Benchmarks are widely used in HPC, for the purposes of assessing the performance of both hardware and software. One problem that benchmarks have traditionally suffered from, is a lack of realistic behaviour, that is representative of real applications. Additionally, large scientific applications, such as those used in this study, can be cumbersome to install and run. One issue which can complicate the execution of applications are dependencies on external entities. Such dependencies might include other applications or libraries, and the necessity to read or write large volumes of data to storage

```
<mum-config>
    <ratio_shortlived_small_sized>50</ratio_shortlived_small_sized>
    <ratio_shortlived_medium_sized>50</ratio_shortlived_medium_sized>
    <ratio_shortlived_large_sized>50</ratio_shortlived_large_sized>
    <ratio_mediumlived_small_sized>20</ratio_mediumlived_small_sized>
    <ratio_mediumlived_medium_sized>20</ratio_mediumlived_medium_sized>
    <ratio_mediumlived_large_sized>20</ratio_mediumlived_large_sized>
    <ratio_longlived_small_sized>5</ratio_longlived_small_sized>
    <ratio_longlived_medium_sized>5</ratio_longlived_medium_sized>
    <ratio_longlived_large_sized>5</ratio_longlived_large_sized>
    <size_small_obj>100</size_small_obj>
    <size_medium_obj>200</size_medium_obj>
    <size_large_obj>300</size_large_obj>
    <lifetime_shortlived_obj>1</lifetime_shortlived_obj>
    <lifetime_mediumlived_obj>1</lifetime_mediumlived_obj>
    <lifetime_longlived_obj>1</lifetime_longlived_obj>
    <sleep_after_shortlived_created>0</sleep_after_shortlived_created>
    <sleep_after_mediumlived_created>0</sleep_after_mediumlived_created>
    <sleep_after_longlived_created>0</sleep_after_longlived_created>
    <warmup_its_static_proc>1000000</warmup_its_static_proc>
    <warmup_its_obj_proc>1000000</warmup_its_obj_proc>
    <phases>
        <phase>
            <total_num_its>1000000</total_num_its>
            <percent_its_static_proc>10</percent_its_static_proc>
        </phase>
        <phase>
            <total_num_its>2000000</total_num_its>
            <percent_its_static_proc>80</percent_its_static_proc>
        </phase>
        <phase>
            <total_num_its>500000</total_num_its>
            <percent_its_static_proc>5</percent_its_static_proc>
        </phase>
        <phase>
            <total_num_its>2000000</total_num_its>
            <percent_its_static_proc>90</percent_its_static_proc>
        </phase>
    </phases>
    <num_its_per_obj_proc>100</num_its_per_obj_proc>
    <seed_value_obj_construction>31</seed_value_obj_construction>
    <obj_string>Hello</obj_string>
    <initial_sleep_s>1</initial_sleep_s>
</mum-config>
```


Figure 7.10: Example of MUM characteristics file.

devices. We believe that there are situations, especially during development and testing stages, when the availability of a model which is representative of the application being developed or tested, in terms of its memory and CPU usage, would be a useful asset.

7.7 Conclusions

From the perspective of application development, GC aids the creation of more secure, robust software, and in less time than would be the case, if explicit memory management code must be written. One of the objectives of modern JVMs is to automatically provide GC that is tuned to the needs of the target application, without requiring any manual tuning to be performed. In other words, JVMs attempt to provide *out-of-the-box* high-performance. In some cases however, it may be possible to achieve better performance through manual GC tuning.

The first step in GC tuning should be to obtain a *memory usage profile* of the target application. JVMs make available a wealth of information that can be used to help understand how well a particular set of GC options match the memory usage behaviour of a particular application. For example, JVMs can be instructed to log the occupancy levels of each of the spaces within the heap, at the time of each collection. This information this might indicate that an execution may benefit from changing the relative sizes of these spaces. There are many high-quality applications available that allow for the visualisation of such data. For the IBM platform, we found the GCMV plugin of the IBM Support Assistant Workbench to be a very useful tool.

We presented our MUM application, which can be used to model the memory usage behaviour of real applications, or to create realistic benchmarks. Real applications typically include phases during which many objects are created, as well as phases during which a lot of computation is performed. To model such behaviour, MUM supports the creation of any number of phases, containing varying levels of object creation and computation.

8

Just-In-Time Compiler

Software is not limited by physics, like buildings are. It is limited by imagination, by design, by organization. In short, it is limited by properties of people, not by properties of the world

– Ralph Johnson

JIT compilers, such as those employed by JVMs, provide runtime environments with the option of compiling part, or all, of code in a Directly Interpretable Representation (DIR) format to code in a Directly Executable Representation (DER) format. Code in a DER format may then be executed natively. In addition to compiling code, JIT compilers typically also apply optimisations to the generated code. An optimising JIT compiler may remain active during the entire execution of an application, applying more targeted and aggressive optimisations. These optimisations may be based on insight gained from executing some code for a period of time, and they are commonly referred to as Feedback-Directed Optimisation (FDO). The full range of dynamic and speculative optimisations that may be employed when executing code in a managed runtime environment, provides Java with one of its competitive advantages over statically compiled languages.

Unlike statically compiled languages, where code is compiled and optimised ahead-of-time, the time required to compile and apply optimisations to code in a managed runtime environment is included in the overall execution time. For this reason, the benefits of performing these tasks must be weighed against their cost, in terms of resources consumed and time taken.

JVMs are free to make use of a JIT compiler or not, and if they do, they may implement JIT compilation in whatever way they wish. Therefore, like GC, the implementation details of JIT compilers differ from JVM to JVM, as do the options available for tuning their activity. Also in common with GC, the topic of JIT compilation predates the birth of Java by decades, and has been the focus of many research projects and publications. In this chapter, we investigate the current status of JIT compilation for the Java platform.

8.1 Background

Traditionally, computer languages were compiled and executed in one of two ways. Either the programming code was first statically compiled to native machine code at compile-time, and then executed natively at runtime, or the code was interpreted by a VM, as illustrated in Figure 2.1.

Executing native machine code is generally much faster than interpreting code written in a DIR format, such as bytecode. Additionally, the cost of compiling and optimising code is paid ahead-of-time, not at runtime. One of the primary drawbacks of statically compiling and optimising code is the limited portability of the generated code, as it is generated for a particular processor architecture. Each processor architecture supports a particular set of instructions, and in order to execute code on a particular architecture it must be expressed using the particular set of instructions supported by that architecture. This means that applications must be compiled for each of the architectures where they might be executed. If an application is to be executed on a new architecture then it must be recompiled for that architecture. Additionally, the advances which have been made with FDO optimisations have shown that, the range of optimisations available to JIT compilers operating in a managed runtime environment is greater than those available to static compilers.

The terms *VM* and *managed runtime environment* are often used interchangeably. According to [Arnold et al., 2005], a managed runtime system wishing to perform selective optimisations requires three key components:

1. A profiler which can provide useful insights regarding the execution of code.
2. A component which can decide which optimisations to apply, to which level of aggressiveness and to which parts of the code. Such systems are typically called an Adaptive Optimisation System (AOS).
3. A compiler that can perform these optimisations.

The use of such runtime systems has a long history, and they have been developed for the execution of a number languages. One of the earliest runtimes systems was designed as an execution environment for a version of Fortran, call *Adaptive Fortran* [Hansen, 1974]. They have also been used for Lisp [McCarthy, 1978], Smalltalk [Deutsch and Schiffman, 1984] and Self [Chambers et al., 1991]. Java, however, was the first language whose success was largely based on the use of a VM. A contemporary of the JVM, and another commonly used VM, is the CLR [Meijer and Goug, 2000], which executes DIR code compiled from the .NET family of languages (see Section 2.1.6.3 for more on the CLR).

In Java, source code is compiled to bytecode at compile-time. In the original JVMs, all bytecode was interpreted at runtime, resulting in relatively poor performance. One alternative to interpreting bytecode at runtime is to compile it to native machine code at some point before runtime. This approach, known as Ahead-Of-Time (AOT) compilation, is effectively what happens with statically compiled languages (except that they are normally compiled directly from source code to native machine code, instead of being first compiled to an intermediate format such as bytecode). A number of tools have been developed for compiling Java source code, as well as Java bytecode, to native machine code using AOT compilation. These include the GNU Compiler for Java [GCJ, 2012], and Excelsior JET [Mikheev et al., 2002]. This approach however, results in the loss of some of the core strengths of the Java platform, including its portability, as the native machine code is machine-dependent; as well the loss of the GC service provided by the JVM, as the native machine code is executed outside of a JVM.

The addition of JIT compilers to JVMs provided the option of compiling some, or all, of the bytecode to native machine code just before it was executed. Early versions of Java JIT compilers attempted to compile all of the bytecode of an application at startup, this approach leads to very significant startup times. To avoid this problem, the MMI approach was introduced.

This allows for some parts of the code to be executed using interpretation, while others are compiled. The decision of which parts of the code to compile is based upon the frequency that each part is called. The use of MMI also allows optimising JIT compilers to spend more time applying optimisations, as they can avoid the time that would of been spent compiling methods that are executed rarely, or never. The unit of compilation for most JIT compilers is methods, in other words, some methods are compiled while others are interpreted. Methods that are called relatively frequently, and therefore constitute a significant part of the execution time of an application, are called *hot* methods, while those that are called few times are known as *cold* methods. Determining which are the hot methods of an application has therefore become a very important task for JVMs.

The compilation and optimisation of code adds an additional overhead. The more aggressive the optimisation, the larger the overhead incurred while applying it. This overhead is *paid for* in the form of slower start-up times, and interruptions to steady-state performance, if optimisations are applied during the steady-state phase. Ironically, as JIT compilers became more sophisticated — supporting more and more optimisations, the overhead that must be paid has increased. However, this overhead will be outweighed by the boost in performance if the optimised code is executed frequently. Like many other aspects of performance tuning, a trade-off must be found between the overhead, and the improvement in performance obtained by these optimisations. The *hotter* the method, the more incentive there is to spend a little more time applying optimisations, as the method is going to be executed a lot. Highly-optimised code is sometimes said to be *high-quality code*, while code which has been compiled without any optimisations is said to be *low-quality code*.

Java supports a number of dynamic features, such as polymorphic virtual method invocation, multiple inheritance through interfaces, and dynamic class loading. Although these features boost developer productivity, they also complicate the task of optimising code, as JIT compilers have to work harder in order to determine which piece of code will actually be executed as a result of a method call.

Despite the portable nature of Java, the optimal set of JIT tuning options for a particular application can depend on the underlying hardware environment. This is due to differences in how hardware implements certain features, such as memory hierarchies. Additionally, the extent to which JVM implementations can take advantage of the available hardware varies from JVM to JVM. Because of these facts, obtaining the maximum performance on a number of different environments can mean having to tune the JIT compiler for each.

8.1.1 Profiling method *hotness*

In order for the JIT compiler to decide which methods should be compiled, and possibly optimised, to a particular level of aggressiveness, it must have some mechanism for determining the frequency that each method is called. A number of profiling techniques have been implemented for this purpose. A primary concern for such profilers is not to add an unacceptably high overhead to the execution of the target application. Profiling techniques can be grouped in two broad categories: lightweight, coarse-grained techniques; and intrusive, fine-grained techniques.

Lightweight techniques are generally used for simply identifying the hot methods in the code. Once these methods have been identified, then an intrusive instrumentation technique may be applied for obtaining an accurate and detailed profile of the identified methods. This instrumentation may focus on certain aspects of the target method, such as argument values, or the

flow of control within the method. The information gained from this may then be used for performing FDO.

Lightweight techniques may be further categorised into sampling, and counter-based techniques. Sampling typically involves the regular inspection of the call stack, to check which method is executing at that time. Counter-based profiling techniques create a counter for each method, and insert code into each method to increment its corresponding counter, every time that the method gets called. Additional counters may be inserted to take into account the presence of loops within the method. Once a counter has reached a certain threshold, the corresponding method is considered *hot*.

JRockit and Jikes JVMs both perform sampling, while the HotSpot and J9 JVMs use a counter technique for identifying hot methods. All use code instrumentation to obtain a more detailed and accurate profile of particular methods.

8.1.2 Overview of compilation and optimisation processes

The process of converting bytecode to optimised native machine code is a long process that involves many intermediate steps. This process is implemented differently by different JIT compilers, however they generally follow a quite similar approach. This approach involves translating the code through a successive number of Intermediate Representations (IRs). Typically, code is translated through three or four levels of IRs, each IR being closer to machine code in its structure. Fundamental to this process, is the translation of the code from a stack-based format to a register-based format. Eventually the code is translated into native machine code. Different optimisations may be applied at each of these levels. Optimisations that are applied to the higher levels of IR are platform-neutral, and it is only in the lowest level that platform-dependent optimisations are applied. Some of the optimisations that may be applied are discussed in the next section.

8.1.3 JIT compiler optimisations

Most JIT compilers support a number of *levels* of optimisations — each level consisting of a number of optimisations performed with a certain aggressiveness. For example, the Jikes RVM offers the levels: -O0, -O1 and -O2, while the TestaRossa JIT Compiler from IBM supports six levels with the identifiers: *noOpt*, *cold*, *warm*, *hot*, *veryHot* and *scorching*. The AOS is the component which can automatically decide which level of optimisations to apply to which methods, during the execution of an application.

The optimisations that may be performed include inlining, constant folding, the elimination of a number of checks, scalar replacement of instance and class fields, and loop unrolling. There exist a class of optimisations which are based on assumptions that a JIT compiler may make regarding the future execution of a method, based on observing the execution of the method for a period of time. Such optimisations are commonly referred to as speculative or opportunistic, an example is code specialisation.

Inlining a method call involves inserting the method code, at the point in the code that method is actually called, thereby reducing the overhead of calling the method. Inlining also provides the benefit of simplifying the code, and it can result in further optimisations becoming applicable to the simplified code. The use of profile-directed inlining has been compared against the use of inlining solely based on static heuristics in a number of studies, including [Suganuma et al., 2002]

and [Arnold et al., 2002]. These have shown that inlining based on profiles gained at runtime outperforms solely heuristic based approaches.

The OO nature of Java can complicate the process of inlining. Due to the fact that methods may be overridden in a subclass, it might not be obvious which code to insert during the inlining process. A technique known as Class Hierarchy Analysis (CHA) is used to determine if any loaded class overrides a particular method.

DCE is an optimisation technique whereby the JIT compiler will first identify sections of code whose execution has no effect on the state of the executing application. The JIT compiler may safely remove such sections of code, thereby reducing the number of operations that must be performed and improving performance. The possibility that the JIT compiler is performing DCE should be kept in mind when performing application benchmarking, as sections of code may simply be *optimised away* by the JIT compiler, without any indication being given.

If a method began its execution in an interpreter, but was subsequently selected for compilation, and was compiled during its execution, then the JVM has two options for how to proceed. One option is to allow the currently executing thread to complete the execution of the method, and invoke the newly-compiled native machine code the next time that the method is invoked, the other option is to switch to executing the native machine code while the method is still executing. It may be advantageous to switch to a compiled version of the code, especially if the method will continue to execute for a relatively long time, this would be true for methods with long-running loops. To handle such scenarios, some JVMs implement a technique known as On-Stack Replacement (OSR), whereby the current interpreter state is captured, and used as the input for the execution of native machine code. OSR should also be taken into account when performing application benchmarking. OSR is support by most JVMs, including the HotSpot and J9 JVMs.

8.1.4 JVM JIT compilers

In this section, we briefly describe the JIT compiler implementations for the leading JVMs. Most of the work of this thesis involved the use of the HotSpot and J9 JVMs, and we therefore concentrate on these.

8.1.4.1 HotSpot JVM

The HotSpot JVM currently includes two JIT compilers, one is used if the Client runtime is being used, and a different JIT compiler is used if the Server runtime is being used. The Client JIT compiler attempts to minimise delays due to compilation and optimisations. The Client JIT compiler was overhauled in Java SE 6, its IR was changed, and a number of additional optimisations were added. These include the ability to take advantage of the machine-dependent Streaming SIMD Extensions (SSE) set of instructions on x86 platforms, thereby improving the performance of floating-point operations. The Server JIT compiler attempts to maximise throughput, at the expense of compilation time, and memory used to store compiled code. It supports a larger number of optimisations, and most importantly, they are performed much more aggressively. As of Java SE 6, by default, the Client JIT compiler waits until a section of code has been executed 1,500 times before attempting to compile and optimise it, while the Server waits until the code has been executed 10,000 times.

First introduced in Java SE 6 Update 25, and later made the default configuration in Java SE 7, the Server JIT compiler now offers startup times of the same level as those of the Client JIT compiler, using a technique known as *tiered compilation*. This involves the Sever making use of the Client JIT compiler to generate instrumented code, which is used for profiling purposes.

HotSpot associates two counters with each method for the identification of *hot* methods. The *invocation* counter records the number of times the method was invoked, and the *backedge* counter is used for recording every time that the flow within the method loops back, which allows methods with loops to be weighted more heavily.

8.1.4.2 J9 JVM

IBM is one of the JVM developers which have been at the forefront of developing JIT compiler optimisations. The design and operation of an early version of the production JIT compiler from IBM is described in [Suganuma et al., 2001]. This system includes a MMI, and supports three optimisations levels. The supported optimisations include FDOs, such as code specialisation. The operation of this system, including a description of the IRs, through which code is translated during the compilation process, is described in [Ishizaki et al., 2003]. These IRs are known as *EBC (Extended ByteCode)*, *quadruples* and *DAG (Directed Acyclic Graph)*. They describe the optimisations that may be applied to each IR, and compare their effectiveness. Later, in [Suganuma et al., 2005], an in-depth description of the IBM JIT compiler, including the techniques that it uses to profile executions, and the optimisations that it supports is given.

The standard IBM JIT compiler was overhauled for Java SE 5.0, its code base was switched to a new project, known as the TestaRossa (TR) JIT compiler. Most of the techniques from the previous JIT compiler were adopted in this new implementation, and a number of new optimisations were also added. Twelve new optimisations that were added at this time are described in [Grcevski et al., 2004]. In [Sundaresan et al., 2006], an account is given of the difficulties and subtleties of providing robust and correct implementations of a number sophisticated optimisations, considering that the target application may be highly multithreaded, may involve substantial class loading and unloading, and may be executed in a range of hardware environments. In particular, they describe the complexities of using code patching, as well as the use of their profiling framework, when executing such an application.

8.1.4.3 JRockit JVM

The JRockit JVM follows a total-compilation approach, whereby all code is initially compiled at runtime. This initial compilation is performed as quickly as possible, following a baseline optimisation approach, and the minimum level of optimisations are applied. As the execution develops, and the hot methods are identified, optimisations are applied progressively to particular methods

8.1.4.4 Jikes RVM

The Jikes RVM uses a timer-based call stack sampling profiling approach. It uses a cost-benefit model for deciding which optimisations to apply. The key inputs to this model are estimates of the cost of apply optimisations, the expected improvement in code quality, and the length of time that the target code is expected to be executed in the future.

8.2 Related work

The inhibiting effect that executing an application in a cluster environment can have on the ability of JIT compilers to optimise code is discussed in [Tejedor et al., 2008]. Within the context of a single execution of a JVM, once a method has been compiled by the JIT compiler, the generated machine code can be used by subsequent invocations of that method. There exist several automatic parallelising and launching frameworks, such as COMPSs (see Section 4.3), which can decompose an application into smaller units of execution, often called *tasks*, which can then be executed in parallel. However, typically these frameworks execute each instance of these tasks in separate JVMs, meaning that each JVM terminates once it has completed its particular execution of the method.

The use of independent and non-persistent JVMs, negatively affects the ability of JIT compilers to apply optimisations in a number of ways. Each of the JVMs operate independently from each other, thus each one pays the cost of compilation, and any optimisations that they apply. Also, and most significantly, as the JVMs typically only execute a single instance of the *task method*, they do not get a chance to apply optimisations based on many invocations of these methods.

The authors proposed two approaches to improve performance. Firstly, they suggest instructing the JIT compiler to apply aggressive optimisations to certain methods — the methods that are to be executed as tasks. This can be used to ensure that all methods will be compiled before execution. Secondly, they proposed the use of *code caches*, which would be used to store compiled, and optimised code. They also discuss the extension of the COMPSs framework to manage the distribution of code caches amongst the distributed components of the system — as it already manages the distribution of input data.

The J9 JVM supports the use of AOT compilation of classes, and the storing and sharing of these compiled classes in code caches known as *AOT caches*. The objective of these caches is to reduce JVM startup time. However, the code stored in these classes cannot be optimised to a very aggressive level, due to the fact that the code must remain usable by any invocation, and therefore, invocation-specific optimisations cannot be applied. For example, optimisations based on argument values cannot be applied, as different arguments might be used in different invocations. The HotSpot JVM includes a similar feature known as Class Data Sharing (CDS), also intended for speeding up startup, and also lacking the ability of storing highly-optimised code caches.

An approach for automatically tuning a JIT compiler is described in [Hoste et al., 2010]. They use the term *optimisation plan* to refer to a set of optimisations, performed to a particular level of aggressiveness. Each level of optimisation that a JVM supports corresponds to a particular optimisation plan. For example, the *scorching* level, which is one of the six optimisations levels available within the TR JIT compiler, includes all optimisations, and these are applied to the maximum level of aggressiveness. They propose a two-step process for tuning a JIT compiler. Firstly, they generate a set of optimisations plans which are Pareto-optimal, in terms of compilation time and code quality, using an evolutionary search algorithm. The second step involves assigning some of these plans to the optimisation levels supported by the JVM, and then using an evolutionary search algorithm to tune the AOS. They tested their approach on the Jikes RVM using a number of benchmarks from the DaCapo and SPECjvm98 benchmark suites for a number of scenarios.

These scenarios include: tuning the JIT compiler for a particular benchmark, and then ex-

ecuting that benchmark; tuning for one benchmark, but then executing a different one; and tuning for a particular hardware environment. They found that their automatic tuning system offers similar performance to a manually tuning approach. Their results showed that their tuning approach is quite dependent on the set of benchmarks used for the tuning process being representative of the target applications. They point out that their approach would also be applicable to other VMs, such as the CLR.

8.3 JIT compiler tuning to improve GASS performance

In order to determine the level of optimisations performed by the JIT compiler of the J9 JVM, when executing the GASS application, a profiling study was performed. One of the key inputs to the GASS application is a file which specifies a number of time intervals. These are the periods of time that must be simulated by the application. There is a method within the GASS application which processes a single interval, therefore, this method is called for each of the intervals. When executed on MareNostrum, GASS is normally executed using the GRIDSs framework, which allows for the automatic parallelisation of applications, through the identification of *tasks* that may be executed in parallel. GRIDSs identifies the GASS method which processes intervals as the method that should be used as the *task* method. However, as described in the previous section, the use of GRIDSs leads to a separate JVM being created for each task, preventing JVMs from using information gained from successive executions of the task method for optimisations.

In order to investigate the extent to which this problem might be limiting optimisations when executing the GASS application, the GASS code was modified slightly to execute the *interval processing method* sequentially within a single JVM. The input of GASS was set up so that this method was executed five times for each execution. A GASS configuration was selected that required roughly fifteen minutes to process each interval. Therefore the full execution required less than 75 minutes to execute. GASS was instrumented to record the total time spent within the *interval processing method* for each interval.

8.3.1 JIT tests 1

For *JIT tests 1*, three test configurations were executed. As described below:

- Sequential — RunSeqGass.slurm — the sequential runs of the *interval processing method*, no profiling was performed.
- Profiling — RunSeqGassHprof.slurm — the sequential version of GASS was executed with the *-Xrunhprof* command-line option, in order to run the Hprof sampler.
- JIT — RunSeqGassJIT.slurm — the sequential version of GASS was executed with the *-Xjit:verbose* command-line option, which requests the JIT compiler to produce an output file showing its activity, including which methods were optimised, and to what level.

Given in Table 8.1, and plotted in Figure 8.2, are the execution times for *JIT tests 1*. The sequential version of GASS was first executed four times, without any profiling, therefore the first four columns refer to identical tests. The 5th test gives the execution of GASS with the *Hprof* profiler was enabled, while in the 6th test, the *-Xjit:verbose* option was enabled.

Method call	Test1 (Seq)	Test2 (Seq)	Test3 (Seq)	Test4 (Seq)	Test5 (Seq + Hprof)	Test6 (Seq + JIT msgs.)
1	893	925	971	1109	1774	893
2	783	798	798	803	1686	794
3	782	802	809	803	1686	772
4	778	792	812	804	1724	779
5	771	801	804	799	1667	813
Total	4007	4118	4194	4318	8537	4051

Table 8.1: Speed-up over successive invocations of *interval processing method* within GASS.

8.3.2 JIT tests 2

In *JIT tests 2*, we attempted to improve performance of this configuration of GASS using a *brute force* tuning option. Having identified a non-documented JIT compiler option, we instructed the JIT compiler to apply the most aggressive level of optimisations to all methods using the option *-Xjit:'*(optlevel=scorching)'*, see Appendix C for a detailed description of this tuning option. The results of this test are given in Table 8.2.

	Test1 (Seq)	Test2 (Seq)	Test3 (Seq + Hprof)	Test4 (Seq + Scorching)
Execution time (s)	4204	4205	9338	4850

Table 8.2: Speed-up over successive invocations of the *interval processing method* within GASS.

8.3.3 Analysis of results

In *JIT tests 1*, the first four tests (all identical tests) showed a decrease in the execution times between the first execution of the main GASS method, and the average of the subsequent executions, this decrease was 12% on average, across the four tests. This decrease is not achieved if separate JVMs are used for each invocation of a method, as happens in some execution frameworks. In these simple tests, we only executed the *task method* five times sequentially, if this number was increased to thousands, then it is likely that the improvement in performance would be much greater.

Overall, the average execution times for the first 4 tests was 4159 seconds. The 5th test, which involved the execution of GASS with the Hprof profiler, showed a very significant increase in execution time, it was 105% longer. Viewing the Hprof output through PerfAnal, as shown in Figure 8.1 revealed that the overall execution time of GASS is distributed amongst a relatively large number of methods. We see that 13.51% of the time was spent in the method *binarySearch*, and 12.27% was spent in the method *lessThan*. By checking the output file produced by the JIT compiler in the 6th test, we can see that these two methods are compiled to the *scorching* and *hot* levels, respectively. The next most frequent methods are mostly from *StrictMath*, as well as some from the *gaiasimu* package.

In *JIT tests 2*, again we see that using the Hprof profiler, with this particular level of profiling, results in a huge increase in execution time — 122% longer. We also see that applying maximum optimisations to all methods actually results in a slowdown in execution time. This is due to

the fact that we forced the JIT compiler to apply the maximum strength of optimisations to all methods, when the methods are first encountered. The time spent applying these optimisations adds to the execution time. Because the compiler applied these optimisations as soon as it encountered each method, it did not have the benefit of any profiling information on the methods. Additionally, we know that the most frequently executed methods were already being optimised to a high degree, even when the *interval processing method* was only called five times. Effectively, we added the overhead of optimising rarely invoked methods, to a high degree, therefore the cost of applying the optimisations outweighed the benefit.

We should point out that, in this case, we only tried a *brute force* approach. If however, the *-Xjit* option was applied in a very selective manner, targeting those methods which are called fairly frequently, but which were not being optimised to a high degree, and also allowing these methods to execute a number of times before applying optimisations, then it is very likely that performance improvements would be gained.

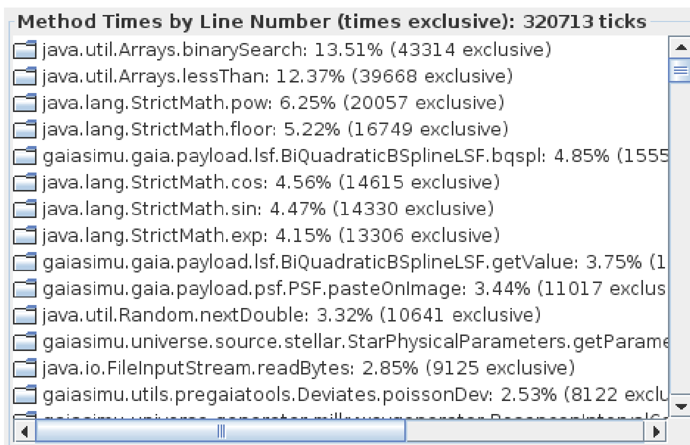

Figure 8.1: Main view with PerfAnal while profiling GASS.

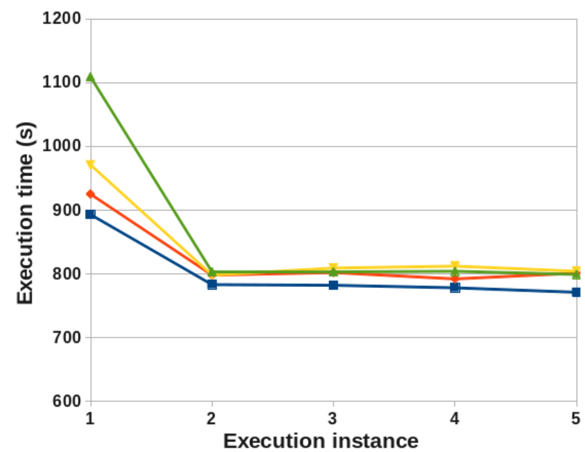

Figure 8.2: Showing execution times of the *interval processing method* over 5 executions.

8.4 Conclusions

The execution of code within a managed runtime environment is very different from the execution of native code that was statically compiled ahead-of-time. The time taken by a JIT compiler to compile code, and apply optimisations, is included in the execution time of the executing application, therefore a balance must be found between applying optimisations, and the cost of actually applying them.

The optimisations available to runtime environments, like the JVM, include optimisations based on insight gained about application behaviour, after having executed the application for a period of time. JIT compilers can also make speculative optimisations, based on assumptions that they *think* will remain true for the entire execution of an application. If however, such assumptions turn out to be incorrect, then the JVM has the ability to recover from the situation by reverting back to a less optimised version of the code. Already, the range of optimisations that are available in runtime environments, such as JVMs, exceed those available to statically compiled languages, and this gap will likely increase further in the future.

9

Monitoring

In computing, the mean time to failure keeps getting shorter

– Alan J. Perlis

In HPC environments, monitoring systems are very important for ensuring the efficient use of the available resources, and for determining if applications are functioning as expected. HPC environments, such as computer clusters, are valuable resources, they have high operational costs, and often many users have to compete for their use. It is important that such resources are used to their full potential. Monitoring is important from the perspective of HPC administrators as well as HPC users, as both groups should be keen to ensure that applications are making the best possible use of the resources that are assigned to them. The importance of monitoring increases for applications that must execute for relatively long periods of time, and applications that consume large numbers of resources.

Monitoring may be performed at a number of different levels. Monitoring at the system level, for example, monitoring CPU usage, shows the processing demands that the application is placing on the system. With respect to the Java platform, monitoring the internal state of the JVM allows users to check if any situations have developed which might warrant intervention, such as the JVM running low on memory, or excessive garbage collection occurring. Finally, monitoring the internal state of applications is also possible. Applications which have been designed, or modified, to expose information to the outside world, for the purposes of monitoring, are said to be *instrumented*.

In many cases, it would be useful to be able to manage the execution of an application, in addition to being able to monitor it. Such a management system might allow users to, for example, change some configuration parameters of the application *on the fly*. JMX is a built-in framework within the Java platform, that allows for the monitoring and management of both JVMs and instrumented applications in real-time.

In this chapter, we discuss the monitoring of Java applications in HPC environments. In Section 9.1 we gave background information on this topic, including discussing the approaches that may be taken, and some considerations which must be taken into account, due to the nature of typical HPC environments. Our initial tests of the JMX framework are described in Section 9.2. We then present the design and implementation details of our monitoring system in Section 9.3, including our approach for monitoring and managing DPAC systems in HPC environments.

9.1 Background

In the context of this work, we use the term *monitoring* to refer to processes put in place to perform regular checks on the status of some properties of a system, while that system is executing. Profiling (as discussed in Chapter 6) and monitoring are related activities, although profiling is an activity that is performed to analyse some specific characteristics of a system. Monitoring however, is an activity generally performed on a live system, for the purposes of checking the progress of an application, and as a safety measure for detecting unexpected situations.

Large distributed HPC environments are more prone to failures than smaller systems. Localised hardware failures are quite common, partly due to the ever-increasing number of computing elements comprising these systems, due to the fact that such systems are increasingly composed of commodity components, and also due to the very high usage that these components receive. The Mean Time Before Failure (MTBF) of the components of such systems can be relatively short. Due to the regularity of component failure, all software running in such environments, including instances of the OS running on each node, the job scheduling and management system, and applications themselves, should ideally be able to tolerate localised hardware failures — assuming that the failures are limited in their extent.

As well as hardware failures, applications themselves can of course behave in unexpected ways due to bugs, and unhandled exceptions. HPC users may wish to monitor the execution of their applications to ensure that they are proceeding as expected, users may wish to be informed if particular events occur, allowing them to take corrective action. Situations which might warrant some corrective action, could be an excessive amount of time being spent in GC. This could result in the application providing reduced throughput. Such situations could be detected if the percentage of overall execution being spent in GC was monitored over time. If a user or system administrator, can detect such situations, then they can take corrective action, improving the efficiency of the system, and preventing fatal errors from occurring.

Monitoring an application that is running in an HPC environment can be more challenging than monitoring an application running on a local workstation, due to the physical nature of the environment, which is typically composed of distributed components; and the means by which applications are executed in HPC environments. Firstly, it must be considered that the application may be executing across many computing nodes rather than simply on a single machine, this means that there are many nodes that need to be monitored. To get a global overview of the status of an application, then its status across all of the computing nodes must be combined together and presented in an aggregated view. If a number of nodes must be monitored from an application running in a single node, then the *monitoring node* will need to retrieve monitoring information from each of the monitored nodes, resulting in extra network activity.

In most HPC environments, users submit their jobs to a job scheduling and management system, which then takes care of the launching of applications in the requested number of computing nodes. Often users cannot connect directly to the computing nodes where their jobs are executing. This limits the ability of users to monitor their jobs. Additionally, user may only be able to access output files generated by their application once that application has finished, and the job management system has copied the output files to a location accessible by the user.

Performance is often the primary concern for applications running in HPC environments, and anything that reduces the performance of an application is viewed negatively. In common with

profiling, the actual process of monitoring an application adds an extra overhead to the workload of the system. This overhead is generally proportional to the granularity of the monitoring that is being performed, in other words, monitoring a system very closely results in more overhead than monitoring the system less closely. This *performance hit* must be balanced against the benefits provided by monitoring.

9.1.1 Java Management Extensions (JMX)

JMX is a built-in framework within the Java platform that allows for the management and monitoring of JVMs, and of the applications running in them. It may be used within a single machine, or it may be used across a distributed environment. It is flexible, extendible and extremely useful. In addition to allowing for the monitoring of JVMs, the JMX framework also allows for the monitoring and controlling of applications running in JVMs. Through a process known as *instrumenting*, applications can be enabled to expose internal properties to the JMX framework, thereby permitting these properties to be accessible through the JMX framework.

In the Java platform, JavaBeans are modular, reusable components which encapsulate some resource or entity, and which conform to a set of conventions allowing for the accessing and modification of their elements. JMX is based on the idea of MBeans (Managed Beans), which are a form of JavaBean. An MBean represents some managed resource, and it exposes some attributes and operations, associated with that resource, to the JMX framework. MXBeans are a special type of MBean that only reference a predefined set of data types, which are defined in `javax.management.openbean`. MXBeans have the advantage that they may be used in a distributed system without requiring remote clients to have access to classes representing the types referenced by the MBean. This is a useful characteristic when monitoring an application using a monitor that may not have access to the source code of the target application. As well as exposing properties and operations, MBeans may be configured to emit notifications when certain events occur. Monitoring applications can then be configured to listen for these events, and taken action when they occur.

The JMX framework is composed of three layers, as illustrated in Figure 9.2, this framework facilities the use of JMX over a distributed system, such as a computer cluster. In distributed systems, communication between the distributed components of the system takes place over the RMI protocol. The *instrumented layer* contains the MBeans, and the associated resources. The *agent layer* contains a component known as the MBeanServer. The *monitoring layer* contains the tools, or clients, used for presenting the retrieved information, and management functionality, to users. The Oracle JDK contains a number of tools including VisualVM, described in Section 6.3.6, which can be used as JMX monitoring clients.

The MBeanServer is the entity with which MBeans must be registered if they are to be accessible through the JMX framework. Each MBean is identified by a unique identifier, known as its *ObjectName*. Monitoring applications wishing to monitor an MBean within a particular JVM can query the MBeanServer to find out which MBeans are available and can interact with these MBeans using a number of possible approaches.

9.1.1.1 Platform MXBeans

Platform MXBeans are a set of MXBeans provided as part of the Java platform which allow for the monitoring of certain properties of the JVM. The package `java.lang.management` defines

a number of Platform MXBean interfaces. It is the responsibility of JVM implementers (such as Oracle and IBM) to implement these interfaces. Although the major JVM implementers provide almost identical implementations of the Platform MXBeans, there can be minor difference between them. For example, prior to Java SE 1.7, the J9 JVM for the PPC64 architecture did not support the method *ThreadMXBean.getThreadCpuTime(long id)*, but this method was supported by the corresponding HotSpot JVM. Platform MXBeans provide information on several topics, including memory usage, GC and JIT compiler activity, and threading.

9.1.2 Simple Network Monitoring Protocol (SNMP)

Simple Network Management Protocol (SNMP) is an industry standard for monitoring resources in a networked environment. The normal SNMP configuration involves a manager process, running on a single host, monitoring a number of devices. A local process, known as an agent, is executed on each managed device, reporting information to the SNMP manager. The information that is available from an agent is defined in a file called a Management Information Base (MIB). These files define the actual data which is available for monitoring and management. MIB files contain a hierarchical tree structure, and each individual element defined, has an associated ID, known as its Object IDentifier (OID). Developers are free to define new MIB files, and add devices to the SNMP framework.

9.1.2.1 SNMP compared with JMX

There is an overlap between the functionality provided by SNMP and JMX, although there are also many significant difference between them. Monitoring *agents* for both these systems are included with most JVMs, and the first step to use either of these systems, is to enable the corresponding monitoring agent. When enabling the SNMP agent, an Access Control List (ACL) file should be provided, specifying the permissions associated with entities. As JMX is part of the Java platform, the process of instrumenting an application, and then making that information available to monitoring applications is more straightforward for JMX than for SNMP.

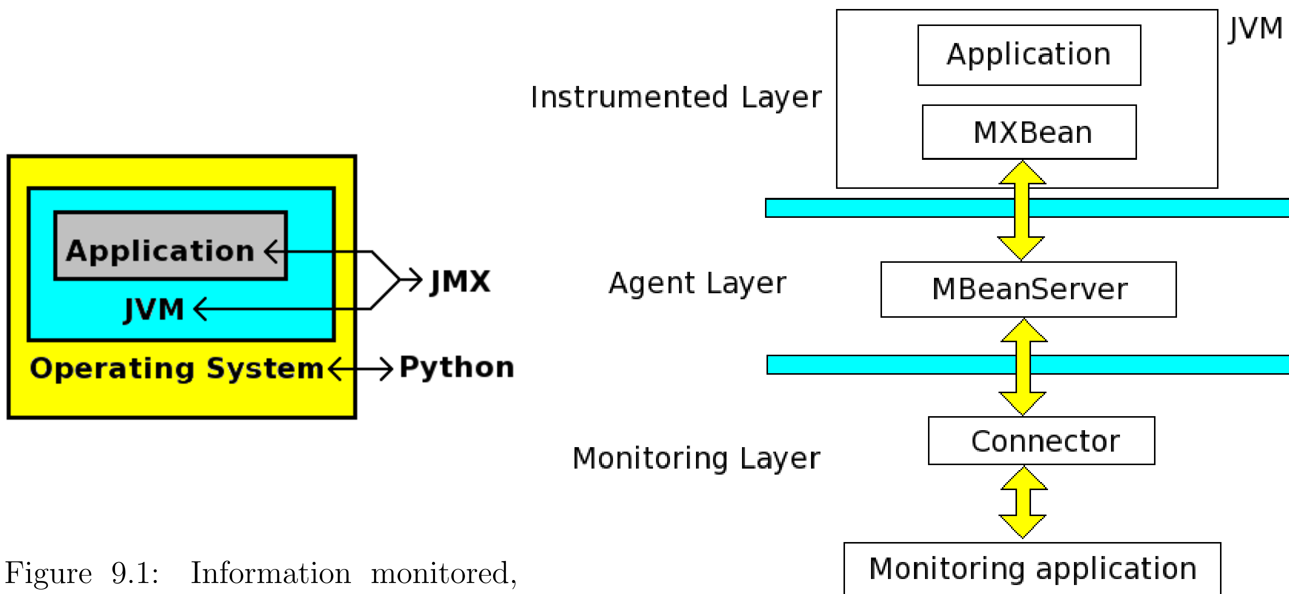

Figure 9.1: Information monitored, and the technology used, by our monitoring system.

Figure 9.2: JMX framework components.

Test	Applications executed	Mon. interval	Execution time (ms)	Execution time increase (ms)	Percentage increase
1	MUM (JMX not enabled)	-	125447	-	-
2	MUM (JMX enabled)	-	128232	2785	2.22%
3	MUM + Mon App	60s	128415	2968	2.36%
4	MUM + Mon App	30s	128816	3369	2.68%
5	MUM + Mon App	10s	129734	4287	3.42%
6	MUM + Mon App	1s	131827	6380	5.08%
7	MUM + Mon App	500ms	133292	7845	6.25%

Table 9.1: Increase in execution time observed when monitoring the target application at varying frequency. In other words, at different levels of monitoring granularity.

9.2 Initial performance tests of JMX

As stated previously, the extra overhead that monitoring adds to a system mainly depends on the granularity of the monitoring. By *granularity*, we mean the amount of information monitored, and the frequency at which this information is requested. Additionally, the amount of effort that must be made by the target system to retrieve the requested information, contributes to the monitoring overhead. A set of tests were performed to better understand the *performance hit* sustained by running a JMX-based monitoring application at the same time as a target application. The main objective of these tests was to determine the rate at which the monitoring overhead increases as the frequency of monitoring requests increases, while requesting a relatively small set of JVM information.

The MUM application (see Section 7.6) was used as the target application. A MUM characteristics file was created, which produced executions of roughly 125 seconds. A basic monitoring application was created that could be configured to request certain information from the JVM using the JMX framework. The frequency at which this monitoring application requests the information was made configurable. The configuration and results of these tests are given in Table 9.1, and are plotted in Figure 9.3. The JVM status information that the monitoring application retrieved is given in Table 9.2

In test 1, an instance of MUM was executed on its own, without enabling JMX monitoring, in order to benchmark the performance of this configuration of MUM, without the effects of any monitoring. The results of this first test were recorded as a reference test set. In test 2, JMX monitoring was enabled, however, no monitoring information was actually retrieved during the execution. Next a series of tests (tests 3 — 7) were performed which involved running MUM along with the monitoring application, and retrieving JVM status information. The frequency at which the monitoring application requested this information was increased with each test. These tests were executed on Miranda (see Appendix A), the machine was restarted before each test, and care was taken to ensure that no other applications were running during the tests.

These tests revealed that simply enabling the JMX framework, even if it is not actually used, results in a 2% performance hit. Retrieving information adds an additional overhead, and there is a linear increase in the execution time of the application, as the frequency of the monitoring requests is increased. It is unlikely that there would be a requirement to monitor an application at intervals less than 1 second, therefore, the increase in execution time due to monitoring, for an application of this type, would be of the order of around 5%.

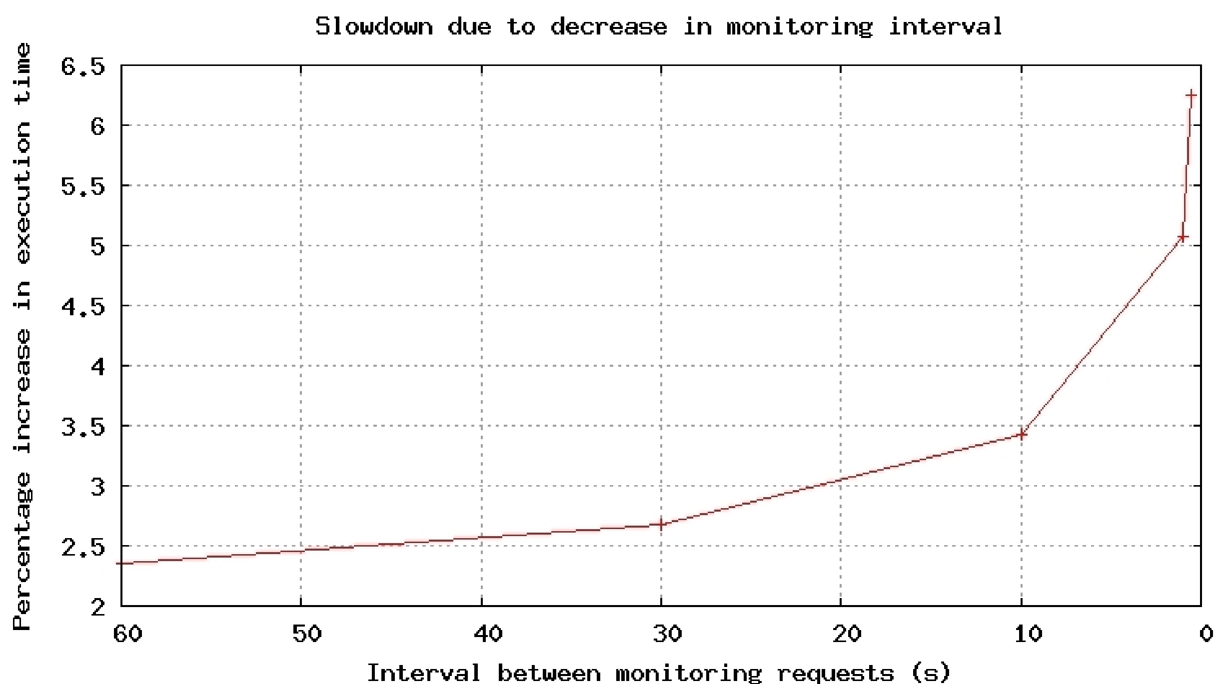

Figure 9.3: Increase in execution time due to increases in frequency of JMX monitoring requests.

Item	Property	Description
1	usedHeapMemBytes	The total amount of memory occupied in the heap. This includes both live objects and garbage objects that have not yet been collected
2	usedNonHeapMemBytes	The current memory usage of non-heap memory by the JVM
3	gcCollectionCount	The total number of collections that have occurred
4	gcCollectionTime	The approximate accumulated collection elapsed time in milliseconds
5	usedMemPoolCodeCache	The current memory usage in the code cache
6	usedMemPoolEdenSpace	The current memory usage in the eden space
7	usedMemPoolPermGen	The current memory usage in the permanent generation
8	usedMemPoolSurvivorSpace	The current memory usage in the survivor space
9	usedMemPoolTenuredGen	The current memory usage in the tenured (old) space

Table 9.2: Information retrieved during JMX performance tests.

9.3 DpcbTools monitoring component - JavaMon

9.3.1 Properties to be monitored

The DpcbTools monitoring component is designed to monitor three distinctive categories of information: system-level information (including CPU, memory, network and disk usage), JVM state information and application-specific information. System-level information is monitored using a Python-based application, we will not discuss this system in detail as it is outside the scope of this work. JVM information, as well as application specific information, is monitored using our JMX-based application, called JavaMon, as depicted in Figure 9.1. There is an overlap between the system-level monitoring and the JVM monitoring, as both of these monitoring systems report CPU and memory usage information. However, the JVM monitoring information only covers JVM usage of these resources, whereas the system-level information monitors total usage of these resources by all processes running on the machine.

9.3.2 Possible configurations

When monitoring an application running on a computer cluster, there are a number of possible monitoring configurations that could be followed, in terms of the relationships between the monitoring application and the target application, and in terms of how the gathered information can be made available to users. Three such configurations are considered here, the first approach, that we refer to as *Monitoring Config 1*, is to run an instance of the monitoring application in each of the computing nodes where the target application is running, and to write the gathered monitoring information to a local file on those nodes.

On a system where users do not have the privileges to connect directly to computing nodes, writing to a local file on the computing node would mean that users cannot view the output from the monitoring application in real-time. To allow for this scenario, the monitoring application could write the information to a file on a shared disk which is visible to users. We refer to this approach as *Monitoring Config 2*. This approach however has a number of drawbacks. If executions are being run which involve many JVMs running on many nodes, then forcing all of the instances of the monitoring application to write to the same storage device could cause scalability problems, as disk performance generally decreases in proportion to the number of concurrent accesses. Additionally, extra network traffic would be generated.

A final approach, that we refer to as *Monitoring Config 3*, is to run a limited number, or a single instance, of the monitoring application. These instances of the monitoring application would be run on a subset of the allocated computing nodes. For example, sixteen nodes might be requested, and an instances of the monitoring application might be executed on two of these nodes. Therefore each of these two instances would be retrieving monitoring information from seven *worker nodes*. Even on systems where users cannot connect directly to worker nodes, the monitoring applications would be able to connect to the other nodes, and retrieve monitoring information. The monitoring application could make this information available in real-time by writing it to a storage device accessible by users. This approach has the advantage of a reduced number of instances of the monitoring application, and means that resources on the *worker* computing nodes would not be consumed by monitoring applications. This approach does however result to extra network traffic between the monitoring application and the instances of the target application, however this is relatively little network traffic, especially considering

the fact that many clusters contain high-performance network interconnects, which provide high-performance inter-node communication.

Choosing the most appropriate of the three mentioned monitoring approaches for a particular application, running in a particular environment, requires the consideration of a number of factors. These include the hardware configuration; the access privileges granted to users; the desired granularity of the monitoring in terms of the period of time between requests, and in terms of the amount of properties to be monitored; and the size of the execution. Our monitoring application, that we call JavaMon, provides the flexibility that it can be configured to operate in any of the three mentioned configurations.

9.3.3 JavaMon

JavaMon may be configured to follow any of the configurations described in the previous section. Taking our execution framework into consideration (as discussed in Chapter 4), an instance of the monitoring application could be executed in each of the NGM nodes. Therefore, each instance of the monitoring application would be responsible for monitoring the state of the nodes in one NG. A further component, which we call the *MonitoringOverviewer* application, is designed to collect the monitoring information from the monitoring applications running in each of the NGM nodes, and to present an aggregated overview of the status of the entire execution. As well as allowing for the viewing of the monitoring information, the MonitoringOverviewer parses the monitoring information, checking if certain events have occurred, and in certain cases, it will raise an alert.

JavaMon makes use of the Platform MXBeans to retrieve JVM information. The exact information retrieved is configurable in the JavaMon configuration file. This configuration file specifies both the JVM information and the application specific information that should be monitored. JavaMon simply retrieves the configured information, and writes it to a log file, utilising the popular SLF4J library to control the logging level. Using this system, the monitoring application is configured with a particular *monitoring level*, also each event which occurs has an associated level. Using the configured monitoring level, and taking the level of the events which occur into account, it is determined what information should actually be logged.

In order to standardise the monitoring information that is gathered by the JVM and application monitoring, we defined a data structure, which we call an *EventRecord*, which defines the set of information that should be recorded each time that the monitoring application polls a particular monitored property. An event record is simply the state of some monitored property at a given time.

A short extract from a log file generated by the execution of JavaMon, while monitoring an execution of the MUM application, is given in Figure 9.4. In this execution, JavaMon was configured to retrieve three pieces of information: used heap bytes, used non-heap bytes and CPU usage information, as indicated in the first line of the log file. The value in the *Time* column gives the number of milliseconds that the monitoring application has been running.

9.3.4 Application instrumentation

MPJ-Cache was instrumented to allow for the monitoring and changing of certain properties of the application, in real-time. There are a number of ways by which these properties may be accessed, one way is to use a monitoring tool which supports the JMX framework, such as

```
14:26:32 INFO      logFile     - |Time|usedHeapBytes|usedNonHeapBytes|cpu|
14:26:32 INFO      logFile     - |63|13360464|6343248|0|
14:26:32 INFO      logFile     - |745|4609200|6647208|0|
14:26:33 INFO      logFile     - |1326|7260632|6715752|33|
14:26:34 INFO      logFile     - |1904|17583288|6732232|92|
14:26:34 INFO      logFile     - |2496|10657224|6738320|84|
14:26:35 INFO      logFile     - |3091|3157392|6750608|86|
14:26:35 INFO      logFile     - |3679|11228528|6769496|74|
14:26:36 INFO      logFile     - |4309|13499080|6809536|84|
14:26:37 INFO      logFile     - |4860|15768928|6825320|90|
14:26:37 INFO      logFile     - |5463|18258680|6838040|93|
14:26:38 INFO      logFile     - |6019|4263320|6838544|91|
14:26:38 INFO      logFile     - |6590|6488584|6827880|90|
14:26:39 INFO      logFile     - |7146|8948704|6828072|87|
14:26:39 INFO      logFile     - |7730|10872232|6846184|65|
14:26:40 INFO      logFile     - |8338|13029272|6860176|85|
14:26:41 INFO      logFile     - |8910|16167008|6891176|85|
```

Figure 9.4: An extract from a log file generated JavaMon.

VisualVM. Figure 9.5, shows the information, or *attributes*, that are exposed by MPJ-Cache through the JMX framework, as accessed using VisualVM. The values shown in black font may be viewed, while those shown in blue font may be viewed and may also be modified. These attributes may be modified using a number of methods that MPJ-Cache exposes to the JMX framework. Using these methods, it is possible to reconfigure MPJ-Cache *on the fly*. For example, it is possible to instruct it to increase the size of the memory cache, by increasing the value of the attribute *MaxMemCacheSizeFromConfig* using the method *modifyMaxMemCacheSizeFromConfig*. Some of the methods exposed by MPJ-Cache are illustrated in Figure 9.6. Application instrumentation is not limited to configuration properties, any internal value within an application can be exposed through this system. The fact that users may call methods which modify the internal state of the running application in real-time, and the fact that the framework supports distributed environments, makes this a very useful system for managing the execution of Java applications.

9.4 Conclusions

The JMX framework has shown itself to be an extremely useful component of the Java platform for implementing monitoring and management tools, and it forms a key part of our JavaMon application. We have used it to retrieve JVM state information, using Platform MXBeans, and we have used it to retrieved information from instrumented applications. Work to further develop JavaMon is still ongoing, however it has already shown itself to be a useful tool. The fact that JavaMon may be configured to operate in a number of configurations, depending on the configuration of the execution environment, increases its applicability to different environments. We are still working on the best way to present an aggregated overview of the status of an execution in a distributed-memory environment.

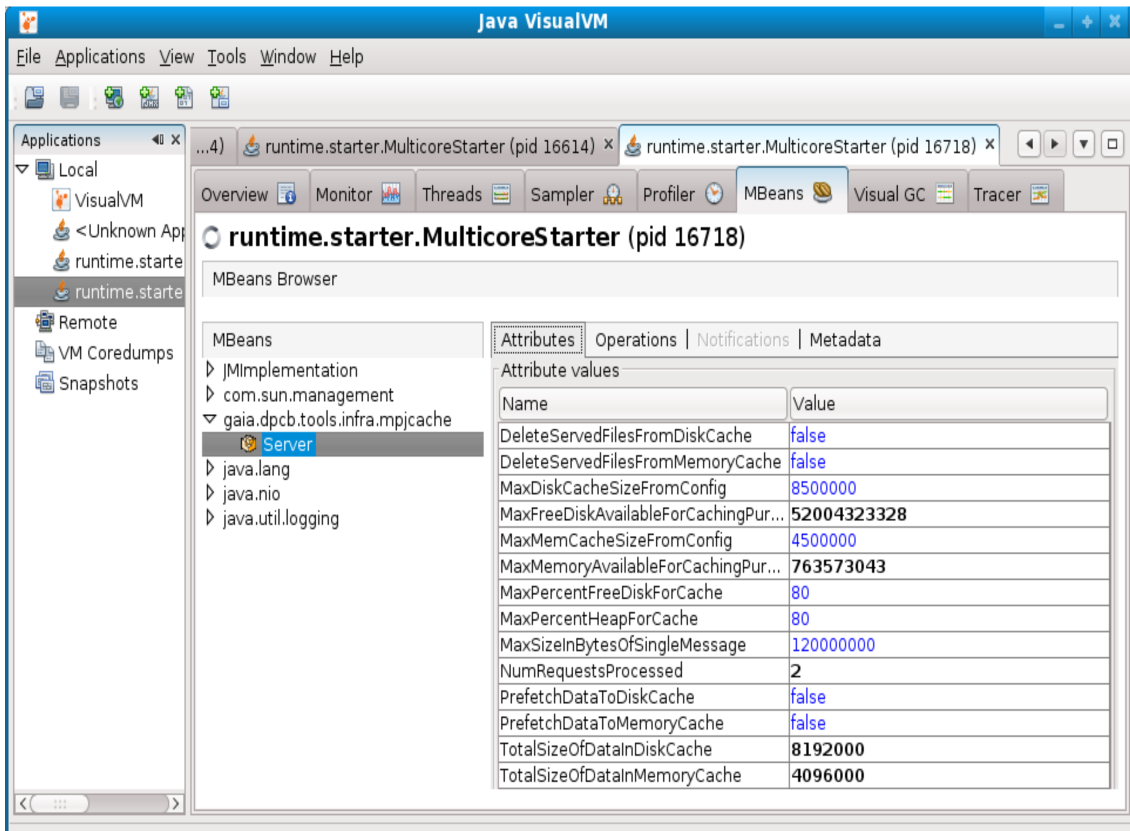

Figure 9.5: Monitoring the status MPJ-Cache attributes in real-time, using VisualVM.

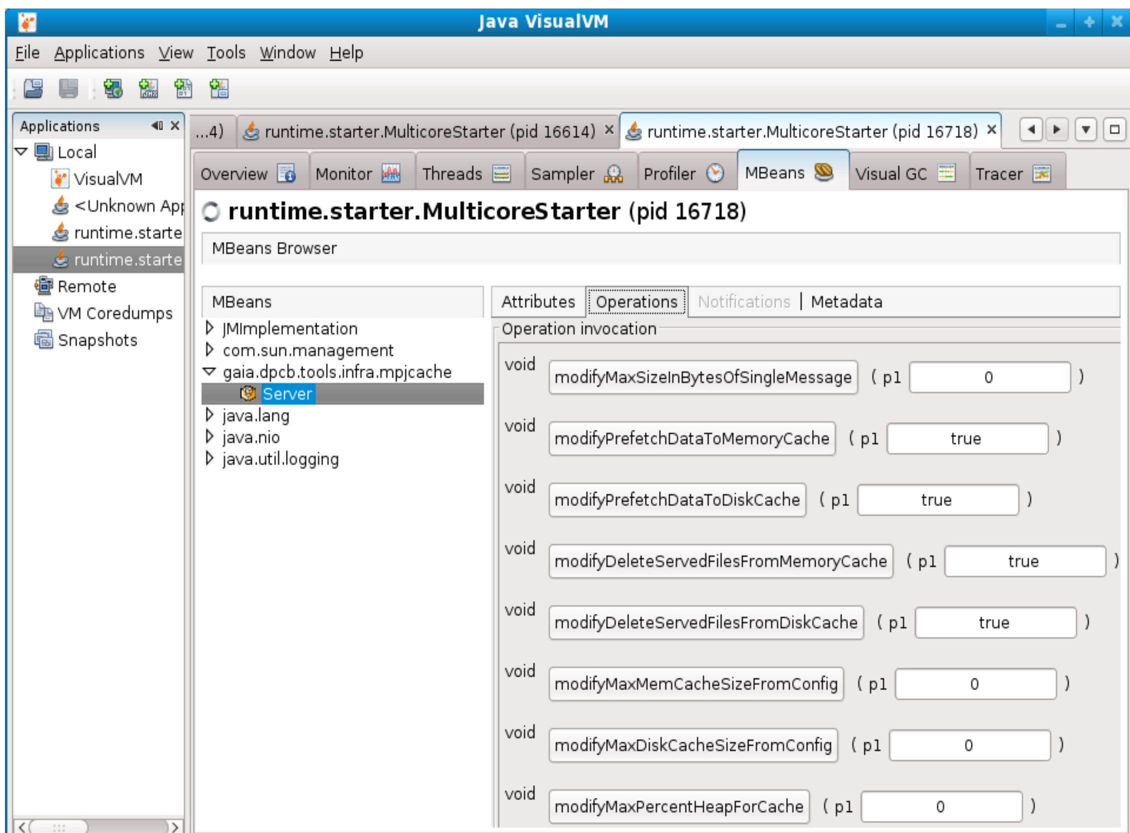

Figure 9.6: Viewing the operations exposed by MPJ-Cache to the JMX framework, which allow for the management of its internal status. Shown in VisualVM.

10

Benchmarking

There are only two kinds of programming languages: those people always bitch about and those nobody uses

– Bjarne Stroustrup

In computing, the term *benchmarking* is used to refer to several related concepts, all of which are concerned with measuring the performance of some component(s). Benchmarking is used in both computer software and computer hardware. The benchmarking of computer hardware is usually performed to obtain an estimation of the maximum computing capabilities of that hardware. To benchmark a software application normally means to compare the performance offered by that application against the performance of some standard application which performs the same task. The term *benchmark* is also used to refer to the application that is used as the standard. Benchmarking is particularly important when trying to estimate the potential throughput of a system.

Benchmarking should be performed following a careful, methodical approach, which takes into account all those factors which could influence the execution of the benchmark, but which are not the intended focus of the benchmark. When benchmarking a particular characteristic of a system, it is important to remove, or at least understand, the influence that other factors could have on the benchmarking results. For example, if the steady-state performance of an application is being benchmarked, then the startup period should not form part of the benchmark period. This can be achieved by including a warm-up period at the start of the benchmark, during which, measurements are not taken. Such steps must be taken in order to ensure the reliability and usefulness of the benchmarking results.

During this thesis, and for the preparations of the DPAC systems at the DPCB, the benchmarking of both computer hardware and software was an important task. It was used for evaluating and comparing applications, as well as estimating the processing and data throughput capabilities that will be needed to execute the DPAC systems.

10.1 Background

In order to compare systems with each other, some standard metrics are required. For benchmarking computer hardware, the metric that is most commonly used is FLOPS (or flops or flop/s). FLOPS is a measure of the number of floating-point operations a given machine (or processing component such as processor or core) can perform per second. The widely referenced

list of the top 500 supercomputers in the world — TOP500 [TOP500, 2012], ranks machines based on their maximum FLOPS capability.

The FLOP count is a measurement of the number of floating-point operations required by a given algorithm, or application, to complete all of its work. Obtaining such an estimation for complex applications, such as the IDU application, is a challenging task. But if the FLOP count is obtained for an application, this metric can then be used to estimate the processing time that that application will require in order to complete its work while running on a computer with a known FLOPS capability. In other words, if we know the number of FLOPS that we have available in our hardware, and we know the FLOP count of the software that we wish to run, then we can get an estimate of the time required to run that application on that hardware (under ideal conditions, that is, with the CPU usage constantly at 100%).

There exist several standard benchmarks, which are widely used in both industry and in academic research. The term *microbenchmark* is used to describe a benchmark used to measure the performance of a particular component or a particular piece of functionality. Microbenchmarks are typically created by the developers of the application that is to be benchmarked.

The regular benchmarking of applications should be incorporated into the development cycle of applications as a means to ensure that performance is not negatively affected by code changes. This is especially important for applications that will be in the development and maintenance stages for long periods of time.

10.1.1 Considerations while benchmarking hardware

When benchmarking the performance obtainable from a given machine, it must considered that there are many factors which could affect the performance of a given benchmark on a given machine, such as input/output, inter-process communication, cache coherence, and memory hierarchy. Therefore, in general, when executing a given application on a given machine, the actual FLOPS achieved on that machine will rarely approach its theoretical peak FLOPS. The net effect of this is that applications generally take longer to complete their work than we might expect, if we were to simply consider the FLOPS of the hardware and the FLOP count of the application.

10.1.2 Considerations while benchmarking Java applications

Benchmarking the performance of an application which is running within a managed runtime environment, such as a JVM, is a more challenging and complex task than benchmarking the execution of a statically compiled executable. The execution of an application within a JVM involves the interaction between the application code and a number of sophisticated components, including the core runtime system, the JIT compiler, and the garbage collector. Additionally, factors such as I/O performance, and other processes which may be running in the execution environment, can all add a degree of non-determinism to the performance obtained while executing an application.

Benchmarking is often performed following a blackbox approach, whereby some simple metric of the performance of the target application is measured — often this metric is simply the time taken to execute the application, but analysis of the execution is not performed. In some other cases however, it may be necessary to investigate the internal processes and phases of the target application. In such cases, profiling can be performed, in conjunction with benchmarking, in

order to provide an insight into how the various components of the system are interacting with each other during the execution of the application (see Chapter 6 for more on profiling). JIT compilers and garbage collectors can normally be configured to output trace files which log their activity. Such output can be used to confirm that these components are not active (or at least, not very active) during benchmarking.

Depending on the configuration of the JVM, it may operate in interpreter mode, in MMI, or it may compile all bytecode to native machine code (see Chapter 8 for a detailed description of the operation of the JIT compiler). As well as compiling code, the JIT compiler may also perform a number of optimisations, these optimisations are performed once the application has been running for a period of time (or more commonly, once a section of code has been executed a number of times). For this reason, it can take applications a period of time before they reach a steady-state. Warm-up periods are normally included within benchmarks to allow the execution to reach a steady-state before actually measuring performance. It is important that the warm-up period involves the execution of the same code, as that which will actually be measured during the benchmarking period. This is to allow the JIT compiler to first profile, and then apply optimisations to that particular code.

One of the optimisations that JIT compilers may perform is DCE. If the JIT compiler determines that a section of code can have no effect on the rest of the execution, then it might remove that code from the execution. The removal of such *dead code* is known as DCE. Microbenchmarks often include sections of code which simply perform some dummy processing. Such dummy processing is typically included to simulate the execution of some real processing. Care must be taken to ensure that such sections of code are actually executed and not *optimised away* because of DCE. One way to avoid DCE is to store the result of the computation performed in that section of code.

It is desirable to avoid garbage collector activity during benchmarking, especially during the execution of microbenchmarks. Such activity includes minor or major collections and heap resizing. To avoid GC activity, the memory usage requirements and behaviour of the benchmark should first be investigated through profiling. Once this knowledge has been obtained, the garbage collector can be configured to ensure that GC activity is minimal during the execution of the benchmark.

10.1.3 Statistical analysis

Statistical analysis of benchmarking results allows for a higher degree of confidence to be attached to the results. Such analysis is especially important when there is a high degree of variability in the results, and when there exist many factors which might be influencing results. This is certainly the case with the execution of Java applications, as the activity of the JIT compiler and garbage collector introduce a degree of non-determinism to executions. The use of statistical analysis techniques allows for evaluations and comparisons of results, even in the presence of a degree of non-determinism.

10.1.4 Benchmarks

In this section we gave a description of a number commonly used benchmarks, and in particular, the benchmarks that were used to evaluate the computing capabilities available at the DPCB.

10.1.4.1 SciMark 2.0

SciMark 2.0 [SciMark, 2012] is Java-based open source benchmark suite designed to measure the performance of numerical algorithms commonly used in scientific and engineering software. More recently, it has been ported to a number of other languages, including C, C# and D. SciMark 2.0 consists of five individual algorithms:

1. Fast Fourier Transform (FFT)
2. Gauss-Seidel relaxation
3. Sparse matrix-multiply
4. Monte Carlo integration
5. Dense LU (Lower-Upper) factorisation

A composite score is calculated for the execution environment, based upon the results of all of these algorithms. Two problem sizes are supported: *normal* and *large*. The normal problem size was designed to allow the state of the execution to fit within the hardware caches of modern machines, thereby minimising memory-hierarchy effects. The larger problem size was designed to measure the performance of applications where the execution state does not fit within available caches.

10.1.4.2 SPECjvm2008

SPECjvm2008 [SPECjvm2008, 2012] is a benchmark suite from the Standard Performance Evaluation Corporation (SPEC) organisation — which has produced many suites of benchmarks for evaluating and comparing computer systems. The primary objective of SPECjvm2008 is to benchmark the performance of JVMs in a particular execution environment (OS and hardware).

SPEC benchmarks include the concepts of a valid and a compliant run. A valid run is one which produces a correct result, a compliant run is one which is executed according to a set of guidelines. Compliant runs are always valid runs. They also include the concepts of run categories, the two main categories are *base* and *peak*. A base run of SPECjvm2008 involves the execution of the benchmarks using default JVM configuration options, a peak run however is intended to show the maximum possible performance obtainable, and therefore allows for tuning of the JVM.

SPECjvm2008 contains many individual benchmarks which are themselves real applications, these are from several fields, including cryptography, data compression, databases and XML handling. SPECjvm2008 also includes the Scimark 2.0 algorithms.

10.1.4.3 NAS Parallel Benchmarks

The Numerical Aerodynamic Simulation (NAS) Parallel Benchmarks NAS Parallel Benchmark (NPB) [Frumkin et al., 2003] are a set of benchmarks designed to assess the performance of highly parallel supercomputers. They were developed and are maintained by the NASA Advanced Supercomputing division. They are derived from code used to solve Computational Fluid Dynamics (CFD) problems. They have been implemented in a number of languages including C, Fortran and Java. There have been three major versions of the NPBs, NPB3 is the latest, and the one used in this work.

10.1.4.4 LINPACK

The LINPACK [Dongarra et al., 2003] benchmark is designed to measure the FLOPS of a system. It measures how fast the system solves a dense N by N system of linear equations. A version of the LINPACK benchmark is used to rank the machines in the Top500 list of supercomputers in the world. Versions of the LINPACK benchmark have be implemented in a number of languages, including C and Java, in this work, we used the Java version.

10.1.4.5 DaCapo

The DaCapo benchmark suite [Blackburn et al., 2006], was developed as an alternative to traditional benchmarks which often lack a sufficiently rich behaviour to simulate the complex interactions which occur during the execution of a Java application. One of the areas in which many benchmarks lack realistic behaviour is in their memory usage behaviour. The DaCapo benchmarks are highly regarded in their ability to replicate memory usage patterns of real applications, and they are often used in studies when comparing the performance of alternative memory management approaches, such as alternative GC policies.

10.1.4.6 IOzone

IOzone [IOzone, 2012] is a C-based, FOSS suite of benchmarks designed for measuring the performance of a filesystem. The benchmark executes a number of I/O operations including read, re-read, write, re-write, read backwards, read strided, and measures the times required for each. IOzone may be configured to use various files sizes and various records sizes. Additionally, it is possible to specify which I/O API the OS should use to perform the benchmarks. This allows for tests to be performed which use the same API as the applications that will run on the target machine. In the context of the preparations of the DPAC systems at the DPCB, data throughput, and therefore data access rates are extremely important considerations. For this reason, the results of the IOZone benchmarks were of particular interest for the preparations of the DPAC systems at the DPCB.

10.2 Related work

The use of rigorous replay compilation for performing the benchmarking of Java applications is discussed in [Georges et al., 2007]. They highlight the level of non-determinism that is often experienced when executing Java applications. One of the main sources of this non-determinism is the sampling technique that JIT compilers use for determining the hot methods. This sampling information is used for deciding when, to what level of aggressiveness, and to which methods, optimisations should be applied. Therefore, the precise performance obtained when executing an application is very sensitive to minor differences between the profiles obtained by the JIT compiler sampler. *Reply Compilation* is a technique designed to minimise such non-determinism. It involves the creation of a compilation plan, which is then used by the JIT compiler for compiling and optimising the code of a benchmark, across multiple executions of the benchmark. Typically, a single compilation plan is used. In contrast, in [Georges et al., 2007], the use of several compilation plans is advocated, and the use of a technique known as *matched-pair comparison* is suggested for performing analysis of the results of executions of benchmarks using multiple compilation plans. They used the Jikes RVM, running a number of the SPECjvm2008 and the

DaCapo benchmarks to test their approach. They found that the use of multiple compilation plans resulted in more reliable benchmarking results, than the execution of multiple instances of the same compilation plan.

An in-depth guide, and a number of tips for benchmarking the execution of applications in the HotSpot JVM is given in [Hunt et al., 2011]. They highlight the importance of checking if the JIT compiler has completed its optimisations before taking any meaningful measurements of application performance. One way to achieve this, is to instruct the JIT Compiler to log its activity, and then confirm that this activity is not occurring during the execution of the benchmark. The HotSpot JVM can be instructed to do this by adding the *-XX:PrintCompilation* option when running an application.

10.3 Benchmarking hardware at the DPCB

10.3.1 Objectives of DPCB benchmarking

A benchmarking study was conducted of the hardware resources available to the DPCB. The primary objective of this study was to get an estimate of the available FLOPS. In particular we were interested in determining the FLOPS available in a computing node of the MareNostrum II supercomputer at the BSC, and in a node of the Prades cluster at CESCA. Knowing the FLOPS that are available within the DPCB, and also knowing the FLOP estimation of the software that must be run at the DPCB, allows for an estimation of the total time required to execute the software.

An additional objective was to determine the available I/O throughput (again at BSC and CESCA), which ultimately determines the minimum time required to process a given amount of data. The I/O benchmarks, additionally, should make it possible to determine the optimum configuration for I/O operations, regarding file size and data transfer records.

Furthermore, it was hoped that the execution of these benchmarks might highlight any aspect of the available hardware which was performing poorly and therefore could act as a bottleneck in the data reduction pipeline. Knowing the performance characteristics of the available hardware would allow us to optimise, or at least tweak, the pipeline to avoid any potential bottlenecks and make best use of the available hardware. The specification of all of the machines used in this benchmarking study are given in Appendix A.

10.3.2 Execution of benchmarks

All of the benchmarks described in Section 10.1.4, except the DaCapo benchmarks, were executed at the DPCB, as part of this study. This section provides some configuration and execution details regarding the execution of these benchmarks. In Section 10.3.3 the results of this benchmarking study are given.

10.3.2.1 MareNostrum

MareNostrum uses the SLURM system to manage jobs. *Job scripts*, which define what must be executed, using what resources, are submitted to the SLURM system. The following extract shows the typical directives which are added to a job script.

```
#!/usr/bin/csh
#
# @ initialdir = .
# @ job_name = SPECjvmTrial
# @ output = SPECjvmTrial%j.out
# @ error =  SPECjvmTrial%j.err
# @ total_tasks = 1
# @ tasks_per_node = 1
# @ cpus_per_task = 4
# @ wall_clock_limit = 02:00:00

<command to execute application>
```

10.3.2.2 CESCA

All of the machines at CESCA use the LSF job management system to manage the scheduling of jobs. We created separate scripts for executing each benchmark on the Prades and the Cadi clusters, the only difference between them is the *-m* argument, which specifies which cluster to use. The following sections show the scripts that we used to run SPECjvm2008 on the Prades and the Cadi clusters.

CESCA - Prades

```
bsub.lsf -app GAIA -J PradesStandardBench -q gaia -M 15500000
-n 32 -R "span[hosts=1]" -m "prades" -oo
/home/gaia/Benchmark/RunSPECjvm2008/logs/pradesStandardBenchLog.txt
-N -u afries@am.ub.es runStandardBench.sh
```

CESCA - Cadi

```
bsub.lsf -app GAIA -J CadiStandardBench -q gaia -M 15500000
-n 8 -R "span[hosts=1]" -m "cadi" -oo
/home/gaia/Benchmark/RunSPECjvm2008/logs/cadiStandardBenchLog.txt
-N -u afries@am.ub.es runStandardBench.sh
```

Note: the last argument passed to the *bsub.lsf* script is the name of another script. This is the job script which is actually executed, in this case, it is called `runStandardBench.sh`. The scripts of both Cadi and Prades refer to the same `runStandardBench.sh`. This script simply launches the benchmark, as shown below:

```
#!/bin/bash
cd /home/gaia/Benchmark
java -Xmx800m -jar SPECjvm2008.jar
```

10.3.2.3 SciMark 2.0

The Scimark 2.0 benchmark was executed using the following 2 configurations:

- SciMark 2.0 with the standard problem size.
- SciMark 2.0 with the large problem size.

10.3.2.4 SPECjvm2008

SPECjvm2008 was executed using the following 3 configurations:

- Trial run — an initial tests to check if all of the components of the Benchmark have been installed and are working correctly.
  Executed with *java -jar SPECjvm2008.jar -wt 5s -it 5s -bt 2 compress*
- Standard execution.
  Executed with *java -Xmx400m -jar SPECjvm2008.jar*.
- Standard execution of the benchmark, with 1 thread.
  Executed with *java -Xmx800m -jar SPECjvm2008.jar -bt 1*.

10.3.2.5 NAS

Two NAS benchmarks were executed:

- The default benchmark.
  Executed with *java -mx300M NPB3_0_JAV.BT -np8 CLASS=A*
- The serial version of the benchmark.
  Executed with *java -mx300M NPB3_0_JAV.BT -serial CLASS=A*.

10.3.2.6 LINPACK

The standard LINPACK benchmark was executed, no configuration arguments are required. Therefore it was simply executed with *java Linpack*. A script was created (shown below) to execute the benchmark 10 times, and an average score was calculated.

```
for ((  i = 1 ;  i <= 10 ;  i++  ))
do
    echo `date`
    echo "Running iteration $i"
    java Linpack
    echo `date`
    echo "-----------------------------"
    sleep 2
done
```

10.3.2.7 IOzone

The default IOzone benchmark was executed. This results in 13 individual I/O operations being performed.

10.3.3 Benchmarks results

Note: In the case of all the benchmarks, a larger score is better than a lower score.

10.3.3.1 SciMark 2.0

The results of executing the two SciMark 2.0 configurations (default problem size, and large problem size) on the several DPCB hosts are given in Figure 10.1.

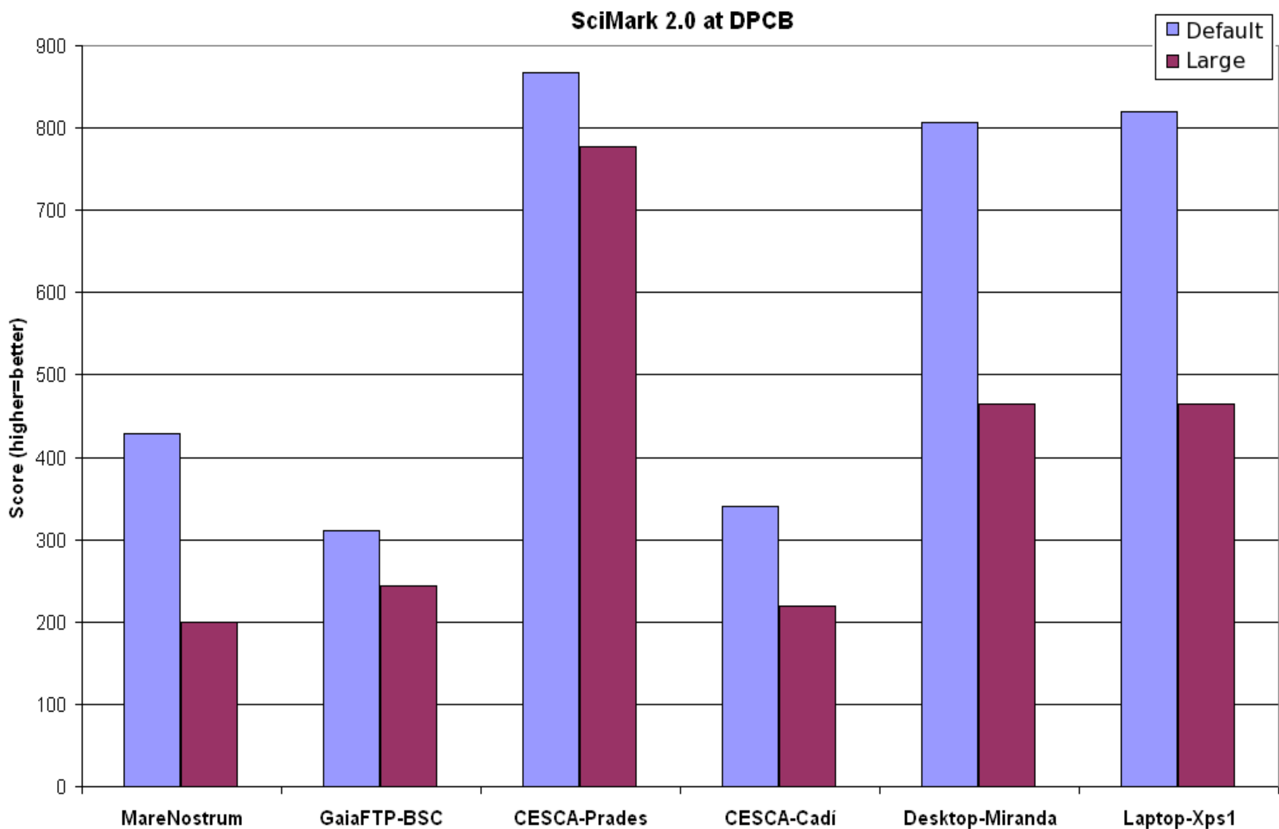

Figure 10.1: Results of SciMark 2.0 on several DPCB hosts.

10.3.3.2 SPECjvm2008

The SPECjvm2008 benchmark produces quite a lot of details in its results file. It provides a score for each of the individual algorithms that were executed and it also provides an overall composite score. Table 10.1 gives the results of executing the three selected runs of SPECjvm2008 on a number of DPCB machines. For illustrate purposes, Figure 10.2, shows the results of each of the algorithms on a MareNostrum II computing node.

—	**MN**	**GaiaFTP**	**Prades**	**Cadi**	**Miranda (HotSpot)**	**Miranda (J9)**	**Xps1**
Trial	46.15	28.73	83.82	47.8	65.22	66.18	62.20
Default	37.36	16.7	105.38	55.69	39.53	34.53	46.6
Default (1 thread)	13.61	11.12	35.18	19.94	25.73	22.87	28.52

Table 10.1: SPECjvm2008 results.

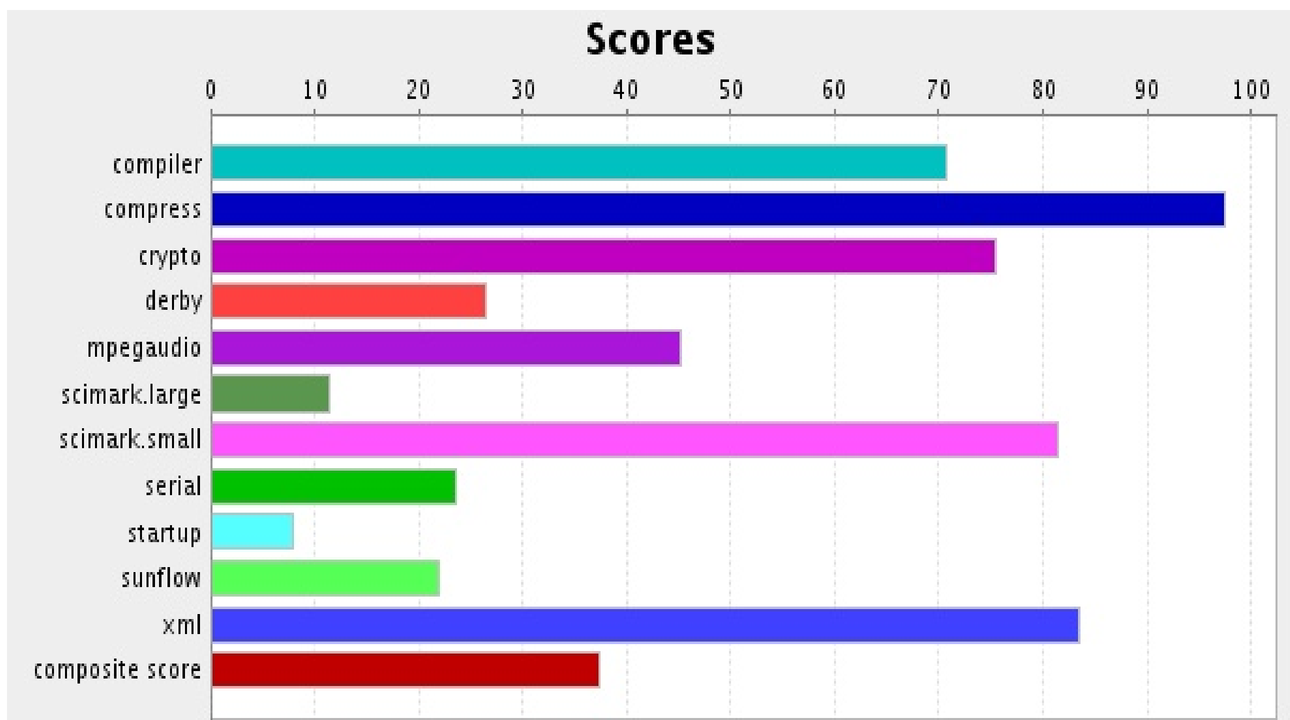

Figure 10.2: Results of SPECjvm2008 on a MareNostrum II computing node.

10.3.3.3 NAS

The following extract is an example of the output produced by the NAS Parallel Benchmark. We are particularly interested in the Million operations per second (Mops) figure, as this represents the number of floating point operations performed per second. Figure 10.3 illustrates the results obtained with this benchmark on the several DPCB hosts.

```
NAS Parallel Benchmarks Java version (NPB3_0_JAV)
Multithreaded Version BT.A np=8
No input file inputbt.data, Using compiled defaults
Size: 64 X 64 X 64
Iterations: 200 dt: 8.0E-4
Time step 1
Time step 20
Time step 40
...
Time step 200
Verification being performed for class A
accuracy setting for epsilon = 1.0000000000000004E-8
Comparison of RMS-norms of residual
0. 108.06346714637165 108.06346714637264 9.205329574672372E-15
1. 11.319730901220835 11.319730901220813 1.8831085525632136E-15
2. 25.974354511582476 25.974354511582465 4.1033323972108074E-16
3. 23.66562254467918 23.66562254467891 1.1409217698756357E-14
4. 252.78963211748658 252.78963211748345 1.2367548507256256E-14
Comparison of RMS-norms of solution error
0. 4.2348416040525025 4.2348416040525025 0.0
1. 0.443902824969957 0.443902824969957 0.0
2. 0.9669248013634564 0.9669248013634565 1.1481999665947505E-16
3. 0.8830206303976548 0.8830206303976548 0.0
4. 9.737990177082924 9.737990177082928 3.648302795746645E-16
BT.A: Verification Successful
***** NAS Parallel Benchmarks Java version (NPB3_0_JAV) BT *****
* Class             = A                                        *
* Size              = 64 X 64 X 64                             *
* Iterations        = 200                                      *
* Time in seconds   = 1002.593                                 *
* ACCTime           = 0.000                                    *
* Mops total        = 167.849                                  *
* Operation type    = floating point                           *
* Verification      = Successful                               *
* Threads requested = 8                                        *
*                                                              *
* Please send all errors/feedbacks to:                         *
* NPB Working Team                                             *
* npb@nas.nasa.gov                                             *
****************************************************************
```

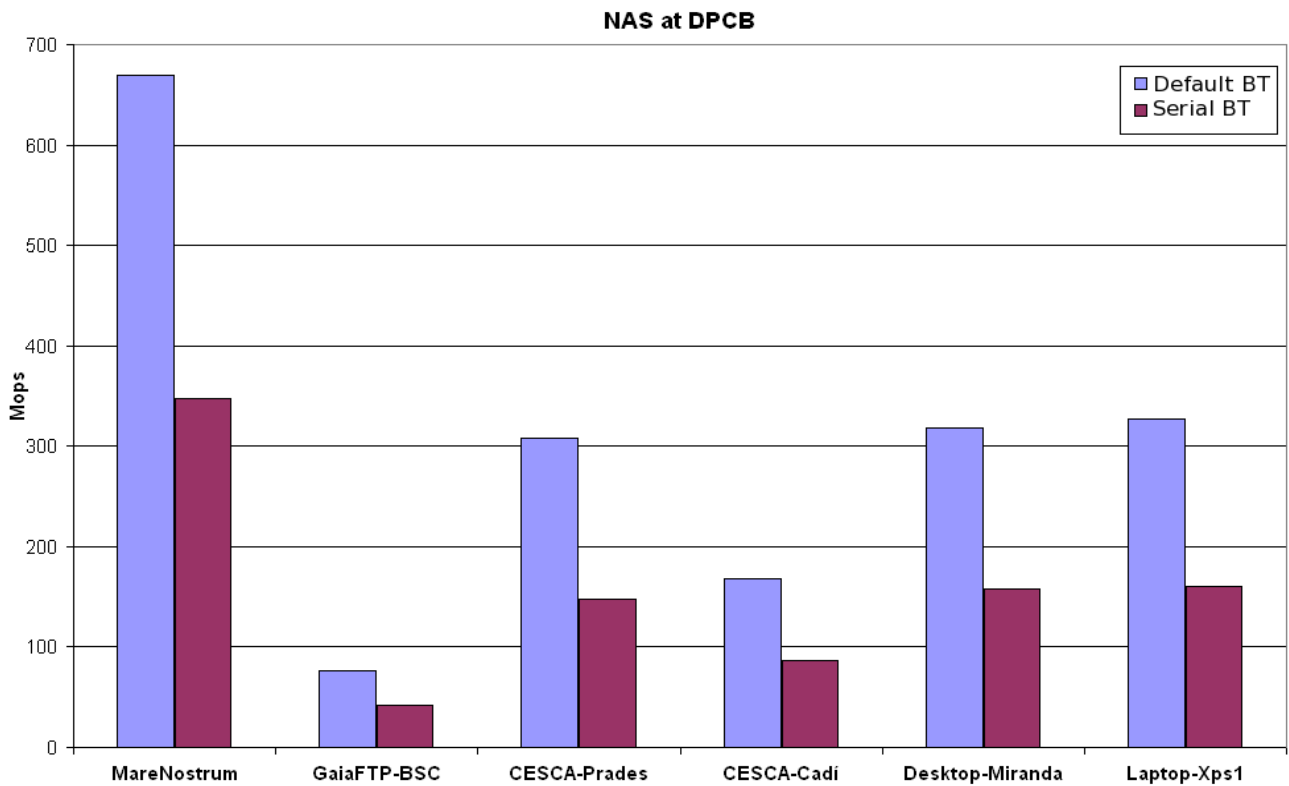

Figure 10.3: Results of NAS on several DPCB hosts.

10.3.3.4 LINPACK

Figure 10.4 illustrates the results obtained with LINPACK on the DPCB hosts. The incredible results obtained for GaiaFTP-BSC are discussed at the end of this report.

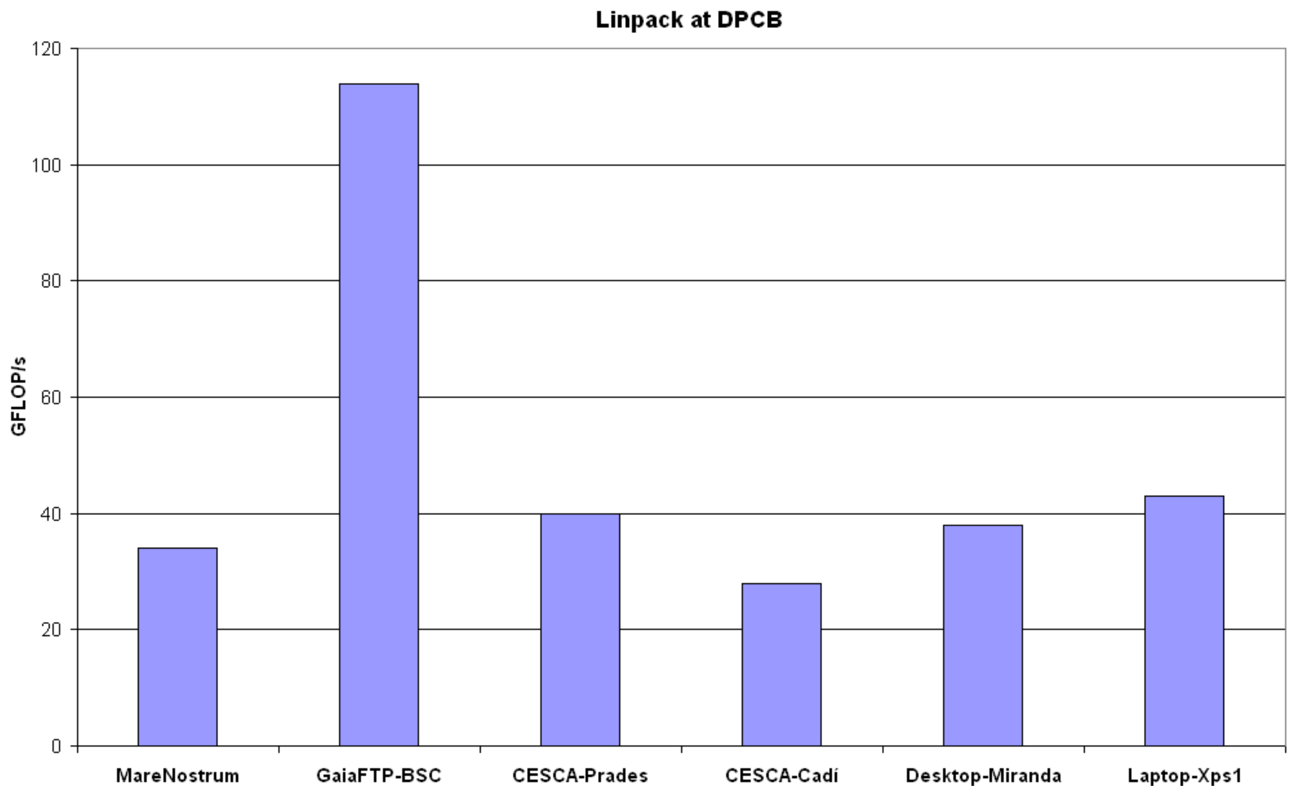

Figure 10.4: Results of LINPACK on several DPCB hosts.

10.3.3.5 IOzone

The standard IOzone benchmark performs thirteen I/O operations. It produces one large output file, the following is an extract from such an output file:

```
Auto Mode
Command line used: ./iozone -a
Output is in Kbytes/sec
Time Resolution = 0.000001 seconds.
Processor cache size set to 1024 Kbytes.
Processor cache line size set to 32 bytes.
File stride size set to 17 * record size.
                                                    random  random
 kB  reclen   write rewrite    read    reread    read   write ...
 64       4    3244  385237  520419   844652  640954  492717 ...
 64       8  248136  576282  1101022   371894 1232453  799377 ...
 64      16  340306  743988  1526887  1828508 1780008 1168097 ...
 64      32  344233  842003  1933893  2278628 2379626 1492919 ...
 64      64  365812  866463  2772930  2801873 2772930 1722886 ...
128       4  316961  371073   392505   615109  766076  496272 ...
128       8  386570  606767  1123617  1319723 1266785  814915 ...
128      16  472262  805138  1508887  1911894 1827299 1206978 ...
128      32  544601  934004  1939522   628061 2238774 1543594 ...
128      64  592699  978253  2248149  2905056 2714132 1802755 ...
128     128  518312 1040839  2985839  3297617 2248149 1967960 ...
256       4  344059  412849   532397   716876  770920  495312 ...
256       8  507966  581719   706964  1267990 1190657  831194 ...
......
```

The first column in this file gives the file size, the second column gives the record length. The term *record length* is another name for *block size* (or *buffer size*). In a filesystem, the block size specifies the size of blocks of data that the filesystem uses to read and write data. Larger block sizes will help improve disk I/O performance when using large files. This happens because the disk can read or write data for a longer period of time before having to search for the next block. On the downside, if smaller files are being used, there is the potential for more wasted disk space.

For each of the thirteen I/O operations supported by IOZone, tests are performed using files, and record lengths of a range of sizes. The record size must be less than or equal to the file size, therefore, as the file sizes increase, the number of possible record lengths also increase. For example, in the case of 64kB files, tests are performed for five record length sizes (4kB, 8kB, 16kB, 32kB and 64kB), and in the case of 128kB files, tests are performed for six record length sizes (4kB, 8kB, 16kB, 32kB, 64kB and 128kB). When the file size reaches 32MB, IOzone stops carrying out tests using record sizes less than 64kB.

The sets of results were plotted using `gnuplot`. A set of scripts were written to automatically plot this data. One such script is shown on the next page, in this case, to plot the *write* performance achieved on a node of the Cadi cluster, assuming that the output is already stored in the file *CadiDataIOzone.out*. The resulting plot is shown in Figure 10.5.

```
#!/bin/bash
OUTPUT="IOzoneCadiWrite.png"
param=$param'set dgrid3d 20,20,3\n'
param=$param'set pm3d\n'
param=$param'set grid\n'
param=$param'set ticslevel 0\n'
param=$param'set term png\n'
param=$param'set output "'$OUTPUT'"\n'
param=$param'set term png enhanced giant size 1024,800\n'
param=$param'set title "IOzone - write - Cadi"\n'
param=$param'set xlabel "kB file"\n'
param=$param'set ylabel "kB record"\n'
param=$param'set zlabel "kB/sec"\n'
param=$param'set logscale x\n'
param=$param'set logscale y\n'
param=$param'set view 15,30\n'
param=$param'set palette defined (-3 "blue", 0 "white", 1 "red")\n'
param=$param'splot "CadiDataIOzone.out" u 1:2:3 with lines\n'
echo -e $param | gnuplot
```

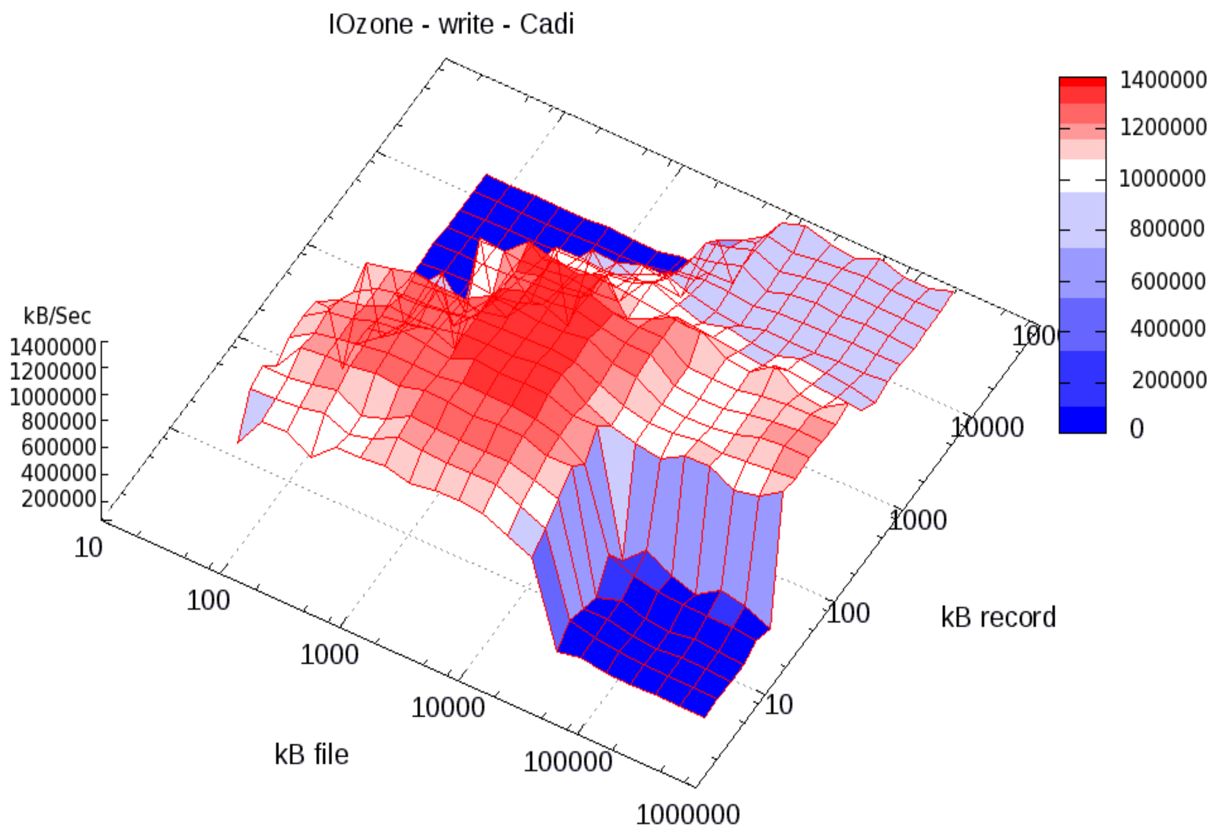

Figure 10.5: Cadi computing node — performance of write.

10.3.4 Analysis of results

Shown in the Table 10.2 is a summary of the benchmarks executed on the hardware platforms. In this table we have chosen to show the default scores that were obtained by running the

Machine	Scimark2	SPECjvm2008	NAS3.0	LINPACK
MareNostrum node	428.11	37.36	669.12	34.33
GaiaFTP	311.20	16.7	77.06	114.44
CESCA Prades	867.52	105.38	307.85	40.39
CESCA Cadi	339.83	55.69	167.85	28.61
Miranda (HotSpot)	807.16	39.53	318.23	38.15
Xps1	820.31	46.6	327.19	42.92

Table 10.2: Summary of benchmark scores obtained on tested platforms.

benchmarks with their default configuration. Also, Figure 10.6 illustrates the overall results obtained with the CPU benchmarks on the several DPCB hosts . The SPECjvm2008 and LINPACK scores have been multiplied by 10 to aid their visualisation.

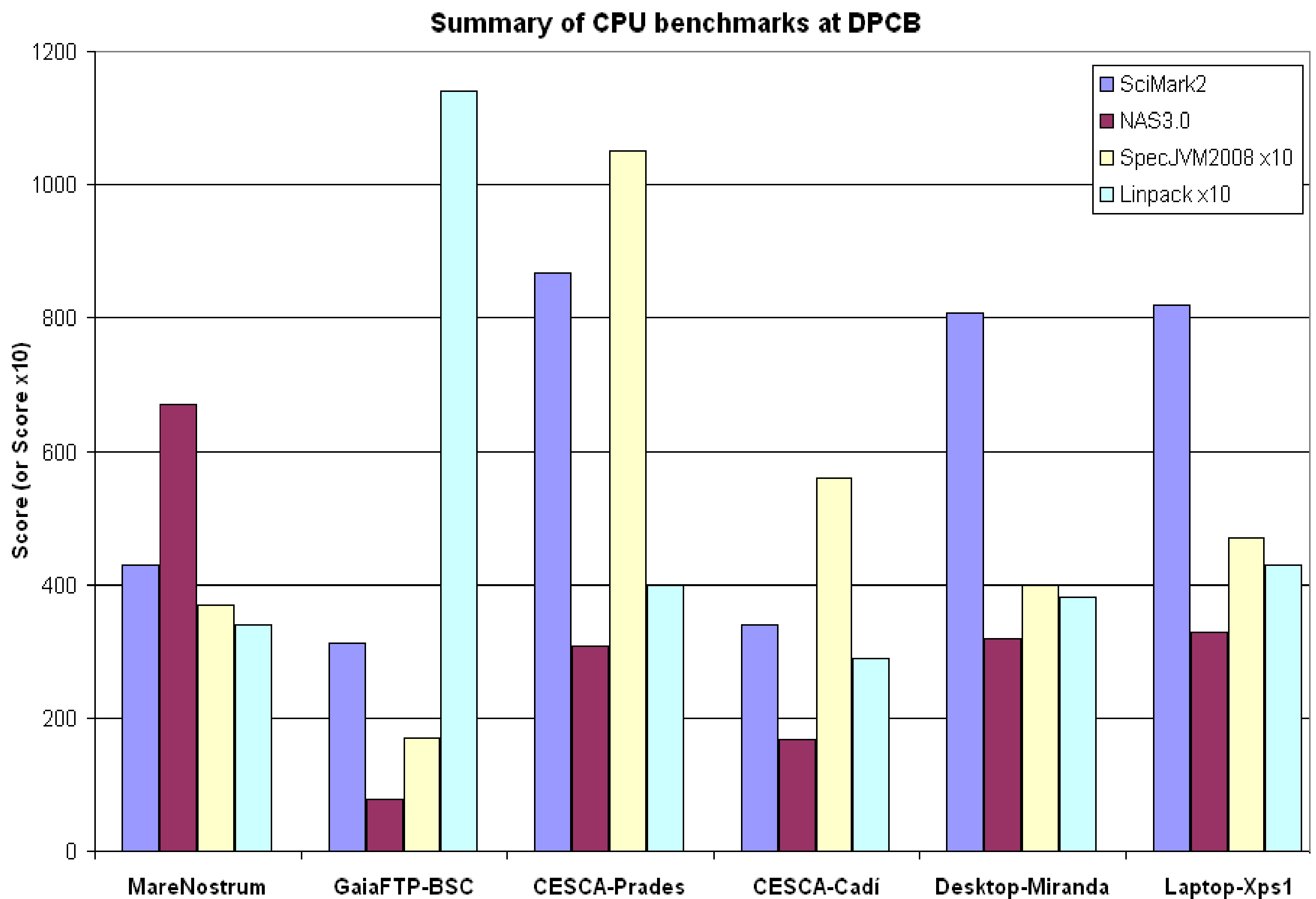

Figure 10.6: Overall CPU benchmarking results on several DPCB hosts.

The results from the Scimark, SPECjvm2008 and NAS3.0 benchmarks are largely in line with what one might expect, and the performance obtained on each machine, relative to the scores obtained on the other machines is fairly consistent across the benchmarks. The results from LINPACK are difficult to explain, as the GaiaFTP machine far outperforms the other machines, conversely GaiaFTP shows the poorest performance in the other benchmarks.

One consideration which we should keep in mind is the JVM that was used in each machine during these tests — HotSpot, OpenJDK or J9. One JVM might perform quite well on a particular machine and therefore the performance achieved by the benchmarks might come

Machine	CPU clock (GHz)
GaiaFTP	3.2
Prades2-29	3.0
Miranda	2.6
Prades30-45	2.6
Xps1	2.6
Prades1	2.5
MareNostrum	2.3
Cadi	1.8

Table 10.3: The benchmarked machines, ranked by the clock speed of their processors.

close to achieving the maximum performance obtainable on the machine, while another JVM, running on another machine might perform quite poorly and therefore not get the best possible performance from that machine.

Also we should keep in mind the fact that, in some of the benchmarks, the clock speed of the processor might be an overwhelming deciding factor, while in some other benchmarks, the processor speed might be less of a deciding factor, and the memory latency or cache size might play a larger role. Shown in Table 10.3 is a list of the machines benchmarked, ranked by the clock speed of their CPUs. As we can see, GaiaFTP is the computer with the fastest clock. Nevertheless it still cannot justify the huge difference with respect to the other computers, so we should simply conclude that there has been some anomaly in the execution of the LINPACK benchmark there.

Regarding input/output benchmarks, the results from the IOzone benchmark give us an indication of what file size and record size we should use if we wish to obtain the maximum I/O performance. Of particular interest are the *read* and the *write* operations performed on MareNostrum, as this will be crucial to the operation of IDU at the DPCB.

The performance of the *read* operation on a MareNostrum computing node is given in Figure 10.7. We can see that there is a ridge of good performance which runs through the plot, representing record sizes of about 128kB to 512kB, as highlighted by the red rectangle. All tests which involved record sizes in this range, regardless of the file sizes used, performed quite well. This indicates that record (or block) size is a very important consideration, possibly even more important than file size, in respect of determining I/O performance.

Regarding the file size, we can see that the highest performance peaks occurred with very small files, in the range of 256Kb to 4MB. As the file size is increased, the performance slightly decreases. Nevertheless, this is a *false effect*, caused by the node caches, which allow an excellent performance for small files (which, thus, can fit not only in the memory caches but even in the CPU caches). We can actually see peaks of up to 3GB/sec. When considering the manageability of data in applications running at the DPCB such as IDU, we should use file sizes of at least 10MB. Combining this with the benchmark results, it seems that IDU should use file sizes of the order of 10MB to 100MB, which may then fit in the node caches for I/O operations — allowing for up to 1.7GB/s. This is anyway an unrealistic value that can only be achieved with repeated reads of small enough files. When moving to larger files we can find the actual capability of the GPFS filesystem: 115MB/s when reading — that is, the limitation of the Gigabit Ethernet that connects the nodes with the GPFS. The writing is a bit slower, reaching 40 to 50MB/s.

In summary, according to these results, IDU should access the disk with I/O buffer sizes of around 128kB to 512kB. There is no specific restriction on the maximum file size, although we

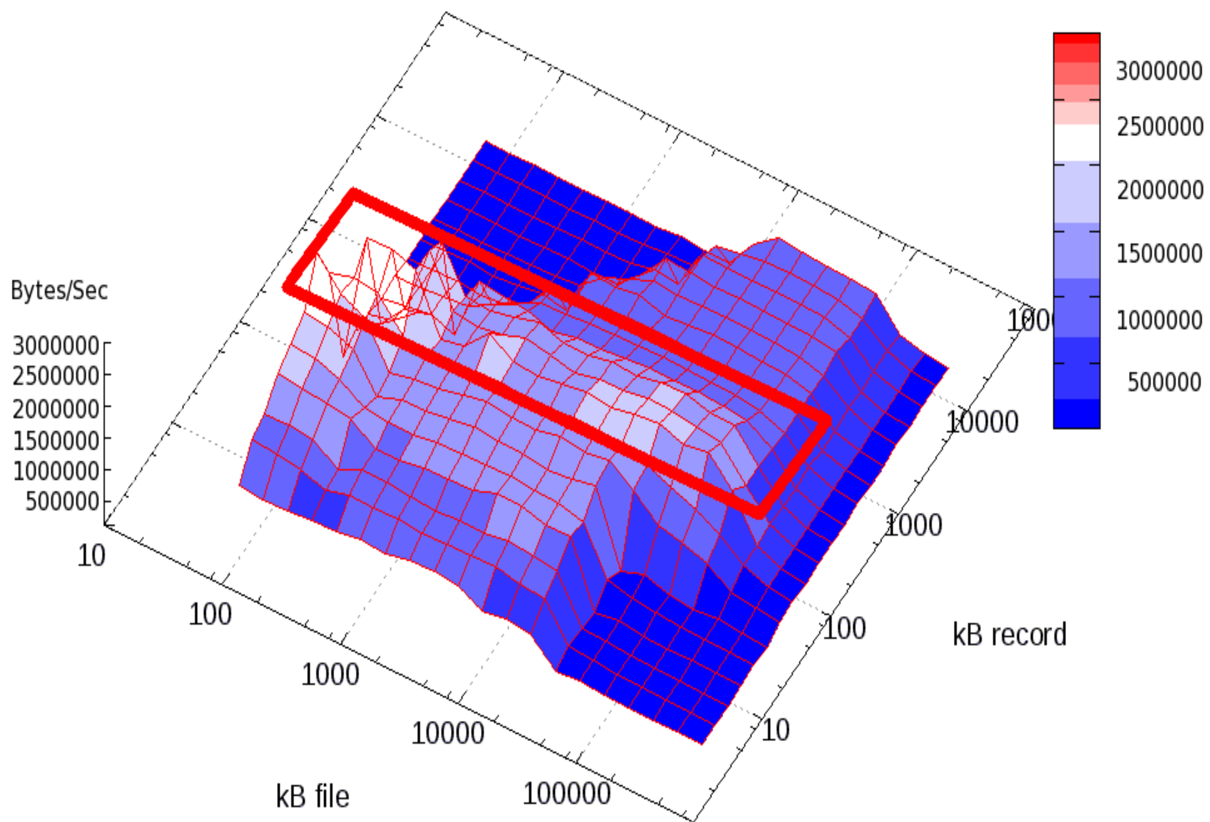

Figure 10.7: Optimal file size and record length for reading data on a MareNostrum II computing node.

recommend not to exceed 1GB. The recommended minimum could be around 10MB per file, mainly imposed by the data manageability in the local DPCB repository. The maximum sustainable throughputs per node are 115MB/s (read) and 50MB/s (write), although several nodes (within limits) accessing the GPFS at the same time may reach higher combined throughputs.

10.4 Conclusions

Performing useful benchmarking can be a difficult task, as often there are many factors which might be influencing the execution of the benchmarks. This is true for both software and hardware benchmarking. In many cases, it might not be obvious what these factors are, and the extent to which they might be influencing the results. In the case of benchmarking Java applications, the activity of the garbage collector and the JIT compiler should be taken into account. To handle such factors, a careful, scientific and consistent benchmarking approach should be followed. In order to increase the confidence that can be placed in benchmarking results, they should be executed multiple times, and certain parameters should be varied in order to observe how such changes affect the result. Additionally, statistical analysis techniques should be performed, when evaluating benchmarking results. Despite the difficulties of benchmarking, it is still a worthwhile activity, including for the purposes of configuring applications to make best use of the available hardware.

11

Conclusions

> *We can only see a short distance ahead, but we can see plenty there that needs to be done*
>
> – Alan Turing

Java has been, and continues to be, a hugely successful computer programming language and platform. It is commonly used for the development of a wide range of software, but it is yet to reach its full potential in the scientific and HPC communities. The reasons for this include the commonly held perception that Java offers lower performance than traditionally used HPC languages. Additionally, there is a lack of awareness of the strengths of the Java platform, and of the tools and libraries which are available to assist the development and efficient execution of Java applications for HPC.

Java applications execute in a complex runtime environment, composed of several sophisticated components, including the core runtime system, a garbage collector and a JIT compiler. The compilation, optimisation and execution model that is used by the Java platform is very different from the model used by statically compiled, and optimised, native code. For the majority of applications, the performance that can be achieved through the use Java is of the same level as that which can be achieved through the use of the traditionally used HPC languages. For some applications, Java will provide the best level of performance.

Typical HPC applications involve the execution of many parallel processes on many processors. Some applications require the processing of large datasets, which might necessitate the distribution of large amounts of data amongst the processors, in the form of a pipeline. The ability of applications to scale to a large number of processes is an important property in HPC. It depends both on the design of the application, and on its ability to take advantage of the available HPC resources. The factors which can limit scalability include poorly-performing inter-process communication, as well as the decrease in performance that can occur when many concurrent processes access a shared storage device. HPC environments typically possess high-speed, low-latency networks. However, in order to take full advantage of such networks, communication libraries which can natively communicate over them are required. There are many approaches that may be followed in Java in order to make use of HPC networks, and there have been many projects with this objective. Many of these projects have implemented some form of message-passing, which in the Java platform is known as MPJ. These projects have generally been of a limited scale, and many are no longer being actively supported. FastMPJ and MPJ Express are two Java-based communication libraries offering high-performance support for several HPC networks.

We present MPJ-Cache as our data distribution middleware for Java applications in HPC environments. It builds on the functionality provided by an underlying implementation of MPJ, thereby taking advantage of any supported HPC networks. It provides a high-level API of functions to Java applications running in HPC environments. To increase performance and functionality, it adds data splitting, prefetching and caching functionality. It also supports both a *pull* and a *push* model of data distribution. Tests of MPJ-Cache have shown that it outperforms alternative methods for the distribution of data amongst the nodes of a cluster.

We have contributed to the development of DpcbTools — a library of tools, designed to assist the development and execution of DPAC systems. It contains a framework for the hierarchical division of computing nodes into groups of nodes known as Node Groups (NGs), and the designation of one node within each NG as the Node Group Manager (NGM), which coordinates the work of the other nodes in its group. It also permits the division of large data processing tasks into smaller tasks, and the distribution of these tasks amongst the worker nodes. The use of this framework, together with MPJ-Cache, allows for the efficient utilisation of distributed-memory HPC environments by Java applications, especially for the processing of large datasets.

A key component of the Java runtime environment is the JIT compiler. Not only can it compile code and apply standard optimisation techniques, but it also can apply a range of speculative optimisations based on runtime feedback, which are not available to statically compiled and optimised code. In other words, a JIT compiler may observe the execution of an application for a period of time, and based on the insight that it has obtained, it might apply certain optimisations tailored to that particular execution. For long running applications, JIT compilers have the opportunity to generate very highly optimised code.

Another of the key components of the Java runtime environment, and one of the strengths of the Java platform, is automatic Garbage Collection (GC). Its presence relieves developers from the responsibility of having to write explicit memory management code, and it leads to cleaner, safer and more modular code, as well as reducing development time. Modern JVMs offer a range of GC policies and tuning options that allow for GC to be configured to meet the needs of particular applications.

The *out-of-the-box* performance of GC in modern JVMs is very high. In the majority of cases, there is not need to perform tuning. If it is deemed necessary to attempt to improve performance through tuning, then a memory usage profile of the target application must be obtained. This will allow for a profile-directed search of the available tuning options. JVM tuning is aided by the fact that the JVM is designed to expose a large amount of profiling information, and there are many high-quality profiling applications available to allow for the profiling of the executions of applications within the JVM. In some cases, we may need GC tuning because of the characteristics of the execution platform. A clear example of this is the vast improvement that we achieved in the execution of the GASS application in a shared-memory supercomputer. By adding a simple command-line parameter when executing the application, we achieved a speedup factor of 4. Not doing so meant not only a large performance decrease for our application, but also an overall performance penalty for other users of the supercomputer as well.

Performing fine tuning in such HPC environments can require some time, especially if our application requires complex deployment procedures, or if it makes use of intensive I/O — which can mask the effect of GC options. We have developed a modelling application, called MUM, that can simulate the memory usage behaviour of real applications. MUM supports the simulation of different memory usage phases during an execution — as real applications typically contain. MUM can be used to simulate the behaviour of real applications, allowing to

test, in an easy and automatic manner, the effect of many GC tuning options in any computing environment. It can also be used for the creation of benchmarks with particular characteristics.

Monitoring the execution of applications in HPC environments is normally more difficult than monitoring applications executing on a local machine. This is mainly due to the distributed nature of such environments, and the fact that obtaining an overview of the entire system requires the aggregation of monitoring information from many distributed nodes. We have presented an application called JavaMon, that makes use of the JMX framework, and allows for the monitoring and management of JVMs and instrumented applications. It is especially useful in environments where users cannot directly access the computing nodes where their applications are executed.

The Java language and platform continues to evolve at a rapid pace. The language specification, the functionality available within the Java class library, the range of supporting tools and libraries, and the performance offered by Java, all continue to be improved and expanded. In this thesis, we have found Java to not only be a viable option for the development of scientific applications for HPC environments, but to be one of the best options, when the wide range of factors which can affect the development, execution and maintenance of software are taken into account.

11.1 Future work

MPJ-Cache has been tested extensively, but it has not actually been used by real Gaia DPAC systems yet. We plan to integrate MPJ-Cache in some of such systems, in particular, for the distribution of some GASS data and for the data handling of IDU. We are also considering to use it for parallelising the execution of an IDU task that requires simultaneous access to different kinds of data, namely, the IDU XM application. We would like to expand MPJ-Cache in a number of ways, including the expansion of the API that it provides.

The upgrade of the MareNostrum supercomputer (from MareNostrum II to MareNostrum III) provides an excellent opportunity to test the software and techniques discussed in this thesis in a different HPC environment. The specification of MareNostrum III is quite different from its predecessor, including a different processor, more cores and memory per node, and a different high performance network — Infiniband. The transition to MareNostrum III should be seamless for MPJ-Cache, as network support is provided by the underlying implementation of MPJ. We will test MPJ-Cache and JavaMon on MareNostrum III, where the GASS and IDU applications will actually run from now on. We will also work on a more user-friendly interface of JavaMon, graphically showing the monitoring of all nodes in a single view.

The GC investigations described in this thesis were largely confined to MareNostrum II, which only possessed 8GB of memory per node. Therefore, the Java heap sizes that were used were less than 8GB. Currently however, some DPAC systems are being executed on machines with much larger amounts of memory, and much larger Java heaps are being used. For example, some current executions of GASS on Pirineus (see Appendix A) are using heap sizes in excess of 40GBs. Several research projects have shown that the best GC policy for a particular application depends on both the characteristics of the application and on the size of the allocated heap. We would like to better understand how well DPAC systems perform when using large heap sizes.

The MUM tool could be expanded in a number of ways, increasing its complexity. The simulation of I/O could be added. This would allow for the inclusion of I/O intensive phases in

simulations, which exist in many applications — actually that is one of the major bottlenecks in DPAC systems. This would further increase the ability of MUM to represent real applications, and increase the range of benchmarks that it could be used to create.

Bibliography

[Allen et al., 2007] Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Stelle, G. L., and Tobin-Hochstadt, S. (2007). The Fortress Language Specification. Technical report, Sun Microsystems, Inc.

[Alpern et al., 2005] Alpern, B., Augart, S., Blackburn, S. M., Butrico, M., Cocchi, A., Cheng, P., Dolby, J., Fink, S., Grove, D., Hind, M., McKinley, K. S., Mergen, M., Moss, J. E. B., Ngo, T., and Sarkar, V. (2005). The Jikes research virtual machine project: building an open-source research community. *IBM Syst. J.*, 44(2):399–417.

[Amedro et al., 2009] Amedro, B., Caromel, D., Huet, F., Bodnartchouk, V., Delb, C., and Taboada, G. L. (2009). ProActive: Using a Java Middleware for HPC Design, Implementation and Benchmarks. *International Journal of Computers and Communications*, 3(3):49–57.

[Arnold et al., 2005] Arnold, M., Fink, S., Grove, D., Hind, M., and Sweeney, P. (2005). A Survey of Adaptive Optimization in Virtual Machines. *Proceedings of the IEEE*, 93(2):449 –466.

[Arnold et al., 2002] Arnold, M., Hind, M., and Ryder, B. G. (2002). Online feedback-directed optimization of Java. *SIGPLAN Not.*, 37(11):111–129.

[Atienza et al., 2008] Atienza, D., Angiolini, F., Murali, S., Pullini, A., Benini, L., and De Micheli, G. (2008). Network-on-Chip design and synthesis outlook. *Integration*, 41(3):340–359.

[Badia et al., 2003] Badia, R. M., Labarta, J., Sirvent, R., Pérez, J. M., Cela, J. M., and Grima, R. (2003). Programming Grid Applications with GRID Superscalar. *Journal of Grid Computing*, 1(2):151 –170.

[Baduel et al., 2006] Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., and Quilici, R. (2006). *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag.

[Baker et al., 1999] Baker, M., Carpenter, B., Fox, G., Ko, S.-H., and Lim, S. (1999). mpiJava: an object-oriented Java interface to MPI. In *Proc. International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP*.

[Baker et al., 2005] Baker, M., Carpenter, B., and Shafi, A. (2005). A Pluggable Architecture for High-Performance Java Messaging. *Cluster Computing and Grid 2005 Works in Progress*, 6(10).

[Baker et al., 2006] Baker, M., Carpenter, B., and Shafi, A. (2006). MPJ Express: Towards Thread Safe Java HPC. In *CLUSTER*.

[Baker et al., 2007] Baker, M., Carpenter, B., and Shafi, A. (2007). A Buffering Layer to Support Derived Types and Proprietary Networks for Java HPC. *Scalable Computing: Practice and Experience*, 8(4).

[Blackburn et al., 2006] Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. (2006). The DaCapo benchmarks: Java benchmarking development and analysis. *SIGPLAN Not.*, 41:169–190.

[Blythe, 2006] Blythe, D. (2006). The Direct3D 10 system. *ACM Trans. Graph.*, 25(3):724–734.

[Borthakur, 2007] Borthakur, D. (2007). The Hadoop Distributed File System: Architecture and Design. The Apache Software Foundation.

[Brecht et al., 2006] Brecht, T., Arjomandi, E., Li, C., and Pham, H. (2006). Controlling garbage collection and heap growth to reduce the execution time of Java applications. *ACM Trans. Program. Lang. Syst.*, 28:908–941.

[BSC, 2012] BSC (2012). Barcelona Supercomputing Center. `http://www.bsc.es` [Checked November 2012].

[Butters, 2007] Butters, A. M. (2007). Total Cost of Ownership: A Comparison of C/C++ and Java. Evans Data Corp - BEA Custom Research - White Paper.

[Carpenter et al., 1999] Carpenter, B., Fox, G., Ko, S.-H., and Lim, S. (1999). mpiJava 1.2: API Specification. Syracuse University Library - http://surface.syr.edu/npac/66/.

[Carpenter et al., 2000] Carpenter, B., Getov, V., Judd, G., Skjellum, A., and Fox, G. (2000). MPJ: MPI-like message passing for Java. *Concurrency, Practice and Experience*, 12(11):1019–1038.

[Cave et al., 2011] Cave, V., Zhao, J., Shirako, J., and Sarkar, V. (2011). Habanero-Java: the New Adventures of Old X10.

[CESCA, 2012] CESCA (2012). *Centre de Serveis Científics i Acadèmics de Catalunya*. `http://www.cesca.cat` [Checked November 2012].

[Chamberlain et al., 2007] Chamberlain, B. L., Callahan, D., and Zima, H. P. (2007). Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21:291–312.

[Chambers et al., 1991] Chambers, C., Ungar, D., and Lee, E. (1991). An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes.

[Charles et al., 2005] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. (2005). X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. *SIGPLAN Not.*, 40:519–538.

[Chen, 2010] Chen, H. (2010). Comparative Study of C, C++, C# and Java Programming Languages. `http://publications.theseus.fi/bitstream/handle/10024/16995/Chen_Hao.pdf` [Checked November 2012].

[Cray, 2011] Cray (2011). Chapel Language Specification - Version 0.8. http://chapel.cray.com/spec/spec-0.8.pdf.

[Damaraju et al., 2012] Damaraju, S., Varghese, G., Jahagirdar, S., Khondker, T., Milstrey, R., Sarkar, S., Siers, S., Stolero, I., and Subbiah, A. (2012). A 22nm IA multi-CPU and GPU System-on-Chip. In *ISSCC*, pages 56–57.

[Dell, 2002] Dell (2002). Load Sharing Facility. `http://www-03.ibm.com/systems/technicalcomputing/platformcomputing/products/lsf/` [Checked November 2012].

[Detlefs et al., 2004] Detlefs, D., Flood, C., Heller, S., and Printezis, T. (2004). Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*, ISMM '04, pages 37–48, New York, NY, USA. ACM.

[Deutsch and Schiffman, 1984] Deutsch, L. P. and Schiffman, A. M. (1984). Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 297–302, New York, NY, USA. ACM.

[Domani et al., 2002] Domani, T., Goldshtein, G., Kolodner, E. K., Lewis, E., Petrank, E., and Sheinwald, D. (2002). Thread-local heaps for Java. *SIGPLAN Not.*, 38:76–87.

[Dongarra et al., 2007] Dongarra, J., Gannon, D., Fox, G., and Kennedy, K. (2007). The Impact of Multicore on Computational Science Software. *CTWatch Quarterly*, 3(1).

[Dongarra et al., 2003] Dongarra, J. J., Luszczek, P., and Petitet, A. (2003). The LINPACK benchmark: Past, present, and future. Concurrency and Computation: Practice and Experience. *Concurrency and Computation: Practice and Experience*, 15:2003.

[Doolin et al., 1999] Doolin, D., Dongarra, J., and Seymour, K. (1999). JLAPACK - Compiling LAPACK Fortran to Java. *Scientific Programming*, 7(2):111–138.

[Expósito et al., 2012] Expósito, R., Taboada, G. L., Ramos, S., Touriño, J., and Doallo, R. (2012). FastMPJ: a Scalable and Efficient Java Message-Pasing Library. *Springer Computing Journal*.

[Fang et al., 2011] Fang, J., Varbanescu, A., and Sips, H. (2011). A Comprehensive Performance Comparison of CUDA and OpenCL. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216 –225.

[Fitzgerald and Tarditi, 2000] Fitzgerald, R. and Tarditi, D. (2000). The case for profile-directed selection of garbage collectors. *SIGPLAN Not.*, 36:111–120.

[Flanagan, 2012] Flanagan, M. T. (2012). Michael Thomas Flanagan's Java Scientific Library. `http://www.ee.ucl.ac.uk/~mflanaga/java/index.html` [Checked November 2012].

[Flynn, 1972] Flynn, M. (1972). Some Computer Organizations and Their Effectiveness. *IEEE Trans. Comput.*, C-21:948+.

[Forum, 1994] Forum, M. P. (1994). MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA.

[Forum, 1997] Forum, M. P. (1997). MPI-2: Extensions to the Message-Passing Interface. Technical report, Knoxville, TN, USA.

[Fries et al., 2011a] Fries, A., Castañeda, J., Isasi, Y., Taboada, G. L., Sirvent, R., and Portell, J. (2011a). An efficient framework for Java data processing systems in HPC environments. In *High-Performance Computing in Remote Sensing (SPIE 8183)*. American Institute of Physics.

[Fries et al., 2011b] Fries, A., Portell, J., Isasi, Y., Castañeda, J., R., S., and Taboada, G. L. (2011b). MPI-based Solution for Efficient Data Access in Java HPC. In *First International Conference on Advanced Communications and Computation (INFOCOMP'2011)*, pages 149–154. IARIA.

[Fries et al., 2010] Fries, A., Portell i de Mora, J., and Sirvent, R. (2010). Java-based communication in a High Performance Computing environment. *EAS Publications Series*, 45(1):103–106.

[Frumkin et al., 2003] Frumkin, M. A., Schultz, M., Jin, H., and Yan, J. (2003). Performance and Scalability of the NAS Parallel Benchmarks in Java. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 139.1–, Washington, DC, USA. IEEE Computer Society.

[GCJ, 2012] GCJ (2012). GNU Compiler for Java. `http://gcc.gnu.org/java/docs.html` [Checked November 2012].

[Georges et al., 2007] Georges, A., Buytaert, D., and Eeckhout, L. (2007). Statistically Rigorous Java Performance Evaluation. *SIGPLAN Not.*, 42(10):57–76.

[Ghazawi et al., 2005] Ghazawi, T. E., Carlson, W., Sterling, T., and Yelick, K. (2005). *UPC: Distributed Shared Memory Programming*. Wiley.

[Goglin, 2011] Goglin, B. (2011). High-performance message-passing over generic Ethernet hardware with Open-MX. *Parallel Comput.*, 37(2):85–100.

[Gosling et al., 2012] Gosling, J., Joy, B., Steele, G., Bracha, G., and Buckley, A. (2012). The Java Language Specification Java SE 7 Edition. `http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf` [Checked November 2012].

[Grcevski et al., 2004] Grcevski, N., Kielstra, A., Stoodley, K., Stoodley, M. G., and Sundaresan, V. (2004). Java Just-in-Time Compiler and Virtual Machine Improvements for Server and Middleware Applications. In *Virtual Machine Research and Technology Symposium*, pages 151–162. USENIX.

[Hansen, 1974] Hansen, G. J. (1974). *Adaptive systems for the dynamic run-time optimization of programs*. PhD thesis, Pittsburgh, PA, USA. AAI7420364.

[Heimsund, 2005] Heimsund, B. (2005). High performance numerical libraries in Java.

[Henderson, 1995] Henderson, R. (1995). Job scheduling under the Portable Batch System. In Feitelson, D. and Rudolph, L., editors, *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 279–294. Springer Berlin / Heidelberg.

[Hertz and Berger, 2005] Hertz, M. and Berger, E. D. (2005). Quantifying the performance of garbage collection vs. explicit memory management. *SIGPLAN Not.*, 40(10):313–326.

[Hoste et al., 2010] Hoste, K., Georges, A., and Eeckhout, L. (2010). Automated just-in-time compiler tuning. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 62–72, New York, NY, USA. ACM.

[hprof, 2012] hprof (2012). The Hprof profiler. `http://publib.boulder.ibm.com/infocenter/javasdk/v6r0/index.jsp` [Checked November 2012].

[Hunt et al., 2011] Hunt, C., Hohensee, P., and John, B. (2011). *Java Performance*. Java Series. Prentice Hall.

[Ibis, 2012] Ibis (2012). Ibis project. `http://www.cs.vu.nl/ibis/` [Checked November 2012].

[IBM, 2011] IBM (2011). IBM on Big Data. `http://www.ibm.com/smarterplanet/global/files/us__en_us__smarter_computing__ibm_data_final.pdf`.

[IOzone, 2012] IOzone (2012). IOzone Benchmark - Java version. `http://www.iozone.org/` [Checked November 2012].

[Ishizaki et al., 2003] Ishizaki, K., Takeuchi, M., Kawachiya, K., Suganuma, T., Gohda, O., Inagaki, T., Koseki, A., Ogata, K., Kawahito, M., Yasue, T., Ogasawara, T., Onodera, T., Komatsu, H., and Nakatani, T. (2003). Effectiveness of cross-platform optimizations for a Java just-in-time compiler. *SIGPLAN Not.*, 38(11):187–204.

[JCuda, 2012] JCuda (2012). JCuda — Java bindings for CUDA. `http://www.jcuda.org/` [Checked November 2012].

[Jette et al., 2002] Jette, M. A., Yoo, A. B., and Grondona, M. (2002). SLURM: Simple Linux Utility for Resource Management. In *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*, pages 44–60. Springer-Verlag.

[JOCL, 2012] JOCL (2012). Java bindings for OpenCL. `http://www.jocl.org/` [Checked November 2012].

[Jones et al., 2010] Jones, R., Hosking, A., and Moss, E. (2010). *Advanced Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Chapman & Hall/CRC Applied Algorithms and Data Structures. Taylor and Francis.

[Jprofiler, 2012] Jprofiler (2012). The Jprofiler profiler. `http://www.ej-technologies.com/products/jprofiler/overview.html` [Checked November 2012].

[Kahan, 1998] Kahan, W, D. J. (1998). How Java's Floating-Point Hurts Everyone Esverywhere. Technical report. Presented at ACM 1998 Workshop on Java for High-Performance Network Computing held at Stanford University http://www.cs.berkeley.edu/ wkahan/-JAVAhurt.pdf.

[Kambites and Bull, 2001] Kambites, M. E. and Bull, J. M. (2001). An OpenMP-like Interface for Parallel Programming in Java. In *Concurrency and Computation: Practice and Experience.*

[Kaminsky, 2007] Kaminsky, A. (2007). Parallel Java: A Unified API for Shared Memory and Cluster Parallel Programming in 100% Java. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8.

[Khronos Group, 2010] Khronos Group (2010). *The OpenCL Specification.* http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf [Checked August 2012].

[Klemm et al., 2007] Klemm, M., Bezold, M., Veldema, R., and Philippsen, M. (2007). JaMP: an implementation of OpenMP for a Java DSM: Research Articles. *Concurr. Comput. : Pract. Exper.*, 19(18):2333–2352.

[Komatsu et al., 2010] Komatsu, K., Sato, K., Arai, Y., Koyama, K., Takizawa, H., and Kobayashi, H. (2010). Evaluating Performance and Portability of OpenCL Programs. In *The Fifth International Workshop on Automatic Performance Tuning.*

[Lagergren and Hirt, 2010] Lagergren, M. and Hirt, M. (2010). *Oracle Jrockit: The Definitive Guide.* Packt Publishing.

[Lea, 2000] Lea, D. (2000). A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, Java '00, pages 36–43, New York, NY, USA. ACM.

[Lea, 2005] Lea, D. (2005). The java.util.concurrent synchronizer framework. *Sci. Comput. Program.*, 58(3):293–309.

[Lindegren et al., 2008] Lindegren, L., Babusiaux, C., Bailer-Jones, C., Bastian, U., Brown, A. G. A., Cropper, M., Høg, E., Jordi, C., Katz, D., van Leeuwen, F., Luri, X., Mignard, F., de Bruijne, J. H. J., and Prusti, T. (2008). The Gaia mission: science, organization and present status. In *IAU Symposium*, volume 248 of *IAU Symposium*, pages 217–223.

[Lindegren et al., 2011] Lindegren, L., Lammers, U., Hobbs, D., O'Mullane, W., Bastian, U., and Hernández, J. (2011). The astrometric core solution for the Gaia mission. Overview of models, algorithms and software implementation.

[Lindholm et al., 2012] Lindholm, T., Yellin, F., Bracha, G., and Buckley, A. (2012). The Java Language Specification Java SE 7 Edition. `http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf` [Checked November 2012].

[McCarthy, 1978] McCarthy, J. (1978). History of Lisp. In *HISTORY OF PROGRAMMING LANGUAGES*, pages 217–223. Academic Press.

[Meijer and Goug, 2000] Meijer, E. and Goug, J. (2000). Technical Overview of the Common Language Runtime.

[Mellanox Technologies, 2007] Mellanox Technologies (2007). InfiniBand Software and Protocols Enable Seamless Off-the-shelf Applications Deployment. `http://www.mellanox.com/pdf/whitepapers/WP_2007_IB_Software_and_Protocols.pdf` [Checked November 2012].

[Mignard et al., 2007] Mignard, F., Bailer-Jones, C., Bastian, U.and Drimmel, R., Eyer, L., Katz, D., van Leeuwen, F., Luri, X., O'Mullane, W., Passot, X., Pourbaix, D., and Prusti, T. (2007). Gaia: Organisation and challenges for the data processing.

[Mikheev et al., 2002] Mikheev, V., Lipsky, N., Gurchenkov, D., Pavlov, P., Sukharev, V., Markov, A., Kuksenko, S., Fedoseev, S., Leskov, D., and Yeryomin, A. (2002). Overview of excelsior JET, a high performance alternative to java virtual machines. In *Proceedings of the 3rd international workshop on Software and Performance*, WOSP '02, pages 104–113, New York, NY, USA. ACM.

[Moab, 2012] Moab (2012). Moab meta-scheduler. `http://www.adaptivecomputing.com/` [Checked November 2012].

[Moore, 1965] Moore, G. E. (1965). Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117.

[Moreira et al., 2000] Moreira, J. E., Midkiff, S. P., Gupta, M., Artigas, P. V., Snir, M., and Lawrence, R. D. (2000). Java Programming for High-Performance Numerical Computing. *IBM System Journal*, 39.

[MPI, 2012] MPI (2012). MPI. `http://www.mcs.anl.gov/research/projects/mpi/` [Checked November 2012].

[Mytkowicz et al., 2010] Mytkowicz, T., Diwan, A., Hauswirth, M., and Sweeney, P. F. (2010). Evaluating the accuracy of Java profilers. *SIGPLAN Not.*, 45(6):187–197.

[Nickolls et al., 2008] Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53.

[Numrich and Reid, 1998] Numrich, R. W. and Reid, J. (1998). Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31.

[ojAlgo, 2012] ojAlgo (2012). ojAlgo Library. http://ojalgo.org/ [Checked November 2012].

[O'Mullane et al., 2011a] O'Mullane, W., Lammers, U., Lindegren, L., Hernández, J., and Hobbs, D. (2011a). Implementing the Gaia Astrometric Global Iterative Solution (AGIS) in Java. *Exper.Astron.*, 31:215–241.

[O'Mullane et al., 2011b] O'Mullane, W., Luri, X., Parsons, P., Lammers, U., Hoar, J., and Hernández, J. (2011b). Using Java for distributed computing in the Gaia satellite data processing. *CoRR*, abs/1108.0355.

[PerfAnal, 2000] PerfAnal (2000). Profiling tool. http://www.oracle.com/technetwork/articles/javase/perfanal-137231.html [Checked November 2012].

[Philippsen et al., 2001] Philippsen, M., Boisvert, R. F., Getov, V., Pozo, R., Moreira, J. E., Gannon, D., and Fox, G. (2001). JavaGrande - High Performance Computing with Java. In *Proceedings of the 5th International Workshop on Applied Parallel Computing, New Paradigms for HPC in Industry and Academia*.

[Qian et al., 2004] Qian, Y., Afsahi, A., and Zamani, R. (2004). Myrinet networks: a performance study. In *Network Computing and Applications, 2004. (NCA 2004). Proceedings. Third IEEE International Symposium on*, pages 323 – 328.

[SciMark, 2012] SciMark (2012). SciMark 2.0 benchmark suite. http://math.nist.gov/scimark2 [Checked November 2012].

[Sestoft, 2010] Sestoft, P. (2010). Numeric performance in C, C# and Java. http://www.itu.dk/people/sestoft/papers/numericperformance.pdf [Checked November 2012].

[Seymour and Dongarra, 2003] Seymour, K. and Dongarra, J. (2003). Automatic Translation of Fortran to JVM Bytecode. *Concurrency and Computation: Practice and Experience*, 15(3-5):202–207.

[Shafi et al., 2009] Shafi, A., Carpenter, B., and Baker, M. (2009). Nested Parallelism for Multicore HPC Systems using Java. *Journal of Parallel and Distributed Computing*, 69(6):532–545.

[Shafi and Manzoor, 2009] Shafi, A. and Manzoor, J. (2009). Towards efficient shared memory communications in MPJ express. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1 –7.

[Shirako et al., 2008] Shirako, J., Kasahara, H., and Sarkar, V. (2008). Languages and Compilers for Parallel Computing. chapter Language Extensions in Support of Compiler Parallelization, pages 78–94. Springer-Verlag, Berlin, Heidelberg.

[Singer et al., 2007] Singer, J., Brown, G., Watson, I., and Cavazos, J. (2007). Intelligent selection of application-specific garbage collectors. In *Proceedings of the 6th international symposium on Memory management*, ISMM '07, pages 91–102, New York, NY, USA. ACM.

[Smith and Bull, 2000] Smith, L. A. and Bull, J. M. (2000). Java for High Performance Computing.

[Soman et al., 2004] Soman, S., Krintz, C., and Bacon, D. F. (2004). Dynamic selection of application-specific garbage collectors. In *Proceedings of the 4th international symposium on Memory management*, ISMM '04, pages 49–60, New York, NY, USA. ACM.

[SPECjvm2008, 2012] SPECjvm2008 (2012). SPECjvm2008 benchmark suite. http://www.spec.org/jvm2008/ [Checked November 2012].

[Suganuma et al., 2001] Suganuma, T., Yasue, T., Kawahito, M., Komatsu, H., and Nakatani, T. (2001). A dynamic optimization framework for a Java just-in-time compiler. *SIGPLAN Not.*, 36(11):180–195.

[Suganuma et al., 2005] Suganuma, T., Yasue, T., Kawahito, M., Komatsu, H., and Nakatani, T. (2005). Design and evaluation of dynamic optimizations for a Java just-in-time compiler. *ACM Trans. Program. Lang. Syst.*, 27(4):732–785.

[Suganuma et al., 2002] Suganuma, T., Yasue, T., and Nakatani, T. (2002). An Empirical Study of Method In-lining for a Java Just-in-Time Compiler. In *Proceedings of the 2nd Java™ Virtual Machine Research and Technology Symposium*, pages 91–104, Berkeley, CA, USA. USENIX Association.

[Sundaresan et al., 2006] Sundaresan, V., Maier, D., Ramarao, P., and Stoodley, M. (2006). Experiences with Multi-threading and Dynamic Class Loading in a Java Just-In-Time Compiler. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 87–97, Washington, DC, USA. IEEE Computer Society.

[Szeliga et al., 2009] Szeliga, B., Nguyen, T., and Shi, W. (2009). DiSK: A distributed shared disk cache for HPC environments. In *Collaborative Computing: Networking, Applications and Worksharing, 2009. CollaborateCom 2009. 5th International Conference on*, pages 1 –8.

[Taboada et al., 2011a] Taboada, G. L., Ramos, S., Expósito, R., Touriño, J., and Doallo, R. (2011a). Java in the High Performance Computing arena: Research, practice and experience. *Science of Computer Programming.*

[Taboada et al., 2007] Taboada, G. L., Teijeiro, C., and Tourino, J. (2007). High Performance Java Remote Method Invocation for Parallel Computing on Clusters. In *Computers and Communications, 2007. ISCC 2007. 12th IEEE Symposium on*, pages 233 –239.

[Taboada et al., 2003] Taboada, G. L., Touriño, J., and Doallo, R. (2003). Performance analysis of Java message-passing libraries on fast Ethernet, Myrinet and SCI clusters. In *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, pages 118 – 126.

[Taboada et al., 2008] Taboada, G. L., Touriño, J., and Doallo, R. (2008). Java Fast Sockets: Enabling high-speed Java communications on high performance clusters. *Computer Communications*, 31(17):4049 – 4059.

[Taboada et al., 2009] Taboada, G. L., Touriño, J., and Doallo, R. (2009). Java for high performance computing: assessment of current research and practice. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 30–39, New York, NY, USA. ACM.

[Taboada et al., 2011b] Taboada, G. L., Touriño, J., Doallo, R., Shafi, A., Baker, M., and Carpenter, B. (2011b). Device level communication libraries for high-performance computing in Java. *Concurr. Comput. : Pract. Exper.*, 23(18):2382–2403.

[Tejedor and Badia, 2008] Tejedor, E. and Badia, R. (2008). COMP Superscalar: Bringing GRID Superscalar and GCM Together. In *Cluster Computing and the Grid, 2008. CCGRID '08. 8th IEEE International Symposium on*, pages 185 –193.

[Tejedor et al., 2008] Tejedor, E., Badia, R., and Albert, P. (2008). Accelerating the parallel distributed execution of Java HPC applications. *Technical Report.*

[The Khronos Group Inc, 2012] The Khronos Group Inc (2012). *OpenGL 4.3 Core Profile Specification.*

[Thiruvathukal, 1998] Thiruvathukal, G. K. (1998). Java Grande Forum Report: Making Java Work for High-End Computing. http://www.javagrande.org/sc98/sc98grande.pdf.

[Tiobe, 2012] Tiobe (2012). Tiobe list. http://www.tiobe.com [Checked November 2012].

[TOP500, 2012] TOP500 (2012). TOP500 Supercomputers. http://www.top500.org [Checked November 2012].

[Van Nieuwpoort et al., 2005] Van Nieuwpoort, R. V., Maassen, J., Wrzesinska, G., Hofman, R., Jacobs, C., Kielmann, T., and Bal, H. E. (2005). Ibis: a Flexible and Efficient Java based Grid Programming Environment. *Concurrency and Computation: Practice and Experience*, 17(7-8):1079–1107.

[Veldema et al., 2001] Veldema, R., Hofman, R. F. H., Bhoedjang, R. A. F., and Bal, H. E. (2001). Runtime optimizations for a Java DSM implementation. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, JGI '01, pages 153–162, New York, NY, USA. ACM.

[Veldema et al., 2002] Veldema, R., Jacobs, C. J. H., Hofman, R. F. H., Rutger, J., Hofman, F. H., and Bal, H. E. (2002). Object Combining: A new aggressive optimization for Object Intensive Programs. In *In Proceedings of the 2002 joint ACM-ISCOPE Conference on Java Grande*, pages 165–174. ACM Press.

[Wang et al., 2009] Wang, F., Oral, S., Shipman, G., Drokin, O., Wang, T., and Huang, I. (2009). Understanding Lustre Filesystem Internals. Technical Report ORNL/TM-2009/117, Oak Ridge National Lab., National Center for Computational Sciences.

[Weiland, 2007] Weiland, M. (2007). Chapel, Fortress and X10: Novel Languages for HPC. Technical report, The University of Edinburgh. http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0706.pdf.

[Welch et al., 2008] Welch, B., Unangst, M., Abbasi, Z., Gibson, G., Mueller, B., Small, J., Zelenka, J., and Zhou, B. (2008). Scalable performance of the Panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 2:1–2:17, Berkeley, CA, USA. USENIX Association.

[Wilson et al., 1995] Wilson, P. R., Johnstone, M. S., Neely, M., and Boles, D. (1995). Dynamic Storage Allocation: A Survey and Critical Review. pages 1–116. Springer-Verlag.

[Xprof, 2012] Xprof (2012). The Xprof profiler. http://docs.oracle.com/javase/7/docs/technotes/tools/windows/java.html [Checked November 2012].

[Yan et al., 2009] Yan, Y., Grossman, M., and Sarkar, V. (2009). JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA. In *Euro-Par 2009 Parallel Processing*, volume 5704 of *Lecture Notes in Computer Science*, pages 887–899. Springer Berlin Heidelberg.

[Yelick et al., 1998] Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., and Aiken, A. (1998). Titanium: A High-Performance Java Dialect. *Concurrency: Practice and Experience*, 10(11–13):825–836. Special Issue: Java for High-performance Network Computing.

[YourKit, 2012] YourKit (2012). The YourKit Java profiler. http://www.yourkit.com [Checked November 2012].

Glossary

Adaptive optimisation An optimisation technique that may be performed when executing code within a managed runtime environment, as opposed to statically performed optimisations performed before runtime. Adaptive optimisations can take advantage of insight gained from running code for a period of time, and use this insight to produce highly optimised code. Such optimisations are sometimes called Feedback-Directed Optimisations (FDOs). An Adaptive Optimisation System (AOS) is a system which can automatically decide when to apply such optimisations. 6, 32, 142

Apache Software Foundation (ASF) A highly successful community of developers working together on free and open source software projects. The ASF developed its own license known as the Apache license, under which its software is released. Amongst the most widely used Apache applications are the Apache HTTP Server, Maven, Ant, Hadoop, and a class library for Java known as Apache Commons. 23, 31, 33, 35, 52, 55

Assembly A very low-level computer language, where each instruction in the code corresponds to an instruction that is supported by some hardware architecture. Assembly is therefore the native machine code for a particular architecture, but expressed in a human-readable format. Assembly is very rarely written or modified by humans nowadays. 20

Bytecode Java code is compiled to bytecode by the Java compiler. There are also bytecode compilers available for other languages including Python, and Ruby. Bytecode makes use of the instruction set of a Java Virtual Machine (JVM) and the same bytecode can be executed on any JVM, regardless of the hardware environment, this provides Java with its portability — one of its key strengths. 5, 6, 20, 25, 27–29, 33–35, 51, 142

C An extremely widely used computer programming language. C was developed in the early 1970s at Bell Laboratories. It is a *general purpose* language and is used for developing a wide range of applications, including scientific applications. It allows programmers access to memory at a relatively low-level, and its instructions were designed so that they could easily map to machine-level instructions, thus offering high performance. 1, 14, 17, 20, 47, 50, 51, 56, 66, 76, 90

C++ A programming language based on the C language, offering an object-orientated programming model, but also supporting the procedural and functional models. The syntax of Java, and later, the C# language, were heavily influenced by the syntax of C++. There are many important differences between C++ and Java, and in many cases Java simplifies, and reduces the possibility of bugs. For example, C++ supports class multiple inheritance, as well as operator overloading, neither of which are currently available in Java. 1, 14, 20, 21, 24, 47, 51, 76

Chapel A computer programming language developed by the Cray company, intended for the development of applications for execution in HPC environments. Offering a global view of data, it includes many features designed to aid the development of parallel applications, and to increase developer productivity. 22

Collective primitive Within the context of MPI, collective primitives are those functions that involve all of the processes within the scope of an MPI communicator. These functions may be for communication, synchronisation or computation. An example of a commonly-used collective primitive is *broadcast*, which allows one processes to send the same message to all other processes. 4, 76, 82, 84, 90

Compliant In the context of Java technology (and technology in general) an implementation may be called *compliant* if it adheres to some specification. Normally, an implementation must pass some tests in order to demonstrate its compliance. For example, a Java Virtual Machine (JVM) must pass the Java Compatibility Kit (JCK) in order to be called a *compliant JVM*. 28, 33, 35–37

Computer cluster In the context of High Performance Computing (HPC), a computer cluster (or simply *a cluster*) refers to a group of computers (or computing nodes) which are connected together using some network technology. A computer cluster can be viewed as a single machine, and such systems are often used to execute large parallel applications. 3, 4, 19, 73, 76, 79, 151, 157

Computing node A hardware unit which normally contains the same computing components as a normal desktop computer, such as a processor, memory, and possibly a local hard disk. The term *computing node* is usually used to refer to those computing units which are grouped together with other computing nodes to form a computer cluster. Often, as is the case in this thesis, the term *computing node* is shortened to *node*. One of the leading trends in HPC in recent years has been the move towards computer clusters with an ever increasing number of computing nodes, and an increase in the number of processors and cores per node. 3, 4, 18, 47, 67, 68, 73, 74, 91, 152

Core A hardware processing unit. Sometimes cores are referred to as processors, however this can lead to some confusion, a processor can contain several cores (See processor). A major tend within HPC, and general computing, is the move towards an ever increasing number of cores per processor. 3, 4, 15, 16, 18, 23, 41, 47, 74, 86, 161

Data Analysis and Processing Consortium (DPAC) The organisation which has responsibility to design, implement and execute the Gaia data reduction pipeline. DPAC is a multidisciplinary organisation, it is mainly composed of academics, but also with significant contributions from the European Space Agency, and the aerospace industry. 39

Data Processing Centre Barcelona (DPCB) One of the Data Processing Centres (DPCs) within the Data Analysis and Processing Consortium (DPAC). It includes computing facilities at *Centre de Serveis Científics i Acadèmics de Catalunya* (CESCA) and the Barcelona Supercomputing Centre (BSC). Members of the DPCB have been heavily involved in the preparation of DPAC systems, and the DPCB will play an important role during the execution of the Gaia data reduction pipeline. 40, 62, 166, 175

Distributed-memory A multiprocessor system in which the total memory is distributed amongst the processors, therefore, each of these sections of memory is local to a particular processor or group of processors. An example would be a typical computer cluster,

in which each computing node has its own memory which may be directly accessed only by the processors in that node. 18, 19, 73, 74, 76, 79

DPAC systems The large group of software applications developed by the Data Analysis and Processing Consortium (see Data Analysis and Processing Consortium). These applications form the Gaia data reduction pipeline. DPAC systems run at a number of distributed data processing centres around Europe. It is a policy within DPAC that, where possible, software should be written in Java. 39–42, 85, 110

DpcbTools One of the DPAC systems developed to assist the execution of other DPAC systems at the Data Processing Centre Barcelona (see Data Processing Centre Barcelona). DpcbTools contains utilities for performing I/O; data manipulation; communication; task creation, scheduling and launching; data visualisation; and monitoring operations. 43, 157

Dynamic compilation The compilation, and possibly recompilation, of code within a managed runtime environment. This term is often used to encompass the (re)compilation, as well as the optimisations that may be performed at JIT compilers. For some applications, more optimal code can be generated if compilation and optimisation is delayed, in order to benefit from more runtime feedback. 6, 142

Ergonomics The ability of JVMs to make intelligent decisions regarding their own configuration, in order to deliver high-performance without the need of tuning by the user. JVMs make these decisions based on the environment (such as the number of available cores) and the characteristics of the application. For example, default heap size, Garbage Collection and JIT compiler settings are selected based on the environment in which the JVM is running. 31, 112, 119

FastMPJ A high-performance, low-latency implementation of MPJ (see Message Passing in Java). Formerly known as F-MPJ, it supports a number of networks, including Infiniband, Ethernet and Myrinet. Its design follows a pluggable architecture, composed of components known as devices. Support for new networks can be added at a certain level without affecting devices at a higher level. 4, 11, 82–84, 90, 100

Floating-point The most widely used mechanism for representing real numbers in computer systems. The standard floating-point specification is defined in the IEEE 754 Standard. The term *floating-point number* is often used to refer to any number represented by floating-point representation, and the term *floating point operation* is used to refer to any operations involving *floating point* numbers. 8, 16, 45, 53, 161

FLoating-point OPerations per Second (FLOPS) A measure of the number of floating-point (see floating-point) operations a given machine (or processor or core) can perform per second. Computer systems are often ranked according to their FLOPS. For example, the top 500 supercomputers list is ordered based on FLOPS, using the commonly used LINPACK benchmark. 8, 53, 162

Fortran A computer programming language widely used in scientific and HPC computing, due to it numerical processing capabilities and the high performance that it delivers. First developed at IBM in the 1950s, it has evolved over successive versions of the language, and continues to be very widely used today. 1, 14, 20, 22, 76

Fortress A computer programming language, initially developed by Sun Microsystems, designed for the development of applications that will be executed in HPC environments. Its

syntax closely resembles mathematical notation, therefore allowing mathematical equations to be expressed naturally. 22

Free software Software covered by a license that gives users the freedom to run, copy, distribute, study, change and improve the software. It is important to realise that, in this context, the term *free* does not mean free in monetary terms (although it almost always is). Instead, the term *free* refers to the freedom that the license grants users. Nearly all *open source* software is *free software*, and vice versa, however these terms do imply different concepts. *Open source* may simply refer to a development methodology, while *free software* refers to a social movement. 33–35, 52, 83

Gaia A space astrometry mission from the European Space Agency (ESA). The Gaia mission has a number of ambitious goals, amongst which is to measure the positions and movements of the one billion brightest stars in the sky. The Gaia satellite will operate for 5 years, scanning the sky with extremely precise and sophisticated instruments. All the data returned to Earth by Gaia will be processed by a complex series of applications, collectively known as the Gaia data reduction pipeline. 2, 9, 38–40, 42

Gbin A file format used within DPAC (see Data Analysis and Processing Consortium) for the storage and transfer of Java objects. It includes a file header section, followed by 1 or more data sections. Java objects are stored in data sections as serialized objects, compressed using the DEFLATE algorithm. 85

General Parallel File System (GPFS) A high-performance shared filesystem, developed by IBM, for use in computer clusters. It is designed to allow many concurrent accesses, while maintaining high performance. It also provides fault tolerance, and optimal use of the storages devices. The technology can involve splitting files into small blocks, and then storing these blocks across several disks, thus obtaining the total combined bandwidth of all the disks. 40, 62

GNU GPL Often referred to as simply *GPL*, is a license agreement under which free and open source software may be made available. This license was created by the GNU project. The GPL license agreement is known as a copyleft agreement, the terms of such a license state that software under the license (including modified versions of the software) may only be distributed under the same license terms. 34, 35

GRID Superscalar (GRIDSs) GRID Superscalar, as well as the more recent COMP Superscalar (COMPSs), are execution frameworks, developed at the Barcelona Supercomputing Centre (BSC), that allow for the execution of applications in a grid or cluster environment. It permits any parallelism within an application to be exploited, through the execution of parts of the application, known as *tasks*, in parallel, using the available computing resources. 66

Heap A section of memory used for dynamic memory allocation. In the case of the JVM, objects that are created during the execution of an application are stored on the heap. Typically, JVMs allow users to configure the initial and maximum sizes of the heap at runtime, and space on the heap is managed by the garbage collector. 50, 83, 97, 113, 118, 124

Hotspot A Java Virtual Machine (JVM) produced by Oracle (formerly by Sun Microsystems). Its name is based on the fact that, as it executes Java bytecode, it attempts to identify

the *hotspots* within the code, it then concentrates its optimisations efforts on these parts of the code. 6, 30, 32, 36, 106, 146, 175

Infiniband A high-performance communication network, commonly used in HPC environments, including in supercomputers. It offers scalable, high throughput and low latency communications. In recent years, Infiniband has come to dominate the HPC network market. 23, 75, 80, 84, 104

Intel The largest, and arguably the leading semiconductor designer and producer in the world. Intel has been at the forefront of several techniques that have sustained the continuing increases in processor performance. 15, 27, 75, 90

Inter-node In the context of communication, refers to communication between processing elements which are physically located in different computing nodes. This could, for example, be through the use of sockets, RMI or MPI. 4, 68, 90

Intra-node In the context of communication, refers to communication between processing elements within the same computing node. This could be through the use of shared memory programming, threads, or through the use of message-passing within a single node. 4, 90

J9 A high-performance Java Virtual Machine (JVM) developed by IBM, and widely used commercially. In this thesis, the terms *J9* and *IBM JVM* are used synonymously. 32, 33, 41, 106, 121, 175

Java Class Library (JCL) A large set of libraries, covering a range of uses, which Java applications may call at runtime. A Java Virtual Machine, together with the Java Class Library, form a Java Runtime Environment (JRE). 28, 29, 33–36, 47, 51, 53, 54

Java Community Process (JCP) An organisation which controls changes to the specification of Java. Composed of representatives from a number of companies and other entities, which all have an interest in Java. 35

Java Concurrency Framework A high-level set of multithreading utilities available in the Java Standard Edition, which provide a set of *ready made* multithreading utilities, generally leading to increased developer productivity and less bug-prone code. 30, 47, 48, 83

Java Development Kit (JDK) A set of tools required to compile and execute Java applications. A JDK includes a Java Runtime Environment (JRE), which in turn includes a Java Virtual Machine (JVM). 27, 30, 108

Java Grande Forum (JGF) An initiative, largely composed of academics, which recognised the potential of Java as a language for the development of software for solving large or *grande* processing challenges, such as those in the fields of mathematics, astronomy and biology. It intended to act as an advisory body — researching and advising on changes and additions to the Java platform, such as the development of better numerical libraries for Java. It has largely been inactive for a number of years. 1, 53, 81, 82

Java platform The collection of applications which allow for the building, compiling, debugging, executing and monitoring of Java applications. The Java platform is independent of the underlying Operating System (OS) and hardware platform. 14, 24, 27–29, 31, 35, 36, 193

Java Virtual Machine (JVM) A system that can execute Java bytecode (see bytecode). Java, like several other languages, was designed to be executed in a managed runtime environment. Java code is normally compiled to bytecode, which is not dependent on any particular CPU architecture or instruction set, instead it is intended to be executed by a JVM. Any machine with a JVM installed can execute the same bytecode. It is up to the JVM to decide how the bytecode should be executed. JVMs gave the Java platform its portability. 28, 146

JavaBean Or simply a *Bean*, is a reusable software component. JavaBeans are Java classes which conform to a set of known conventions, which allow for their inspection and use in a standardised manner. 30, 153

Job In the context of High Performance Computing (HPC), the term *job* is used to refer to a single unit of work that should be executed by the system. Requests from users result in the creation of new jobs. Normally, a job scheduler is in charge of deciding if, and when, jobs are executed, based on a number of factors, including the privileges of the user making the request, and the availability of resources. 41, 59, 67

JRockit A JVM known for its high performance and scalability. It is distributed with a powerful set of analysis, profiling and debugging tools called JRockit Mission Control. At the time of writing, many features from JRockit are currently being incorporated into the HotSpot JVM, while many of its tools are being ported to the OracleJDK, and are expected to be present in JDK 8. 32, 33, 108, 146

MareNostrum A supercomputer at the Barcelona Supercomputer Center (BSC). MareNostrum II was decommissioned in October 2012. At the time of writing (November 2012) a *new* MareNostrum is currently being built (referred to as MareNostrum III in this thesis). Like its predecessor, the new MareNostrum is a distributed-memory machine, with roughly 3000 computing nodes, 16 cores per computing nodes, and an Infiniband network. 40, 41, 62, 85, 90, 100, 121, 166

MBean Or *managed bean*, is a software component used to represent some resource inside a Java Virtual Machine (JVM) such as an application, or a system property, or a service. MBeans are used by Java Management Extensions (JMX) to allow for the interaction with managed resources. 7, 31, 49, 153

MBeanServer A server process that is executed as part of Java Management Extensions (JMX) technology. MBeans which are to be made accessible through JMX are registered with an MBeanServer. Remote applications may then use the MBeanServer to communicate with MBeans. 7, 153

Message Passing in Java (MPJ) Any implementation of a message-passing system using Java. The standard Java libraries do not provide support for message-passing, although there have been a number of implementations of MPJ. These implementations followed various approaches, some using JNI to make use of MPI libraries, while others have followed a totally Java-based approach, interacting directly with native high-performance network libraries. 4, 74, 81–84, 90

Message Passing Interface (MPI) A message passing system widely used in computing for enabling communication between within parallel software. It is particularly used in distributed-memory systems. MPI bindings are available for a number of languages including Fortran, C, C++ and Java. 4, 76

Moab A job scheduling system (also known as a workload management system) produced by Adaptive Computing. Moab is widely used in HPC, including by many supercomputers in the TOP500 list. It offers a range of sophisticated job scheduling features including *backfill*, and advanced reservation, as well as providing statistics and diagnostic features. 59, 62

MPJ Express One of the most widely used implementations of Message-Passing in Java (MPJ). Developed in Java, its design follows a pluggable architecture of devices at different levels within the application. It provides support for several HPC network interconnects, and supports inter-node as well as intra-node communication. 11, 79, 82–84, 90, 100

MPJ-Cache Our data distribution middleware, offering a high-level API of data transfer methods to Java applications in HPC environments. It includes features to minimise accessing of shared storage devices, including the prefetching and caching of data. It makes use of an implementation of MPJ to provide efficient data communication over high-performance networks. 10, 11, 44, 73, 74, 90, 91, 97, 99, 100, 158

Mutator In the context of Garbage Collection (GC), the term *mutator* is sometimes used to refer to the user application, because it mutates the state of the system. The term *collector* is sometimes used to refer to the GC process. 111

Myrinet A high-performance low-latency network technology used in HPC. The Myrinet network protocol involves less overhead than Ethernet, for example, and provides lower latencies, and higher throughput. 5, 41, 62, 64, 68, 75, 80, 83, 84, 88

Native machine code Code which can be directly executed by a particular hardware architecture is said to be *native machine code* for that architecture (for that processor). Computer software is generally written in a relatively high-level language, such as C or Java. However, in order to execute software, it must be compiled to native machine code at some point. 6, 8, 25, 28, 29, 31, 34, 141, 142

New Input/Output (NIO) A Java I/O library available in the Java Class Library since J2SE 1.4. This library added to the functionality already available in the Java IO library, including better buffer management, the channels abstraction and non-blocking I/O functionality. 50, 83

Node Group (NG) In the context of DpcbTools (see DpcbTools), a NG is a group of computing nodes which are said to form a group of worker nodes, and containing a single node which is designated as the Node Group Manager (see Node Group Manager). 10, 67, 88

Node Group Manager (NGM) A computing node within a Node Group (NG), that is in charge of coordinating the activity of the other nodes in that NG, including the scheduling of tasks, and the management of data. 10, 67, 88

Object-Oriented (OO) A programming paradigm whereby entities are represented by objects, and the processes related to those objects are represented by functions (or *methods*). It dates back to the 1960's, and is supported by a large number of languages including Smalltalk, C++ and Java. It is generally accepted that the OO design offers many benefits, amongst which are high developer productivity, and the development of modular and extendible code. These are especially important for large software projects. 24, 56

Open core model A product development strategy that involves an application being made available as open source, and also made commercially available as a closed implementation,

but with extra features included which are not open source. In this way, the application owner gains the benefit of collaborating with the open source community, while still offering the application as a commercial product. 36

Open source The term open source is used to refer to software whose source code is freely available, and may be inspected, modified, and redistributed. Open source software is normally distributed with an open source license that grants these rights. The GPL is a widely-used open source license. There exists a huge community of open source developers, many of whom devote their spare time to open source software development as they consider it a fair and just method for granting access to software. A software product which is not open source is sometimes known as a *closed implementation*. 22–24, 31, 33–36, 52, 83

OpenJDK OpenJDK (Open Java Development Kit) is a free and open source implementation of the Java programming language. It is the result of an effort started by Sun Mircosystems to make Java an open source project. The OpenJDK is now the reference implementation of Java, and the project is being contributed to, by Oracle, IBM, as well as many other companies, groups and individuals. 24, 31, 33, 35, 36, 175

OpenMP An Application Programming Interface (API) that supports the development of parallel software for execution on shared-memory environments. Using OpenMP involves the insertion of directives into the code, which specify which sections of code should be executed in parallel. One advantage of OpenMP is that existing code can be parallelised in a progressive manner, with relative ease. 19, 38, 74, 76, 79, 83

Oracle One of the largest technology companies in the world, involved in the development of a range of computer hardware and software systems. In January 2010, Oracle purchased Sun Microsystems, thereby becoming the *owner* of Java technology. 22, 24, 27, 30–33, 35–37, 153

Oracle JDK A closed implementation of the Java platform made available by Oracle. Largely based on the OpenJDK codebase, but with extra features included. 36

Point-to-point primitive Point-to-point primitives are a set of functions within the MPI specification for the communication of data between two endpoints. Typically, these functions involve one of endpoints sending information, and the other endpoint receiving information. There are many variations on this basic *send—receive* model, for example, functions may be blocking or non-blocking, and buffering may, or may not, be used. 4, 76, 84, 90

Process Strictly speaking, a process is an instance of an executing program. Each executing process is allocated a separate address spaces. A process may comprise multiple threads. Occasionally, the terms process and thread are used interchangeably, however this should be avoided as it can lead to confusion. 3, 5, 19, 68, 73, 74, 76, 77, 85, 90

Processor A hardware processing unit. The exact meaning of the term *processor* depends on the context, and it has changed over the years. Originally, processors were considered a single processing unit, however, it is almost always the case nowadays that processors contain several individual processing units, referred to as *cores* (see core). Intel defines a processor as a unit that plugs into a single socket on the motherboard, regardless of whether or not the processor has multiple cores. The cores within a processor may be tightly or loosely coupled, depending on the resources which they share, such as memory

caches. Sometimes the terms *processor* and *core* are used interchangeable. 3, 4, 15–17, 27, 32, 33, 41, 47, 73, 74, 161

Python A general purpose programming language, supporting a number of paradigms, including Object-Oriented (OO) as well as functional. In recent years, Python has been gaining acceptance by the scientific community as a valuable development platform. Python may be compiled to bytecode and executed in a JVM. 29, 31, 43, 71, 157

Remote Method Invocation (RMI) A mechanism built into Java technology that allows for the invocation of methods in objects existing in remote Java Virtual Machines (JVMs). Additionally, objects may be passed between JVMs, thus allowing RMI to be used as a means for implementing communication in a distributed system. 30, 80

Scalability A measure of how well a particular approach, or system, maintains the same level of performance, as the *size* of executions are increased. The size of an execution may be increased in a number of ways, including through the use of more processes. Scalability is an increasingly important consideration in HPC, as the number of available processors increases. In the case of computer clusters, the scalability of the inter-node communication system used, such as an MPI implementation, often acts as one of the main factors determining the scalability that can be achieved. 76, 85, 90

Shared-memory A system in which several processing units may access the same memory space. In such an environment, processes may use shared-memory as a means to communicate with each other. OpenMP is widely used in shared-memory environments for this purpose. Systems purely based on shared-memory generally do not scale as well as distributed-memory systems. Some applications use a combination of shared-memory and message-passing communication utilities in order to improve scalability. 18, 19, 74, 76, 79, 80, 83

Socket In the context of computer networks, the term *socket* is used to refer to an endpoint over which communication can be sent and received. Socket programming refers to the utilisation of sockets to allow processes to communicate with each other. For applications with complex communication patterns, socket programming normally requires a large programming effort in order to coordinate the communication. 50, 80

Source In astronomy, the term *source* or *point source* is used to refer to any object that appears as a single source of light. Typically, and especially in the context of the Gaia data reduction, sources are stars. 9, 38, 39, 42, 56

Steady-state An application is said to have reached a steady-state after it has already been executing for a period of time, and its performance is remaining relatively stable. In this context, performance is typically measured by throughput, or sometimes, by responsiveness. Java applications generally reach a steady-state once the JIT compiler has applied the majority of the optimisation that it will apply, and once its memory usage behaviour is remaining relatively constant. When profiling or benchmarking the performance of an application, it is considered good practice to allow the application to reach a steady state before measuring its performance, otherwise measurements taken will not reflect normal application behaviour. 134, 163

Sun Microsystems, Inc. A company that developed computer hardware, software and services. The Java programming language was developed, and later managed for a number of years by Sun. Sun was purchased by Oracle in 2010. 22, 25, 27, 30–33, 35, 36

Technology Compatibility Kit (TCK) A suite of tests designed to determine if an implementation correctly implements the specification of a Java Specification Request (JSR). 35

Thread A unit of processing. The implementation details of threads can differ from system to system, however, generally speaking, threads are contained within processes. A process may contain multiple threads, each thread typically shares the same resources, most importantly, multiple threads may share the same memory space. The fact that threads share resources means that synchronisation of threads is sometimes required. Threads are sometimes referred to as *light weight processes*. 3, 5, 19, 83, 84

X10 An extension of the Java language, developed by IBM, which adds a number of additional constructs, that are designed to assist the development of parallel software for HPC environments. These constructs include *places*, *regions* and *distributions*. 19, 37, 38

Acronyms

ACL Access Control List. 154

AGIS Astrometic Global Iterative Solution. 42, 56

AOS Adaptive Optimisation System. 20, 142, 144, 147

AOT Ahead-Of-Time. 142, 147

API Application Programming Interface. 11, 19, 28, 76, 80, 91

APU Accelerated Processing Unit. 17

ASCII American Standard Code for Information Interchange. 107, 108

BCI ByteCode Injection. 107

BCL Binary Code License. 35, 36

BLAS Basic Linear Algebra Subprograms. 54

bps bits per second. 75

BSC Barcelona Supercomputing Center. 40, 62, 166

BW Bandwidth. 101

CAF Co-array Fortran. 19, 20

CCD Charge-Coupled Device. 38

CDS Class Data Sharing. 147

CESCA *Centre de Serveis Científics i Acadèmics de Catalunya*. 40, 166

CHA Class Hierarchy Analysis. 145

CIL Common Intermediate Language. 21

CLR Common Language Runtime. 21, 110, 142, 148

CMP Chip Multiprocessor. 18

CMS Concurrent Mark-Sweep. 116

CNES Centre National d'Etudes Spatiales. 53

COMPSs COMP Superscalar. 66, 78, 147

CPU Central Processing Unit. 1, 16, 75, 105, 151, 157, 158

CU Coordination Unit. 39

DAL Data Access Layer. 64

DARPA Defense Advanced Research Projects Agency. 22

DCE Dead-Code Elimination. 137, 145, 163, 206

DER Directly Executable Representation. 28, 141

DIR Directly Interpretable Representation. 28, 141

DMA Direct Memory Access. 83

DPAC Data Analysis and Processing Consortium. 39–41, 59, 62, 161, *Glossary:* Data Analysis and Processing Consortium

DPC Data Processing Centre. 39–41, 62

DPCB Data Processing Centre of Barcelona. 40, 42, 43, 62, 63, 85, 90, 166, *Glossary:* Data Processing Centre Barcelona

DPCE Data Processing Centre of ESAC. 40, 42, 63

DSM Distributed Shared Memory. 18, 78

EC Executive Committee. 35

ESA European Space Agency. 2, 9, 38, 39

FDO Feedback-Directed Optimisation. 141, 142, 144, 146

FIFO First In First Out. 71, 98

FLOP count FLoating-point OPerations count. 8, 162

FLOPS FLoating-point Operations Per Second. 8, 161, 162, 166, *Glossary:* floating-point operations per second

FOSS Free and Open Source Software. 34, 51, 52, 165

FOU Field Of Use. 36

FSF Free Software Foundation. 34

FTP File Transfer Protocol. 78

G1 Garbage First. 116

GASS GAia System Simulator. 41, 42, 59, 120, 148

GB Gigabyte. 39, 41, 85, 89

GbE Gigabit Ethernet. 41, 75, 84

Gbps Gigabits per Second. 75, 102

GC Garbage Collection. 6, 9, 21, 28–30, 33, 34, 47, 110, 111, 179

GCM Grid Component Model. 65, 66

GCMV Garbage Collection Memory Visualizer. 108, 133, 134, 140

GHz Gigahertz. 15, 176

GNU GNU's Not Unix (GNU is a recursive acronym). 34

GOG Gaia Object Generator. 41, 59

GPFS General Parallel File System. 41, 62, 63, 85, 88, 89, 100–102, 176, *Glossary:* General Parallel File System (GPFS)

GPGPU General-Purpose Graphics Processing Unit. 17

GPL General Public License. 34–36, *Glossary:* GNU GPL

GPU Graphics Processing Unit. 16–18, 23

GRIDSs GRID Superscalar. 66, 148, *Glossary:* GRID Superscalar

GUI Graphical User Interface. 108

HDFS Hadoop Distributed File System. 24

HJ Habanero-Java. 37

HPC High Performance Computing. 3, 14, 17, 19, 22, 45, 46, 62, 64, 75, 78, 151, 152

HPF High Performance Fortran. 20, 22

HSM Hierarchical Storage Management. 41

HTTP Hypertext Transfer Protocol. 78

I/O Input/Output. 2, 29, 50, 66, 73, 74, 77, 80, 88, 165, 168, 176

IBM International Business Machines. 20, 24, 27, 33, 35, 37, 41, 76

ICRS International Celestial Reference System. 42

ICT Information and Communication Technologies. 24, 56

IDE Integrated Development Environment. 37, 51

IDT Intermediate Data Treatment. 42

IDU Intermediate Data Updating. 41, 43, 59, 63, 85, 87–89, 162, 176, 193

IEEE Institute of Electrical and Electronics Engineers. 53

IMB Intel MPI Benchmark. 90, 204

IR Intermediate Representation. 144–146

J2EE Java 2 Enterprise Edition. 30

J2ME Java 2 Micro Edition. 30

J2SE Java 2 Standard Edition. 30, 53

JAR Java ARchive. 28, 51

JCK Java Compatibility Kit. 35, 36

JCP Java Community Process. 2, 24, 30, 31, 35, 36, *Glossary:* Java Community Process

JDI Java Debugger Interface. 49

JDK Java Development Kit. 5, 8, 27, 30, 31, 35, 36, 108, 153, *Glossary:* Java Development Kit

JFR JRockit Flight Recorder. 32

JFS Java Fast Sockets. 80, 81

JGF Java Grande Forum. 1, 38, 53, 54, 65, 81, 82, *Glossary:* Java Grande Forum

JIT Just-In-Time. 5–9, 11, 29, 30, 32, 33, 162

JMX Java Management Extensions. 5, 30, 44, 49, 71, 100, 109, 151, 153, 157

JNI Java Native Interface. 17, 51, 55, 80–84

JPDA Java Platform Debugger Architecture. 49, 107

JRE Java Runtime Environment. 8, 11, 27, 29, 32, 35

JRMC JRockit Mission Control. 32, 33, 108

JSPA Java Specification Participation Agreement. 35

JSR Java Specification Request. 35, 46

JVM Java Virtual Machine. 3, 25, 27–36, 50, 51, 78, 83, 97, 106, 110, 151, 153, *Glossary:* Java Virtual Machine

JVMTI Java Virtual Machine Tools Interface. 5, 30, 49, 107

JWS Java Web Start. 30

kB Kilobyte. 84

LAPACK Linear Algebra PACKage. 55

LGPL Lesser General Public License. 34, 65

LOA Large Object Area (Alternatively known as *Large Object Space (LOS)*. 114

LRU Least Recently Used. 71, 98

LSF Load Sharing Facility. 59, 167

MB Megabyte. 85, 89, 204

MDB Main DataBase. 40, 41

MIB Management Information Base. 154

MIMD Multiple Instruction Multiple Data. 18

MISD Multiple Instruction Single Data. 18

MMI Mixed Mode Interpreter. 6, 29, 142, 143, 146, 163

MMTk Memory Manager Toolkit. 34

Mops Million operations per second. 171

MPI Message Passing Interface. 4, 5, 11, 19, 41, 73, 75–77, 79, 81–83, 90, *Glossary:* Message Passing Interface

MPJ Message Passing in Java. 5, 11, 44, 70, 73, 74, 77, 81–83, 90, 100, 178, *Glossary:* Message Passing in Java

MPMD Multiple Program Multiple Data. 18

MTBF Mean Time Before Failure. 152

MUM Memory Usage Model. 3, 9, 11, 136, 158, 179

MX Myrinet Express. 75, 80, 82–84, 90, 201

NaN Not a Number. 53

NAS Numerical Aerodynamic Simulation. 164, 168

NG Node Group. 10, 67, 68, 88, 100, 158, *Glossary:* Node Group

NGM Node Group Manager. 10, 67, 68, 88, 158, *Glossary:* Node Group Manager

NIC Network Interface Card. 75

NIO New Input/Output. 50, 80, 83, *Glossary:* New Input/Output

NPB NAS Parallel Benchmark. 164

NUCC Non-Uniform Cluster Computing. 37

NUMA Non-Uniform Memory Access. 18, 37

OID Object IDentifier. 154

OO Object-Oriented. 21, 22, 24, 25, 38, 56, 79, *Glossary:* Object-Oriented

OOM Out Of Memory. 42, 121, 132, 208, 209

OOP Ordinary Object Pointer. 208

OS Operating System. 25, 27, 29, 51, 71, 75, 152

OSR On-Stack Replacement. 145, 206

PB Petabyte. 41, 62

PBS Portable Batch System. 59

PGAS Partitioned Global Address Space. 18–20

PMAT Pattern Modeling and Analysis Tool. 108

RDMA Remote Direct Memory Access. 75

RDMA Remote Memory Access. 77

RMI Remote Method Invocation see. 30, 74, 80, 81, 153

RVM Research Virtual Machine. 33, 146

SCI Scalable Coherent Interface. 80

SE Standard Edition. 30, 31, 46, 48, 145

SIMD Single Instruction Multiple Data. 18

SISD Single Instruction Single Data. 18

SLF4J Simple Logging Facade for Java. 137, 158

SLURM Simple Linux Utility for Resource Management. 59, 62, 86, 167

SMP Symmetric Multiprocessing. 18, 79

SNMP Simple Network Management Protocol. 154

SOA Small Object Area. 114

SPEC Standard Performance Evaluation Corporation. 164

SPMD Single Program Multiple Data. 18–20, 37, 76

SSE Streaming SIMD Extensions. 145

STW Stop-The-World. 112–115

TB Terabyte. 39, 42

TCK Technology Compatibility Kit. 35, *Glossary:* Technology Compatibility Kit

TCP/IP Transmission Control Protocol/Internet Protocol. 5, 75, 80, 84

TLAB Thread Local Allocation Buffer. 117

TLH Thread Local Heap. 117

TR TestaRossa — IBM JIT compiler. 33, 146, 147

UMA Uniform Memory Access. 18

UPC Unified Parallel C. 19, 20

VM Virtual Machine. 25, 33–35, 73, 141, 142

WOCA Write Once Compile Anywhere. 21

WORA Write Once Run Anywhere. 21, 25, 51

XML Extensible Markup Language. 68, 94, 99, 137

List of Figures

2.1 Simplified overview of Java's compilation and execution processes, compared with those of C. The most obvious difference is the simplification of the compilation process at compile-time for the Java platform, and its more sophisticated runtime environment. . . . 26
2.2 How the key components of the Java platform relate to each other. . . . 28
2.3 DPAC Coordination Units (CUs). . . . 40
2.4 The flow of data between DPCs. . . . 40

4.1 IDU Task flow — showing the constituent tasks which comprise IDU, the order in which these tasks must be executed and the data dependencies between these tasks. Figure taken from *IDU Software Design Description* document. . . . 61
4.2 Overview of how the nodes in a distributed-memory system, such as MareNostrum, can be grouped using our node management framework. . . . 69

5.1 Scaling the number of computing nodes and job files. . . . 87
5.2 Scaling the number of job files. . . . 88
5.3 Processing time of an IDU prototype over a fixed amount of data, when using 1 to 16 nodes, and accessing the GPFS disk directly. . . . 89
5.4 MPJ-Cache components and test setup. . . . 92
5.5 Client-server communication protocol following the basic *pull* protocol. . . . 93
5.6 Client-server communication protocol following the basic *push* protocol. . . . 94
5.7 The protocol used when operating in the *push* mode of operation and using FileGroups. . . . 95
5.8 FileGroups configuration file — configured to distribute three FileGroups, consisting of different numbers of files. . . . 96
5.9 Showing the status before any files have been distributed, the cache has been populated according to the order in which files will be distributed. . . . 98
5.10 During the distribution of files, the cache contains some files which have already been sent to clients, and also some files which are yet to be sent. . . . 99
5.11 Following directly after the previous figure, the server has sent another file to a client, but it has not yet removed any files from the cache or retrieved new files. 99
5.12 Following directly after the previous figure, the server has sent another file to a client, but it has still not removed any files from the cache or retrieved new files, as it has been busy all the time. . . . 99
5.13 The server has experienced a period when there were no idle clients, therefore, it had an opportunity to perform some maintenance on the cache. It removed some files that were already sent to clients, and it retrieved some new files. . . . 99
5.14 Comparison of GPFS against MPJ-Cache, when using 1 NG, small files (1MB) and frequent requests (1ms). . . . 101

5.15 Total aggregate data rate for GPFS and MPJ-Cache, using different Node Group configurations. 100MB files have been used in this case. 103

6.1 The *VisualGC* tab within VisualVM, showing the occupancy levels of each of the generational spaces within the Java heap, while a particular application is executing. 109

7.1 Main spaces in the generational heap layout of the HotSpot and J9 JVMs. . . . 115
7.2 Execution time of low density configuration of GASS, with the gencon policy, on the J9 JVM 1.6 and 1.7. 122
7.3 Execution time of high density configuration of GASS, with the gencon policy, on the J9 JVM 1.6 and 1.7. 124
7.4 GASS - Testset4 - change in execution times as the percentage of heap used for the young space is varied. 127
7.5 IDU-ipd - Testset4 - change in execution times as the percentage of the heap used for the young space is varied. 130
7.6 IDU-crb - Testset2 - change in execution times as the maximum heap size is increased, for two GC policies: *optthruput* and *gencon*. 132
7.7 Heap size and the number of used bytes after each collection, during the execution of GASS. 135
7.8 Heap size and the number of used bytes after each collection, during the execution of IDU-crb. 135
7.9 An extract from a log file generated by MUM. 138
7.10 Example of MUM characteristics file. 139

8.1 Main view with PerfAnal while profiling GASS. 150
8.2 Showing execution times of the *interval processing method* over 5 executions. . . 150

9.1 Information monitored, and the technology used, by our monitoring system. . . 154
9.2 JMX framework components. 154
9.3 Increase in execution time due to increases in frequency of JMX monitoring requests. 156
9.4 An extract from a log file generated JavaMon. 159
9.5 Monitoring the status MPJ-Cache attributes in real-time, using VisualVM. . . . 160
9.6 Viewing the operations exposed by MPJ-Cache to the JMX framework, which allow for the management of its internal status. Shown in VisualVM. 160

10.1 Results of SciMark 2.0 on several DPCB hosts. 169
10.2 Results of SPECjvm2008 on a MareNostrum II computing node. 170
10.3 Results of NAS on several DPCB hosts. 172
10.4 Results of LINPACK on several DPCB hosts. 172
10.5 Cadi computing node — performance of write. 174
10.6 Overall CPU benchmarking results on several DPCB hosts. 175
10.7 Optimal file size and record length for reading data on a MareNostrum II computing node. 177

Appendix A - Machine Specifications

MareNostrum II (decommissioned October 2012)

- 2560 JS21 blades (computing nodes)
- 2 Dual-core CPUs per node - IBM Power PC 970MP processors at 2.3 GHz
- Total cores: 10240
- Cache info: L1: 32kB Data cache + 64kB Instruction Cache, L2: 1 MB
- Memory per node: 8 GB
- Large central shared disk: General Parallel File System (GPFS)
- Local storage: 30GB in each node
- Networks: Myrinet and Gigabit Ethernet
- Java version:
  - `java version "1.6.0"`
  - `Java(TM) SE Runtime Environment (build pxp6460sr3-20081106_07(SR3))`
  - `IBM J9 VM (build 2.4, J2RE 1.6.0 IBM J9 2.4 Linux ppc64-64 jvmxp6460-20081105_25433 (JIT enabled, AOT enabled)`
  - `J9VM - 20081105_025433_BHdSMr`
  - `JIT - r9_20081031_1330`
  - `GC - 20081027_AB)`

Gaia Interface Server at the BSC - gaiaftp

- 2 Intel(R) Pentium(R) D CPU 3.20GHz
- Memory: 2 GB
- Local storage: 2 TB
- Java version:
  - `java version "1.6.0_0"`
  - `OpenJDK Runtime Environment (build 1.6.0_0-b11)`
  - `OpenJDK Client VM (build 1.6.0_0-b11, mixed mode)`

CESCA - Prades

All of the nodes within the Prades cluster contain Xeon CPUs, however there are three speeds of processors and three amounts of RAM, as shown below.

CESCA - prades1

- 8 Intel(R) Xeon(R) CPU E5420 @ 2.50GHz (quad core)
- Cache: 256 kB of L1 and 12 MB of L2
- Memory: 16 GB
- Storage: NFS + local disk
- Java version:
  - `java version "1.6.0_16"`
  - `Java(TM) SE Runtime Environment (build 1.6.0_16-b01)`
  - `Java HotSpot(TM) 64-Bit Server VM (build 14.2-b01, mixed mode)`

CESCA - prades2-prades29

- 8 Intel(R) Xeon(R) CPU E5472 @ 3.00GHz (quad core)
- Cache: 256 kB of L1 and 12 MB of L2
- Memory: 32 GB
- Storage: NFS + local disk
- Java version:
  - `java version "1.6.0"`
  - `OpenJDK Runtime Environment (build 1.6.0-b09)`
  - `OpenJDK 64-Bit Server VM (build 1.6.0-b09, mixed mode)`

CESCA - prades30-prades45

- 8 Intel(R) Xeon(R) CPU X5550 @ 2.67GHz (quad core)
- Cache: 256 kB of L1 and 12 MB of L2
- Memory: 48 GB
- Storage: NFS + local disk
- Java version:
  - `java version "1.6.0"`
  - `OpenJDK Runtime Environment (build 1.6.0-b09)`
  - `OpenJDK 64-Bit Server VM (build 1.6.0-b09, mixed mode)`

CESCA - Cadi

All of the nodes within the Cadi cluster contain the same CPU, and the same amount of RAM.

- 4 Dual Core AMD Opteron(tm) Processor (275 2210.198 MHz)
- Cache: 64 kB of L1 for data and 1 MB of L2, each core
- Memory: 16 GB
- Storage: NFS + local disk
- Java version:
  - `java version "1.6.0_21"`
  - `Java(TM) SE Runtime Environment (build 1.6.0_21-b06)`
  - `Java HotSpot(TM) 64-Bit Server VM (build 17.0-b16, mixed mode)`

CESCA - Pirineus

- SCI Altix UV1000
- 224 Xeon X7542 (2,66 GHz, 64KB, 256KB, 18MB) (14,3000 Gflop/s)
- Total cores: 1,344
- Memory: 6 TB shared memory
- Java version:
  - `java version "1.7.0_06"`
  - `Java(TM) SE Runtime Environment (build 1.7.0_06-b24)`
  - `Java HotSpot(TM) 64-Bit Server VM (build 23.2-b09, mixed mode)`

Miranda (local workstation)

- Local workstation
- Intel(R) Core(TM)2 Duo CPU E6750 @ 2.66GHz (32-bit)
- Memory: 4 GB
- HotSpot Java version:
  - `java version "1.6.0_14"`
  - `Java(TM) SE Runtime Environment (build 1.6.0_14-b08)`
  - `Java HotSpot(TM) Server VM (build 14.0-b16, mixed mode)`
- J9 (IBM) Java version:
  - `java version "1.6.0"`
  - `Java(TM) SE Runtime Environment (build pxi3260sr4-20090219_01(SR4))`
  - `IBM J9 VM (build 2.4, J2RE 1.6.0 IBM J9 2.4 Linux x86-32 jvmxi3260-20090215_29883 (JIT enabled, AOT enabled)`

- J9VM - 20090215_029883_lHdSMr
- JIT - r9_20090213_2028
- GC - 20090213_AA)
- JCL - 20090218_01

Xps1 (laptop)

- Intel(R) Core(TM)2 Duo CPU P9600 @ 2.66GHz (32-bit)
- Memory: 2 GB
- Java version:
  - java version "1.6.0_22"
  - Java(TM) SE Runtime Environment (build 1.6.0_22-b04)
  - Java HotSpot(TM) Server VM (build 17.1-b03, mixed mode)

Appendix B - Communication Tests

Initial communication test results

Pingpong - MPJ Express

Pingpong - MPJ Express - inter-node communication

Configuration:

- MPJ Express version 0.36
- Number nodes: 2
- Number processes: 2

```
MPJ Express (0.36) is started in the cluster configuration
Starting process <0> on <s32c1b01>
Starting process <1> on <s32c1b02>
 PingPongBench with 2 process______________________________
 T(s)     T(us)  Bw(MB/s) NPROCS log2(NPs) size  log2(size)
0.000025   25       0.0396   2      1       1          0
0.000025   25       0.0791   2      1       2          1
0.000025   24       0.1613   2      1       4          2
0.000027   26       0.3009   2      1       8          3
0.000026   26       0.6101   2      1       16         4
0.000027   26       1.1984   2      1       32         5
0.000027   27       2.3547   2      1       64         6
0.000028   27       4.6282   2      1       128        7
0.000028   27       9.2564   2      1       256        8
0.000029   28      17.8957   2      1       512        9
0.000031   31      32.5376   2      1       1024      10
0.000033   33      61.5766   2      1       2048      11
0.000043   43      94.3949   2      1       4096      12
0.000065   64     126.3226   2      1       8192      13
0.000109   108    150.7006   2      1       16384     14
0.000246   245    133.4359   2      1       32768     15
0.000397   397    164.9927   2      1       65536     16
0.000705   705    185.8539   2      1       131072    17
0.001317   1316   199.1147   2      1       262144    18
0.002822   2821   185.7911   2      1       524288    19
0.005800   5800   180.7813   2      1       1048576   20
```

```
0.011593   11592  180.9002   2      1       2097152   21
0.023193   23193  180.8407   2      1       4194304   22
 PingPongBench ******* TEST COMPLETED *********
Stopping process <1> on <s32c1b02>
Stopping process <0> on <s32c1b01>
```

Pingpong - FastMPJ

Pingpong - FastMPJ - intra-node communication

Configuration:

- FastMPJ version v0.1.0
- Number nodes: 1
- Number processes: 2

```
FastMPJ v0.1.0 started with <omxdev> and <2> processes.
Starting process <0> on <s15c5b01>
Starting process <1> on <s15c5b01>
 PingPongBench with 2 process______________________________
 T(s)   T(us)   Bw(MB/s) NPROCS log2(NPs)  size  log2(size)
0.000010  9     0.1049      2      1          1      0
0.000010  9     0.2021      2      1          2      1
0.000010  9     0.4043      2      1          4      2
0.000010  9     0.8085      2      1          8      3
0.000010  9     1.6171      2      1         16      4
0.000010  9     3.2342      2      1         32      5
0.000010  9     6.4683      2      1         64      6
0.000010  9     12.9366     2      1         28      7
0.000010  9     25.8733     2      1        256      8
0.000004  3     134.2177    2      1        512      9
0.000003  3     306.7834    2      1       1024      10
0.000005  4     429.4967    2      1       2048      11
0.000013  13    306.7834    2      1       4096      12
0.000015  14    554.1893    2      1       8192      13
0.000020  19    838.0424    2      1      16384      14
0.000027  27    1205.6049   2      1      32768      15
0.000038  37    1739.7336   2      1      65536      16
0.000062  61    2114.4454   2      1     131072      17
0.000124  123   2114.4454   2      1     262144      18
0.000484  484   1082.1965   2      1     524288      19
0.001104  1104  949.4919    2      1    1048576      20
0.002141  2141  979.3023    2      1    2097152      21
0.004355  4354  963.1110    2      1    4194304      22
 PingPongBench ******* TEST COMPLETED *********
```

Pingpong - FastMPJ - inter-node communication

Configuration:

- MPJ Express version v0.1.0
- Number nodes: 2
- Number processes: 2

```
FastMPJ v0.1.0 started with <omxdev> and <2> processes.
Starting process <0> on <s15c5b01>
Starting process <1> on <s15c5b03>
 PingPongBench with 2 process_______________________________
 T(s)   T(us)   Bw(MB/s) NPROCS log2(NPs)  size  log2(size)
0.000012  12     0.0807     2      1         1      0
0.000012  11     0.1678     2      1         2      1
0.000012  12     0.3226     2      1         4      2
0.000013  12     0.6214     2      1         8      3
0.000012  12     1.3031     2      1        16      4
0.000012  12     2.5811     2      1        32      5
0.000013  12     4.9710     2      1        64      6
0.000018  17     7.2550     2      1       128      7
0.000018  17     14.5100    2      1       256      8
0.000019  18     27.5318    2      1       512      9
0.000021  20     48.8064    2      1      1024      10
0.000024  24     84.2150    2      1      2048      11
0.000031  31     130.1505   2      1      4096      12
0.000054  53     152.0342   2      1      8192      13
0.000093  93     175.3048   2      1     16384      14
0.000161  160    203.9154   2      1     32768      15
0.000353  352    185.7283   2      1     65536      16
0.000617  616    212.5893   2      1    131072      17
0.001151  1150   227.8308   2      1    262144      18
0.002210  2210   237.2193   2      1    524288      19
0.004327  4327   242.3166   2      1   1048576      20
0.008569  8568   244.7438   2      1   2097152      21
0.017039  17038  246.1616   2      1   4194304      22
 PingPongBench ******* TEST COMPLETED *********
```

Pingpong - MX utilities

These tests were executed using utilities which form part of the MX installation. These were executed by starting two terminals, either on the same machine, or on different machines, depending on the test configuration (intra-node or inter node).

Pingpong - MX Utilities - intra-node communication

Configuration:

- MX utilities

- Number nodes: 1
- Number processes: 2

Two terminals were started on the same node (login4), and the following commands were executed:

```
login4:> <PATH_TO_MX>/bin/tests/mx_pingpong -e 2
```

```
login4:> <PATH_TO_MX>/bin/tests/mx_pingpong -d login4:0 -M 2 -E 10000000 -r 2
```

```
Starting pingpong send to host login4:0
Running 1000 iterations.
   Length   Latency(us)    Bandwidth(MB/s)
        0       1.600          0.000
        1       1.081          0.925
        2       1.097          1.824
        4       1.093          3.661
        8       1.076          7.435
       16       1.124         14.235
       32       1.135         28.206
       64       1.137         56.288
      128       1.246        102.688
      256       1.499        170.781
      512       2.357        217.271
     1024       2.446        418.557
     2048       3.339        613.357
     4096       5.112        801.330
     8192       8.696        942.042
    16384      15.983       1025.057
    32768      29.038       1128.433
    65536      52.968       1237.287
   131072     102.186       1282.674
   262144     209.685       1250.183
   524288     465.964       1125.167
  1048576    1069.626        980.320
  2097152    2165.739        968.331
  4194304    4461.526        940.105
  8388608    8879.108        944.758
```

Pingpong - MX Utilities - inter-node communication

Two terminals were started on two different nodes (login4 and login5), and the following commands were executed:

```
login4:> <PATH_TO_MX>/bin/tests/mx_pingpong -e 2
```

```
login5:> <PATH_TO_MX>/bin/tests/mx_pingpong -d login4:0 -M 2 -E 10000000 -r 2
```

Configuration:

- MX utilities
- Number nodes: 2

- Number processes: 2

```
Starting pingpong send to host login4:0
Running 1000 iterations.
   Length   Latency(us)    Bandwidth(MB/s)
        0         3.901          0.000
        1         3.864          0.259
        2         3.918          0.510
        4         3.862          1.036
        8         3.904          2.049
       16         3.913          4.088
       32         3.938          8.125
       64         5.453         11.737
      128         5.449         23.491
      256         8.024         31.902
      512         9.988         51.264
     1024        13.801         74.198
     2048        18.580        110.226
     4096        28.325        144.607
     8192        47.878        171.102
    16384        87.952        186.283
    32768       154.380        212.256
    65536       300.278        218.251
   131072       565.042        231.969
   262144      1094.259        239.563
   524288      2161.769        242.527
  1048576      4271.034        245.509
  2097152      8508.790        246.469
  4194304     16980.155        247.012
  8388608     33917.871        247.321
```

Pingpong - Intel MPI Benchmark (IMB)

Pingpong - Intel MPI Benchmark (IMB) - inter-node communication

Configuration:

- Intel MPI Benchmark 3.2
- Number nodes: 2
- Number processes: 2

```
   #bytes #repetitions      t[usec]   Mbytes/sec
        0         1000         3.78         0.00
        1         1000         3.88         0.25
        2         1000         3.89         0.49
        4         1000         3.88         0.98
        8         1000         3.85         1.98
       16         1000         3.95         3.86
       32         1000         3.95         7.72
```

64	1000	5.17	11.80
128	1000	6.18	19.74
256	1000	8.06	30.27
512	1000	10.11	48.32
1024	1000	13.82	70.65
2048	1000	18.59	105.09
4096	1000	28.38	137.65
8192	1000	47.91	163.06
16384	1000	87.11	179.38
32768	1000	153.74	203.27
65536	640	297.84	209.85
131072	320	563.51	221.82
262144	160	1094.64	228.38
524288	80	2151.72	232.37
1048576	40	4269.73	234.21
2097152	20	8507.97	235.07
4194304	10	16975.14	235.64

Note: there is an inconsistency between how IMB calculates the bandwidth and how the other tested applications calculate bandwidth. The issue is that IMB benchmark considers that 1 MB/sec is 1048576 bytes per second, but in transmission units, 1 MB/sec is generally consider to mean 1000000 bytes/sec. Therefore, in order to compare IMB with the other applications, it must be adjusted accordingly. For example, the final value in the the table above, i.e. the bandwidth for the transfer of 4194304 bytes of data, should not be 235.64 MB/sec, but in fact it should be 247 MB/sec.

Pingpong - Intel MPI Benchmark (IMB) - intra-node communication

Configuration:

- Intel MPI Benchmark 3.2
- Number nodes: 1
- Number processes: 2

#bytes	#repetitions	t[usec]	Mbytes/sec
0	1000	1.07	0.00
1	1000	1.25	0.76
2	1000	1.24	1.53
4	1000	1.26	3.03
8	1000	1.23	6.18
16	1000	1.28	11.91
32	1000	1.27	24.04
64	1000	1.27	48.16
128	1000	1.33	91.48
256	1000	1.58	154.76
512	1000	2.48	196.77
1024	1000	2.42	403.71
2048	1000	3.14	621.41
4096	1000	4.69	833.60

```
   8192         1000         7.91       988.12
  16384         1000        14.27      1095.22
  32768         1000        11.62      2690.26
  65536          640        18.82      3321.56
 131072          320        33.32      3751.11
 262144          160       132.19      1891.16
 524288           80       401.09      1246.59
1048576           40      1035.31       965.89
2097152           20      2050.22       975.50
4194304           10      4166.85       959.96
```

Pingpong - summary of results

Application	Latency	Bandwidth
MPJ Express	23.193 μms	180.841 (MB/s)
FastMPJ	17.038 μms	246.162 (MB/s)
MX utilities	16.980 μms	247.012 (MB/s)
IMB (MPICH-MX)	16.975 μms	247.00 (MB/s)

Table 11.1: Pingpong tests (point-to-point communications).

The results of FastMPJ, MX and IMB (which uses MPICH-MX) are almost identical in the PingPong test. MPJ Express however shows roughly 35% higher latency and 25% less bandwidth.

Collectives

Using the 3 applications: MPJ Express, FastMPJ and IMB, we executed a number of collective communication tests. The results of the *Broadcast* tests are given in Tables 11.2 and 11.3.

Data	MPJ Express		FastMPJ		IMB (MPICH-MX)	
	Latency	BW	Latency	BW	Latency	BW
2MB	43.6 μms	48.1 MB/s	25.8 μms	81.3 MB/s	21.9 μms	96.0 MB/s
4MB	87.5 μms	47.9 MB/s	52.4 μms	80.0 MB/s	42.7 μms	98.2 MB/s

Table 11.2: Broadcast test - 8 processes, 1 process per node (therefore 8 nodes in total).

Data	MPJ Express		FastMPJ		IMB (MPICH-MX)	
	Latency	BW	Latency	BW	Latency	BW
2MB	N/A	N/A	50.6 μms	41.5 MB/s	21.4 μms	98.1 MB/s
4MB	N/A	N/A	100 μms	41.9 MB/s	38.7 μms	108.3 MB/s

Table 11.3: Broadcast test - 128 processes, 4 process per node (therefore 32 nodes in total).

Appendix C - JVM Tuning Options

Selected JVM command-line options

Most JVMs offer a large number of command-line options. Many of the options provided by JVMs are unique to a particular JVM, however for the most fundamental options, such as setting the initial heap size, JVMs try to use the same options where possible. In the case of most JVMs, standard command-line options, which tend to be stable across many releases, are prefixed by "*-X*". Non-standard options, which may be newly added, or experimental, are generally prefixed by "*-XX*". In the HotSpot JVM, some options, which are referred to as diagnostic options, are only accessible if the option *-XX:+UnlockDiagnosticVMOptions* is first enabled, while some other options require *-XX:+UnlockExperimentalVMOptions*.

Profiling GC and JIT compiler activity

Selected profiling/logging options		
J9	HotSpot	Description
-verbose:gc	*-XX:+PrintGC* OR *-verboseGC*	Basic GC logging
-Xjit:verbose	*-XX:+UnlockDiagnosticVMOptions* *-XX:+LogCompilation*	Log information related to the JIT compiler activity

Table 11.4: Selected HotSpot and J9 profiling/logging options.

Additional HotSpot Profiling options

- Log each GC along with some details:
  -XX:+PrintGCTimeStamps -XX:+PrintGCDetails
  -XX:+PrintAdaptiveSizePolicy -Xloggc:<GC log filename>
- Log information related to why, and for how long, application threads were paused:
  -XX:+PrintGCApplicationStoppedTime -XX:+PrintGCApplicationConcurrentTime
  -XX:+PrintSafepointStatictics
- Print extensive information on JIT compiler activity. This is useful when investigating if certain optimisations such as OSR or DCE have been performed: *-XX:+PrintCompilation*

GC tuning options

GC policies			
HotSpot 1.6		J9 1.6	
Policy	Description	Policy	Description
-XX:+UseSerialGC	Serial GC	-Xgcpolicy:optthruput	Optimised throughput GC
-XX:+UseParallelGC	Parallel GC	-Xgcpolicy:gencon	Generational concurrent GC
-XX:+UseParallelOldGC	Parallel GC in both young and old area	-Xgcpolicy:optavgpause	Optimised avg. pause times GC
-XX:+UseConcMarkSweepGC	Concurrent mark-sweep	-Xgcpolicy:subpool	Subpool GC
HotSpot 1.7		J9 1.7	
-XX:+UseG1GC	Garbage First GC	-Xgcpolicy:balanced	Balanced GC
		-Xgcpolicy:metronome	Metronome GC

Table 11.5: HotSpot and J9 GC policies.

Selected tuning options		
J9	HotSpot	Description
-Xms	-Xms	Initial size of the heap
-Xmx	-Xmx	Maximum size of the heap
-Xmn	-Xmn	Initial and maximum size of the heap
-Xmns	-XX:NewSize	Initial new space size
-Xmnx	-XX:MaxNewSize	Maximum new space size
-Xmn	-Xmn	Initial and maximum new space size
-Xmos	-XX:NewRatio	Initial old space size
-Xmox	-XX:NewRatio	Maximum old space size
-Xmo	-	Initial and maximum old space size
-	-XX:PermSize	Initial size of permanent generation
-	-XX:MaxPermSize	Maximum size of permanent generation
-Xmaxf	-XX:MaxHeapFreeRatio	Free space % that triggers heap shrinking
-Xminf	-XX:MinHeapFreeRatio	Free space % that triggers heap expansion
-Xgcthreads	-XX:ParallelGCThreads	Number of GC threads (Parallel GC)

Table 11.6: Selected HotSpot and J9 GC tuning options.

Selected additional tuning options

HotSpot JVM

- Limit the total GC overhead (default value is 98%): *-XX:+UseGCOverheadLimit*

- Enable the most recently developed optimisations. The effects of all these optimisations may not be fully understood, therefore, enabling this option may introduce some stability issues: *-XX:+AggressiveOpts*
- 64-bit JVMs allow much larger heap sizes than are permitted in 32-bit JVMs, however an additional overhead is introduced. The pointers used to refer to Java objects, known as Ordinary Object Pointers (OOPs) must increase in size from 32 bits to 64 bits. This results in fewer OOPs being cached, which leads to more cache misses and ultimately a reduction in performance. This option enables the use of 32-bit OOPs on a 64-bit JVM. It improves performance by reducing cache misses: *-XX:+UseCompressedOops*
- Use incremental garbage collection. Provides relatively short GC pauses. This policy may be removed in future versions: *-Xincgc*
- Set the percentage of the total execution time that should be spent in the mutator, as opposed to performing GC: *-XX:GCTimeRatio=n*
- Set the maximum number of milliseconds of any application pause due to GC: *-XX:MaxGCPauseMillis=n*
- Set the ratio between survivor space sizes and the eden space size *-XX:SurvivorRatio=n*
- Set the target percentage of occupied survivor space after each GC: *-XX:TargetSurvivorRatio=n*

J9 JVM

- Set the minimum or maximum amount of bytes by which the heap should be expanded by at any one time: *-Xmine<size>*, *-Xmaxe<size>*
- Set the minimum or maximum percentage of the execution time that should be spent in GC. If the actual percentage of time drops below the minimum, then the JVM tries to shrink the heap. If the actual percentage of time rises above the maximum then the JVM tries to expand the heap. The default values are 5% and 13% respectively: *-Xmint<percentage>*, *-Xmaxt<percentage>*
- Disable the throwing of an OOM exception if excessive time is spent in GC. This option should be used with care as it can lead to very poorly performing executions (where a very high percentage of the execution time is being spent in GC) being allowed to continue: *-Xdisableexcessivegc*
- Set the number of JIT compiler threads: *-XcompilationThreads<number>*
- To compile a method, or group of methods, to a particular optimisation level:

  -Xjit:'<method name>(count=<count>, optlevel=<level>)'

  - *Method name* specifies a particular method, or a group or methods. The * character may be used as a wildcard character for the method name.
  - *count* specifies the number of times that a method should be executed before it is compiled.
  - *optlevel* specifies the level at which the method(s) should be optimised. The supported optimisation levels are: *noOpt*, *cold*, *warm*, *hot*, *veryHot* and *scorching*.

  Note: Instructing the JIT compiler to compile a particular method does not result in methods that are called from that method also being compiled.

Appendix D - Configuration files

MPJ-Cache Configuration

Item	Type	Description
cache name	String	Name of a cache which should be created
implClass	String	Name of the class that implements the cache
policy	String	Policy used for maintaining the cache
maxsize	Integer	Max. No. items permitted in the cache
maxNumRequestsToHandle	Integer	Max. No. requests that the server will handle (only used in the pull mode)
maxMemCacheByteSize	Integer	Max. total size of the memory cache
maxPercentHeapForCache	Integer	A limit on the total size of the memory cache
maxPercentFreeDiskForCache	Integer	Max. percentage of the heap to use for caching of data. This cannot be too high (for example 98%) otherwise not enough space will be left for allocating memory to newly created objects, and OOM errors may occur
maxDiskCacheByteSize	Integer	Max. total size of disk cache
pathOfFilesLocalDisk	String	Path of disk cache data
pathOfInputFilesRemoteDisk	String	Path of remote files
numServerThreads	String	No. server threads (not currently used)
maxSizeDataInBytes	Integer	Limit on the maximum single message size
modeOfOperation	Integer	The mode of operation that should be used: 0 = push mode, 1 = pull mode
useFileGroups	Boolean	Should FileGroups be used to send groups of files together
fileGroupsList	String	Full path of a file which contains the list of FileGroups
prefetchDataToMemCache	Boolean	Server should prefetch data to the memory cache
prefetchDataToDiskCache	Boolean	Server should prefetch data to the disk cache
deleteServedFilesFromMemCache	Boolean	Server should delete prefetched data in the memory cache
deleteServedFilesFromDiskCache	Boolean	Server should delete prefetched data in the disk cache

Table 11.7: Configuration parameters for MPJ-Cache.

Item	Type	Description
numFilesPerClient	Integer	Number of files that each client should request
numRepsPerClient	Integer	Number of times that each client should repeat the test
shouldDelayBetweenRequests	Boolean	Clients should perform a delay (or sleep) between each request. This delay is used to simulate the processing of data, that would occur with a real application
randomDelayBetweenRequests	Boolean	The delay between requests should be random
maxDelayBetweenRequests	Integer	If there is a random delay between requests, then it is calculated as a random number between 0 and maxDelayBetweenRequests, if there is a non-random delay, then the delay is exactly equal to this value, given in milliseconds
initialClientDelay	Integer	If set to 0, then there is no initial delay, if greater than 0, then there is an initial delay, which is calculated using the formula: delay = initialClientDelay * rank
communicationSystemToUse	Integer	If set to 0, retrieve data from a MPJ-Cache server, alternatively bypass the server and read data directly from disk
pathOfRemoteInputFiles	String	Path of files on the remote disk (only used if clients are to bypass the server, which can be configured using the communicationSystemToUse parameter)
methodForObtainingFileList	Integer	Method by which clients should obtain the list of available files: 0 = request list from the server using a message, 1 = from a file
clientShouldSelectRandomFiles	Boolean	Clients should randomly select files from the list of available files, the alternative is that they should select files based on the order of the files in the list of available files
pathOfFileWithListOfFiles	String	Path of a file containing a list of the available files. This is only used if clientMethodForObtainingFileList = 1
numThreadsToCreate	Integer	Number of client threads (not currently used)
writeRetrievedDataToFile	Boolean	Clients should write the data that they have retrieved to a file
modeOfOperation	Integer	Mode of operation to be used, currently supported modes, and their identifiers are: 0 = push mode, 1 = pull mode
useFileGroups	Boolean	FileGroups should be used
pathToStoreFiles	String	Path where the client application should store files
deleteStoredFiles	Boolean	Clients should delete the files that it has received before terminating

Table 11.8: Configuration parameters for MPJCacheTestClient application.

MUM characteristics file fields

Item	Type	Description
ratio_shortlived_small_sized	Integer	Ratio of short-lived, small-sized objects
ratio_shortlived_medium_sized	Integer	Ratio of short-lived, medium-sized objects
ratio_shortlived_large_sized	Integer	Ratio of short-lived, large-sized objects
ratio_mediumlived_small_sized	Integer	Ratio of medium-lived, large-sized objects
ratio_mediumlived_medium_sized	Integer	Ratio of medium-lived, medium-sized objects
ratio_mediumlived_large_sized	Integer	Ratio of long-lived, large-sized objects
ratio_longlived_small_sized	Integer	Ratio of long-lived, small-sized objects
ratio_longlived_medium_sized	Integer	Ratio of long-lived, medium-sized objects
ratio_longlived_large_sized	Integer	Ratio of long-lived, large-sized objects
size_small_obj	Integer	Size of small-sized objects (bytes)
size_medium_obj	Integer	Size of medium-sized objects (bytes)
size_large_obj	Integer	Size of large-sized objects (bytes)
lifetime_shortlived_obj	Integer	Lifetime (iterations) of short-lived objects
lifetime_mediumlived_obj	Integer	Lifetime (iterations) of medium-lived objects
lifetime_longlived_obj	Integer	Lifetime (iterations) of long-lived objects
sleep_after_shortlived_created	Integer	Sleep period after creating short-lived object
sleep_after_mediumlived_created	Integer	Sleep period after creating medium-lived object
sleep_after_longlived_created	Integer	Sleep period after creating long-lived object
warmup_its_static_proc	Integer	Number of warmup iterations of static processing
warmup_its_obj_proc	Integer	Number of warmup iterations of object processing
total_num_its	Integer	Number of iterations to perform
percent_its_static_proc	Integer	Percentage of iterations consumed by static processing
num_its_per_obj_proc	Integer	No. iterations of object processing
seed_value_obj_construction	Integer	Seed value for object creation
obj_string	String	String passed to object constructors
initial_sleep_s	Integer	Initial sleep before all processing

Table 11.9: Description of the fields in a MUM characteristics file.