ALGORITHMS ACCELERATION OF PATTERN-MATCHING IN MULTI-CORE ARCHITECTURES

David Ródenas Picó

Dipòsit Legal: T-1350-2011

UNIVERSITAT ROVIRA I VIRGILI
ALGORITHMS ACCELERATION OF PATTERN-MATCHING IN MULTI-CORE ARCHITECTURES
David Ródenas Picó
DL: T.1350-2011

David Ródenas Picó

ALGORITHMS ACCELERATION OF PATTERN-MATCHING IN MULTI-CORE ARCHITECTURES

DOCTORAL THESIS

directed by Dr. Francesc Serratosa Casanelles

Departament
d'Enginyeria Informàtica i Matemàtiques

UNIVERSITAT ROVIRA I VIRGILI

Tarragona
2011

Index

List Of Publications Of This Thesis

[1] D. Ródenas et al., "Optimizing NANOS OpenMP for the IBM Cyclops multithreaded architecture," *19TH IEEE International Parallel & Distributed Processing Symposium (IPDPS 2005)*, 2005.

[2] D. Rodenas, X. Martorell, J. Costa, T. Cortes, and J. Labarta, "Running BT Multi-Zone on non-shared memory machines with OpenMP SDSM instead of MPI," *Proceedings of the XVI Jornadas de Paralelismo*, Sep. 2005.

[3] D. Ródenas et al., "Exploiting multilevel parallelism using OpenMP on a massive multithreaded architecture," *Journal of Embedded Computing*, vol. 2, p. 141–155, Apr. 2006.

[4] P. Carpenter, D. Rodenas, X. Martorell, A. Ramirez, and E. Ayguade, "Code generation for streaming applications based on an abstract machine description," *Universitat Politécnica de Catalunya, UPC-DAC-RR-CAP-2007-3*, 2007.

[5] F. Cabarcas, A. Rico, D. Rodenas, X. Martorell, A. Ramirez, and E. Ayguade, "A module-based cell processor simulator," *3rd HiPEAC Advanced Computer Architecture and Compilation for Embedded Systems. ISBN: 978-90-382-1127-5*, 2007.

[6] A. Rico, F. Cabarcas, D. Rodenas, X. Martorell, A. Ramirez, and E. Ayguade, "Implementation and validation of a Cell simulator using UNISIM," *3rd HiPEAC Industrial Workshop, IBM Haifa, Israel*, 2007

[7] F. Cabarcas, A. Rico, D. Rodenas, X. Martorell, and A. Ramirez, "CellSim: A Validated Modular Heterogeneous Multiprocessor Simulator," *Proceedings of the XVII Jornadas de paralelismo*, Apr. 2007.

[8] P. Carpenter, D. Rodenas, X. Martorell, A. Ramirez, and E. Ayguadé, "A streaming machine description and programming model," in *Proceedings of the 7th international conference on Embedded computer systems: architectures, modeling, and simulation*, Berlin, Heidelberg, 2007, p. 107–116.

[9] P. Carpenter, D. Rodenas, A. Ramirez, X. Martorell, and E. Ayguade, "Code generation for streaming applications based on an abstract machine description." IST ACOTES Project Deliverable D2.2, May-2007.

[10] P. Carpenter, A. Ramirez, X. Martorell, D. Rodenas, and R. Ferrer, "Report on Streaming Programming Model and Abstract Streaming Machine Desription 1st version." IST ACOTES Project Deliverable D2.1, Sep-2007.

[11] F. Cabarcas, A. Rico, D. Rodenas, X. Martorell, A. Ramirez, and E. Ayguade, "CellSim: A Cell Processor Simulation Infrastructure," *4th HiPEAC Advanced Computer Architecture and Compilation for Embedded Systems. ISBN: 978-90-382-1288-3*, 2008.

[12] D. Rodenas, R. Ferrer, X. Martorell, and E. Ayguade, "ACOTES Stream Programming Model," *4th HiPEAC Advanced Computer Architecture and Compilation for Embedded Systems. ISBN: 978-90-382-1288-3*, pp. 19-22, Jul. 2008.

[13] P. Carpenter, A. Ramirez, X. Martorell, D. Rodenas, and R. Ferrer, "Report on Streaming Programming Model and Abstract Streaming Machine Desription Final version." IST ACOTES Project Deliverable D2.2, Sep-2008.

[14] D. Rodenas, F. Serratosa, and A. Solé-Ribalta, "Graph Matching on a Low-cost & Parallel Architecture," *Iberian Conference on Pattern Recognition and Image Analysis, IbPRIA 2011, LNCS 6669*, p. 508–515, 2011.

[15] D. Rodenas, F. Serratosa, and A. Solé-Ribalta, "Parallel Graduated Assignment Algorithm for Multiple Graph Matching based on a Common Labelling," *Graph based Representations, GbR2011, Münster, Germany, LNCS 6658*, pp. 164-174.

[16] D. Rodenas, F. Serratosa, and A. Solé-Ribalta, "Massive Parallel Graduated Assignment Graph Matching Experiences on Low Power Architectures," *Submited to IJPRAI*.

Glossary

annotation based programming model: programming model which extends an existing programming models with annotations, as directives or comments. Annotations improves the existing information and the capacity of taking advantage of underlying architecture without changing application structure significantly.

cache: small memory with a small latency that contains a partial copy of main memory data.

cluster: group of one or more processing units sharing the same memory on a distributed memory system.

core: part of a microprocessor which is composed by one or more execution threads, functional units and data cache.

distributed memory: system with multiple process units which each process unit has its own memory and restricted access to memory of other process units. It is required explicit communication between process units in order to receive or to send required information.

execution thread: set of ordered instructions with its own register file.

functional unit: process hardware which executions mathematical computations usually real computations.

hardware: physical part of a computer, normally part that electronically computes a function.

heterogeneous system: system with multiple process units which has more than one instruction set and characteristics.

homogeneous system: system with multiple process units which all its units share the same instruction set and characteristics.

many-core processor: a multi-core with many cores.

multi-core: micro-processor containing two or more cores.

multi-threading (processor): processor that contains multiple execution threads at any of its cores.

multiprocessor: system composed with multiple processors interconnected, and, if not is explicit, with shared memory and homogeneous.

process unit: core.

programming model: set of tools of compilation and programming.

register file: set of small and very fast memories directly accessible by assembler instructions used as temporal values, it includes the program counter which contains the address of next instruction.

shared memory: system which all its process units have access with no restrictions to all available memory, but small time penalizations.

software: data that encodes instructions to execute an algorithm.

system with lineal memory address: system which all its process units can access to memory of all other units without restrictions but paying a great time penalization. It is possible found specific instructions designed to send and receive information as memory distributed.

thread: execution thread.

Figure Index

Index Of Tables

Index Of Algorithms

UNIVERSITAT ROVIRA I VIRGILI
ALGORITHMS ACCELERATION OF PATTERN-MATCHING IN MULTI-CORE ARCHITECTURES
David Ródenas Picó
DL: T.1350-2011

Abstract

Pattern matching algorithms are a classification task of pattern recognition that attempts to assign each input value to one of a given set of classes. Some of these algorithms use graphs because they have more capacity to capture the knowledge of the model but their comparison or matching is also more computationally expensive. This restriction makes them computationally not suitable for real-time applications.

In the current market scenario desktop computer architectures have evolved towards supercomputing architectures. These architectures generally provide a vast computing capabilities, but they require parallelise existing applications in order to take advantage of existing hardware. Consequently, applications and algorithms must be modified and adapted in order to take advantage of all available resources.

Algorithms and programs must be redesigned to be able to work on parallel environments. In order to redesign algorithms, programmers and algorithm designers must be aware of architecture limitations and must know parallelism techniques. The parallelisation of any program is usually a very complex tasks, but in many cases these require an expert programmer who knows specialised techniques for parallelism. We focus on OpenMP. OpenMP is a programming model which allows to parallelise an application by adding just few directives or comments. This programming model is very easy to use, and almost a non-expert on parallelisation can use it in order to achieve a good parallelisation and resource usage. It was initially designed for supercomputers with shared-memory with multiple processors instead of multiple cores.

We group research on this thesis on two steps: 1) for developing tools to allow non-expert programmers to take advantage of parallel architectures and 2) for applying extracted knowledge and create a version of graph pattern-matching

algorithms able to run quasi real-time on current desktop computers, focusing on those having low power consumption.

We use supercomputers as a starting point. They have been computing parallel programs for many years and almost all their users are no computer scientists. We have selected OpenMP from all available tools because it is one of the tools with better usability. Supercomputers are not desktop computers. Supercomputer programs and benchmarks does not include our target algorithms.

Firsts steps of this thesis compare multi-core processors with supercomputers. General purpose multi-core, now available on almost any desktop computer, are very close to shared-memory multiprocessors supercomputers. We use OpenMP as a target programming model, and we use existing and well known parallel supercomputing benchmarks in order to validate our affirmation. We will explore some critical differences as cache behaviour and we will solve how to overcome them and have good results on multi-core. We also will focus on one kind of parallel applications which has multiple levels of parallelism, having an external level of parallelism working with coarse-grain parallelism. Firsts distributed-memory and heterogeneous multi-core processor have been introduced, and we try to validate OpenMP on these architectures. First we prove that OpenMP is able to have a good performance on distributed-memory architectures. In this case we found that it have a special good performance on programs with multiples levels of parallelism. We also realise that OpenMP has an important lack of expressiveness for distributed-memory and heterogeneity. We propose a programming model derived from OpenMP able to extract streaming parallelism from serial applications. This model introduces two clauses able to convert a serial program into a streaming program. Proposed directives are able to create a graph representation of a streaming program, nodes are executing kernels and edges communication nodes. In that point we realise that multi-core can become very complex, and we collaborate in the creation of a modular simulator of heterogeneous multi-core. We contribute with an abstraction which allows to connect each module in any configuration. We expect from this simulator to help to create architectures closer to programmer needs and programming models restrictions.

In the lasts steps of the thesis we use extracted knowledge to effectively create parallel versions of the pattern matching algorithms. We focus on graph matching algorithms and we implement them on desktop computers. Target computer used

are desktop computers, but we do not limit our implementations to the general-purpose CPU: our target processors are either main processor and graphic processor. Main processor is a shared-memory homogeneous multi-core, very close to our firsts steps, on the other hand, graphic processor unit is a massive parallel processor which has distributed-memory and heterogeneous environment. We adapt previous OpenMP like tools to these final architecture and we use them to parallelise serial algorithms. We also introduce two common techniques in parallel programming which allows to redesign existing algorithms but without changing algorithm results. We show a methodology which allows to apply previous techniques, it transforms program equations in order to obtain an optimal parallelisable performance. We also show how to take advantage of existing private memory inside graphic processors but without rewriting the application. We evaluate presented algorithms and transformations to show how they effectively use underlying existing resources on desktop computers.

Nowadays desktop computers are indeed desktop supercomputers, not only by its performance, but also because its complexity and programmability. We show how a programming model can help to create parallel applications and how this applications can take advantage of existing hardware. One thing we have for sure, serial programs will not use efficiently existing hardware on desktop computers. This thesis has the objective to help to overcome this limitation.

Chapter 1. Introduction

Classification is a task of pattern recognition that attempts to assign each input value to one of a given set of classes. Pattern recognition algorithms generally aim to provide a reasonable answer for all possible inputs and to do inexact matching of inputs. Pattern recognition is studied in many fields such as psychology, cognitive science, computer science and so on. Depending on the application, inputs of the pattern recognition model or objects to be classified are described by different representations. The most usual representation is a set of real values but other common ones are strings, trees or graphs. These structures have more capacity to capture the knowledge of the model but their comparison or matching is also more computationally expensive. The distance between a pair of strings or trees is computed in polynomial time; nevertheless, the computation of the distance between a pair of graphs is exponential respect the number of vertices. For this reason, some algorithms that compute the distance between graphs have been presented obtains a sub-optimal distance. Although these last algorithms have a polynomial computational cost, the real run time is not acceptable for some applications such as fingerprint classification, on-line face identification or robot navigation, between others.

Nowadays desktop computer architectures have evolved towards supercomputing architectures. These architectures generally provide multiple processors and complex memory hierarchy [17]. A simple desktop computer may contain tens of small processors called cores [18], some of them present at main processor [19], but most of them are present as auxiliary coprocessors like graphical processors [20]. Main processor is usually a general-purpose processor [21]: a processor that contains at least one core able to execute almost any algorithm with an

acceptable trade off in efficiency. As a counterpart, some specialized algorithms, like video decoding or 3D computations can not be executed on reasonable time and real-time is not possible on such processors. Specialized cores [22], like graphical processor cores, are provided in order to speed-up these algorithms. As the number of cores grows, the complexity of the memory hierarchy and its interconnection network also grows. Consequently, applications and algorithms must be modified and adapted in order to take advantage of all available resources.

Most graph-matching algorithms are designed to be executed on a single core and general-purpose processor [23-25]. Consequently, they are not designed to take advantage of all available resources on current desktop computers. On the other hand, as the number of cores on desktop computers grows, the execution speed of a generic core remains unchanged [26], that means that classical graph-matching algorithms have no faster execution on improved architectures. Graph-matching algorithms must be redesigned to take advantage of all present resources in order to achieve real time applications.

There are three major challenges on new paradigm programming: work distribution, data distribution and synchronization [27]. The implementation of classical algorithms assumes that there is only one processor, which means that all instructions are executed in sequential order. Adaptation of these algorithms to the parallel paradigm starts in the identification of independent algorithm steps, those that can be executed concurrently. Each step access to a certain group of data, data can be accessed locally or must be transferred from another location. All steps must process data to achieve the final algorithm result, which means that synchronization is required to ensure a correct computation. Algorithms parallelisation is a complex task that usually requires a deep knowledge of the underlying architecture and, for the same reason, their performance are limited to computers with analogous architectures.

Compilers do not transform or adapt algorithms automatically. Automatic parallelisation might be perfect to this task: they are able to transform automatically applications to their underlying architecture. Unfortunately there is not much practical application: a compiler does not know programmer intentions, compiler can not go beyond information presented at compile time [28-30]. Some recompile-just-in-time techniques are able to overcome this limitation by collecting statistical

data from current execution. This technique allow to modify code from the original program in order to obtain statistically re-engineered code. Therefore, once again, the compiler does not know the programmer's intentions, just the programmer's instructions. Because of that, the compiler is not able to know for sure which transformations are safe and coherent with ideas beyond the original code. There is only one solution: programming model must allow the addition of relevant information to the compiler. The compiler must use these provided information to optimize a program to the underlying architecture. Nevertheless, while defining this new programming model, it has to be considered that there are a large number of programmers that implement their algorithms in current languages, and so, any addition of information must respect existing programming models and take advantage of already written code.

The aim of this thesis is to present a new research on multi-core architectures applied to graph-matching algorithms in order to easy their adaptation. The starting point are supercomputers: they already have multiple processors and a large list of programming models and compilers. Supercomputer's programming models and compilers help non expert programmers to take advantage of supercomputers resources. The first steps of this thesis are to compare graph-match algorithms against typical supercomputer algorithms and check their similarities. The second step is to compare multi-processors against multi-core processors. They can be quite similar, but memory hierarchy and cache behaviours can affect algorithms performance. As third step we will compare and adapt shared memory multi-processor programming models to distributed memory. Our aim is to increase distributed-memory usability. The fourth step is to develop a model to help to understand current architectures and close incoming architectures. The fifth step is to compare data acquisition from graph-matching algorithms to streaming applications. The sixth step is to adapt existing multi-processor programming models to streaming applications. The final step is to validate created programming model by presenting a version of the graph-matching algorithms which takes advantage of present resources on a desktop computers with a reasonable effort to a programmer or algorithm designer.

Section 1.1. Current Desktop Processors

Since the first desktop computers to first years of XXI century all market processor has followed the same tendency: each new generation has improved significantly the execution time of programs in comparison to previous generations. These improvements have been implied an exponential improvement related to time. As a consequence, algorithm design and programming has used the same tools to describe tasks for decades.

Limitations over instruction level parallelism (ILP) and energy consumption has broken this tendency [26]. Nowadays desktop computers increases the number of cores and execution units in order to maintain the exponential improvement, but, now it is required to re-engineer existing programs and algorithms to use additional execution units.

1.1.1. Architectures

We focus on three main topics: multi-core, homogeneous versus heterogeneous, and memory hierarchy (distributed versus shared memory).

Market has demonstrated that multi-core are present on most of our homes and offices, we have to deal with them. Multi-core processors are quite close to multiprocessors, both have similar structure: multi-core are a kind of multiprocessor embedded into a simple chip die. Multi-core distances are shorter than multiprocessor, so communication and synchronization are faster (between cores). As a counter part, multi-core have less room for cache memory for each core and memory bandwidth, both are shared between all cores of the same die.

Homogeneous systems are simpler to deal with than heterogeneous systems. Homogeneous systems have basically the same unit processors replicated, and any of them can execute the same functions at almost the same speed. Heterogeneous are quite more complex, they have many kinds of unit processors. Each unit processor can have its specific characteristics. The main advantage of heterogeneous architectures is that they have specialized computing units, so there are some computations are many times faster. As a drawback, each unit processor from a heterogeneous architecture can execute only a set of functions. The programmer must decide how to create each set of functions and how may them interact. As the

number of cores grows, there are more chances to have specialized cores, like happens on CellBE, Fusion, Larrabee, or even on desktop computers with CPU and GPGPUs.

Memory hierarchy is usually defined by multiple levels of cache. As smaller is the memory component faster it is. Main memory usually is slow in comparison to the processor. In order to reduce latency there are caches between memory and processor. Caches near to the processor are smaller, but at the same time they are faster. Closer caches to the processors (Level 1, and Level 2) are usually embedded on the same die with the processing units. That is the big problem of caches and multi-core/multi-processor: multiple processing units are sharing the same main memory, so they should have a consistent view of its content. Unfortunately, it means that either all of them share the same caches or all caches must be aware of all memory changes. There are two schemas to face this problem of memory hierarchies: shared memory (add some kind of coherence protocol) and distributed memory (avoid the problem and be inconsistent). Shared memory assumes that all processors are sharing the same data, so changes must be coherently notified to all caches. On the other hand, distributed memory slices memory into multiple isolated regions. It avoids the problem, so there is no need to maintain coherency between all caches. Shared memory is easier to use, programmer does not need to know where is the data located, just use it. Distributed memory require a strict control about data location and synchronization, programmer must decide where to store data, and how synchronize parameters and results. In other words: someone has to solve the problem. Shared-memory implies that the architecture solves the problem, distributed-memory implies that the programmer must solve the problem of data distribution.

Original desktop computers started as single core homogeneous (there is only one processor, so there is only one kind of processor) shared memory (there is only one coherent cache hierarchy), but they are now multiple core homogeneous shared memory [19], and it seems that they are turning heterogeneous with distributed memory [31]. We also can consider that they are becoming dual processor systems, one generic CPU and one general-purpose graphics processing unit [20].

Section 1.2. Algorithms And Parallelism

Most algorithms are designed to be executed on a single core and general-purpose processor. Consequently, they are not designed to take advantage of all available resources on current desktop computers. By not taking into account available resources, it appears a gap between effective algorithm performance and potential algorithm performance. This gap will increase at the same rate that cores count on computers increases.

There are many approaches to solve the gap between existing algorithms and existing architectures. The first and naive way is to rewrite completely existing algorithms and applications to work in those new architectures. This approach is very complex, requires a great effort, and also requires a large amount of qualified people able to transform algorithms. As we have already stated: a perfect approach would be automatic transformations, like automatic parallelisation, but there is a lack of required information at compile level. Our selected approach is using annotations.

Annotated programming model is a programming model based on an existing model, but annotations are added to increases the available compile time information. A good example of annotated programming model is OpenMP. OpenMP is designed as a set of annotations over C, C++ and Fortran applications. OpenMP targets multiprocessors and split the algorithm into multiple parallel computations. In order to enable parallel computation on OpenMP, programmers should add a few annotations on their programs. OpenMP will adapt the application to the many multiprocessors and use the underlying tools and thread libraries automatically.

1.2.1. Parallelization Tools

In order to parallelise an application we have to transform an application to use the underlying architecture, but we also must study the application to know which processes are critical to parallelise.

This parallelisation task requires many tools, each of them covers a partial set of requirements and some of them overlaps on their features. This tools are, from lower levels to higher levels: compiler, dynamic linker, auxiliary libraries (like libc), threads

library, synchronization libraries, communications libraries, runtimes, profilers, profiling libraries, profiler visualizers, and simulators.

Compiler is the most basic tool, it translates an algorithm to binary code. Compilers usually work with the higher level code: transforms code to low level operations. This transformation looses most of high level information. This information includes variable names, loops structures, … The compiler is the first responsible to take advantage of underlying computer architecture. As a consequence it requires the maximum information about the algorithm and target architecture.

Execution environment is not fixed, even the same binary can run on machines with different configurations (as an example, each machine can have a different amount of physical memory). Compiler does not generate a binary for each architecture (mainly because it is not practical). Instead of this, compilers use auxiliary libraries (such libc or a run-time) to adapt the binary to a specific environment. Dynamic linkers are key on this process, they link, and completes the binary, with the most suitable library for the executing architecture.

Thread libraries and synchronization libraries are usually highly coupled and usually shipped as a single library (for instance pthreads library). There are two main classifications: user threads and kernel threads. User threads are threads controlled by user space, they are usually mapped onto a single kernel thread, and I/O operations may block all of them. Kernel threads are threads controlled by the operating system and they can be mapped over multiple physical threads or processor units. Most threads libraries mix user threads and kernel threads to achieve optimum performance.

Synchronization libraries are highly coupled to thread libraries, the main motivation is to select the most suitable wait policy and perform it. Default wait policy is usually suspending thread execution; this action requires the exact knowledge of threads implementation in order to change active thread. On the other hand, for high performance applications waiting kernel threads policy is based on active waits. They use this kind of waits in order to resume work as fast as possible. Two main synchronization primitives are mutexes and barriers. Mutex ensures mutual exclusion on critical shared structure manipulations. Some mutexes are replaced by the compiler with atomic instructions, if there are atomic instructions for

the enclosed operation. Barriers are used to synchronize execution of multiple threads. Threads are collaborating on the same problem, for each step or point of synchronization a barrier primitive is used. Barrier ensures that all threads have reached a specific point.

Communications libraries are required for distributed memory systems, but they are uncommon on shared memory systems. There are four main primitives: send, receive, broadcast and reduce. Send and receive are designed to send or receive a simple piece of data, usually point to point. Broadcast sends the same data to many execution threads. Reduce primitive summarises many data into a single result, it receives a data set from many locations and performs the reduce operation (for example a summation operation). Synchronization primitives are often synchronous, that means that they require to use a synchronization library in order to wait on receive operations, if data is not yet present, or send operations, if there is not enough buffers to store the result (or if the operation is defined as completely synchronous).

Runtimes are libraries designed to support the compiler or the behaviour of a programming model. Runtime library provides a new set of primitives related to each programming model, it allows to express high level concepts. These primitives use other libraries to spawn threads, synchronize and communicate data. Compiler does not need to know how to deal with this low level libraries, it needs to know which primitives are implemented by the runtime. For example, there is an OpenMP runtime primitives defining a parallel region. Runtime decides how many threads are created to execute this parallel region. Compiler just creates a binary using this primitive. But there is a more important point, the resulting binary does not contain information limited to thread creation (as it should happen using low level libraries as pthreads), it also has information about parallel regions structures, and this information can be used to adapt the binary to each environment.

Annotated programming model and compilers are based on hints or comments over serial working code. That means that a starting point for an annotated program is a common serial well known and tested application. Annotations are just comments, they can be added one by one ensuring a correct behaviour. At the same time, they can be disabled (just removing or using common comments) to return to the serial behaviour.

Profiling tools are useful to achieve a good performance on parallel architectures. Amdahl's law [32] states that an application 95% parallelisable can be speedup x20, but no more (even if you use a thousand processing units). As a consequence, it is important to know which parts can be easily parallelised and how many threads can be used. Profiling tools are more than time watches: they report important information like data movements, false sharing, or load balancing. This information is usually collected while the application is running helped by profiling libraries. In addition, some profiling libraries also includes user events, they help to add runtime information or programmer relevant information.

Collected profiling information is usually studied by the user or programmer. Some times this information is also used by some compilers and runtimes to optimize future executions. Provided information allows to know the impact of the architecture on the program. There are some tools that just list some statistics, but there are some other tools that visualises the execution.

Simulators are one step further on profiling tools. They use profiling information and architecture information in order to extrapolate the behaviour on different architectures. Simulators also helps to predict algorithm performance on experimental architectures, when real hardware does not exist yet.

Section 1.3. Graph Matching Algorithms

One of the objectives of graph matching algorithms is the computer vision. In this premise real-time response is the requirement for the viability of an algorithm. There are many algorithms close to be executed on real time; in order to make them viable, these algorithms must be able to extract the maximum performance from current computer architectures.

Computer vision process is usually split in two processes: image acquisition and preprocessing and image recognition. Image acquisition transform a raw image information (a sequence of pixel colours) into a more usable information (as for example border maps, frequency maps or regions of adjacency maps). Image recognition tries to understand acquired information, in order to perform this task, it designates symbolic information to acquired information.

There are many processes and algorithms to do computer vision. From all of them we have selected a graph matching algorithms. These algorithms first require to transform acquired image into a graph representation. A typical transformation process is to convert input image into a segmented image by regions. Once this is converted, an adjacency regions graph is created. Result graph is matched against other representative graphs in order to detect known patterns.

Section 1.4. Streaming And Scientific Applications

High performance computing is known by its effort to extract the maximum performance from underlying computer architecture. Moreover, high performance underlying architectures usually involve multiple processing units and complex memories hierarchies. For this reason, their target applications are a good candidate as a starting point for multi-core. These applications are usually scientific applications, and their structure seems to be very close to graph matching algorithms.

On the other hand, data acquisition systems are usually designed to work in real-time on embedded devices. These devices can have multiple processors and specialized units in order to achieve real-time and low consumption requirements. Embedded systems usually works with streaming programs, these programs are designed to process a continuous flow of data. Streaming programs are very close to acquisition image and transformation algorithms of computer vision, and even many of them are already converted to streaming programs [33].

Beyond instruction level parallelism (performed by processors themselves), there are two main kinds of parallelism on applications: task parallelism and data parallelism. Task parallelism assumes that there are multiple tasks (or functions), these functions can be executed simultaneously and independently. Data parallelism assumes that each task replicates the same computation over each element of a large set of elements, computation for each element can be performed simultaneously and independently.

There are two main operations on data parallelism: parallel map and parallel reduction. Parallel map applies the same operation over each element from a large set of elements. Parallel reduction applies an associative operation over all elements

of a large set. As a result, parallel map obtains a set of elements. Each result element is computed in parallel, computation must be independent from other elements. All elements can (not as a requirement) be computed in parallel. An example of parallel map is to multiply per two all elements of a set. Parallel reduction computes as a single result element. This result is the computation of an associative operation over all elements of a set. In order to parallelise these operations, multiple associative operations can be performed over distinct sets of elements simultaneously (they must be independent). Resulting partial results are combined with the same associative operations until obtain a final result. An example of parallel reduction is summation of all elements of a set.

Streaming applications and algorithms are usually designed to exploit first task parallelism and, if there are enough free resources, then data parallelism. Streaming programs exploit task parallelism with techniques known as pipelining: data processing is split in stages, each stage computes over the result of the previous stage and produces the data required for the next stage; all stages are computed simultaneously for different time sliced sets of data.

Scientific applications and algorithms are usually designed to exploit data parallelism. These applications usually computes over a large sets of data, and usually computes iteratively over the set until reach to a solution or expected state. Each data element is computed in parallel, but the application must wait to finalise the processing of all data elements before starting a new iteration. A good example is a weather simulation: a grid represents values from the atmosphere (for example pressure, humidity or temperature). For each grid cell it computes the next time values using contiguous cell values. In order to simulate a large time weather prediction, this process is repeated many times. All cells can be processed in parallel for one algorithm iteration, but results of this process are required to start the next step.

Chapter 2. Objectives

The objective is to find a set of tools and directives to help algorithm designers in computer vision and pattern matching. These tools and directives will allow to algorithm designers take advantage of current underlying computer architectures and their possibilities. We want to demonstrate that existing tools for supercomputers are helpful for multi-cores. We will focus on those tools that do not require a deep knowledge about computer architecture. We expect that this kind of tools are suitable for described task with an acceptable performance. We will evaluate these tools on many environments or architectures. Recommendations will be performed in order to improve their efficiency.

We define two main criteria to evaluate tools and directives: usability and performance. We expect from tools to be usable by people with no deep insights of underlying architectures. Adapted algorithms and applications must run with an acceptable performance. We are looking for a trade off between this two criteria: performance and usability. For example, it is worthy to lose a 20% of maximum performance if it can be achieved by a standard algorithm designer with a small effort. We do not expect to require an architecture expert plus a great effort on the algorithm or program parallelisation and distribution.

Usability criteria is related with required knowledge: the possibility to exploit a many kinds of architectures with the same source, and of course, the possibility of incremental code transformation and debugging. Usability criteria will benefit those solutions which hide architecture details and focus on algorithm semantics. We expect from an algorithm designer to know insights from its algorithm, but not to know from architecture details. That means that tools should be focused on algorithm semantics better than architecture details. Tools should allow to provide

enough information to take advantage of current architecture. If the additional information is good enough, it should be useful for many kinds of architectures. On the other hand, we can not expect from a programmer to adapt a large application or algorithm at once to a new architecture. For this reason tools may help to change parts of the algorithm or application incrementally; at each step, the algorithm designer or programmer may be able to test if provided information is correct, and if results (algorithm and performance) are expected. If at any step the application do not works properly, we expect that the programmer will debug the application in order to solve any problem.

Performance criteria is more related with the possibility of increasing application performance on future processor architectures than to have a very good performance in a current or specific architecture. For years we have assumed that a single threaded will increase its performance over each new processor generation, now we know that it is not longer true. Current market trends seems to indicate that the number of cores and available threads will increase, in this scenario, a good performance criteria is scalability. Scalability studies how an algorithm or program is able to keep improving its performance in the same ratio that cores or hardware threads increases. A good scalability will ensure that future processors, with an increasing number of cores, will be able to speedup the application.

Section 2.1. Graph Matching And Scientific Applications

Scientific applications have been used on supercomputers and many architectures for decades. In other words, scientific applications are a very well known problem. On the other hand, research and practical uses of graph matching algorithms on supercomputing environment are not common. In other words, graph matching algorithms are still in research and there are many potential problems to solve. By using scientific applications, instead of graph matching algorithms, on first steps of this thesis, we can reduce potential unexpected problems.

In our research we want to face only one problem at a time. When we have started this research there was little applicability of supercomputer tools in multi-cores. Doing graph matching and using supercomputer tools in multi-cores faces two

problems at a time: 1) supercomputing tools on multi-core, and 2) graph matching with supercomputer tools. There are two approaches: 1) port graph matching to supercomputing using supercomputer tools, or 2) port a well tested scientific application with supercomputing tools to multi-cores. Approach one has a considerable drawback: supercomputers are really very expensive; moreover, results on supercomputers are not extensive to multi-core. Approach two, per contra, makes available a very large list of bibliography to solve the problem. We have selected approach two.

First steps of this thesis will start doing research with supercomputing scientific applications. As our objective is to port graph matching and computer vision algorithms to multi-core, we first will compare scientific and vision algorithms in order to find an equivalence.

Hypothesis 1: *Graph matching algorithms have common characteristics with scientific applications used at supercomputing.* There are many kinds of applications from scientific world used on supercomputing. Most of these applications shares patterns and structures, in addition they are focused to intensive computations. For example, scientific applications patterns and structures differ from classical administration applications.

Objective 1: *Prove that graph matching algorithms have common characteristics with scientific applications.* From all available scientific applications, we focus only on a set of applications that 1) have potentially common characteristics with target graph algorithms and 2) have been successfully adapted to many supercomputing environments using tools that satisfies our criteria of usability and performance.

Section 2.2. Annotated Programming Model Over Multi-Core

A homogeneous multiprocessor with shared memory is the most usable supercomputer architecture. Annotated programming models for these architectures have one of the best trade off between usability and performance. We want to achieve the same trade off in multi-core architectures.

Multi-processor are basically multiple processors of the same kind and characteristics interconnected sharing the same data. In other words: all threads have the same computations capabilities and can use the same data simultaneously. Complex memory operations like data synchronization are hidden from application programmer by hardware.

Annotated programming models uses annotations in order to extend available information at compile time. As less information is required, and less specific, more usable is the programming model. Most popular multi-processor annotated programming models are born in multi-processor homogeneous shared memory architectures. Shared memory implies no need for data partitioning or explicit communications annotations (like any standard application). Homogeneous property allows to execute the same code in any arbitrary hardware thread.

Shared memory homogeneous multi-processor annotated programming model, as OpenMP, focus on annotations to express algorithm parallelism. These annotations help the compiler to identify which parts can be executed in parallel, in other words, which parts have no dependences. It allows the compiler and the runtime to decide how many threads use and how to perform synchronizations. This decisions can be different for each architecture, ensuring a good performance under many architectures. At the same time, OpenMP can hide architecture to the programmer. These properties enable usability and performance.

We focus only in tools for multi-processor homogeneous shared memory. Multi-processor and multi-core differs on communication and synchronization velocity, cache size, and memory bandwidth. On the other hand, multi-processor and multi-core both run multiple hardware threads. A set of annotations based on expressing algorithm parallelism help the compiler to take advantage of hardware threads, no matter if threads are multiple processors or multiple cores. We expect to overcome differences like communication velocity or cache by changes on the compiler/runtime. We also expect to allow applications to exploit multi-core improved communication and synchronization latencies.

Hypothesis 2: *Supercomputer architectures based on multiprocessors have similar characteristics to multi-core and multi-threading processors in order to execute scientific applications.* Both architectures execute multiple hardware threads and have multiple functional units allowing true parallelism. But, on the other hand, there are some

important differences related to communication and synchronization (multi-core are faster) or related to memory hierarchies (multiprocessor have more high speed cache capacity). Communication and synchronization speed should open new opportunities on fine grain parallelism (parallelism which more frequent synchronizations), but low capacity on cache might affect negatively to execution of large data sets. We expect that low capacity on cache might be compensated by high speed synchronizations.

Objective 2: *Prove that multi-core processors present similar characteristics to supercomputer architectures and they execute successfully scientific applications.* Our objective is not to create a new adaptation of scientific applications for multi-core architecture, but use the same tools for multiprocessors on multi-core processors. We have explained in main objectives that selected scientific applications use tools that provides usability and performance. We focus on programming models based on annotations, which provides desired usability. Target scientific applications must execute successfully (with good performance) on multi-core with minimum changes. Runtime and compilers might be adapted in order to help performance.

Section 2.3. Annotated Programming Model Over Distributed Memory

As the number of processors increases and memory hierarchies becomes more complex architectures are more likely to be distributed memory ([17], [31], [34]). Annotated programming models are not designed for distributed memory, and they do not specify data movement. We want to study if it is possible to use annotated programming models under distributed memory and which changes are required.

Shared-memory is logical view of the physical memory. When there are multiple processors or cores, multiple caches, and many levels for replication, there is not a unique real data view. Each cache can contain many versions of the same data. When we talk about shared memory systems, or either we have a unique memory hierarchy, or we have a very complex piece of hardware that creates a logical shared memory.

Shared memory illusion is usually kept by cache coherence protocols. These protocols coordinate all caches for all levels sharing the same memory in order to synchronize values. Special hardware, implemented at caches and communication buses, maintain the coherency between all caches. When memory hierarchies are very complex, this hardware is also very complex and coherency becomes very expensive.

Distributed memory architectures avoids shared memory complexity by not implementing it. Distributed memory architectures are organized in clusters: groups of computing resources (like processors, nodes) sharing the same memory. Distributed memory architectures have a specialized communication network in order to transfer data between all nodes of the same system. This network allows to send and to receive data explicitly from hardware threads of the network. Each node logical memory is like an independent computer which it is connected through a network. Nodes uses the network to send and receive data to complete its computation in collaboration with all other nodes.

Distributed memory architectures expect from the application programmer and from the algorithm designer to deal with multiple memories. All complexity and responsibilities taken from hardware architecture are given to the programmer. For this reason programming for such architectures is harder than programming for shared memory architectures. Programmer must distribute the data and the program itself into multiple nodes, in addition it must design its communication and synchronization.

Distribution work made by the programmer is usually better than any distribution performed automatically. Programmer knows perfectly application behaviour, programmer can foresee program necessities and rearrange application behaviour in order to reduce required communication. On distributed memory architectures, latencies are usually high and all synchronizations and communications must be minimised. This minimisation impact is so important that is the key to the design of distributed algorithms and distributed applications. Communications and synchronizations becomes critical. The viability of an application usually depends on the capacity to minimise such communications on a large number of nodes. Usually best policy is trying to keep together data and computations, in other words, best policy is having a high locality.

Annotated programming models were designed to simplify multi-processor programming. Shared-memory architectures are simple to use, and in order to avoid complexity, most annotated programming models focus on shared-memory. These programming models relay on shared-memory in order to avoid explicit data communication annotations and data distribution.

Some annotated programming models have experimental run-times that allows to simulated a shared memory on distributed memory architectures, they are called software-distributed-shared-memory systems (SDSM). These run-times uses software techniques to emulate shared memory. For example, not locally present memory pages can be invalidated on the current node and page fault can be used as a substitute for a cache miss. SDSM run-times allow that applications created for run shared memory architectures to work on distributed memory systems. Unfortunately, most of these applications are not able to have a good performance and scalability on SDSMs. Some benchmarks even presents a slowdown: performance is worst two nodes (more execution resources) than one single node. This behaviour is not surprising: shared memory applications are not designed to work on distributed shared memory and they are prone to compute with pieces of data spread through all cluster, in other words, they have a low locality. Locality was important but not critical on shared-memory architectures.

Distributed shared memory systems faces two problems: 1) some decisions about the design of applications are not aware of distributed nature of architecture and 2) there is a large latency to receive required data not present locally. Solving problem 1 requires the expertise from the algorithm designer or from the application programmer, so some applications must be rewritten in order to increase its locality. Nature of problem two is a little more tricky. All shared-memory hardware threads stalls (halts) when data is not ready (for example, if data is not present on the cache, thread must wait until data is received). When data is not locally present, it is likely that it is being used by another node, the latter node must send the data to the former in order to be used.

Prefetch technique allows to send or receive a data from another cluster before the execution requires the data. Prefetch technique is able to avoid thread stalls for data by advancing data communication. Data communication can be performed in parallel with computations to hide its transference cost. A good prefetch

performance is able to hide all communication costs and show a performance comparable to shared architectures. A poor prefetch performance can send wrong pages and invalidate them on source node. As a result, source node will have to ask again for data that originally was present. Distributed shared memory run-times uses prefetch, but annotations does not have information related about data movements, so they have to perform predictions.

Hypothesis 3: *Annotation based programming models are also valid on environments of distributed memory*. Algorithms and applications designed to have a high locality, in addition to good prefetch predictors (or some annotations) can have a good usability and performance on distributed memory architectures.

Objective 3: *Prove that annotation based programming models can effectively (usability + performance) run on distributed memory architectures.* Addition information required by the programming model should be minimal and do not require a great effort on application restructuration. We deal with a trade off between application transformation and performance. We expect a little effort from the algorithm designer or programmer in order to increase locality and performance, but not the same complexity required by explicitly programming for distributed memory.

Section 2.4. Heterogeneous Processor Simulator

Not every architecture is designed to have a good usability. There are many architectures designed to be used by expert programmers on such architectures, usually limited for target tasks. Small changes on those architectures can jeopardize the maximum performance, but can enable better results from non expert programmers. One of the biggest problems is distributed memory, but distributed memory having a global linear address space can improve its usability. In other words, a mechanism to use memory remotely, even at expenses of paying a large latency.

As more complex is an architecture, more knowledge it is required from the algorithm designed and application programmer of underlying architecture. General-purpose old fashioned processors allows to achieve a good performance

without any special knowledge. Some specific architectures have special features that allows performances many times better than any general-purpose processor. Unfortunately to achieve such performance it is required to use such special features. Using these features requires a high knowledge of architecture. On most of the specific architectures, if especial features are not used, their performance can be lower than generic architectures.

As the number of cores grows, there are more chances of having specialized cores. Having 16 cores identical for general-purpose is not a good idea: there are many tasks running on general-purpose computers that can take advantage of special cores (graphics processing, video streaming, …). But there are not too much tasks able to take advantage of multiple identical general-purpose threads. It even is likely to have dedicated cores for each kind of task, in order to achieve a high performance.

There are two main problems of heterogeneous architectures: each core is specialized in computing a kind of code, and they usually have distributed memory. It has two consequences: the programmer must deal with code distribution (which function or task is executed by which core) and data distribution (where data will be located and how will be transferred).

Having specialized cores implies that the programmer or algorithm designer must decide how to use each core. In other words, the programmer must create applications for multiple processor kinds on the same code. It is not usable, kinds of cores can change between processor generations as available and require a very complex task for a programmer. This problem is already present on general-purpose processors, like Intel ones: specialized vectorial instructions have evolved, specialized programs must be rewritten in order to take advantage of each generation (MMX, 3DNow!, SSE, SSE2, SSE3, SSSE3, AVX, CVT16, FMA3, FMA4, XOP). Some compilers enable automatic vectorisation (they choose for a target architecture which instructions to use), but their usually require to use annotations like #pragma ivdep (equivalent to #pragma omp parallel for) in order to enable some optimizations (in this case, the programmer reports that there are no dependences between loop iterations, even if compiler can not ensure). Nowadays, maximum support in this line is to allow the addition of some annotations in order to enable optimizations (ignored if the underlying architecture does not support some

features) or describe requirements in some parts of the code (used to decide target processor).

Distributed memory is a difficult problem. In Objective 3 we try to show that there is possible to use distributed memory as shared memory, so programming task can be lighter, but we also add one requirement: some mechanism to allow specify address to be read. Usually all pointers on distributed memory architectures focus only on certain nodes: all pointers are relative to one node and there is no information related to which machine has the information. That limitation requires from the user to know where a pointer points, to which node memory. This information usually is nowhere else than the programmers mind, and prone to errors.

There is a concept, global linearly addressable memory, that allows to have global pointers, even in distributed memory. It can be seen as an architecture with distributed memory which pointers mix local address to current relative node plus a cluster id. On this kind of architecture, addresses from other clusters can be read (with a simple load or store instruction) but they may be slow and require special instructions in order to keep local pointers fast.

Global linearly addressable memory allows to increase usability: a programmer can iteratively optimise an application. First the programmer creates an application assuming shared-memory (even if it is very slow), and then, iteratively step by step, the programmer transforms global accesses by local accesses. With this model, distribution on firsts steps may have poor performance, but the programmer can start to distribute data (or add annotations for such finality) without writing all process from scratch (or even paying small penalties for few distributed accesses).

There are already some architectures that are distributed and they are a good challenge for theirs programmers. We state that to modify such architectures in order to have global linearly addressable memory may be worth. This change is not free, it requires to pay some kind of penalty. We propose to use a simulator in order to modify the hardware and to evaluate the performance for non expert programmers.

A simulator for a heterogeneous multi-core is relative new. Most simulators provides capability to simulate one processor, with one kind of ISA, with one memory hierarchy. Distributed heterogeneous multi-core are a bigger issue. We require a modular simulator able to create almost an arbitrary hierarchy with

multiple local addressable memoires and cache hierarchies. In order to show the usability of global linearly addressable memory architectures, first, we need to simulate a distributed memory processor, validate it, and perform modifications. Results must be evaluated in order to know how changes impacts on usability and on performance.

Hypothesis 4: *Small changes on architecture can help to improve usability.* There is a trade of between usability and architecture performance. Some architectures have very good performance results with almost any code (like generic architectures), but some other architectures need deep knowledge of the architecture in order to achieve the maximum performance. It is possible to modify architecture in order to increase usability, but on the other hand, the architecture can decrease its maximum performance. We state that some changes on architecture can create a good balance between two possibilities: to make available good performance ratios with low knowledge of underlying architecture.

Objective 4: *Present changes in the architecture that increases usability on complex architectures, even if we loose partial hypothetical maximum performance.* In order to satisfy this objective we use a simulator and other architectures in order to prove candidate changes. Changes must show how concepts like global linearly addressable memory on distributed memory can help to increase usability with an acceptable cost.

Section 2.5. Graph Matching Preprocessing And Streaming Applications

Streaming applications have perfect properties for heterogeneous distributed memory architectures: they have multiple independent kernels (functions) with explicit data flows. Graph matching preprocessing algorithms are in fact image acquisition programs. This kind of programs usually present streaming characteristics. Streaming applications have been used on multiple research and are deployed on multiple commercial system. If we stablish that they have common characteristics with graph matching preprocessing algorithms, we can use a well known algorithms in order to perform firsts steps with streaming architectures.

As we stated above on section 2.1 we want to face a only problem at a time. There is a large research on some streaming applications on many kinds of architectures. There are also a large number of streaming programs deployed on commercial systems. On the other hand, there is not an annotated programming model to create streaming applications. So we stablish once again three-step research: select streaming applications comparable to graph matching preprocessing, use streaming applications to define a programming model for streaming applications, apply resulting knowledge to graph matching preprocessing.

Streaming applications are based on independent kernels with explicit communication data flows. A kernel is like a function, executed in its own thread, within a loop whose iterations are computed each time that a data set is received. Kernel memory is private, this memory is not accessed by other kernels. All communications between kernels are performed using synchronous data channels, or specialized asynchronous directives. Kernels computations are performed over incoming data from synchronous data channels. Their results are sent to other kernels through outgoing synchronous data channels.

Synchronous data channels communicates kernels, they represent a data flow. There are three main operations over synchronous data channels: push, peek, and pop. Data channels are flows of one single type of data. Push operation is applied to outgoing synchronous data channels, it adds one data element to the channel. Push operation is synchronous, it can stop its kernel execution if there are not enough available space to receive data. Pop operation reads and discards one element from a synchronous data channel. Pop reads elements in the same order that push adds elements. If a synchronous data channel is empty, pop blocks its kernel execution. Peek operation reads one arbitrary non discarded element from a synchronous data channel. If required element is not yet pushed, peek blocks its kernel execution. Peek operation is performed carefully because it decides the minimum number of elements of synchronous data channels buffers. Peek operation can induce deadlocks when there are loops or cycles in the data flow.

Some architectures do not allow kernels to block. For this reason, the number of consumed elements and produced elements for each kernel must be known by runtime.

Any streaming application is in fact a graph: each node is a kernel, and each edge is a synchronous communication data channel. This structure is very malleable: nodes and edges can be grouped in order to share affine resources. If a compiler knows the streaming application structure and properties, it can decide how map each node and edge to available architecture resources. If a graph matching preprocessing algorithm can be expressed as a streaming graph, it can be implemented as any streaming program and take advantage from streaming optimisations.

Hypothesis 5: *Image acquisition algorithms for a graph matching algorithms presents similar characteristics to streaming applications.* Streaming applications are focused on data streams treatment and transformation, they usually acquire data from external devices or large data files and process elements as an ordered sequence. Data results are produced as input is read. Given a time-stamp, current results only depends on current and previous inputs, but are independent from incoming future results. These applications base its behaviours on multiple processors with private memories. We state that graph matching preprocessing are in fact image acquisition algorithms and the same parallelisation techniques can be applied. As a consequence these applications can be executed successfully on heterogeneous multi-core distributed memory systems.

Objective 5: *Prove that image acquisition algorithms for computer vision have similar characteristics to streaming applications.* Image acquisition algorithms must present characteristics from streaming applications relevant to time sliced data processing, multiple kernels, and private memories for kernels.

Section 2.6. Annotation Based Programming Models And Streaming

Giving information about data transferences can help the compiler and the runtime to improve the performance on distributed memory systems. Streaming applications are conceptually equivalent to multiple kernels (independent processing tasks) and data transfers channels. Annotations can be defined in order to extend the

information provided by the programmer to the compiler. These extensions can allow the compiler to manipulate a standard application as a streaming application.

Streaming applications are basically graphs: their nodes are execution kernels and their edges are synchronous data channels. A kernel is a function, in other words, a delimited region of code. A synchronous data channel is a data flow from one kernel to another, in other words, a kernel result stored into an intermediate memory and an input data read by another kernel.

Kernel functions are usually activated (executed) depending on inputs. Producer kernels are able to generate many elements without an input, as for example a kernel of device reading. Another kernels can consume all elements without generating elements for another kernel (as for example a kernel of device generator). Kernels must be executed at specific frequency, usually inputs define this frequency.

Kernels also generates output elements. These elements are generated each time that the kernel are activated. The number of generated elements depends of each kernel. There are two kinds of kernels: 1) kernels that generate a constant number of element given a number of input elements, and 2) kernels that generate an arbitrary number of elements (some times depending of input values). Former kernels allow a large number of optimisations due to the kernel scheduling can be precomputed even in compile time. The latter requires a dynamic scheduling.

Streaming exploits task parallelism naturally: each kernel by definition is a function which task can be executed in parallel with another kernels. Required synchronization is given by synchronous data channels. Some kernels also allow data parallelism. Data parallelism can be exploited on streaming applications by replicating kernels (multiple instances of the same kernel) and slicing data streams; as an example kernel instance 1 computes even elements, and kernel instance 2 computes odd elements. Each kernel instance is a new kernel. Regardless every instance of the same kernel performs the same computation, each instance has its own private state (private memory). This state is not shared with other instances. For this reason, data parallelism is usually restricted to those kernels which state does not need to be shared among instances. It can be achieved by not having state, or by replicating computations to update state.

Peek operations on synchronous data channels are very common on streaming applications. Peek is performed in order to access data already present on input

streams. Objectives of peek operation are to avoid kernel state variables and reuse streaming buffers. A classical use of peek is to implement very efficiently sliding windows. The implementation of an average of last four inputs, as an example, can be computed with a state variable or by peeking the last four elements on the input data channel. First implementation does not allow to use data parallelism, but peek solution allows data parallelism. Peek must be used carefully: it requires to access to data from multiple activations. First activation has to wait for data equivalent to many activations. For this reason first activation occurs after all required elements are received and ready to be peeked. Input elements and output elements ratios are lost. For example, if a slider window of 4 elements is required to generate one element (as the 4-average example), it is not a 1:1 ratio because three first inputs are not producing an output (it should be a ratio about 1:0.9999, the exact ratio depends on the number of processed elements).

Most streaming programs also use non synchronous data in order to control some non critical aspects, as for example volume level. These data produces no kernel activations. Asynchronous data are written externally as soon as possible by other processes or kernels themselves checks for updates.

Annotations on streaming applications must allow the compiler to transform serial programs into streaming ones. Annotations must be able to provide required information in order to identify kernel functions, input and output synchronous data channels, peeks over input channels, state variables, how to create multiple instances of a single kernel, and which are asynchronous variables. With all this information, the compiler and the runtime should be able to generate the equivalent streaming graph from the serial program. A program converted to a streaming graph can be adapted to any architecture capable to execute streaming programs.

Given the complexity of creating a streaming program, we give a great relevance to usability. Streaming programming model was created by imitation of electronic circuits: each kernel was a physical component, and each data channel was a wire connecting two components. On the physical world, wires carries continuous signals (non discrete signals). Input and output ratios are always perfect; there is no such as a failure because there is a wire having no data. On software world, data channels carries discrete elements. Ratio of element consumption has to be controlled perfectly. A producer which produces more elements that can be consumed can

produce a buffer overflow on the output. A producer which produces less elements that are consumed can produce a dead-lock if there are cycles. Simple linear data-flows has not such problems, but feedback data loops and convergence of many flows is critical to control producer and consumer ratios.

In order to achieve usability we must ensure two conditions: incremental construction and control about consumer/producer ratios. Annotated programming models are incremental, but changes only affect to current region. A streaming application connects many parts of the application graph. If a new kernel is defined, its synchronous data channels must be created and connected to its producers and to its consumers. Kernel can not work if required synchronous connections are missing or are miss-connected. At the same time, this connections must ensure that consumer/producer ratios are kept, and as a consequence, the frequency of kernel invocation and its ratio is the expected. These two points leads us to one conclusion: create connections between kernels are a very critical task on usability. And this task requires to interconnect parts remotely.

Creating a new kernel, or removing an existing kernel (for debugging purposes, as example) requires two actions: to define a kernel and to define correct channels and connections. As a consequence required annotations must allow to specify minimum information and let to the compiler reconstruct from semantics remaining required information. Kernels can be defined as a block of code (defined as a function to be executed as a kernel), but to create synchronous data channels must require to connect at least two kernels and comply with all requirements for legal connections. Annotations should be based on kernel definition and provide enough information in order to let the compiler build the application graph.

Hypothesis 6: *Annotation based programming models can be used to describe streaming algorithms.* Annotation based can be extended to describe required information to split streaming algorithms into multiple kernels and data flows. Annotations about kernels must be able to select independent algorithms parts (kernels) with private data with input and output for data flows. Annotations should be able to link dataflows from multiple kernels and create data flow paths. Annotations should assume a minimal effort from the algorithm designer or programmer. Compiler must be able to identify different parts correctly and generate a streaming applications able to be executed on streaming architectures.

Objective 6: *Demonstrate that there is a small set of annotations able to create streaming applications from serial algorithms.* Selected serial applications (applications that have specialized streaming versions) can be converted effectively to streaming applications (not necessary the same). Compiler must be able to do this transformation using existing serial code plus annotations. Transformations must keep global objectives (usability and performance). Annotations must be simple enough and versatile in order to have a high degree of usability: annotations must be able to be introduced iteratively (step by step) and minimise the possibility of introducing bugs for non expert streaming programmers. Result programs performance must be comparable to existing performance of streaming programs.

Section 2.7. Graph Matching On Current Architectures

We have been focused on providing usability and performance on incoming architectures. We expect to have discovered enough properties in order to take advantage of acquired knowledge and apply to graph matching algorithms and image acquisition. We pretend present graph matching algorithms adaptations as a prove of concept. At the same time, we also want to present some directives for future developers and designers in order to maximize their capacity of taking advantage of incoming architectures.

We have researched in two main lines of algorithms: scientific applications and streaming applications. Each of this kind of applications have different requisites and different properties: streaming applications are able take advantage of small tasks with small memories, scientific applications require random access to large amounts of data. It is likely that streaming applications are suitable for almost any architecture, but scientific applications must be carefully designed in order to maximize locality.

We expect to implement image acquisition algorithms as a streaming program. In fact, nowadays, these kinds of algorithms are already used on digital television. We expect no major problems. The objective is to identify a set of tasks and channels in order to create the application streaming graph. Steps used should be create a serial application (in other words, a common application for a general-purpose processor),

and then, add required annotations in order to define the application streaming graph.

Graph matching algorithms, on the other hand, requires access to larger data sets, and there are not too many scientific applications using graphs in their implementation. The first step should be to transform the original algorithm into an algorithm able to exploit maximum locality. Once the algorithm is transformed, it must be encoded as a serial application. This application must run on a general-purpose processor without any change. At this point, required annotations must be added in order to provide enough information in order to let the compiler parallelise and distribute (if it is required by the architecture) the application.

All process must show that the discovered knowledge is applicable to the original problem. Final results must provide and acceptable performance and be enough usable. Performance must show that the application is able to scale in many architectures and provided annotations are enough. The same annotations must allow the compiler to adapt the application to each tested architecture. Usability should be good enough to allow a programmer (non expert on architecture but assisted by the presented tools) to reproduce experiments.

Hypothesis 7: *All presented techniques can be applied to target algorithms: image acquisition algorithms and graph matching algorithms.* Until this point we have studied similarities between this to target algorithms and state of art algorithms on supercomputers. We have hypothesised that graph matching algorithms are similar to scientific supercomputing applications and image acquisition algorithms are similar to streaming algorithms. We also have hypothesised that scientific applications can be executed efficiently on multi-core processors using usable tools. We have also stated that streaming applications can be expressed efficiently with annotation based tools. As a result we hypothesise that graph matching algorithms execute efficiently as scientific applications do, and image acquisition algorithms execute efficiently as streaming applications do.

Objective 7: *Prove that computer vision techniques based on a graph matching algorithms can benefit from incoming computer architectures.* They should be able to execute on multi-core with a high degree of scalability and in a programming model able to

allow incoming programmers to express their knowledge about algorithms in a manner that they can effectively use underlying architecture.

Chapter 3. State Of The Art

Section 3.1. Graph Matching

We focus on graph matching algorithms and its efficient implementation on current multi-core architectures. Classification is a task of pattern recognition that attempts to assign each input value to one of a given set of classes. Pattern recognition algorithms generally aim to provide a reasonable answer for all possible inputs and to do inexact matching of inputs. Pattern recognition is studied in many fields such as psychology, cognitive science, computer science and so on. Depending on the application, inputs of the pattern recognition model or objects to be classified are described by different representations. The most usual representation is a set of real values, but other common ones are strings, trees or graphs. These structures have more capacity to capture the knowledge of the model but their comparison or matching is also more computationally expensive. The distance between a pair of strings or trees is computed in polynomial time; nevertheless, the computation of the distance between a pair of graphs is exponential respect the number of vertices. For this reason, some algorithms that compute the distance between graphs have been presented that obtain a sub-optimal distance [24]. Although these last algorithms have a polynomial computational cost, the run time is not acceptable for some applications.

Graph structures have more capacity to capture the knowledge of the model but their comparison or matching is also computationally more expensive. Sometimes in graph based pattern recognition applications, given a set of graphs, which all represent equivalent or related structures, it is required to find global consistent

correspondences among all those graphs. These correspondences are called a Common Labelling (CL). Algorithms like [25] and [35] does pair matching and reconstructs a general correspondence, other algorithms like [36] uses Graduated Assignment [24] to generate the CL by matching all graph nodes to a virtual node set in a polynomial time.

Graduated Assignment Graph Matching. We denote a pair of attributed graphs by G^p and G^q. Attributes on vertices are denoted by G_a^p and G_i^q. In our application, attributes are a bi-dimensional value representing the position of the node. Attributes on the edges (arcs) are denoted by A_{ab}^p and A_{ij}^q. In our application, arcs do not have attributes, then $A_{ab}^p \in \{0,1\}$ and $A_{ij}^q \in \{0,1\}$ take binary values representing the absence or presence of an edge.

We define a matrix F by $F_{ai} \in \{0,1\}$ such that $F_{ai}=1$ if node a of G^p matches node i of G^q and $F_{ai}=0$ otherwise. F represents an isomorphism between a pair of graphs. Moreover, we define the compatibility between two nodes as $C_{ai}^{pq} \in [0,1]$. Due to the binary nature of the attributes on the edges, the compatibility between two edges is represented by the product of them $A_{ab}^p \cdot A_{ij}^q$.

Most of the algorithms that compute an error tolerant isomorphism between two graphs aim to minimize an objective function. The objective function usually has the following form:

$$E^G\left(G^p, G^q, F^{pq}\right) \in [-1,0] = \frac{-1}{R(R-1)} \sum_{a=1}^{R} \sum_{b=1}^{R} \sum_{i=1}^{R} \sum_{j=1}^{R} P^{pq}[a,i] \cdot P^{pq}[b,j] \cdot C_{aibj}^{pq} \text{ given } a \neq b, i \neq j \quad (3.1)$$

The objective function relates the isomorphism given by the probability matrix P^{pq} with the cost given by the function C_{aibj}^{pq}. The probability matrix P^{pq} represents F^{pq} in a continuous form. The cost $C_{aibj}^{pq} \in [0,1]$ measures the compatibility of labelling nodes a and b of G^p to nodes i and j of G^q plus the compatibility of labelling the corresponding edges between them. In our application, we define $C_{aibj}^{pq} = A_{ab}^p \cdot A_{ij}^q \cdot C_{ai}^{pq} \cdot C_{bj}^{pq}$. Due to the high computational cost that it is needed to find the minimum value of the energy in equation 3.1, it is usual to approximate it at point $\left(P_f^{pq}\right)^0$, using Taylor series expansion:

$$E^{G}\left(G^{p},G^{q},P^{pq}\right)\approx\left(E^{G}\left(G^{p},G^{q},P^{pq}\right)\right)'=\sum_{a=1}^{R}\sum_{i=1}^{R}\sum_{b=1}^{R}\sum_{j=1}^{R}\left(P^{pq}[a,i]\right)^{0}\cdot\left(P^{pq}[b,j]\right)^{0}\cdot C_{aibj}^{pq}-\sum_{a=1}^{R}\sum_{i=1}^{R}\left[\sum_{b=1}^{R}\sum_{j=1}^{R}\left(P^{pq}[b,j]\right)^{0}\cdot C_{aibj}^{pq}\right]\left(P^{pq}[a,i]-\left(P^{pq}[a,b]\right)^{0}\right) \tag{3.2}$$

Analysing equation 3.2 it is deduced that:

$$\operatorname{argmin}\{E'\}\equiv\operatorname{argmax}\left\{\sum_{a=1}^{R}\sum_{i=1}^{R}Q_{ai}^{pq}-P^{pq}[a,i]\right\} \tag{3.3}$$

where the P and Q are obtained as follows [24]:

$$\forall_{a=1}^{R}\forall_{i=1}^{R}P^{pq}[a,i]=\exp\left(\beta Q_{ai}^{q}\right)\ \text{ and }\ Q_{ai}^{pq}=\left[\sum_{b=1}^{R}\sum_{j=1}^{R}\left(P^{pq}[b,j]\right)\cdot C_{aibj}^{pq}\right] \tag{3.4}$$

The Graduated Assignment [24] algorithm is probably the most popular algorithm to compute a suboptimal solution for the graph matching among others. It minimises the objective function equation 3.1 in a suboptimal way by means of approximating the energy as in equation 3.3. The problem is equivalent to the quadratic assignment one where Q represents a cost matrix, and P represents the stochastic matrix which contains the desired assignation probability. Algorithm 3.1 shows the main schema of the Graduated Assignment [24]. Update function obtains the probability matrix as in equation 3.4 and Normalise function makes the probability matrix double stochastic [37].

Algorithm 3.1: Graduated assignment.

Input G^{p} and G^{q}

Initialise P^{pq} and β
repeat
repeat
$P^{pq}=\text{Update}\left(P^{pq},A^{p},A^{q},C^{pq}\right)$
$P^{pq}=\text{Normalise}\left(P^{pq}\right)$
until P^{pq} convergence
$\beta=\beta\cdot\beta_{r}$
until $\beta>\beta_{f}$

Algorithm 3.2 shows an implementation of *Normalise* function based on the Sinkhorn method [37] as follows,

Algorithm 3.2: Sinkhorn stochastic matrix transformation algorithm.

repeat

$$\forall_{a=1}^{R} \forall_{i=1}^{R} P^{pq}[a,i] = \frac{P^{pq}[a,i]}{\sum_{x=1}^{R} P^{pq}[a,x]} \quad (3.5)$$

$$\forall_{a=1}^{R} \forall_{i=1}^{R} P^{pq}[a,i] = \frac{P^{pq}[a,i]}{\sum_{x=1}^{R} P^{pq}[x,i]} \quad (3.6)$$

until P^{pq} convergence

Graduated Assignment Common Labelling. Graduated Assignment [36] is one of the algorithms considered to have a good run-time performance between most popular common labelling algorithms. This algorithm approximates a distance and a labelling between multiple graphs using a polynomial time method respect the order of the graphs. The result of the CL algorithm is a set of probability matrices { $P_h^1, P_h^2, \ldots, P_h^N$ } that represents, for each matrix, the probability of matching a node of one of p graph to a virtual node. Since any p matrix P_h^p values are continuous, a discretisation process of the probability matrix [38] is applied to obtain the final labelling between graph nodes.

Given a set of graphs {G^1, G^2, ...,G^N} (that have R vertices) and their respective adjacency matrices {A^1, A^2, ...,A^N}, the general outline of the graduated assignment common labelling is shown in algorithm 3.3 and 3.4.

Algorithm 3.3: General diagram of the Graduated Assignment Common Labelling.

$\beta = \beta_0$
Initialise P_h
begin Do until $\beta \geq \beta_f$
  begin Do until P_h convergence
    $P_f^{pq} = P_h^p \cdot (P_h^q)^T$
    $Q = Approx\_Q(P_f, P_h, C)$
    $P_h^p[a,w_1] = \exp(\beta \cdot Q_{a,w_1}^p)$
    $P_h^p = Stochastic(P_h^p)$
  end
  $\beta = \beta \cdot \beta_r$
end
M^{pq} = Discretise P_h^p

Algorithm 3.4: *Approx_Q* function description.

for $2 \le p \le N$
 $Q^p = 0$
 for $1 \le q \le N \wedge p \neq q$
  for $1 \le a,i \le R$
  $v_1 = 0$
  for $1 \le b,j \le R \wedge b \neq a \wedge j \neq i$
   $v1 = v1 + P_h^p[b,j] \cdot C_{aibj}^{pq}$
  end
  for $1 \le w_1 \le R$
   $Q_{a,w_1}^p = Q_{a,w_1}^p + v_1 \cdot P_h^p[i,w_1]$
  end
 end
end
end

C_{aibj}^{pq} represents the compatibility of labelling edge *(a,b)* of graph G^p to edge *(i,j)* of graph G^q and their respective ending nodes. In order to optimize C_{aibj}^{pq} computation it is defined as:

$$C_{aibj}^{pq} = \frac{1}{1 + C_{ai}^{pq} + C_{bj}^{pq} + dist\left(A_{ab}^p, A_{ij}^q\right)} \tag{3.7}$$

C_{ai}^{pq} is the precomputed distance between vertex *a* from graph *p* and vertex *i* from graph *q*, *dist* function determines the distance defined by the existence of graph *p ab* edges and graph *q ij* edges.

Function *Stochastic* obtains a double stochastic matrix [36] using the Sinkhorn method [37] (see algorithm 3.2).

Section 3.2. Benchmarks

Benchmarks are the key point for the evaluation of a new architecture or technique. They are able to obtain a objective quantitative value for a established evaluation. Benchmarks results can be used to compare many solutions in order to provide an objective criteria. We have selected a set of well known benchmarks in order to evaluate the objectives of this thesis.

Benchmarks are designed to evaluate some specified characteristics and only on some environments. Each specific benchmark has one target evaluation and environment. Choosing the right benchmark is key the evaluate or results and progress in the right direction. In this section we will present benchmarks used for this thesis. For each benchmark we present its definition, its evaluation objective and target architectures. We also show some benchmarks characteristics which might impact on our evaluations.

NASA Advanced Computing Parallel Benchmarks. *NASA Advanced Computing Parallel Benchmarks (NPB [39]) are a set of small programs designed to evaluate the performance of parallel supercomputers.* This benchmark measures performance of supercomputers by measuring the execution time of 5 characteristic programs. It is also provided a serial version for each parallel program in order to compute the speedup, relation between serial execution time and parallel execution time. Benchmarks programs are derived from computational fluid dynamics (CFD) which are computational intensive, original applications for these programs are weather forecast, Monte Carlo applications, and many others. These programs are originally implemented in Fortran or C, and have available many classes in order to scale to many sizes. Nowadays there are many flavours and versions of NPB covering many architectures and programming models.

Programs from NPB are: multigrid (MG) benchmark, conjugate gradient (CG) benchmark, 3-D FFT PDE (FT) benchmark, integer sort (IS) benchmark, and simulated CFD applications lower-upper diagonal LU benchmark, scalar pentadiagonal (SP) benchmark, and block tridiagonal (BT) benchmark. For each benchmark program there are 6 data size available: Class A, Class B, Class C and Class D (from smaller to bigger) and two extra classes for very small data sizes Class S and Class W.

Benchmark programs are designed as a kernel that operates over a data iteratively until it reaches a result. The execution starts with data initialization, and one kernel program kernel invocation. These steps are not timed and the first kernel invocation is not timed in order to warm up system data caches. After these steps the program benchmark starts the clock, invokes the kernel for hundreds of iterations (benchmark program dependant) and stops the clock. The final step is to evaluate the correctness of the result and display the execution time. This final evaluation is resumed as two

messages "SUCCESSFUL" and "UNSUCCSESSFUL". Only successful execution times can be considered. An unsucessful appears when the evaluated environment breaks the program (as an example a too aggressive optimisation), in this case then execution time is not valid. The correctness of result also considers that computer real numbers operations order are not exchangeable, but as most parallel implementations change the order, a tolerance error factor in results is implemented.

NPB benchmarks versions used on this thesis are: NPB 3.0 for OpenMP [40] and NPB multizone (NPB-MZ) [41]. NPB 3.0 was designed for OpenMP. It uses a serial version from NPB 2.3 [42] for serial version programs and parallel version programs. In addition parallel version programs add OpenMP directives to implement parallelism. All benchmark programs, but IS, are in Fortran. Characteristics of each benchmark program can be found on table 3.1.

Table 3.1: NPB 3.0 [40] characteristics.

	Size (num. iterations)	
Benchmark	Class S	Class W
LU	12 (50)	33 (300)
MG	32 (4)	64 (40)
SP	12 (100)	36 (400)
BT	12 (60)	24 (200)
CG	7000 (15)	14000 (15)

NPB-MZ [41] is focused on CFD programs: SP, BT, and LU. It provides a new set of programs called SP-MZ, BT-MZ, and LU-MZ. These applications are inspired from programs targeted to the MPI programming model, but implemented with OpenMP. This version, instead of parallelising the whole kernel at once, it splits the computational space into multiple zones (see figure 3.1). Each iteration first computes each zone independently, and then, it exchanges values from boundaries in order to synchronize results between zones. This benchmark is designed to exploit multiple levels of parallelism, in this case we have two levels of parallelism: one coarse-grain parallelism between zones, and other with fine-grain parallelism inside each zone. First parallelism level is to compute each zone in parallel, the underlying idea is to assign each zone to a set of close processors or threads in order to increase performance of memory hierarchy. Second level is inside each zone, in this level the

previous assigned set of threads compute, in parallel, the same zone. This level of parallelism have the same behaviour than NPB 3.0 for OpenMP [40] but replicated for each zone. SP-MZ benchmark zones characteristics can be found on table 3.2, it presents a very regular mesh of zones. BT-MZ and LU-MZ presents almost the same characteristics. We focus on BT-MZ. BT-MZ zones characteristics for Class W and Class A can be found on tables 3.3 and 3.4 respectively. The main challenge that they present is that zones are not balanced: each zone has a different size and as a consequence require a different computing power. This situation creates an unbalance between requirements from each zone that must be solved by the correct assignation of resources for each zone. As an example, Class A zone 16 is almost 20 times larger than Class A zone 1, as a consequence it should require 20 more resources.

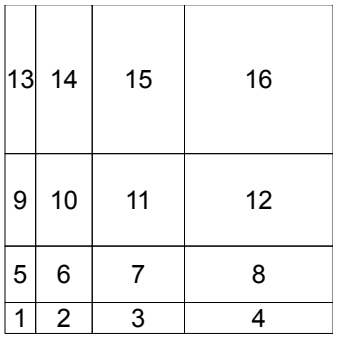


Figure 3.1: NPB-MZ [41] BT-MZ visual mesh of zones.

Table 3.2: NPB-MZ [41] program SP-MZ Class W zone characteristics.

zone	x-dim	y-dim	z-dim	elements
1-16	16	16	8	2048
Total				32768

Table 3.3: NPB-MZ [41] BT-MZ Class W zone characteristics.

zone	x-dim	y-dim	z-dim	elements
1	6	6	8	288
2	11	6	8	528
3	18	6	8	864
4	29	6	8	1392
5	6	11	8	528
6	11	11	8	968
7	18	11	8	1584

zone	x-dim	y-dim	z-dim	elements
8	29	11	8	2552
9	6	18	8	864
10	11	18	8	1583
11	18	18	8	2592
12	29	18	8	4176
13	6	29	8	1392
14	11	29	8	2552
15	18	29	8	4176
16	29	29	8	6728
Total				32768

Table 3.4: NPB-MZ [41] BT-MZ Class A zone characteristics

zone	x-dim	y-dim	z-dim	elements
1	13	13	16	2704
2	21	13	16	4368
3	36	13	16	7488
4	58	13	16	12064
5	13	21	16	4368
6	21	21	16	7056
7	36	21	16	12096
8	58	21	16	19488
9	13	36	16	7488
10	21	26	16	8736
11	36	26	16	14976
12	58	26	16	24128
13	13	58	16	12064
14	21	58	16	19488
15	36	58	16	33408
16	58	58	16	53824
Total				243744

Synthetics: memory copy and matrix multiplication. *These two synthetics benchmarks are used in order to stablish environment memory access limitations.* Both measure the time required to perform an operation given a data size and a number of repetitions, they can expose some characteristics of the underling architecture. Memory copy benchmark is an intensive benchmark which only copies a region of memory to another. This benchmark exposes bandwidth limitations between memory and processors. Matrix multiplication (see algorithm 3.5) is a common task inside many algorithms and programs. Its main characteristic is the k iteration over matrix B: each

iteration access to a different row. As a consequence, memory access pattern for B is not linear so cache can not hide successfully memory latency (low data locality). Matrix multiplication helps to determine how an architecture is able to work with complex access patterns to memory as an opposition to memory copy benchmarks. Some architectures present *B* matrix transposed, so all memory accesses are consecutive as it is on A matrix. In this case *B* matrix transposition must be taken into account in benchmark computation.

Algorithm 3.5: Matrix multiplication algorithm.

```
for 1 ≤ a , i ≤ N do
   C[a][i] = 0
   for 1 ≤ k ≤ N do
     C[a][i] = C[a][i] + A[a][k] * B[k][i]
   end for
end for
```

Synthetics: tolower, wordhash. *These synthetic benchmarks compute over an input stream of data and produce an output stream, they target is to evaluate the ability to parallelise through a stream of data.* Target architectures are usually embedded and streaming oriented architectures. These benchmarks have in common an input stream data. Each one reads an input stream and performs multiple transformations step by step. As a result it obtains another stream of data. A parallelisation of this benchmarks may take advantage of data and function parallelism. Function parallelism will consist in the computation in parallel of all steps but with a different input each time.

Algorithm 3.6: Tolower algorithm.

```
while read(&a) do
   if 'A' ≤ a ≤ 'Z' then
     a = a - 'A' + 'a'
   end if
   write(a)
end while
```

Tolower algorithm is shown at algorithm Error: Reference source not found and the data stream is at figure 3.2. It just reads one element, evaluates if it is lowercase, and transforms into lowercase if not. The output is the lowercase string. This

benchmark is a minimum program of a stream processing: one consumer step, one transformation step, one producer step.

Figure 3.2: Tolower stream graph.

Wordhash benchrmark algorithm can be found at 3.7, and stream processing graph at 3.3. This example uses complex data structures: word (as a collection of characters). It first computes the lowercase (as the previous synthetic benchmark), next it collects a word, and then computes a complex hash functions. At last it writes the output value. This benchmark tries to show the ability to collect multiple elements into a single one, split output from one step to two parallel steps, and finally collect two elements from two steps into the final result. This benchmark produces less elements than consumed.

Algorithm 3.7: Wordhash algorithm.

```
while read(&a) do
  tolower(&a)
  word[n] = a; n = n + 1; if a = EOF then n = 0
    v = hashA(word)
    w = hashB(word)
    h = hash(v, w)
    write(h)
  end if
  write(a)
end while
```

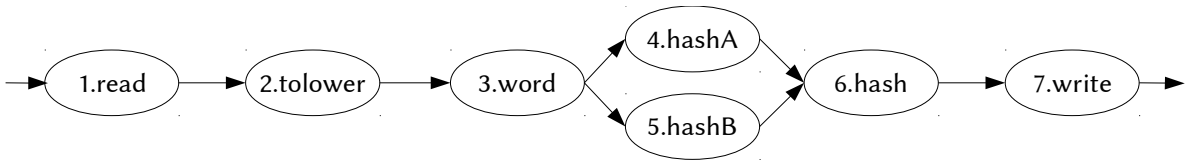


Figure 3.3: Wordhash stream graph.

FMradio benchmark. *This benchmark program decodes a raw signal from FMradio and computes the wave sound corresponding for a determinate channel.* As tolower and wordhash program benchmarks it is used to evaluate the ability of streaming

programs to take advantage of streaming architectures. FMradio benchmark is under GPL license. It comes from GNU Radio [43] application examples. FMradio initial structure is designed as a collection of filters (or application data processing steps) plus a runtime which creates a glue synchronizing and invoking each step as data is available. This benchmark is a pure streaming application with no serial equivalence available. Later, this kernel was extracted from GNU Radio by Marco Cornero from STMicroelectronics. In this extraction all dependences with GNUradio were removed. Extracted code was modified in order to be able to run in serial without the runtime, but it still require large structures representing data streams. Moreover, output varies depending of the internal data stream representation lengths. Later in this thesis it was modified in order to create a pure serial algorithm as wordhash or tolower which processes element by element.

FMradio filter structures is shown at figure 3.4 , the source is about 500 lines and it is available under GPL license. This benchmark is organized in ten transformation steps. Each step has a different cost and a correct load balancing is required on parallel environments. Steps from FM_QD_Demod to SubMultSq are executed 8 more times than other steps. The same computation (FFD) is reused on multiple steps, but with different configurations. The filter FFD is the most time consuming task. The parameter specified is the size of the sliding window. The larger is the sliding window, heavier is the computation, there are more elements to process. The FMradio benchmark presents unbalance between transformation steps, stream splitting, stream join, multiple consume ratios, and filter reuse.

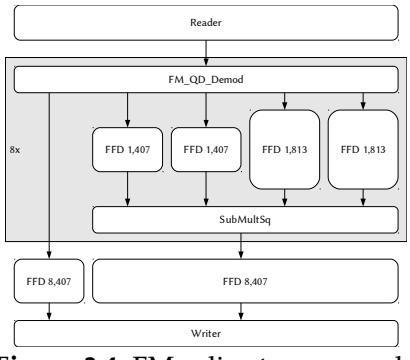


Figure 3.4: FMradio stream graph.

IEEE 802.11a benchmark. *This benchmark program decodes a raw radio signal of a Wifi IEEE 802.11a transmissions and computes the received data packet.* This benchmark is provided by Nokia as part of its collaboration in the European project ACOTES IST-034869. The code is not available to the scientific community and it is proprietary. This program benchmark is provided as standard C application, and unlike FMradio, as a simple serial application. It presents a structure more complex than FMradio, and it uses more signal control variables. We use this benchmark to evaluate the expressiveness of a programming model and its ability to scale.

StreamIt cookbook [44]. *Is a collection of a programs written as a collection of filters used to evaluate the expressiveness of a programming model.* StreamIt cookbook is written by the MIT university and contains near to a dozen different programs. Each program is presented as a graph of streaming filters connected together to compute one result. Applications used on this thesis are echo, fft filter, equalizer, and others. As FMradio StreamIt cookbook programs are provided as a pure streaming applications, we have converted these programs into serial programs.

Section 3.3. Architectures

Architectures are the underlying hardware which is responsible of the execution of a program. Program behaviour and execution are tied to the architecture. The same program on different architecture can have different behaviour and execution time. Programs can be rewritten or changed to take advantage of underlying architectures. By choosing right architectures and appropriate programming techniques programs can speedup its execution time in many orders.

We present architectures used on this thesis and their characteristics. These architectures are used to perform benchmark evaluations. They are also modified in order to obtain a better performance as a part of current reseach. We focus on those characteristics which has a direct impact on our evaluation. As the range of characteristics is very large we focus on multi-core chips with multiple cores and multiple threads and in its memory hierarchy.

IBM BlueGene/Cyclops [45-47] architecture. *This is a simulated architecture of a configurable processor with 32 cores, 4 threads per core, and main memory on chip plus a configurable cache per core.* This architecture was presented by IBM at 2002 as a part of the BlueGene supercomputer project. Cyclops was a prototype for a supercomputer on a chip as a processor for the whole system.

The objective of BlueGene project is to create a supercomputer powerful enough to stand between the 5 first positions of the Top500 [48] best supercomputers of the world. The supercomputer is designed as a large array of computers connected by a specific hyper-toroidal network. Each computer, or node, from the supercomputer is designed to have low power consumption, and prepared to be replaced by newer processor generations. First working BlueGene supercomputer had more than 100 thousand nodes of BlueGene/L processor, a low power two core processor.

BlueGene/Cyclops was conceived as an architecture for the execution of applications with high degrees of parallelism. It is achieved by integrating on the same chip several thread units and sharing memories with a small latency. This strategy is currently used by other architectures also used on this thesis as CUDA [49].

BlueGene/Cyclops was defined before the standardisation of multi-core nomenclature. All related work and publications of this thesis name BlueGene/Cyclops as a massive multi-threading architecture (one of the first prototype of a general purpose processor with more than 100 threads). Cores are named as a quads or thread groups. Hardware threads are also named as thread units.

Figure 3.5 shows the default configuration of the BlueGene/Cyclops processor. As default configuration it has 32 cores, 4 threads per core, one data cache per core, one floating point unit per core and one instruction cache for each pair of cores. It also have an embedded DRAM memory a special hardware for communications and off-chip memory.

Figure 3.6 shows the memory hierarchy of the Cyclops processor architecture. The main difference between current multi-core architectures and Cyclops architecture is that each core can access to data stored on all other caches. Current computers cores can access only to its own memory, a protocol of memory coherence or converting caches into core private memory is used to overcome this limitation. Cyclops allows

to use data caches from other cores but with a latency penalty: accesses to local caches are faster than accesses to remote caches.

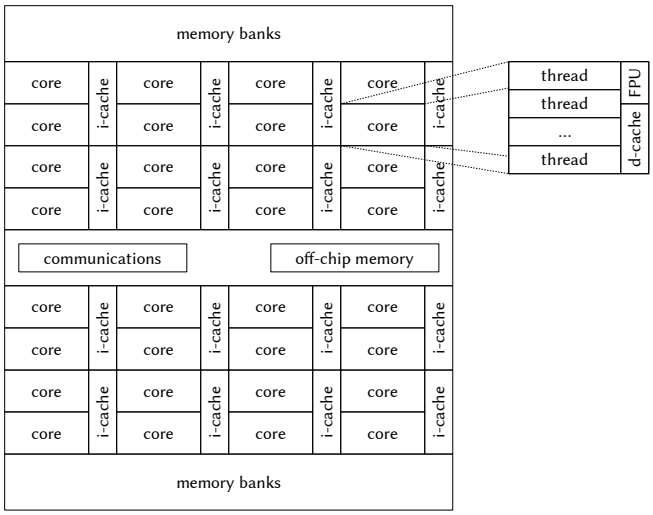


Figure 3.5: Overview of the BlueGene/Cyclops processor architecture.

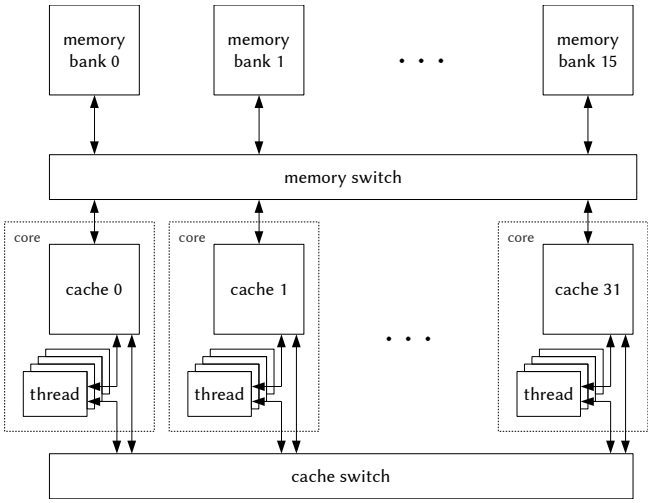


Figure 3.6: Overview of the BlueGene/Cyclops memory hierarchy.

Cyclops have no global directory for all data stored on processor caches. It uses the data access address to localize the cache responsible for the storing of its data. It has two address modes, one for private memory, and other for shared memory. Private memory is used when a special range of the effective memory addresses is accessed. When a data is requested by any thread of one core it is always stored on its local cache. Private memory allows duplicate data on data-caches under the responsibility of the programmer. Shared memory mode is used otherwise.

Hardware ensures that each main memory address maps directly to a single data cache. Selected data cache is determined by the memory address, hardware uses a static scrambling bijective function to map memory addresses to data caches. This mechanism allows to use all caches as a single very large cache. Hardware determines the destination cache as follows: bits 0 to 5 of the effective address determines the byte inside the line cache, bits 6 to 10 determines the line among all data cache lines and bits 11 to 15 determines the data cache who owns the line.

As BlueGene/Cyclops was a prototype, we present on table 3.5 the configuration that we have used. The configuration presented is the default configuration but we have increased the available memory to 256MB in order to run benchmarks. Table 3.5 (a) presents the memory latencies for each kind of access and (b) shows the exact count of elements, sizes, and configurations for simulator components.

Table 3.5: BlueGene/Cyclops prototype configuration used on this thesis.

(a) memory latency

Memory access type	Latency
Local cache hit	6
Local cache miss	24
Remote cache hit	17
Remote cache miss	36

(b) components

Component	# units	Params./unit
Cores	32	4 threads, 1FPU, 1 data cache
Threads	128	single issue, in order, 500 MHz
FPU	32	1 add, 1 multiply, 1 div/sqrt
Data-cache	32	16 KB, 8 way assoc., 64-byte line
Instr.-cache	16	32 KB, 8 way assoc., 32-byte line
Mem. banks	32	8MB each, total 256 MB

BlueGene/Cyclops architecture provides a full environment support for developers. This environment is based on Linux Red Hat 7. The most important tool is the simulator of the architecture, evaluated and verified at [46]. The simulator is provided with source code under a privative license, it is ready to be modified or adapted in order to change configurations. This simulator is also ready to measure almost any performance metric, as for example kinds and distributions of cache accesses. It is also provided a cross-compiler tools based on GNU toolkit. These tools provides a gcc 2.95.3 and a binutils 2.11.2 retargeted for the BlueGene/Cyclops. In addition to these tools a raw debugger is provided able to debug a program running inside the simulator at any of the threads by following assembler instructions. The binutil addr2line or objdump -DS proves to be very useful on this condition. Finally, it is also provided a library for OS emulation. This library is linked to the program to run inside the simulator and provides a basic OS routines. As the BlueGene/Cyclops

is conceived to work inside a large network of thousands of processors (as [48]), the OS library reserves one thread for the execution of network communication routines. As a consequence, there is a maximum of 127 threads available for the programmer on an environment with a configuration of a total of 128 threads.

Kandake cluster architecture. *Kandake is a cluster of common market computers connected through a high performance Myrinet network.* Kandake is a supercomputer designed research on distributed systems. It has 8 computing nodes. Each node has two processors at 266MHz and has available 128MB memory. Memory is not shared between nodes, each node can only access to its own memory. Nodes can be communicated explicitly through a high performance myrinet network. This network provides a very small latency and a large bandwidth.

One typical requirement of parallel programs is the necessity to communicate partial results between execution units. Communications on embarrassing parallel programs are usually negligible, unfortunately, most of parallel programs requires an efficient communication network in order to obtain partial results. Programs explicitly wait for data from their parallel execution units. While partial results production is determined by the processor speed, wait synchronizations are determined by the network speed. If the processor is very fast, proportionally to the network, waiting for data can have a large impact. A good relationship is able to speedup programs with a high degree of dependence in parallel programming.

This architecture is very specific: processors are small, but the network is fast. This feature provides a reasonable relationship between processor performance and network capabilities (clusters, in contrast, usually have a very poor processor-network relationship). Multi-core architectures usually shares this good relationship: processors are fast, but networks between cores are built inside the same chip, so they are also very fast. Kandake architecture is suitable to simulate multi-core distributed architectures, it just scales size and time scale.

Cell Broadband Engine [31]. *Cell processor is an initiative of Sony, Toshiba and IBM in order to create an heterogeneous multi-core processor suitable for gaming and video processing.* Cell Boradband Engine is not one processor but a specification of multi-core heterogeneous processor. It describes kinds of processors, memory hierarchy, internal processor networks and external connections. The original idea is to provide

a framework able to design any kind of multi-core processor just selecting convenient cores for each task. Cell processors can be composed in order to obtain a specific processor for a specific program. This capacity of specialisation is the main cause of a high degree of proliferation papers about specialized cores.

The first implementation of this processor was presented as the main processor of the Sony PlayStation3. Its initial concept was to develop a processor with a main generic core to act as a controller and 8 more vector processors known as synergistic processor elements. This processor was able to achieve performance peaks ten times better than other processors developed at the same year. As a counterpart it required a tedious and careful programming and not all applications are suitable for this processor. In fact, this is not a surprising due to the target program was gaming and video.

The second noticeable implementation was presented as the main processor of the Microsoft Xbox 360 [50]. Instead of having one general purpose processor and 8 vector processor, it has three general purpose processors and no vector processor. This architecture assumes that main computations will performed on the main processor, and graphics computations are performed in a separated graphics processor unit. This architecture shows the versatility of the Cell Broadband Engine specification capabilities.

The overview of the cell processor is presented at figure 3.7. All components are connected through the EIB (element interconnection bus), the main network of the processor. SPEs, PPE (L2 and PPU see figure 3.8) and memory and input/output interface are connected to the EIB. Main processor core is the PPE, which is a generic processor. SPEs are vector processor cores for specialized computations. Memory and i/o interfaces are designed as a bridge for off-chip communications.

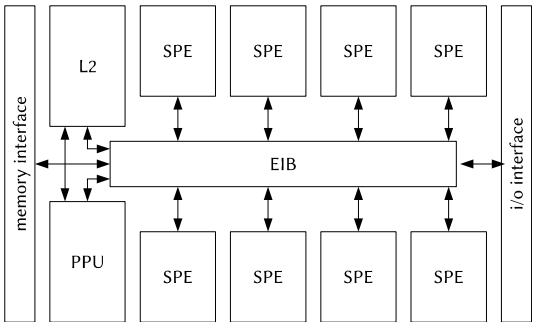


Figure 3.7: Cell B.E. Sony Playstation3 processor implementation block diagram.

EIB (element interconnection bus) is indeed the most important element in the processor. It is focused on streaming processing. EIB interconnects all elements in a four-ring segmented buses. Each ring has one direction and can be used simultaneously for multiple communications, if they do not overlap on the same cycle. Each element before using the bus issues a command, a bus arbiter decides how to perform communications and how to extract the maximum bandwidth. The maximum bandwidth of the whole EIB is 96B/cycle. Each Cell B.E. element has a connection to the EIB. These connections have a maximum bandwidth of 16B/cycle, including memory. There is a possible configuration which is to use memory and i/o interfaces simultaneously to access memory, in this case, the maximum bandwidth with memory is of 32B/cycle, 3 times slower than maximum EIB bandwidth. If we consider that data has been consuming from L2 and memory, we have a maximum of 48B/cycle, 2 times slower than maximum EIB bandwidth. As a consequence we deduce that cell implementation architects has assumed that the full bandwidth of the EIB is achieved in internal communications. This kind of communications are suitable for stream processing where results from one processor are sent directly to another processor, without using main memory. In other words, maximum bandwidth can only be achieved by direct data transfers between processors.

PPE (power processor element) is a generic core processor. The PPE is a simplified PowerPC processor in order to die (silicon chip) size restrictions. It is a 64-bit in order processor, with two execution threads. The L2 is a 512 KiB 64 bytes per line cache. The PPU L1 cache has 64 KiB, 32 KiB for data cache and 32 KiB for instruction cache. The PXU (processor execution unit) has a FPU (floating point unit) shared between both execution threads. The PXU provides of AltiVec [51] vectorial instructions extensions and one vectorial processor unit. Without vectorial instructions, at 3.2GHz, the PPE can achieve up to 6.4 GFLOPS of double precision. With vectorial instructions, at the same 3.2GHz, the PPE can achieve up to 25.6GFLOPS of single precision. This is very important to take account, there is some literature comparing PPE against SPE, but, although SPE are only vectorial, they compare non-optimized scalar PPE results against optimized vectorial SPE results. Moreover, the PPE is less sophisticated than other processors of the same period, so its performance is not as good as other architectures, and this must be also considered on benchmarks.

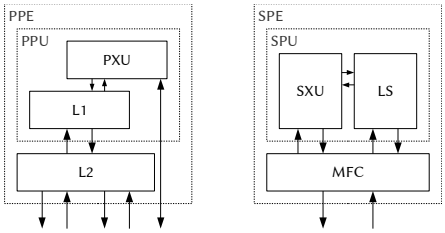


Figure 3.8: Cell B.E. PPE (main processor) and SPE (auxiliary vector processors) blocks diagrams.

SPE (synergistic processor element), also called accelerator, is a specialized vector processor. It is a simple processor of one single thread with in order execution. The SXU has a 128 bank register, register size is 128 bits. SXU instruction set is different from the PXU and only provides vector instructions. As the PXU AltiVec it can achieve, at 3.2GHz, 25.6 GFLOPS of single precision floating point. SXU has no direct access to main memory, it only can access to local storage (LS). Access to LS are performed in a modulo addressing mode, so all addresses are valid. Only lower bits of the LS access address are used, an MMU is not required and there are not any invalid memory access. Local storage is a small memory and fast. It is usually defined as a self managed cache. LS size is 256KiB, this space is shared between instruction and code. Data can be copied from/to main memory to/from the LS through the MFC, a DMA controller. Copies are asynchronous and the SXU can query the MFC about the status of the transfers.

MFC (memory flow controller) is a very sophisticated DMA controller. It is fully programmable and some times is considered as the third kind of core inside the Cell. MFC is programmed by the SXU, but also by the PXU. MFC exposes a set of registers to the virtual address space in order to be programmed by any other device. MFC also exposes state registers to the SXU from its SPE. MFC is designed to do more than one memory copy simultaneously. SXU or PXU can program one or more data transfers based on effective addresses. Each of these transfers can copy a data from main memory to the local storage, from the local storage to main memory, but also from one local storage to another, or even to any valid effective address. The MFC is able to overlap many data transfer commends, reschedule, in order to take maximum advantage from EIB. It also provides of commands to tag transfers, stablish barriers and fences, and query the status of a transfer. Experimentally has been shown that MFC memory transfers bust be greater than 1 KiB in order to use the maximum EIB bandwidth [46].

Cell software-development-kit (SDK), operating system, and environment for research consist on a Linux Fedora Core 4. In addition, IBM has also provided MAMBO [52], a full-system cell-simulator. The operating system provides a specialized library called libSPE in order to provide a framework to use the SPEs. OS and applications are executed on the PPE, as a common PowerPC processor. When an application requires to use any SPE, it uses the libSPE in order to launch kernels into any SPE. The OS provides an abstraction layer in order allow manage multiple applications using SPE, although the SPE change of context is expensive.

As SPE and PPU does not share the same instruction set architecture (ISA) it is required to compile two different binaries for an application. It is provided a full compilation environment and tool kit from GNU for a PowerPC; it is also provided a cross-compiler version in order to create binaries of the SPE. The program is composed by two binaries. Both binaries can be combined in a single file, or provided as many files. If binaries are combined in a single file, the file is a PowerPC binary (for the PPU) which stores SPE binaries as global data. LibSPE is responsible to load SPE binaries, either from independent files, or from program global data.

BladeServer JS21 blade computing node. *We have used single Marenostrum supercomputer nodes in order to perform some tests and benchmarks.* BladeServer JS21 is a computer with two processors IBM PowerPC 970MP sharing memory. Each processor is 64 bits and dual core. Each core has 1MB of L2 cache. The processor speed is 2.3GHz, and the blade has 8GiB of memory. The operating system is SuSe Linux 9 from Novell. Main memory is shared with memory coherence protocols among all cores of the architecture.

Sur computer architecture. *Sur is a dual processor computer based on Power5 [53].* Sur computer haves two Power 5 processor. Power 5 is a RISC multi-core multi-threading 64 bits processor. It is composed by two cores, each core has two execution threads. L1 4-way set associative d-cache and 2-way set associative i-cache for each core. In core shared L2 cache 1.9 MB, 10-way set associative. L3 directory is on-chip in order to speedup accesses to L2. Main memory is shared with memory coherence protocols among all cores of the architecture.

CUDA enabled NVIDIA GPGPU + Intel CPU desktop computer architecture. *This architecture is present on most desktop computers. It consist of a standard Intel compatible*

CPU and a graphical processor. The graphical processor is able to execute generic programs. Nowadays most desktop computers are composed by a generic processor (CPU) and by a graphics processing unit (GPU, see figure 3.9). We are interested on this architecture because is close to supercomputers, in fact, current GPU have thousands of hardware threads running simultaneously. Both, CPU and GPU are connected to the main memory. Although some GPU have its own memory, both CPU and GPU have access to main memory. Access to memory is restricted by latency, but also by the bandwidth. Data bus bandwidth defines the maximum data that can be transferred from/to main memory to/from any processor.

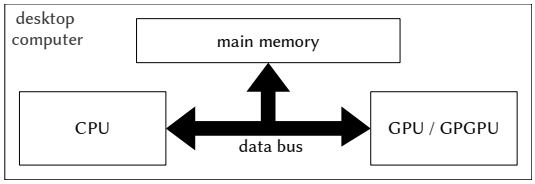


Figure 3.9: Current desktop computer overview.

Although for many years desktop computer CPUs were simple processors, current CPUs are composed by multiple cores and a complex cache hierarchy (see figure 3.10). Each core can have one or more physical threads and its own memory cache. A memory-coherency protocol is implemented in order to provide the same memory space for all threads.

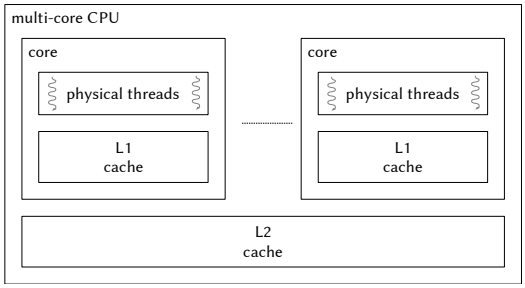


Figure 3.10: Overview of a desktop computer multi-core CPU.

Current desktop computers GPUs are in fact a General Purpose GPUs (GPGPU). GPGPU, in contrast to simple GPU, are capable to execute an arbitrary code and they are not limited to graphical processes. Current GPGPU are specialised to intensive computations, mainly addressed to graphic tasks. They are able to execute simple functions usually called kernels. GPGPUs are massively multi-threaded architectures, they have thousands of physical threads.

NVIDIA GPGPUs are composed by several multiprocessors (see figure 3.11), each one has multiple cores and a shared memory. Cores are processing units that compute thread instructions. Shared memory can have multiple data accesses simultaneously. Shared memory size is small, but it has a very low latency.

NVIDIA GPGPUs characteristics are organized by CUDA Compute Capabilities. We have focused on a low power GPGPU with CUDA Compute Capability 1.1. This architecture assumes that there are 8196 threads per multiprocessor, 96 threads per core, and shared memory size is 16384 bytes.

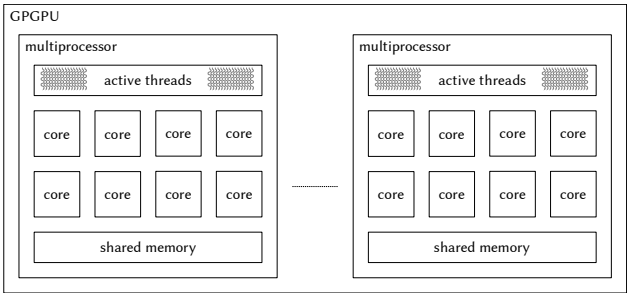


Figure 3.11: NVIDIA GPGPU architecture overview.

NVIDIA GPGPUs are massively multi-threaded architectures. The idea underlying of this architecture is to hide memory latency by executing many threads. In addition, memory is organized as a memory banks and usually optimized to fetch a row of data with consecutive addresses. As each thread executes a scalar operation, NVIDIA introduced the concept of coalescence. Threads are ordered, as memory is ordered. Coalescence ensures that consecutive threads accessing consecutive memory addresses will take advantage of memory organization.

Almost any CPU have vector-instructions, NVIDIA GPGPU is close to a vector processor but does not have visible vector-instructions. Vector programming is very complex and forces to organize execution and memory in basis of multiples of vector size. NVIDIA GPGPU can be said that operates in vectors, but, instead having instructions to operate with many data simultaneously, it defines multiple threads. For each vector element there is one thread. All threads of the same vector must execute the same program, but they are not forced to perform all operations. They are just suggested to do so. Threads computing over the same vector are called warps. These warps (32 threads, or even half warps, 16 threads) are supposed to be

executed together. If one thread of a warp diverges from the execution, the performance is affected, but the behaviour is correct.

Table 3.6 discuss our target architectures details. ViewSonic is the architecture of a nettop, a computer designed to be part of a home cinema. It is a very small computer and has a very low power consumption. MacBook Air is corresponding to the model released by Apple on October 20 of 2010. It is a light laptop computer with low power consumption. GA-965P-DS4 is a desktop computer of 2008 targeted for gaming. We have not executed benchmark on its CPU processor due to we had limited access. NOX is a desktop computer targeted for intensive computations. It has a processor with many threads, and CUDA Compute Capability 2.1 on the GPGPU.

Table 3.6: Intel + NVIDIA GPGPU desktop computer architectures used on this thesis.

Computer	Proc / GPGPU	GHz	Power	Cores	Threads	Bandwidth
ViewSonic	Intel Atom 330	1.6	8W	2	4	5 GB/s
ViewSonic	NVIDIA 9400	1.1	10W	16	1536	5 GB/s
MacBook Air	Intel Core 2 Duo	0.97	35W	2	2	10 GB/s
MacBook Air	NVIDIA 320M	2.16	14W	48	4608	10 GB/s
GA-965P-DS4	NVIDIA 8800GT	1.65	> 50W	96	10752	53 GB/s
NOX	Intel i7 950	3.0	130W	4	8	21 GB/s
NOX	NVIDIA GT 430	1.4	49W	96	3072	21 GB/s

Section 3.4. Tools

Tools are software artefacts, or just specifications implemented in some way, which helps as a fulcrum of any research. Our research have relied on some of them, which give us the means to progress with a good environment. We have prioritized those tools which are open source, or at least we had access to its sources. Some presented tools have been modified in order to adapt them to our research, but at the same time, we have contributed with them to the scientific community.

Tools used in this thesis are: compilers, runtimes, simulators and tracing utilities. As compilers we understand from simple C or Fortran compilers to full

programming models. Runtimes are auxiliary libraries which help to implement some programming model. Runtimes help to spawn (create) parallelism, synchronize the execution flow, or even to communicate data. We use, and later contribute, hardware simulators. Simulators provides us the possibility to adapt hardware and research for better programming models beyond existing hardware limitations. Tracing utilities are usually known as profilers, but a profiler is the most simple form of a tracing utility. Profilers are designed to obtain general statistics in order to know which parts of a program needs to be improved, and some related information. Tracing utilities goes beyond this point, they are designed to record execution in a trace of events and posterior analysis. They allow to visualise the execution and gives a better understanding of real problems and hazards.

We present a set of tools that we have used on this thesis. We have ordered them by kinds of research and kinds of tools. We start with tools used for homogeneous shared memory multi-core, then with homogeneous distributed memory, simulator, heterogeneous distributed streaming, and finally tools used for pattern-matching algorithms.

OpenMP [54]. *OpenMP is a parallel programming model designed to parallelise C, C++ and Fortran programs on homogeneous multiprocessor architectures.* It is uses the same parallelism than a user threads library, but, it frees the programmer to use a low level library. OpenMP allows programmers to add few directives to enable parallelism. Directives are like comments, they report to the compiler that some parallelisations can be performed without breaking the application semantics. Compiler generates low level thread library calls automatically from existing directives enabling parallelism.

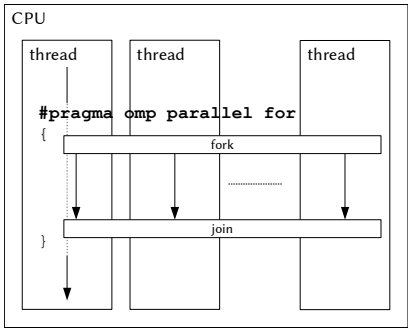


Figure 3.12: OpenMP fork/join thread execution model.

OpenMP thread execution model is fork/join (see figure 3.12). Each time that the serial execution reaches a parallel annotation, OpenMP splits the execution into multiple parallel threads. When a parallel execution reaches the end of a parallel annotated region, it waits for all threads and resumes a serial execution.

OpenMP has many directives and clauses. Each of one has a specific semantic and is suitable for many structures. Although, from all directives, there is one directive construct which is usually the first step and the most useful: "*parallel for*". This directive helps to create a basic parallel region from iteration constructs. It also has the optative clause reduction, which helps to summarize a computation with many elements into a single element. Thus, from all OpenMP directives we emphasise on the following OpenMP directive:

#pragma omp parallel for [**reduction(**OP:*r*)]
for $i_0 \leq i \leq i_f$ **do**
  ··· for body
end for

Parallel for directive creates a parallel region and distribute the following for construct iterations across multiple threads. Each thread executes the for body given a subset of *i* iterations. All *i* iterations are executed once and only once by all threads. Optionally a reduction operation can be performed: it summarises a single value *r* as an operation OP over a large set of values corresponding for each *i* (an example of reduction is a summation).

OpenMP NanosCompiler [55] based on Parafrase-2 [56]. *OpenMP NanosCompiler is a source to source compiler. It compiles (translates) Fortran programs with OpenMP annotations into a Fortran programs with calls to an OpenMP runtime.* A source to source compiler is a compiler able to transform one program into another. Even nowadays, where compilers are able to compile by default OpenMP, they are difficult to modify or adapt for research terms.

Any OpenMP compiler basically translates directives into a code which explicitily creates the parallelism and makes any runtime library calls required. The OpenMP NanosCompiler transforms source code to a code with runtime calls, after this transformation it generates a new Fortran code but with code transformed in order

to implement OpenMP directives. There are three advantages: generated code can be modified, runtime library can be changed, and there is not a target architecture.

OpenMP NanosCompiler generated code is basically the original code reorganized plus runtime library calls. The OpenMP NanosCompiler processes all parallel directives and extracts code within parallel regions. For each region it creates one or more function containing the original code. These functions have as parameters the variables and values required for the correct execution of the extracted code. Original code and parallel OpenMP directives are removed and replaced by runtime library calls in order to implement the required behaviour.

Even if we have no access to the compiler source, source generated by OpenMP NanosCompiler is plain Fortran code. Generated code can be modified or adapted manually by the researcher. This possibility allows us to adapt or change the resulting code in order to do further changes or tests. This possibility allows us to change or introduce concepts in the programming model.

Resulting transformed code is compiled to a binary by the native compiler. In this process, the source is linked to a runtime library. This library is responsible of many parallelism decisions, such as scheduling, lock policy, … We can link against any runtime which satisfies interfaces assumed by code transformation. As a consequence we can select or adapt a runtime able to make desired experiments.

OpenMP NanosCompiler just transforms source to source, thus it requires a native binary compiler. The generated source is just Fortran, no additions and no OpenMP directives are present. This resulting source can be compiled again with any compiler, so it can be compiled to any target. This makes the result of this compiler as portable as the original program and the runtime library.

NthLib user-level thread library [57]. *NthLib is a portable thread library used as a OpenMP runtime for the OpenMP NanosCompiler.* NthLib is based on nano-threads library, and it is ported to several platforms, including BlueGene/Cyclops, Linux/DSM, Linux/Pentium, Linux/IA64, IRIX/MIPS, AIX/POWER and SPARC/Solaris. NthLib supports C and Fortran. It also has support for multiple levels of parallelism and for some experimental OpenMP clauses as *groups* [58].

We have used NthLib because we had access to the sources, it was ported to our target architectures, it is compatible with OpenMP NanosCompiler, and because it has suport for nested parallelism.

Nested parallelism is one of the key points of this thesis. We mix two levels of parallelism with two grains of parallelism in order to obtain better locality and achieve better results. In order to have efficient nested parallelism, the NthLib has available two kinds of threads: nano-threads and work-descriptors. Nano-threads are full threads, as they can be pthreads. Nano-threads have their own stack, their own dependency list, and they require saving/restoring context in order to use them. They have the same flexibility than pthreads. On the one hand, it also implements work-descriptor [59]. Work-descriptors are a kind of list of task, each task is basically a function call with parameters. They are not really a thread library, but tasks can be scheduled and executed serially by any real thread. If there are more than one thread, we can execute multiple word-descriptors simultaneously. The main limitation of work-descriptors is that they have been executed on the same stack than his parent, and they can not block and switch to another work-descriptor (they have no context change).

OpenMP NanosCompiler is adapted to produce source with function calls to NthLib runtime using both kind of threads. An extra clause on the OpenMP Fortran source hints which kind of threads are used on source generation for each parallel construct. By default it uses nano-threads for all parallel constructs (by doing this OpenMP NanosCompiler ensures that the result will always work). We need nano-threads for grain parallelism and for outer levels of parallelism, so for those corresponding parallel constructs nano-threads are fine. For inner parallelism and fine grain parallelism nano-threads can imply a great overhead, so we select which parallel constructs can be converted into word-descriptors. We provide of additional compiler the compiler by adding the corresponding clause.

NthLib is a library with a high degree of versatility and very efficient, it implements some OpenMP clauses as GROUPS and also by implementing two kinds of threads it helps us to create parallel programs with a very good performance.

NthLib runtime for Cyclops [60]. *NthLib was initially ported as a prototype to BlueGene/Cyclops architecture with a reasonable performance.* The objective of the porting of the NthLib for cyclops was to made available the OpenMP programming model for BlueGene/Cyclops architecture. This environment was based on the OpenMP NanosCompiler, which is executed on a host machine (a intel laptop), and generated (transformed) code is compiled to Cyclops binary by the gcc cross-compiler toolset.

Benchmarking for scalability usually tests the same benchmark with a different number of threads. If we select to execute a benchmark with less user threads than available hardware threads, we might decide how to place them on the processor. Given that we have multiple threads per core (and FPU), it is reasonable to control how threads are mapped and which FPUs are shared.

NthLib for Cyclops has an extra scalar numeric parameter called stride. User threads and Cyclops hardware threads are numbered. If no stride is specified it maps user threads to the hardware thread with the same number. If stride is specified it assigns one user thread to a hardware thread every stride hardware threads. If stride mapping reaches the maximum number of threads, it starts again reusing free hardware threads. Cyclops threads are ordered by cores, so, if there is 4 threads per core, and 32 cores, threads 0, 1, 2 and 3 are mapped to core 0, threads 4, 5, 6 and 7 are mapped to core 1, and so on. As an example, with stride 1, threads 0, 1, 2, and 3 are mapped to core 0, 32, 64 and 96 are mapped to core 0. If there are less user threads than cores on the last configurations, there should be a maximum of one user thread per core.

Cyclops simulator [45]. *Cyclops simulator is the parametrizable simulator of the IBM BlueGene/Cyclops processor.* The processor architecture, simulator and development environment is described on the previous section.

Paraver [61]. *Paraver is a trace visualizer tool.* Its name comes from spanish "for see" and it was designed literally for this task. The underlying idea of Paraver is first to run the application with some kind of execution tracer, and then visualize the behaviour of the application at offline. Paraver was initially developed by the European Center for Parallelism of Barcelona (CEPBA) under the supervision of Dr. Jesús Labarta.

Paraver traces are text files describing the application behaviour. Each line describes a state, an event or a communication at given time stamping and thread identifier. These traces are usually generated by specialised libraries and then converted into Paraver.

State line describes a change of program state. Some of most common states are *idle*, *running* and *waiting* for communication. Running state describes that at given

timestamp a thread started computations. At idle state means that at given timestamp a thread begins to wait for some kind of synchronization. Waiting for communications also specified if waiting for input data or for output data. States are assumed to persist on the same thread until a new state is registered for the same thread.

Events are located at given timestamp and thread identifier. Events are described as a pair kind-value. Each event can report any kind of information and they can be user-defined. There are no restrictions for values, but some times value 0 is used in order to report that a previous event was finished. Events can report function calls and exits, cache misses, partial IPC, and any other relevant information.

Communications describes data transfers between two thread identifiers. It is composed by a communication channel, a data packet size, thread identifier for the sender, thread identifier for the receiver, and timestamps for synchronization. Time stamps are designed in order to have information about all synchronization involving a communication. It gives enough information to know if there is data contention on the sending, or if the receiver was waiting. This information is also combined with the state in order to reflect special states for idle in case of waiting for communications.

Paraver is a visual tool and all trace information is presented visually. It has many kinds of visualisations as statistics, 2D grid displays, or linear time views. Views are no tied to predefined values or event. Paraver uses mathematical functions in order to decide how to draw traces on visualisations. Some of these mathematical functions can be state value as is for linear time view (as first step to know when the application was running), event stack composite of function calls for linear time views (which uses value 0 in order to compute exits), or even a map of page faults of pairs of thread identifiers and memory regions for 2D views (for analyse memory behaviour).

All Paraver configurations and associated views and mathematical functions can be combined or even saved for future reuse. In fact Paraver provides some of more useful configurations.

libFASTparparaver. *This is a library designed to collect manually information from parallel programs for Paraver.* LibFASTparparaver is a user library which allows to collect information about the execution of a program. This library is optionally used

by the OpenMP NanosCompiler to generate a binary which produces a trace of its execution. This library contains function calls to determinate the state of the execution and add events. Although OpenMP NanosCompiler is able to generate function calls to the libFASTparparaver, this library is also available for the user to generate any trace required. This library has no special requirements but its functionality limited only to shared memory architecture and only is able to generate trace states and events.

MPI [62]. *Message passing interface (MPI) is a library interface to run programs on distributed memory clusters.* MPI is actually implemented on many architectures and almost any scientific application for clusters is written using it. MPI hides many architecture details (as node configurations, networks, …) and it has been proved to have a very good scalability [39].

MPI programming model is based on multiple copies of the same program running on every node (computer) of a cluster. Each node executes the same binary, but each instance has a different identifier. MPI assumes that there is no distributed memory and all information and result exchange must be done through library calls (passing messages). MPI library calls allows to exchange information but also synchronize. There are many MPI primitives, some examples are: point to point communication, broadcasting, reduction, and more.

Main difference between MPI and OpenMP is that MPI forces to rethink algorithms to work with distributed memory. The programmer must decide how the algorithm is split and how to synchronize data and control. As a reward, programmer can decide exactly how to parallelise and distribute it and consequently MPI can achieve a very good performance. As a counterpart, MPI applications are more difficult to write and debugging involves many nodes (computers), its states, and network state.

NanosDSM and OpenMP [63]. *Distributed shared memory (DSM) systems are libraries designed to emulate a shared memory architecture on distributed shared memories.* As OpenMP requires shared memory to work. A DSM can enable OpenMP on distributed memory clusters by fulfilling the requisite of shared memory. DSMs usually relies on page fault mechanism in order to emulate memory-coherency across nodes of a cluster. Using DSM over a cluster, it opens the opportunity to use

OpenMP and as a consequence easy programming and debugging of parallel applications.

We have worked with NanosDSM [63]. NanosDSM is considered to be an everything-shared DSM and it uses all Nanos infrastructure to provide an efficient solution. OpenMP NanosCompiler is adapted in order to generate special function calls for the NanosDSM runtime. NthLib is also modified to be in touch with NanosDSM. NanosDSM relays on the compiler and NthLib runtime to control the full architecture.

As MPI, NanosDSM us running on all nodes (machines) of the cluster. When a OpenMP physical thread is spawned, NanosDSM select a node to execute the corresponding threads and sends the function pointer and the stack pointer. Automatically at this point the thread starts to execute, even if the stack content or function code is not available on the node. If at any time the program is not able to execute it produces a page fault. NanosDSM captures page faults and solves it. As a result, NanosDSM is able to control all memory accesses and maintain memory coherence across machines of a cluster.

NanosDSM uses page faults to implement memory-coherency. Each memory logical page is owned by its master. The master controls the correct coherency of the page. A page can have two possible states: shared for reading or exclusive access for modification. If a program thread starts to read a page that is not present on its node, it causes a read page fault, NanosDSM captures it, it request to the master a copy for read, NanosDSM restores the content of the page, and atomically makes it available to the current node. If the page was previously acquired for an exclusive write, the master changes page protections to only read, and changes page state to shared read. If write is requested, it invalidates all other copies, notifies the new state of the page and changes page protections to read/write.

NanosDSM presents a very interesting architecture. It emulates an architecture with a cache line of one page and a high communication latency.

UNISIM [64]. *UNIted SIMulation environment (UNISIM) is a simulation infrastructure which helps to create modular simulators and reusable components.* Unisim simulator is designed to provide a real modular simulator. While other simulators like systemC

claims to be modular, but each configuration require specific control modules, Unisim is designed to distribute control as well as modules.

Unisim is conceived as a collection of modules connected by ports. Modules can be almost plain C++ classes or a composition of other modules. Both kind of modules have some special variables called ports and also can be parametrized. Unisim uses C++ templates to make modules parametrizable. C++ templates are used in order to define variable number of ports, variable number of subcomponents, … All components are connected by the Unisim runtime, which constructs the simulator instancing all classes and computes all required behaviour. Unisim also adds a special kind of port called clock. This port allows to synchronize all the simulator through a clock and construct pure sender modules.

Unisim has its own protocol to connect modules using input-output ports (shown at figure 3.13). Each port has a type (C++ type) and three signals: data, accept and enable. Unisim uses these three signals to distribute the control. Output ports from one module are connected to input ports of other module. To communicate to connected modules, the output port first sends the data, second if the input port receiver of the data can accept it sends back the signal accept as true, and third the sender module can enable or disable the data setting enable signal to true or false. Data is used to communicate the value of the connection, accept is designed to control data contention (for example, receiver may have the input buffers full), and finally the enable helps to build routers by sending the same data to multiple modules, but just enable to receive data to one module.

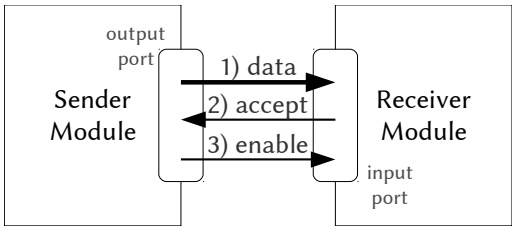


Figure 3.13: Unisim connection model based on ports and three signals.

Unisim is a cycle oriented simulator. Although Unisim can simulate combinational circuits thanks to the triple protocol. For each simulation cycle, all signals must be written once and only once. When a signal is written it fires a trigger on the receiving signal which executes a function to handle the signal. Each signal trigger is handled as a small independent process. This is a very flexible policy but

also make programming more difficult by distributing module behaviour across many functions.

Unisim runtime is responsible to activate all signal processes, including those signals activated by the Unisim clock signal. At the beginning of the cycle, Unisim runtime activates all rising edge processes connected to the clock. These processes may write the data value of output ports. When it finishes, some receiver data ports have been activated, and Unisim runtime executes all activated input data processes. As a consecuence more data ports can be written and activated as well as accept signals. Unisim runtime activates all pending data processes and also executes all activated accept processes. After this step enables can be activated, as well as more data and accepted signals. Unisim continues executing all actived signals until there is no more activations (ensured by the limitation of one change per cycle). After the last activation, actives all falling edges processes. These processes can active more signals, and once again Unisim runtime starts to execute all activated processes until the system stabilises. At the end, Unisim checks that all values have been activated.

Unisim is a very flexible and powerful simulator infrastructure. In addition, Unisim is distributed under an open source license GPLv2 and it has available a large list of modules able to built a simulator from scratch in short time.

LibSPE [65]. *The libSPE is the library provided by IBM for the Cell B.E. processor in order to execute and control processes at SPE accelerators (see figure 3.8).* As we have presented previously the Cell B.E. has two main kind of cores: a general-purpose core and accelerators. For the point of view of a program the architecture of the Cell B.E. is just a PowerPC: the OS is running at the PPE and it executes 64 bit PowerPC binary code. On the other hand, SPE are accelerators, they are not designed to execute whole applications or an OS, they just are designed to accelerate some processes. LibSPE is the bridge from a classical PowerPC to a Cell B.E.: libSPE allows to run processes or functions at SPE accelerators from the PPE core.

There are two versions of the libSPE. The first version SPE status and execution was managed by the library, the second version the SPE status must be executed and monitored by the program. As a short explanation of the second version, it requires that the programmer creates multiple threads in order to control each SPE accelerator status. We focus on the first version.

LibSPE is not much different from a threads library. Main functions are designed to load a function binary into memory (it differs from PowerPC binary), execute the function present with selected arguments, and wait for a result or query for an error. As other threads libraries, it also has synchronization primitives and communication primitives. Unlike thread libraries, libSPE has two sets of interfaces: one interface for the PPE and other interface for the SPEs. These two interfaces are designed for different roles, for example, the PPE is responsible of creating SPE processes.

LibSPE also has all required functionalities to work with MFC and local storage (SPE memory, LS). In order to work with the MFC, it provides primitives that starts DMA memory transfers, mailbox communication, query MFC state, and signalling. A specific primitives for the LS is available from the PPE, it allows the program to map the SPE local storage into an effective address. Mapping the LS to an effective address makes visible the local storage for all the program as a simple memory region. Main utility is to enable DMA transfers between SPEs.

MCXX Mercurium Compiler. *MCXX Mercurium Compiler is a LGPL licensed C/C++ source to source compiler which allows to transform source codes following a set of rules. MCXX is mainly divided in a front-end, engine, and modules.* The front-end is the C/C++ parser. This front-end is designed to parse a small snipset of code (function, instruction, declaration or expression) or a full file. The engine coordinates the module task. MCXX Mercurium Compiler is designed to be modular, each module represents a compilation phase, many phases can be added in order to create complex compiling pipelines.

MCXX Mercurium Compiler was designed for directive processing (but not limited), but at the same time, it is designed to remove the necessity to work transforming the AST (abstract syntax tree representing the code). A classical module, or transformation phase, is to provide to the engine a callback for each directive. Each time that the directive is found by the engine it invokes the callback with the primitive AST and context. For example, if the callback phase function wants to warp the original code by a loop, the function just creates a string with the code of the loop and as a body the pretty print of the AST primitive content, and replace the primitive AST by the string of the new code. MCXX Mercurium Compiler automatically solves variable references, declarations, and other inconsistencies that

might appear. This behaviour makes the compiler very flexible and useful even for non compilers experts (you just print the code as you should write).

MCXX Mercurium Compiler is shipped with support for OpenMP, software transactional memory, Cell SuperScalar, loop transformation utilities, functions instrumentation and instrumentation for pthreads, and many other examples.

Note: MCXX Mercurium Compiler is different from previous Mercurium Compiler. MCXX is the re-engineering of the previous Mercurium Compiler, but it is more reliable, more flexible, supports C++, and templates are plain C++ classes.

NVIDIA CUDA programming framework [66]. *NVIDIA compute unified device architecture (CUDA) is a parallel programming model based on C and C++ designed to program NVIDIA GPGPUs*. CUDA is designed to simplify the programming of the NVIDIA GPGPU architecture. This architecture is very complex compared with common commercial software: it is massively parallel and usually has available thousands of threads. Consequently CUDA is focused to deliver the power of high parallel architectures to common programmers.

The programming framework is focused (as OpenMP does) from the point of view of massive parallelisation of loops. CUDA has decided to conceptualise loops as logical spaces, if you have two or more nested loops you have a two or more dimensional space. If those loops are independent between iterations, they can be converted into a CUDA kernel invocation. CUDA kernels are just like C functions but they are executed in the GPGPU and called asynchronously from the standard C program.

As memory hierarchy is a big challange on almost all architectures CUDA has decided to have a mix between Cyclops and Cell B.E. memory architecture. CUDA assumes that there is a shared memory for a group of threads and modifies C data typing in order to make this memory visible. The design objective is to have a self managed cache which allows to hide memory latency and save memory bandwidth by high reuse of shared memory content between threads. This shared memory has visibility only for groups of threads (called blocks), other groups (or blocks) can not access to this memory. Although that shared memory is private, CUDA allows to access to main memory data, but paying a high penalty for the latency. In order to hide this latency, CUDA relays on tens of threads in order to reduce the pennalty.

CUDA uses a logical execution space (see figure 3.14) as an abstraction of the GPGPU architecture. This abstraction maps physical threads, core memory, and physical cores into logical threads, block memory and blocks. When the sequential code reaches a kernel invocation, it configures a logical grid of blocks defined by the programmer and launches its execution on the GPGPU. A kernel code is executed concurrently by all threads of the defined grid of blocks. Each block is physically mapped into a GPGPU multiprocessor, and many blocks can be mapped into the same multiprocessor. The threads of a block are executed in the cores of one multiprocessor and the block memory is mapped into the shared memory of the multiprocessor (figures 3.14 and 3.11).

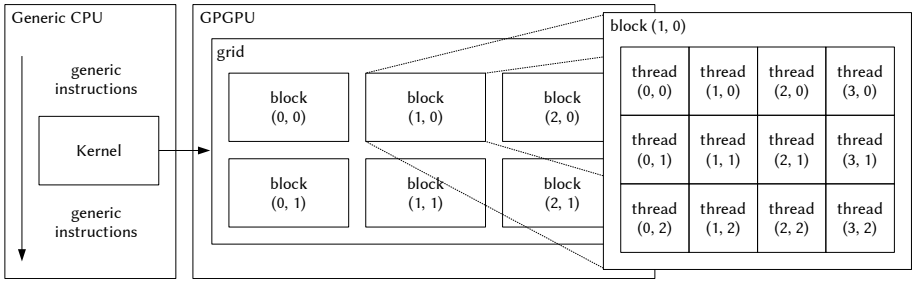


Figure 3.14: CUDA logical execution space.

This logical space is configured in the kernel invocation by the generic CPU program, but each thread has one position inside a block, and the block has one position inside the configured grid. As each thread has a logical position, this position is available to the programmer. This position can be consulted from kernel functions by two predefined variables. These two variables are designed as three dimensional points, each point (each variable) has the x position, the y position and the z positions. All threads which block position coordinate has the same value, shares the same shared memory. It can be used with the thread position value in order to coordinate partial results between threads and coordinate them into the computation of a result.

There are no directives on the CUDA programming model. This programming model is based on function encapsulation: code intended to be executed in parallel is isolated in a function. These functions (named kernels) are executed massively parallel. Each invocation (function call) to a kernel has a configuration which describes the logical grid (see figure 3.14). Kernels access to kernel local variables (equivalent to function local variables), shared variables (variables stored at block

memory shared by all threads of the same block) and global memory (variables whose value is stored on main memory accessible by CPU and GPGPU).

An illustrative example of CUDA is the following:

parallel for $1 \leq a$, $i \leq$ N **do**
 $C[a][i] = A[a][i] + B[a][i]$
end do

This algorithm translated into a parallel kernel as follows:

```
__device__ f_kernel(float C[N][N], float A[N][N], float B[N][N]) {
  int a = blockIdx.x, i = threadIdx.x;
  C[a][i] = A[a][i] + B[a][i];
}
f_kernel<<<N, N>>>(C, A, B);
cudaThreadSyncrhonize();
```

In this case, we have decided to parallelise *a* through blocks and *i* through threads. When *f_kernel* kernel function is invoked it configures a grid with *N* blocks, and each block with N threads. As we do not require more dimensions, we only use the *x* dimension of both spaces (blocks and threads). Block position is used as a index of the loop, and *x* thread position is used as *i* value of the loop. The cudaThreadSynchronize is used to wait for the result of the kernel invocation. In CUDA, by default, all kernel invocations are computed in parallel with general CPU execution.

CUDA has been proved to be a very reliable programming framework and a very powerful programming model. There are lots of community research on CUDA, community adapts and optimises existing algorithms to CUDA architecture. CUDA, like OpenMP, express parallelism through loops. CUDA, instead of reusing a loop construct, adds kernel configuration and invocation, which is equivalent to OpenMP parallel for directive. Maybe a step further to emulate OpenMP should give even better usability to CUDA.

Chapter 4. Related Work

The main challenge of this thesis is the viability of multi-core processors for the common programmer. In a few years, multi-core processors has become common and cheap. As processors complexity grown, processors designers realised that more hardware will not achieve better performance, so they just replicated the processor. At least, from the theoretical point of view of the problem, it just could speedup programs as well as previous solutions. But the reality was different [26]. Programmers does not want to care about threads and parallel programming. For many years parallel programming was not a requirement. As a consequence the programmer effort in parallelisation was almost superfluous and very expensive. At this point, programmers tendency and processor tendency started to diverge.

Nowadays we have a vast collection of code, programs, components and many kinds of artefacts. Most of these artefacts were created assuming serial processors. These artefacts did not care about threads or parallelism, but neither their original algorithms or underlying theories. As far as processor designers were able to stand, they decide to support better serial execution. It takes no advantage in supporting a non used features. But this situation ended, and as consequence hardware designers tried to create parallel hardware as close as possible to the mainstream programmers and their necessities.

In the beginning of common market multi-cores there was a great divergence of possibilities. Not all multi-cores tried to target the same applications and the same programmers. Probably the first steps on this direction were vector-instructions, but that was just the beginning. These instructions required a little effort from an expert programmer. Their use was limited to critical sections to give a very significant improvement on many applications. Slowly some architectures limited to replicate

the number of cores, just enough to increment the number of processes to be executed simultaneously. Others based their architectures on vector processors which were designed to increase vector-instructions performance, and others, tried to simplify a very complex hardware just adding thousands of threads.

These new challenges brought the need to create a synergy between computer architectures and programmers habits. This synergy brought new programming models as foundations of the interrelation between programmers and architectures. The main objective of programming models is to hide as most as possible the architecture particularities, try to give a view as close as possible to well known single core processors. At this point, like architectures, many programming models have been emerged and compete for the mainstream programming environment. The key point to achieve the success is the capacity to give a useful environment able to take advantage of existing code and programmer knowledge, but also a very competitive motivation to embrace the change.

Some programming models and architectures are highly dependent nowadays. Most programming models are specific for just one architecture, or in other words, just too expensive to use or slow on others. This heterogeneity of models forces programmers to be expert on many programming models if they want take advantage of each hardware available to them. Nowadays, simple desktop computers, are often provided with two different architectures: simple generic multi-core, and GPGPUs.

In this section, we will present some of the programming models and how they target their architectures, or even how they try to overcome their architecture. This research is the basis for this thesis and many of them have influenced the presented work. There is a large list of programming models and architectures, and each of them was designed or even improved to solve a different problem. Each piece of knowledge apported is one step closer to the general objective which is give multi-core for the mainstream programmers.

From single-core to multi-core. Before multi-core processors come multi-threaded processors. These multi-threaded processors were called multi-streamed processors, because they were designed to execute more than one stream of instructions. They based their design on the three main components of a single processor: register bank,

arithmetic logic and control logic. Register logic and arithmetic logic where simple components which specialised purpose. First processors map assembler instructions directly on these two resources: each instruction operates a selected registers over a determined arithmetic. Logic for this kind of processor architecture was really simple. Later, as processors start to parallelise instructions (using techniques like segmentation or out-of-order processors) the control logic began to be more complex. The control logic started to translate logic instructions to real hardware. This translation becomes complex and expensive.

The observation from multi-stream processors researches was the following: there is more hardware involved in computing the control of the parallelisation of a single thread (stream of instructions) than the required hardware to duplicate the stream of instructions. The main idea was simple: instead of create more and more complex control hardware, they execute multiple threads with the same hardware. When one thread gets stalled (it has to wait because any control risk), the processor resume the execution of another thread. This though was impulsed the researches into the multi-core era, and the same though that impulsed this thesis on the beginning.

One of firsts works into this direction comes from Eggers et al. at [67]. They experiment with the number of threads in order to find the exact ratio of the benefit of using multi-streamed processors in front of complex computers. The idea of this work was to prove the viability of the previous assessment and to compute the number of hardware threads that can be added in order to take advantage present hardware. They have shown that with the hardware that they had available, it was possible to add up to four hardware threads with a very good efficiency. Later, at [68] they have found that there was a misbehaviour in the cache performance.

If multi-streaming processors had shown that they helped to increase the number of parallel instructions executed by hardware they also have been found a serious drawback: cache seemed to diminish its effectiveness when multiple threads are executing on the same processor.

Far to be unexpected this feedback was quite logical. Multi-streaming processors, or current multi-cores, are sharing the same data bus with the memory. Most of previous works with multiple hardware instruction streams were conducted on multiprocessors. Multiprocessors are really many independent cores, each core has its own connection to the memory, its own cache, and to summarise: its own

independent resources. Multiprocessors gives to each instruction stream (thread) its own cache and memory bandwidth. As a consequence threads are no competing for resources as caches and they can scale properly, even better, in some cases because the total of available cache increases there appears an effect called super-linearity [69]: the parallel program is more than the number of processors used faster (speedup, parallel speed/serial speed) is greater than the number of processors). This effect appears because the size of the available cache for the program execution increases as the number of processors and caches are many times faster than main memory. But this changes on multi-streaming processors or multi-core. In this scenario a single processor execution have to share the same cache with all the present threads, and these threads are competing for the resources.

The effect of many threads in the same core are presented at: [70-72]. They present that more threads have more misses. They have tested many benchmarks on multi-threading architectures with a different number of threads. Their work shown that as the number of threads is increased, the number of cache misses also growns. This proved that threads were really competing for the same results. Gulati et al. [73] extended this work to try to characterise benchmarks and their impact on multi-threading architectures. They have shown that the ratio of the increasing cache misses were dependent to the benchmark, and each benchmark had its own cache interference patterns.

OpenMP over multi-core processors. One of the most important contribution for multiprocessor system is OpenMP [74]. OpenMP is a programming model designed for shared-memory multiprocessors. OpenMP has some very special properties: scalability, incremental parallelisation, portability, high level API, data parallelism and performance oriented. All these properties made the OpenMP an excellent programming model for this kind of multiprocessors.

Most current multi-core CPUs and multi-core and multi-threading processors presented at the beginning are very close to share-memory multiprocessor: they have a common vision of the memory and multiple threads which can be exploited by calls to a user threads library. This similarity made OpenMP a very good candidate, the only restriction to execute OpenMP is the capability of having a user thread library plus a shared-memory view of the system. From this point of view

multiprocessors are very close in semantics, and OpenMP can be implemented on a multi-core. Martinez et al. at [60] (as we have presented above in section 3.4, tools) have ported an OpenMP implementation to the IBM BlueGene/Cylops processor. This work shows that it was possible to execute OpenMP on multi-core architectures. Unfortunately performance were not as good as expected, benchmarks not scaled as well as expected. The processor has hundreds of threads, but the average speedup is of x15.

One of the largest architectures based on multiprocessors with shared-memory is constellations. This architecture assumes multiple homogeneous processors, but a large computer, where each memory bank has a variable distance/latency with each processor. This architecture is very close to cluster, there are multiple nodes, each node can have multiple processors with shared-memory, and there is a fast interconnection network between all nodes. Constellations, in contrast to clusters, provides a shared-memory illusion. They have a specialised hardware to manage memory accesses and provide memory-coherency.

On constellations not all accesses have the same latency. A processor accessing to a local memory (memory present on the same node) it obtains the data faster than if it is present into a remote memory. Programmers of constellations architectures must be aware of memory accesses penalties. In addition, many programming models (or even languages through specialised directives) added special commands in order to decide data position. As a consequence, programs must be written in order to localise maximum working data at the same processor which will process it.

Most of NAS parallel benchmarks for OpenMP (NPB for OpenMP [40], presented above in section 3.2., benchmarks) were not implemented to take maximum advantage on such architectures. These benchmarks access more or less uniformly to all address space, in other words, all processors requires to access to all memories. As a consequence there is no better data distribution: every body pays a penalty to access remote memories.

Fortunately NPB were implemented originally on MPI. MPI works on distributed-memory and applications must be rebuilt in order to work with local memory, access to remote memory is so expensive to be considered. As a consequence, NPB for MPI was already written to use small memories. Gonzalez et al. presented at [75], [76] the NAS MultiZone parallel benchmark for OpenMP (NPB-MZ, more details are

presented above in section 3.2., benchmarks). This was a new version of OpenMP NPB based on the NPB MPI benchmarks. This version of the benchmark was constructed on two levels: first level (outer level) emulates the work distribution across clusters, each parallel group computes the same region, and the second level (inner level) emulates the local computation of each node. As results show, the benchmark is able to take advantage of localised memory access and speedup the process.

Constellations can be compared with multi-cores: multi-cores are composite by many cores each one with its L1 cache (as each cluster has its memory) and each core has multiple threads accessing to the same memory (as processors of the same cluster have fast access to local memory of the cluster). We expect that coarse-grain parallelism will allow us to overcome this challenge.

OpenMP over distributed memory systems. We focus on OpenMP as a programming model in this thesis because it has many important advantages (as scalability, incremental parallelisation, portability, high level API, data parallelism and performance oriented), unfortunately it has a very important drawback: it requires shared-memory.

Nowadays all top 10 supercomputers [77] are distributed (non-shared) memory computers. Distributed computers are more complex to program (it forces the user to do data distribution) so there is a good motivation for such architectures: performance. Distributed computers have a very good performance, and for a very good reason: it is very difficult to maintain memory coherency on large systems. Distributed computers have no shared memory, so they do not need to keep an imaginary unique view of memory. The programmer is forced to encode in this environment and must keep every required data or result where it is required. The complexity is moved to the user, who, potentially, knows better the program and its behaviour.

Distributed computers problem is double: they demand to modify the program in order to face distributed memory, and moreover, they also visualise the computer as many different machines making difficult the program construction and debugging. The former is also faced partially on shared-memory, as we have explained that not all the memories have the same speed so keep things locally is better. But this is not

mandatory, it is only a recommendation: the program works in any case, the difference is the performance. The former problem is also important, most supercomputer users are not expert programmers, but to program, run, and follow the execution of a program across multiple machines is quite difficult. Moreover, the programmer must follow each execution, and try to know what is doing each machine and single program instance at each step. A common problem as a barrier misplaced can be a great issue.

Distributed shared-memory systems (DSM) are software able to give a view of shared-memory onto distributed memory systems. In this case, the addition specific hardware it is not required: a software controls the memory-coherency. The main advantages of DSM is that it have none of previous exposed distributed systems drawbacks, the main disadvantage is that DSM are not able to adapt any program to a distributed memory as good as programmers do.

The DSM needs to know the structure of the programs in order to increase the efficiency of the communications. The main problem is the following: if a determinate thread program needs a data outside its cluster, it freezes until the DSM recovers it. A programmer of distributed memory will reduce this waits as much as possible, and will replace some parallel structures by efficient distributed memory primitives. With no information, DSM can not assume any structure inside of a program beyond a collection of instructions and data, and consequently can not replace any of them by the proper primitive. A simple hint about a reduction operation would help to save lots of unnecessary data movements.

OpenMP was proposed as programming model for DSM at [78], [79]. The assumption is the following: OpenMP knows the parallel program structure and it is able to replace some structures with proper directives and add information to have more efficient data movement. The theory proposed is heading in the right direction, but resulting performance is not as good as expected. Unfortunately, many of successful OpenMP benchmarks had a very poor performance on DSM systems. They have too many memory dependences to have a good scalability on DSMs.

DSM where limited in order to achieve better performance. The first limitation was to separate data spaces into two regions: private memory for each node and shared-memory for all the system. In this case, variables stored on private memory were fast and not subjected to the network distribution primitives. Amza et al. [80]

presented TreadMarks, a DSM with a relaxed memory consistency model. As Lo et al. [81] shown, to weak some memory coherence characteristics are able speedup applications. The underlying ideas under both authors is to ignore some restrictions, and allow an inconsistency between memory values among nodes. Under this premise, benchmarks need to be rewritten, but there is a new hazard: memory inconsistency. Memory inconsistency can be controlled by the programmer carefully and avoid any arising problems, but programmer must know exactly, not only about values stored in the memory, but also about they consistent state. Basumallik et al. at [82] gone one step further: instead of executing the program under a software emulating a distributed system, the compiler translates an OpenMP program directly to an MPI program. They also have assumed the same restrictions than previous works, but in addition, they require to expand directives with memory hints.

Costa et al. at [63] have proposed a very different approach compared to the mainstream research: use an everything-shared DSM (no private memory regions) and use a full memory consistency model (no need to rewrite existing programs). Their assumption is one step further into the de detection of the OpenMP structures: all layers of the OpenMP programming model must collaborate in order to achieve a good performance. In this case, the compiler, the runtime and the DSM software were interconnected. Some presented solutions are the assumption of a cache line size as logical page size, the use of loop iterations to predict the presend (send data to future consumers before they request) of data, and adjust scheduling of iterations to avoid false sharing. Presented results were good but limited to some benchmarks with characteristic behaviours.

Beyond the research field OpenMP is already used on production at supercomputers. As we have commented MPI has a very good performance on distributed memory system, but for shared-memory OpenMP is a more suitable model. MPI bases its execution of ditributed memory, this is a multiprocess single thread program execution in multiple machines. MPI can use multiple processors or threads on a shared-memory machine, but it requires one independent process for each thread. As this solution is very expensive, a hybrid programming model is used to obtain the better performance from both models. Many researchers, as [83], [84], has concluded that an MPI+OpenMP hybrid model can achieve a very good

performance. MPI is used to create one process for each memory independent node, and OpenMP is used to spawn parallelism inside a single model.

One implementation of a benchmark on MPI+OpenMP for our thesis is the NPB MPI+OpenMP [85], implementation. There are two levels of parallelism presented. The outer level is performed by the MPI programming model. This parallelism level is responsible to distribute zones across the supercomputer. The inner level of parallelism is performed by OpenMP. OpenMP parallelism is analogous to the OpenMP implementation of NPB benchmarks. This solution is practically analogous to the previously presented NPB-MZ nested parallelism. This work is very relevant because it reveals some important points: 1) OpenMP is used even on distributed-memory architectures (cluster can have multiprocessor nodes), 2) NPB MPI+OpenMP and NPB-MZ has analogous implementations, 3) NPB-MZ is more simple and it can be used as previous step, and 4) when memory is close it is better to take advantage of present data using it by as many threads as possible.

Almost every solution proposed for DSM requires some sort of hardware support. Balart et al. at [86] proposes to use a compiler transformation in order to allow the software itself simulate shared-memory. They replace every access to the memory by a convenient primitive of memory access. They present the case for the Cell B.E. which accesses to remote memory are performed through DMA accesses. The efficiency is not as good as hardware supported DSM.

A multi-core modular and configurable simulator. Multi-core era is about of processors with many cores. When the number of cores is high, the number of possible internal architectures grows and also the possibility of the specialisation of any of present threads. In many ways a multi-core with tens of threads is as complex as a supercomputer and also is limited to the same criteria. The main difference between multi-core processor and a supercomputer is not the scale, it is the price and the availability. Nowadays multi-core are available for the mainstream programmers. These programmers might be able to program them and take advantage of multi-core possibility.

Perhaps the most previous quality for a multi-core is not the performance, but the capacity to extract maximum performance from its programmers. A multi-core able to run 10 times faster is not useful if the target programmer it is not able to extract

this performance. When Cell B.E. was publicly presented a great problem was spotted: it is very difficult to program. Nowadays, although its performance and power consumption, the Cell B.E. project was cancelled at 2009 [87].

Our claim is that a processor performance is not the theoretical peak performance, the processor performance is given by the ability of the programmer to extract performance from the processor. As there are a vast possibilities of multi-core processor configurations, it is possible that a few changes may reduce peak performance, but the same change will indeed increase the performance achieved by their programmers. Our objective is to present those small changes that will help to change the ability of the programmer to obtain better performance. For example, many scientific applications are not ready for clusters, and users execute many instances of the same program on multiple nodes (they do not want/know to use MPI), but most of them are able to parallelise their applications using OpenMP. We have seen previously that some works are able to transform one kind of architecture to another, they use software emulations. Unfortunately most of hardware required for the emulation are not present on multi-core processors.

Simulators were developed in order to test and verify new architectures before its implementation. First simulators were simple, they just had to simulate an architecture with one simple thread, one core and one memory hierarchy. With the arrival of multi-cores, specialised processor simulators become more difficult to build. Nowadays a simulator involves the execution of multiple streams of instructions, multiples levels of cache and memory-coherency, buses simulation for memory transfers, and many other components not present on old fashioned single-cores. Current simulators must face the a wider range of exploration space, many cores compositions and distributions, many memory hierarchies and multiple memory address spaces. They must be created modularly and each module must be easily reused for further designs or space exploration.

One of the most spread simulator, also usually applied to industry, is SystemC [88]. It is in fact a simulator infrastructure, and, in many characteristics this simulator is close to UNISIM [64] (we have presented it above in section 3.4, tools). UNISIM was build to supply lacks from SystemC. Another simulator designed to be modular is the M5 simulator [89]. This simulator is written in C++ and python (in many aspects its architecture is comparable to the GNUradio [43]), C++ is used to program

modules and python is used to describe the composition of modules. We do not use this simulator as infrastructure of our simulator because there was no modules available for PowerPC cores.

Our target architecture to be simulated is a Cell B.E. like. This is a very powerful processor but it lacks of enough versatility. The main problem is that the effort required to encode an application in order to have an acceptable speedup is too complex. We wanted to propose changes in the architecture in order to easy such development.

MAMBO [52] is the Cell B.E. simulator provided by IBM. This simulator is a full system simulator, able to boot a RedHat Linux operating system, and it has a very good accuracy in its simulated performance. It is also able to execute very fast with a reasonable simulation time. The objective of MAMBO was to make available the Cell B.E. architecture before its commercialisation. The drawback is that there is no source code available for the community and it can not be modified, so in many terms, it is as useless as the processor itself for architecture design exploration. Concurrently with this thesis was presented SimCell [90]. This japanese simulator do not included the PPE and had no simulation of the memory hierarchy. Their simulation objective differs from ours: they want to test different SPE configurations or kinds, and its programs. So their design exploration space was limited to reconfigure accelerators.

Simulator validation. Simulators are a very good tool for research on space exploration, but results are useless if simulators have no validation. Simulator must validate its behaviour in order to ensure that results are correct [91]. A proper validation can ensure that there is not hidden bugs that changes the simulation, or even there no are important restrictions or bottlenecks not implemented on the simulator.

We focus on the Cell B.E. as a simulator. One of the most complex components, and at the same time totally undocumented (we assume due to industrial secrets) was the EIB (the bus interconnection of all other processor components). Moreover, having buses of this complexity are not common on most simulators, so, while most of the other components have been validated and tuned extensively on most of the literature, there was to little literature about simulating multi-core processors internal connection high speed bus.

In order to do an extensive validation of the simulated interconnection bus we required an extensive performance analysis. We used results and benchmarks from Jimenez et al. [92]. They presented an extensive set of communications through the real Cell processor in order to characterise its behaviour. The other problem is that, although we know that the interconnection bus was a four rings based bus, we did not know the details. We use the bus analysis from computer networks of Girona et al. [93] in order to simulate the bus. In their work they shown that any kind of network bus can be simulated with an exact number of broadcast buses.

Streaming programming programs. As we deep in the study of multi-core and its characteristics, it is more evident that they are becoming more complex [17]. Future processor generations will add more kind of cores (specialised to many tasks) and complex memory hierarchy. Due to the expensiveness of maintain memory-coherence is likely that they will have distributed memory and complex.

This complexity becomes more urgent on mobile devices or processors. Specialised hardware, in other words, specialised cores and specialised communication buses, reduce significantly the power required to compute tasks whose hardware was designed for. As an example of this necessity was the creation of the ACOTES [94] project. This project, leaded by Nokia, Philips, NXP, Silicon Hive and STMicroelectronics focused on the creation of an infrastructure for mobile computing, but at the same considering the programmer experience.

As more complex the architecture more difficult is to find a programming model able to extract the performance. The programming model must be able to take advantage of heterogeneity and compile and execute each code to its more suitable core. Moreover, as this architecture becomes more complex, the variety of possible target architectures usually is greater. The compiler and the programming model must face this reality and adapt the program to have a good performance at each variation of the architecture.

Fortunately exists one programming abstraction able to face this complexity. It is the streaming programming model. One of the first (if not the first) descriptions where presented by Lee and Messerschmitt at [95]. This model assumes two main components: kernels and communication streams. Kernels are assumed as isolated processes running independently. Communication streams act as data transfers but

also as process controllers. Communication streams are point to point channels from kernels to kernels. They are usually defined statically and represents a continuous infinite flow of data. Communications streams are the only way to communicate kernels.

Streaming programs are able to execute naturally on distributed architectures, take advantage of specialised cores and even improve cache reuse [96] on shared memory architectures. They work naturally on distributed architectures because there is no communications between running threads, only explicit communications through streams. Kernels represents small functions. Some of these kernels can be easily vectorised, and even some of them can be classified by required computing resource. It is also possible to provide many alternatives (as for example function libraries) in order to find for each candidate which kind of core is the most suitable kernel. It helps to adapt these applications for changing heterogeneous architecture. Moreover, even communications are explicit and many times it is provided enough information to know the type, frequency and bandwidth and even source and target. It can allow to choose the most suitable hardware communication channel, as can be mailboxes, DMAs, and any other kind of bus. Kernels focus on the consumption of data coming from input streams and production to output streams and kernels activations are related to those streams. Blocking techniques can be applied in order to fit data in cache lines. Moreover, kernels can be fused in order to take advantage of already present data on cache lines. This optimisations can create super-linearity effects.

Streaming programming models. Like DSM systems, a streaming program requires information about its structure. If the programming language or framework does not provide extra information about programs kernels and streams, compiler and runtime can do very little in order to extract some structure. It is required that the programming language and environment adds information about the program structure.

Almost all programming models or libraries for create pure streaming applications are designed as two parts: one to create kernels, another to create connections. In manner of speaking they are as building blocks and glue. Kernels are basically the execution of the program or a determinate basic task. All kernels conform the building blocks of the program. The glue is the connection of kernels

into a unique application. Usually most of the programming models allow to create components by fusing kernels. These components can be glued again in greater components. Finally a program is the glue of all components into another component.

GNUradio [43] is one of the first libraries designed to build streaming applications. It was created in order to process radio signal directly through software and make hardware independent from changes in the standard. GNUradio has a large collection of filters (and its sources) to build many kinds of streaming programs. It includes the source of the FMradio benchmark used on this thesis. GNUradio framework consists of two parts: one for kernels and another for glue. Kernels are written in C++ a set of libraries designed to build streaming applications. The glue is written in python, it just creates the program structure. GNUradio provides a runtime to execute the program. It even has some modules ready to run on FPGAs in order to obtain relevant speedups.

One of the firsts programming models/languages for building streaming programs is StreamC/KernelC [97]. This language was created in order to provide an infrastructure to program the Imagine processor [98], a kind of massive multi-core data flow oriented processor. This programming model is structured into two languages: StreamC and KernelC. Both languages are designed as close as possible to C or C++. StreamC is the language specialised on the creation of communications and the building of the program. StreamC is the glue, it literally instances modules (kernels or collections already connected of kernels) and creates the connections between them, becoming a new module available, or even the main program. KernelC is the language used to create a kernel. It is programmed like a function which arguments and results are input and output streams, and its body is the processing of those streams. StreamC/KernelC has a greater control over the program than the GNUradio.

StreamIt [44], [99] is a programming model/language. It is inspired on StreamC/KernelC but modified to handle some hazards and to allow maximum control of the execution to compiler and runtime. The main difference with StreamC/KernelC is that it combines both in a single source. In addition they have limited input and output streams to a single flow of real elements. Multiple streams can be emulated with special collectives for data distribution or collection. Even with

these limitations StreamIt has a large list of applications successfully ported. The main contribution of StreamIt is the analysis about how a streaming program can be modified and adapted to the underlying architecture [99]. In this work, Gordon et al., explains how a streaming program can take advantage of task, data and pipeline parallelism. The pipeline parallelism is obtained by executing each stage of the stream process for a different temporal set of data in parallel, like processors does with instructions streams. The task parallelism is achieved by executing concurrent independent kernels simultaneously. Data parallelism is achieved when the same kernel is instantiated multiple times (or just manipulated to appear so) in order to process multiple input data simultaneously. In the case of StreamIt data parallelism can only be achieved when a kernel does not have state, in other words, when there is no local variables required to process the following element. As this limitation is represented as a stream from the kernel to itself. This cyclic stream presents a heavy dependence, it limits program parallelisation.

OpenMP like streaming languages. Previous presented models creates a new language to create streaming programs. Our target is to present a solution as close as possible to OpenMP. We believe that OpenMP characteristics are desirable, and the addition of streaming information to a plain C program should allow the creation of programs able to take advantage of streaming characteristics.

One of the firsts adaptations of OpenMP to implement pipelined executions was presented by Gonzalez et al. at [100]. They have exposed that there are some complex dependencies between tasks which can not be handled by OpenMP. They have proposed two new directives for OpenMP: *pred* and *succ*. This two clauses creates a pipeline execution from the producer of data to the consumer of the same data. There is no concept of stream, they relay on the shared-memory to store results and retrieve it later. In this case, the communication is limited to control. This two new directives are tied to de *parallel for* and to a *for* loop iteration. Each time that the *pred* directive is reached it hints which iteration has completed, and consequently its information is available on shared-memory. The *succ* marks which iteration is expected to be reached in order to read the result from the shared-memory. If the information is not available it stops the execution. Both, *pred* and *succ* are just one step of stream communication. Because there is no real stream and no kernels,

synchronization is defined on runtime and there is no such producer and consumer kernels connected. Any thread can be a consumer, any thread can be the producer.

Grid SuperScalar [101] (GridSS) is designed for the grid. GridSS execution is based on multiple binaries running on multiple computers with files as only mean of synchronization. The binary execution and file synchronization is based on the same superscalar model to solve dependences from supersclar computers. The idea is that these tools can be ported to all levels of the execution, even to the grid. Each program of the GridSS is constructed as many small programs communicated through files. The only communication performed between these small programs are files. They use files as input, and files as output results, but there is no other kind of communication of synchronization. All GridSS processes where controlled by a single C program or Script which represents all binaries invocations as simple function calls whose parameters and results are files. The GridSS runtime controls programs distribution, file synchronization and copy and, in some cases, also duplication and recovery. GridSS is not a streaming programming model and its results are not as we understand streaming. But it is very close, each program is like a kernel, and it also has the capacity to work on a heterogeneous environment where some programs can only be computed on some architectures and some communication lines are faster than others. The only thing which does not converts this programming model to streaming is the lack of capacity to create stable communication channels: each function invocation is independent and the runtime creates its dependence when is executed. When the program is executed, dependences disappears and no structure can be reused.

The Cell SuperScalar [102] (CellSS) programming model is based on annotations and targets heterogeneous and distributed memory multi-cores. It is also based on the same principle of GridSS and it also uses the same proposals: superscalar model to solve dependences from supercomputers can be ported to all levels of the execution. In this case, instead of having small programs, CellSS uses functions. This functions declarations are annotated with OpenMP like directives, these annotations complement the C definition of a function in order to express if any array parameter value is for input or for output. Arrays are like GridSS files, and the same rules are also applied. In this case, the serial program executes normally. When a function with its declaration annotated is reached, the CellSS computes function dependences

and schedules it to be executed in any of existing available resources. The same from GridSS similitude and differences are applied. In this case CellSS uses some kind of true streaming if the same function is called many times, it transfers the data and results continuously from parallel functions, or it even creates direct transfers between cores. The main drawback of CellSS is the scheduling. While GridSS works over network and scheduling overhead is negligible, CellSS works inside the same processor. Communications inside the processor and computations are so fast that the overhead of scheduling is very significant. It is possible that there is room for many optimisations, but because the model does not observe tasks as a persistent kernels consuming a stable flux of data, it must reschedule functions and data flows each time that an annotated function is reached.

Chapter 5. New Contributions

The aim of this thesis is to create or adapt a programming model in order to make multi-core processors accessible by almost every programmer. This objective includes existing codes and algorithms reuse, debuggability, and the capacity to introduce changes incrementally. We face multi-cores with many architectures including homogeneity versus heterogeneity and shared-memory versus distributed-memory. We also contribute by exposing real algorithms and applications and showing how some of them can be used for quasi realtime applications. For each section we present one step of this research, we introduce which publications support our thesis, and we expose our contributions.

Section 5.1. Multi-Processor Tools Over Multi-Core Homogeneous Shared Memory

Contributions exposed on this section were presented in publications [1] and [3]. The former relates to the optimisation of multi-core architecture and optimisation of the OpenMP execution and its viability on multi-core homogeneous processors. The latter relates to the use of multiple levels of parallelism in order to take advantage of tight communications of multiple threads of the same core.

Multi-core processors and OpenMP. *In [1] we demonstrate the viability of the multi-core processors and we relate its viability to the existing multiprocessor architectures.* Previous work ([60]) had shown the possibility of executing OpenMP over the many-core (multi-core with many cores) IBM BlueGene/Cyclops architecture, but its results did

not shown a good scalability. Using this previous work as a basement we do a scalability analysis, a cache usage study, analyse bottlenecks, and propose two solutions in order to increase parallelism. As a result we demonstrate the viability of multi-core for running OpenMP programs by showing a good scalability.

Our first step is to reproduce original experiments and run a scalability study. We present a study of scalability using NAS 3.0 benchmarks [40]. For each benchmark program, given a variable number of threads, we compare the behaviour to the serial version of the benchmark with no parallel overhead. Comparisons are measured as speedups. Programs are executed from 1 thread to the maximum of threads of parallelism available for each program. Most of the benchmarks fails to scale properly as they were expected in basis of multiprocessors results.

We present a detailed statistics of the cache behaviour of analysed programs. Many of previous work has spotted the cache as the bottleneck for multi-core processors ([68], [70-72]). We present a statistics of how programs access to caches. For each program we show its characteristic signature in the cache usage. We also present statistics about cache hit ratio and access counts. These statistics are also presented and shown for a varying number of threads, showing how the use of threads impacts on the use of the cache.

We present an analysis about how cache affects to the program execution on multi-core. By examining previous results, we establish that threads, program stack mapping, cache organization and cache associativity are highly relevant. We state from experiments that with the same program, and the same input, if we increase the number of threads the number of cache hits decreases. Program characteristics presented shows that program threads stack mapping has a relevant effect on the diminishing of cache performance. Used architecture helps us to evaluate how large caches (like L2 caches) are related to the program performance. We show that the associativity of the cache is critical, there is a high number of cache conflicts and the associativity requires to be related to the number of hardware threads. Results shows that a 4-way cache is good for 1 thread but not for tens of threads.

Previous analysis and results spotted that cache was effectively the main issue which prevents multi-core performing good results. We focus on the proposal of realistic solutions. Solutions can modify the software, or can modify the hardware solutions. Our solutions have been ensured to be realistic. For example, we have

avoided solutions like: having 4-way associativity is good for 1 thread, so for 128 threads we need 512-way associativity (128 x 4). Even if such cache can be implemented, the cost of the implementation is too expensive. The objective of our solutions is to avoid increasing the associativity.

We present that program stacks behaviours is one of the most relevant problems. Analysis of cache behaviours had shown that one of the most relevant effects on the cache misbehaviour are accesses from program stacks. Usually operating system and all libraries tries to align structures to multiples of powers of 2. We have proven that this behaviour, unfortunately, is catastrophic: many stacks are aligned in the same cache-line, so their execution conflicts. In addition, as happens with most of the OpenMP programs and other programs with regular parallelism, executions of all threads follows approximately the same execution path. It results in a catastrophic behaviour because all threads collides on the same cache lines simultaneously. This problem increments with the number of threads, given that the number of thread stacks increases, and as a consequence possible conflicts increase.

First solution presented is to modify how stacks are placed in order to avoid conflicts. We propose to introduce a dis-align relative to thread stacks themselves in order to avoid stack accesses collisions. We show that a determined dis-align between stacks can place these stacks strategically in order to avoid collisions.

Second solution proposes to change cache scrambling function in order to distribute stacks across different cache region. This solution has three objectives: avoid stacks cache conflicts, localise stacks closers to the core responsible for that stack and avoid data parallelism conflicts. As our target architecture cache has different latency depending of the thread location, we will locate stacks closer to its threads and study the effect. Data parallelism slices large data into a regular blocks, if this blocks are multiples of the cache size there can appear also a conflict between threads on cache access. Cache scrambling function is the responsible to decide, given a memory access address, which cache line storage is responsible for given address. If cache is 4-way associative and scrambling functions sends 5 accesses to the same cache line storage, one access is discarded. Usually, the scrambling function discards the bits which references a byte inside a cache line, and uses the number of cache line storage to get the following lower bits. This create a modulo space which accesses cache line storages at reasonable distance may collide. We change

scrambling function in order to use higher bits corresponding to the stack spacing. This solution have better performance than the software solution and it satisfies all three objectives.

As a result and thanks to all the analysis we conclude that multi-core can be effectively used by providing the tools required to increase the performance of applications on many-core processors.

Multi-core processors and multi-level parallelism. *In [3] we demonstrate that multi-level parallelism programs can take advantage of multi-core processors characteristics*. Previous presented work we have proved that OpenMP can achieve good performance on multi-core BlueGene/Cyclops architecture. This work tries to go one step further and achieve an even better performance by using a program which parallelism mimics multi-core architecture. As part of this research we have re-analysed previous results, ported multi-level parallelism programs to the multi-core architecture, we ported and effectively use the OpenMP compiler for nested parallelism, we studied programs and how the distribution of work affects to the performance, and we determined under which parameters is better to share cache for inner parallelism. We use previous research in order to have an optimised infrastructure.

The placement of the most used stack section is critical and OpenMP can help to detect it. We based our experiments in previous hardware solution for a good performance. We have discovered that parallel 1 thread execution is faster than serial programs (which has no overhead for parallel zone creations or further synchronizations). As part of the analysis we have discovered that as part of our solution the OpenMP runtime places new threads aligned to the closer cache. As a consequence, when the application spawns one thread as part of a parallel region, the stack for the new thread is mapped strategically on the cache. Usually, parallel regions encloses the most critical and time consuming tasks. These tasks are now perfectly aligned and can use effectively closer cache. We have discovered that OpenMP allows to realign stacks in order to place critical functions on closer caches for each core.

We port the OpenMP implementation for multiple levels to a multi-core architecture. There are previous versions of the OpenMP of multiple levels but none

ready for multi-core. We adapt the NthLib [57] extension for multiple level, included the support for the experimental *groups* clause [58] (described above in section 3.4. tools). Within the modifications we finish the porting of two kinds of spawning parallelism of NthLib and we also add architecture dependent primitives for synchronization and idle threads. The latter implies that spin-locks (often used on supercomputing) are complemented with directives to sleep waiting threads.

We port NPB-MZ 3.0 for OpenMP to a multi-core architecture. We use our OpenMP compiler to compile the multi-zone benchmark for the multi-core environment. We test it, we verify results and, in addition, we also modify benchmark programs to take advantage of experimental groups clause. We focus con class W.

We run an extensive benchmarking of NPB-MZ including 4 parameters and 3 statistics and present a summary of results. We run the three programs included on MZ benchmarks: SP-MZ, BT-MZ and LU-MZ. Results from LU-MZ are not presented due to its similarities to BT-MZ. For each program we vary the number of threads used, the number of groups of zones (from 1 to 16 for the outer level of parallelism, having total threads divided by groups as the number of threads per group), and the distribution of user threads in hardware threads (using from one thread per core up to four threads per core). As a results, we present statistics about scalability (by varying the number of threads and size of the groups), and statistics about cache usage including application characteristics (including changes in the number of threads and groups), number of accesses and which kinds, and cache hit %.

We reveal that NPB-MZ BT-MZ and LU-MZ programs are limited by load balancing up to speedup close to x30. Loop collapsing over code is required for a better parallelisation. Our benchmarks use small classes and as a consequence they are not designed to scale using a large number of threads. Benchmark SP-MZ presents a good scalability, up to speedup of 90 by using 127 threads and only 32 FPUs. The main characteristic SP-MZ is the regularity in its parallel regions and a good thread balancing. Each zone of SP-MZ has the same size and the execution time is almost the same for all zones. BT-MZ and LU-MZ are unbalanced. We focus on BT-MZ. We compute using limits on its *for* directives that there is a zone very large and it is not sliced in enough threads. If we assume that the maximum number of threads

is used, the most time consuming thread of the BT-MZ requires 1/29 of the serial execution time. As a consequence the maximum theoretical speedup is of 29. Our results gives a speedup slightly better than 30, thanks to cache locality. Load balancing limits speedup and parallel limits. We propose to perform loop collapsing in order to increase the available parallelism and decrease the size of non-parallelisable region.

We expose that as more threads we use, more accesses to cache are performed. Our results shows that as we increase the number of threads it also increases the number of thread accesses. We detect three reasons to increase the number accesses: more stacks replicated, more pressure over shared variables, and more synchronization operations. Each thread has its own stack, and its own variables. Although many of variables are smaller because they have been distributed, many others for temporal computations are replicated, and, as a consequence, the cache usage. There is also a set of global variables, accessed by all threads. If we have more threads, there are more simultaneous accesses for those variables. As a consequence increases the number of accesses. At last, each thread must be synchronized with other threads, most of these synchronizations are based on spin-locks, and all of them are based on shared memory. As a consequence, the number of accesses and collisions also increases.

Greater number of groups have better performance and less overhead. We test benchmarks varying the number of threads and groups. For a given number of threads, we test many group distributions. Results shows that experiments having a large number of groups have better performance than experiments having a small number of groups (with lots of threads). We detect that bigger groups requires less communications and less expensive synchronizations. More groups also creates smaller global data to share easing pressure over cache.

Experiments show that is better to share cache when there is a small number of threads. We previously have determined that the greater is the number of threads or lesser is the number of groups, greater is the number of accesses to the cache and its conflicts. We have determined that with a low number of threads it is better to share the same cache and the same core between all of them. We can use the same cache for many threads to speed communications. But, when the number of cores increases, the required amount of memory and cache pressure also increases, and as

a consequence the number of cache hits decreases. At this scenario, it is better to give a larger cache to each thread than try to reuse the same cache to speed communications.

Section 5.2. Annotation Based Programming Model Over Distributed Memory

Contributions exposed on this paper are presented in publication [2]. This publication presents a novel comparison of MPI versus OpenMP but taking into consideration MPI-like applications. We demonstrate that an application almost prepared for MPI can be executed efficiently on a DSM system but with all the benefits from de OpenMP programming. As an example, we take a nested programming model and we intend to emulate the behaviour of a multi-core with distributed memory between nodes.

OpenMP can be run efficiently on distributed-memory architectures. *In [2] we demonstrate that an OpenMP program can be as efficiently as an MPI program on a distributed-memory architecture.* MPI programs are designed to have low communication rates (coarse-grain parallelism) and a great cohesion inside each node (fine or medium coarse parallelism, if there is any parallelism). In addition, a programmer also must deal with a distributed environment and distribution primitives. We state that the same version of the algorithms can work on DSM efficiently, but, in addition, MPI drawbacks as distributed environment or distribution primitives are not required. In order to prove this, we optimise the OpenMP at the DSM and parallelism runtime, but we do not modify the original program.

As we have done on multi-core, we also port the OpenMP with support to multiple levels of parallelism to a DSM. As we have done before we base our porting on an existing DSM and already ported NthLib. Existing ported libraries only supported one level of parallelism. We implement the additional support for nested parallelism and *groups* clause.

We port NPB-MZ 3.0 for OpenMP to DSM. We port the multi-zone benchmarks, but we will focus our research on BT-MZ. This benchmark presents (as LU-MZ) a irregular zone shapes which creates challenges relative to load balancing.

We execute BT-MZ and present traces for analysis. We present a trace of memory accesses and a trace of tasks execution of BT-MZ class A. Trace memory allows us to detect conflicts in thee zones: thread stacks, NthLib runtime library (due to locks and synchronizations), and global data of the program. We present two kinds of analysis of BT-MZ task executions: one with a maximum of one node for each zone, and another one for two nodes for zone 16 (the zone computation is split among two nodes). We determine that spliting zones creates so many communications that it can not run efficiently (the execution time almost doubles).

We present an optimisation for work distribution. As part of the memory-accesse conflicts come from stacks and from synchronizations, we decide to re-implement NthLib runtime library to handle remote communications. The optimisation is performed for nano threads (threads with stacks). This optimisation consist in: 1) creating a work-descriptor for the task (it includes the function to execute and stack parameters), 2) sending of the descriptor through network to the corresponding node, and 3) replacing finalisation synchronization by a message from the executing node to the parent node. This optimisation removes almost all memory conflicts from stacks and conflicts from the library.

We optimise runtime library synchronization primitives to use message locks between nodes and spin-locks inside each node. Previous implementation of DSM and NthLib based all locks on messages, so their implementation are fast to synchronize multiple nodes. But this implementation is slow to synchronize threads inside the same node due to message passing overhead. We implement a double lock system: spin-lock for local locks and remote-lock to synchronize remote.

We optimise paddings inside application to consider a page as a line size. Most multiprocessor programs consider the line size in order to avoid false sharing conflicts. Usually small paddings are added to align data structures to line size (just few tens of bytes). DSM library uses page faults to emulate shared memory, so we might consider a page size as a line size. As we have detected conflicts in the use of global structures, we modify paddings from program structures to assume that the

line size is the logical memory page size. It removes all conflicts from structures not involved on zone borders communications.

The maximum speedup for BT-MZ class A is obtained with 5 nodes. We present traces of the execution of BT-MZ. We have determined that execution time of zone number 16 takes 1/5 of the whole serial execution time. As a consequence, if we want to keep zones inside nodes (in order to exploit coarse grain parallelism), the benchmark execution time can not be faster than zone number 16 execution time. We find that to split the zone number 16 into two nodes harms the performance.

Detection of reads for immediate updates does not help to split zones. We have detected that a significant part of the overhead comes from reads before updates: the program reads a value from a page, and immediately after it writes the result on the same page. The problem is that the first time, it requests the page shared and it pays one overhead. The second time it requests the page again, but for write permissions, it pays the second overhead. We develop a predictor which uses the instruction for read failed to annotate if there is an immediate update after. If the predictor has recorded that a read should do a posterior update, it request the page for write permissions and paying only one overhead. The effect of the predictor is almost negligible.

The execution of the optimised OpenMP is competitive against MPI results. We have evaluated the BT-MZ before optimisation, after optimisation and the same BT program for MPI. We have compared the three executions. The first version before any optimisation has not competitive results, but it scales. This demonstrates that coarse-grain parallelism can be well supported on distributed-memory by OpenMP. Optimisations have been proven to be very successful by achieving results competitive to the MPI.

Better control over zone synchronisation and communication can increase performance. Almost all unnecessary communications has been removed with optimisation. The only synchronisation remained (shown in figure 6.6) are zones update and access over some accesses to parent data. These updates are usually well known by the user, as a consequence, the user can hint data transfers and requirements. We propose the addition of directives to hint data sending and data receiving instead of waiting the page fault.

Section 5.3. Heterogeneous Modular Multi-Core Simulator

Contributions exposed on this section were presented in publications [5], [6], [11] and [7]. We performed a tutorial of the simulator at conference PACT 2007. These publications expose the design, the construction and the validation of a heterogeneous-processor modular simulator. The simulator is built as blocks connected by a glue which allows to change the configuration. The glue is the memory-access abstraction concept which establishes a unique and common protocol for all the modules and the main contribution of this thesis. This simulator intends to cover the necessity to adapt and configure multi-core processors for programs. The idea is to be able to find a trade off between peak performance and usable performance.

Memory-access as the glue of the simulator. *The memory-access is the cornerstone of the design of the simulator, it provides a common framework to interconnect every module in almost any configuration.* As the multi-core becomes more complex, it can combine many kinds of processors, but also can have a very complex memory hierarchy. Each module can be any processor of any kind, a memory storage, or a complex bus, but usually all of them, soon or later, communicates the same information: a piece of data. Memory-access has been designed to mimic the behaviour of a IP network but inside the simulator. As all modules must be designed to handle memory-accesses packets, they can be interconnected in any distribution.

We reuse ports semantics from UNISIM but we present a unique interface for all ports called memory-access. Modules can have many ports, but each port uses a memory-access as interface. Modules have ports, for receiving data or to send data. This behaviour is inherited for UNISIM infrastructure [64]. Unfortunately UNISIM infrastructure is not sufficient to create a real modular simulator: all modules must use the same protocol and interfaces. We have decided that all ports of our infrastructure has only one type: a memory-access. Modules use ports to send (output ports) and receive (input ports) memory-accesses. As part of UNISIM infrastructure, input ports can reject the receiving of memory-accesses.

We require a set of addresses for each module. On a network, each node know which address it has and programs knows the address of the node where to send information. As part of the glue, we require each module of the simulator to have an

address range, and if it can route memory-accesses, neighbours addresses. Each module will implement its functionalities based on memory-accesses to its own address range.

The memory-access description is the source address, the target address, the kind of access (read or write) and the data involved. The only interaction between modules are these memory-access. The kind of access determines if there will be a write command on a determinate address, or a data will be read from a determinate address. If memory-access address destination matches with the current module, it applies the memory-access command. Read commands usually fills the memory-access data with the requested state, and returns it, using the source address as target and write access. Write commands changes the state of the module with the data provided by the memory-access.

We define routers or memory-accesses buses as modules connecting many modules. Routers modules have many ports and for each port can have many more modules, and consequently, many candidate target addresses. These modules must have a list of target addresses and forward accesses to each module. Routing modules should be able to send to any of their output ports the corresponding memory-access, defined by the target address. Thus, as part of its configuration, in addition to connections, it has output ports addresses. In addition, routers can emulate pauses, delays, latencies, congestions, and any other

Our infrastructure is described around memory-accesses, not module functionalities. Modules can be as complex as required, but the infrastructure only emulates memory-accesses. This proposed infrastructure is designed to handle memory-accesses. There is no restriction about module shape, functionalities or timing. A module can be a functional simulator of a processor, a full simulator of each functional unit, or just an interpreter of a trace from an execution. The only requirement is to be able to produce and consume their respective memory-accesses and emulate timing for a correct simulation of the whole system.

Full power processor simulator and infrastructure. *In [5] we present the main building blocks of the heterogeneous simulator and its functionality.* We have developed a first version of the simulator for the PPE and the memory hierarchy. We also develop the OS emulation and the elf loader. PPE is verified by executing in the simulator the “hello world” program. This publication also presents another independent

simulator focused on the Cell B.E. SPE built by another group of researchers. We eventually fuse both works under our thesis infrastructure.

We design the simulator to be heterogeneous multi-core with PPE (common processor) and SPE (accelerators). We implement each module following the interface for the infrastructure based on memory-accesses. There are two processor modules, one for the PPE based on PowerPC, and other for SPE based on Cell vector processors.

The provided simulator is developed by two teams. There is one team for the development of SPE and another for the development of the PPE. This publication presents the simulator as two independent simulators not connected. The other team implements the SPE. Our team is responsible for the PPE, memory hierarchy, memory, and simulation configurability.

We implement a module for memory storage. This is a simple module which has an array (to store data), and receives memory-accesses as operations to perform over the array. This emulates a simple memory.

We implement the PPE and its memory-access interface. As part of the simulator we develop the PPE. This is the functional simulator of the cell general-purpose core. It is based on a PowerPC 405 with some additions of vector-instructions. The PPE reads instructions from memory and interprets them. The result of instructions can be a system call (resolved by the PPE itself) or memory-access to resolve load and store instructions. In addition the PPE has special memory-access target addresses for: set state and receive answers to read requests. State exposed registers contains the PC and if the PPE is running or halted.

We provide an operating system emulation based on continuations and memory-access for user space. We implement the operating system of the simulator based on UNIX system calls. We allow the program to perform some actions as access to files (or file descriptors as the standard output for debugging), get information, or even call exit. System calls are executed in native code, but their parameters (data from application address space) is acquired from the simulator using memory-accesses. When a system call requires a data access from the simulated program, it executes the system call on other context stack, and when an access is required, this is emulated by the PPE by memory-accesses. While the access is being solved, the context stack of the system call is halted. When the memory access finishes the

system call is resumed. No instructions are interpreted from PPE while the system call is executing. This mechanism allows system call programmers to program the system call without states in the same way that they can program it on any machine.

We provide an elf loader able to load static linked programs. We implement an elf loader to load binaries based on static linkage. We do not support dynamic libraries. The elf loader works as a system calls and uses memory-accesses to store binary and data into memory inside the simulator (it does not need to know about the shape of the simulator) and uses memory-access to program the CPU responsible to start the execution. This elf loader is not modified since this stage, and it has been working for many configurations of the simulator, proving the benefits from memory-accesses.

We decided an effective address space for applications of 32 bits and 64Kb for page size and translation for effective to physical pages. We have decided these parameters to simplify and speedup address translation. With 32 bits of address space and pages of 64Kb we are able to create a translation table of 64K positions. As the effective space is common for all the simulator, we use the mapping for all simulator. The emulation of effective addresses is required by the elf loader, but also it is required to map SPE memories or to map memory regions for double buffering. Almost all communications inside the simulator are performed over physical addresses as it should work on a real processor. Effective addresses are relevant for processors simulators.

A simple “hello world” application is simulated. We implement this application as a proof of concept that all the PPE system is working. The elf loader maps binary into memory. It programs the PPE to the start PC and set the state running. The PPE is able to execute the program and emulate its instructions. Finally the PPE performs the write system call (printf function is performed by the native libc) and exit system call.

Full Cell B.E. modular simulator. *In [6] we present the simulator working and a first validation of its behaviour.* Previous work we have presented the PPE simulator and a modular simulator infrastructure. We develop the compiling infrastructure, the libspe and libpthreads compatibility, and the optimisation of the PPE to reduce the execution time. Memory-accesses are adopted as a communication standard for all the simulator. The team responsible for the SPE integrates their work in our simulator and focus their effort on interconnection bus and MFC.

Other team implements the EIB emulation and MFC using memory-accesses. Their previous SPE simulator has not implemented the MFC and it was not based on memory-accesses. They implement the EIB as a k-bus to interconnect modules using memory-access and enabling a high degree of functionality. They also implement the SPE DMA memory controller called MFC. They also map registers from MFC to be accessed through memory-accesses.

We provide source compatible compiling infrastructure. We decide to not construct a full system simulator and give compatibility to Cell B.E. programs through source compatibility. Programs targeting the simulator must be recompiled to use specific libraries created for this purpose.

We develop a library compatible with libSPE. Although the other team was the responsible for the SPE, SPEs are programmed and controlled from the PPE. We implement an interface compatible library for libSPE which simulates the functionality of the library inside the simulator. This library has parametrized the physical addresses of SPE modules in order to access them from the PPE. It maps SPE physical addresses to effective addresses for the local storage (memory from PPE) and for MFC registers. It loads the binaries for SPE kernels from files into PPE local storage. It also programs (writes) MFC registers to start execution at desired PC. It also queries MFC registers to be noticed when the execution finishes.

We develop a library compatible with pthreads library. The simulator is intended to have more than one PPE in order to emulate a general-purpose multi-core. We use pthreads (as we use on common processors) to start threads on other processors. We implement pthreads library by satisfying its API in order to execute functions in other PPEs. Our pthreads library has a list with PPE and its register states. When a thread is created it modifies the PC and the state of a free PPE.

We implement a cache with invalidation protocol. We have to add cache support. Not all memory accesses can be cacheable. We create a list (like the effective to physical address map) which decides which physical addresses are cacheable and which physical addresses are not. We implement a parametrizable cache. Cache is able to invalidate its data if it receives an empty data for a memory address. Bus must notify to the cache of such invalidations.

We perform a functional validate the PPE. We create a set of programs in order to validate that the PPE has the expected behaviour. Programs emulates scientific

applications but also uses some specific simulator functionalities, as SPE programming.

We perform a performance analysis of the simulator. Although the overhead introduced by the UNISIM infrastructure, we have optimised the PPE to execute up to 5 millions of instructions per second.

Validation of the heterogeneous and modular simulator. *In [11] and [7] we present the validation of the simulation and a proof of concept of its modularity.* A simulator can not be trustful if it is not well validated. In this publication we validate the simulator against performance benchmarks. We also provide a some tested configurations.

As our first intention was to construct a Cell B.E. simulator we focus on Cell B.E. simulators to validate its performance. The most important piece for the validation is data communication inside the bus. We use the benchmarks from Jimenez et al. [92]. These benchmarks are focused on memory transfers from inside of the processor. We compile the same benchmarks for the simulator and we execute. We compare results in order to prove the correct functionality of the simulator.

We demonstrate simulator modularity. Although our first objective is to emulate the Cell B.E. (the first massively distributed heterogeneous processor) we also want to perform architecture exploration space with the compiler. We build several possible architectures and we present it. Some of the presented architecture configurations are the Cell B.E. itself, general-purpose multi-core, heterogeneous modules with different kinds of accelerators or DMAs, multi-cores with scratch-pads and caches for SPEs.

Section 5.4. Annotation Based Programming Model For Streaming Applications

Contributions exposed in this section were presented in publications [4], [8], [9], [10], [13] and [12]. These papers relate the construction of a OpenMP like streaming programming model. We have designed a programming model for streaming programs, and built a compiler demonstrate its viability. We present a programming model able to take advantage of streaming characteristics presented by Gordon et al.

at [99], but at the same time able to take advantage of existing code and programmers.

The first version of this work was presented on the ACOTES meeting at Eindhoven, October 2006 [103]. We just have presented some ideas developed in a blackboard on summer of the same year as the starting point of a streaming model based on kernels and communication flows.

Many publications presented on this section also talks about an Abstract Programming Machine. This part belongs to Paul Carpenter. It presents an abstraction of a streaming program and also an abstract processor architecture. The idea of combining two works, the work on this thesis and the Carpenter's abstract programming machine, is to make possible to the compiler to adapt the code to any underlying architecture.

Basis for the OpenMP like streaming programming model. *In [4], [8] and [9] we present a programming model OpenMP like designed to build streaming programs from serial programs.* We use the OpenMP as a start of a streaming programming model. We pursue the same benefits of the OpenMP and we use the same kind of annotations to hint kernels and possible communications to the compiler. We provide an algorithm to transform such annotations on serial code in order to demonstrate that it can be performed. As a proof of concept of the result we also provide some examples converted and analysed.

We present the first streaming programming model based on OpenMP. We design a programming model able to take advantage of existing code. We assume that there is no automatic parallelisation, and we do not require an expert knowledge of streaming programming. The presented model is designed to be compatible with OpenMP and it is also based on directives and clauses working on top of serial valid code.

We implement almost all desired properties from OpenMP. Our programming model is designed to have scalability, allow incremental parallelisation, portability (in fact has better portability than OpenMP, due to there is no shared-memory assumption), high level, performance oriented and data parallelism. Data parallelism is implemented by computing multiple elements of the stream in parallel, it only can be applied as on StreamIt if there is no state on kernels (we have solved this problem on later works). It allows incremental parallelisation by allowing to identify kernels

one by one and creating and routing streams automatically based on serial program semantics.

Only three main directives are required. In order to obtain an easy to use programming model we define three core directives: taskgroup, task, and input/output. Taskgroup directive (originally named pipeline) defines a region where streaming tasks are contained, it allows to mix serial parts with streaming parts. Task directive identifies the code of a streaming kernel. This kernel is executed as many times as it has been reached on serial code. Tasks are persistent and they keep their state between invocations. Input and output directives defines an input or output port for a stream. It is based on variables whose value is received or sent. Stream is created by the compiler.

We present an algorithm to transform the application based on directives. As part of the demonstration of the viability of the programming model and as part of the definition, we supply an algorithm which describes how to transform the application. This application explains step by step how tasks are identified, ports are added, and how stream connections are created automatically.

We present a method to optimise streaming graph. The result of the previous algorithm can produce limited results: it can produce cycles in streams graph when they are not required. We propose a method which can be easily implemented on intermediate representation to remove unneeded cycles and optimise connections. It exposes consumption and production from/to a stream as consumption and production instructions. Consumptions are placed as soon as possible, productions as late as possible. If a consumption and production of the same value are consecutive, it creates a bypass.

We expose directives for explicit communications implementations. We present input and output as task clauses and stand-alone directives. We state that if all input and output directives are mandatory, produced graphs from the adapted original algorithm are always optimised. This schema can be reused on DSM for explicit communications.

We present a stream implementation with no locks. In order to make a proof of concept we implement a stream library based on no locks. This library uses two counters in order to determine the number of elements produced and the number of elements consumed. Atomic updates are used in order to update values for

production or consumption. Consumer waits for elements if there are not present, and given a buffer size the producer waits for enough room.

We implement tolower and worddhash as a proof of concept. In order to demonstrate that the algorithm and directives are sufficient to build a streaming application we transform two programs. We apply the algorithm manually to tolower and wordhash algorithms and we developed parallel versions which uses streams in its executions.

We present execution traces with different optimisations. We present Paraver traces in order to show the program behaviour. We also present zooms of the traces in order to show details. We have developed three versions of presented programs: 1) direct transformation by applying the algorithm, 2) stream graph optimisation by applying optimisation method, and 3) optimisation by applying blocking to kernels. The first version contains cycles, the second version remove cycles, and the third version takes advantage that there are no cycles to enclose task kernels into loops and reduce the number of communications.

Presented programming model is designed to generate the graph at compile time. Our main objective is to define a programming model able to do the same transformations that presented by Gordon et al. at [99]. For this reason the programming model is designed to work at compile time (unlike OpenMP). The idea is that almost all required transformations transforms binary code. Although these changes can be performed by a very sophisticated runtime, the overhead can become too expensive.

First definition of the programming model. *In [10] we present the first definition of the programming model, clauses and directives.* Previous work has been focused on presenting main semantics and directives, but there are many other functionalities required by the industry not present on the previous work. We define all directives and clauses of the programming model and theirs behaviour.

We present first formal definition for each element. This definition is inspired in the OpenMP standard definition and includes detailed information about all directives and clauses.

We define the first following directives: taskgroup, task, port (as a replacement for standalone previous input and output directives), for_distribute (to use existing

loops as part of kernels control loop) and update (to update asynchronous pseudo-shared variables).

We define the first version of clauses. Some clauses are deprecated on posterior versions. We define the following clauses: input, output (to create ports), import, export (to use user-managed streams accessible directly from programmers code), target-input, target-output (to allow create explicit graph connections), firstprivate, lastprivate, pivate (to handle state), async (to declare pseudo-shared variables), shared (to declare real shared variable), requires (to specify specific properties required by the code).

We define the acolib interface for program generation. Directives and clauses are translated to a lower level library. Acolib is the definition of the primitives used by the streaming program defined by directives. These primitives are designed to be used as part of the intermediate representation of the compiler in order to be subjet to low level transformations and optimisations.

We present examples of stream graph optimisation. We present many examples of the utilisation of the programming model, their translation, and how their stream graph is optimised.

Full definition of the programming model. *In [13] we present the first definition of the programming model, clauses and directives.* Previous definition was the first step but there was many requirements from stream programs not achieved. We define the execution and memory model, new semantics to correct problems with state definition and directives for an effective handling of data parallelism.

We define the memory and the execution model. We define the memory and the execution for tasks. This model assumes that tasks can be created at the same time that taskgroup streaming environment is reached. Tasks execution is driven by control stream inputs. If no direct connections are performed, task kernel invocation is performed as many times as serial program executes it. There are defined four kinds of memory: private (private variables which value does not survive from one invocation to another), state (private variables which values are kept between invocations), pseudo-shared variables (variables whose values can be synchronized between tasks, but synchronisation is performed lazily), and true shared variables (as default OpenMP, they limit where task can be placed if two shares the same variable).

We define programming model directives. We define the following directives: taskgroup, task, teamreplicate (replicates computations to keep state up to date with data parallelism), port, for_replicate, peek (maps an array to a stream queue), update, check (ensures that a pseudo-shared variable value is updated), ivdep, dividesby and aligned (to boost and guide automatic vectorisation).

We define programming model clauses. We define the following clauses :input, output, bypass (enhance graph optimisation), private (private variables), state, copyinstate, copyoutstate, initialisestate, finalisestate (for state variables, whose initial value is obtained from serial program or it uses an initialisator or finalisator to compute it), async, sync (for pseudo-shared variables), shared, team, inputreplicate (enables and defines inputs behaviour for teamreplicate) and requires.

Although we have defined clauses for constants, they do not appear on this specification. These clauses allow to define variables whose value is constant for all the execution and it allows many optimisations.

We present an extended of the acolib in order to translate all directives and clauses to intermediate representation. In this case some functions are proposed to be exposed to the user to have better control of some processes.

The working version and demonstration of the programming model. In [12] we present an almost complete working version of compiler and reference library of the programming model. In previous works we have defined, studied, verified, and iterated over the definition of the programming model. In this work we present a fully functional compiler able, automatically without human interaction, to translate a serial program with annotations to a streaming program. We also implement a library able to perform almost all the acolib primitives in order to emulate a distributed-memory processor and demonstrate its effectiveness. We evaluate serial programs transformed into streaming, and we present a set of synthetic examples as a guide of the programming model.

We annotate FMradio, Wifi 802.11a and FFD filter. As a demonstration of the correct behaviour of programming model, compiler, and acolib we annotate serial version programs of FMradio and Nokia Wifi 802.11a. We also extract the FFD filter from the FMradio application as a stand-alone benchmark. All these programs are compiled correctly using the compiler. We link to our acolib implementation in order to demonstrate its functionality.

We present an evaluation of the created streaming programs. We use serial programs (ignoring directives) to evaluate the streaming programs. For each program we study the capacity to correctly detect and create the application graph (not published for Wifi program due to copyright restrictions), we execute with our simulation to demonstrate that scalability is possible. We also demonstrate data parallelism on the FFD filter.

We present an incremental construction example. We present the construction of a streaming program from serial program step by step. A plain C serial code is first annotated with the taskgroup directive which defines the zone to be streamised. It allows to mix serial and streaming zones. Next step defines task kernels by adding task directives. Finally we show how the graph is optimised.

We present an assisted loop blocking. We present the possibility to use the *for_replicate* directive to reuse a task outer loop as a control loop of the task. It enables direct loop blocking optimisation by ensuring multiples of number of invocations for the kernel. One of the best utilities of the loop blocking is to have a good vectorisation: automatic vectorisation is usually performed over loops with independent iterations. This vectorisation can be ensured if tasks are prone to have the same behaviour for all elements, and if those tasks are able to exploit data parallelism.

We present the advanced state handling. We present state related clauses to handle states. We present examples of state complex variables initialised and finalised by serial code inside each task.

We present asynchronous update of variables. We present how some variables can be updated remotely and checked. They help to give a kind of shared-memory view, but the programmer knows that values are not always up-to-date.

We present how to use stream queue. The main challenge of this option is how to access to specific hardware as stream queues but reusing at the same time serial code. We show how the peek directive is able to use an existing array as the stream queue. Peek reduces the overhead of copying stream queue and at the same time it reduces the required state variables.

We present data parallelism support for tasks with states. We present an example which is able to exploit data parallelism and use at the same time state variables. Presented example illustrates the utilisation of the teamreplicate directive and the

inputreplicate clause. The execution trace of the example is also shown to illustrate the mechanism.

We present how elements are balanced between two data parallelised tasks. Each task that exploits data parallelism creates multiple instances of the same task. Each instance is computing a part of the input data. This data is distributed across all instances in order to have load balancing. If we have two consecutive tasks exploiting data parallelism, and they have a different number of instances, results must be communicated in order to ensure load balancing. We define and show module computations in order to select how to send each result.

In addition, presented as part of these work, we want to emphasise three more contributions: a solution for consumer/production ratio, collaboration in the Multicore Streaming Framework, and the implementation of the FMradio.

We provide a solution for the consumer/producer ratio. One of the most important problems is the consumer/producer ratio. Almost every streaming programming model suffers from this problem. Usually stream programs are designed as a circuit: there are many components connected and each connection is a flow of electrons. But there is a problem, on streaming problem there are not almost infinite flow of electrons, there a finite set of data. Most of streaming components define a ratio between production and consumption. This ratio define how many elements are produced related to how many elements are consumed. It should ensure that no problems like one component receiving data from two flows, but consuming one of the flows at half peace. If it happens a data overflow, then data can be lost. This effect can be seen on most of video decoders as a lack of synchronisation between audio and video solved by stopping and playing again from the current point. The solution proposed by other programming models is to ask to the programmer to be careful. Our solution is easy: follow serial program semantics. This effect is impossible on serial programs, so its conversion to streaming does not break this semantic and programmer is protected against these failures. Nevertheless, we allow expert programmers to create direct connections. In this case, consumer/production ratio problem can appear, but it will be created by the expert user, not by the average programmer.

Collaboration in the definition of the IBM Multicore Streaming Framework (MSF). As final stages of this work we have collaborated with IBM Haifa research labs in

order to use MSF as a back-end for our streaming programming model. We have defined the only requirement to interrupt a task and resume later, as part of the requirement for incremental streaming program construction.

We write the serial version of the FMradio. We have as a reference application the FMradio, an adaptation from the GNUradio source. The problem is that this application was developed as streaming application, with explicit streams and tasks. We have converted the FMradio from streaming program to a serial program, able to work on serial. The serial version developed by us of the application is used later by the literature as their reference implementation.

Section 5.5. Graph Matching On Current Architectures

Contributions exposed on this section were presented in publications [14], [16] and [15]. All three papers are related to the parallelisation of graduated assignment algorithms targeted to desktop processors, focusing low power architectures. Our target processors are either main processor and graphic processor. The firsts two papers relates to the parallelisation of the graduated assignment graph matching algorithm. The third paper is related to the parallelisation of the common labelling algorithm.

Graduated Assignment Graph Matching Parallellisation. *In [14] and [16] we perform two parallelisations of graduated assignment graph matching for low power consumption architectures.* We present a methodology to transform an algorithm into a parallel algorithm: we use two basic transformations, loop tiling and reordering, and three OpenMP like primitives for parallelisation. We also determine for the graduated assignment algorithm that we require two different parallelisations: one focused on large graphs, and another focused on multiple small graphs. For each parallelisation we apply the methodology step by step. Resulting algorithms are implemented and compared against the serial version.

We present two OpenMP like directives to express parallelism in two levels. Parallelism on NVIDIA CUDA have a duality of strategies: coarse grain parallelism for loosely coupled task, and fine grain parallelism for highly coupled tasks. We

develop a hybrid OpenMP like directive, based on OpenMP for directive, which allows to express data parallelism. This directive allows to specify if the specified level is block parallelism (coarse-grain parallelism) or thread parallelism (fine-grain parallelism). In addition we also add support to reduce operation. We contribute with the required semantics to compile the same code by an OpenMP compiler, or by a CUDA compiler. Reduction operation is implemented following the CUDA reference implementation [104].

We present an OpenMP like directive to express local data on CUDA. CUDA programming model provides block memory, which is assumed that has a very low latency and can be accessed by all threads of the same block. This is a private memory, not supported by OpenMP standard (implies distributed memory). We define an OpenMP like directive, inspired on previously presented peek directive (see section 5.4.). This directive defines a region of a larger matrix to be copied to local memory. It uses an access pattern to each index of the matrix to define when the local copy is accessed. Optionally a set of permutation of indexes can be specified for reordering dimensions of the sub-matrix stored at block memory. It allows to ensure high-performance by data coalescence. Synchronization are added in order to ensure that there is no data hazards. If statement contains any sub-matrix element modification, it flushes the whole sub-matrix to main memory.

We present a methodology to transform serial algorithms to parallel algorithms. There are many ways to transform a serial program into a parallel program. Many of these techniques are applied automatically by compilers, but lack of information prevents them. Although some algorithms can be completely transformed, or rebuilt from scratch, to create a parallel new algorithm, we present a methodology based on two transformations which helps to parallelise a program without changing its semantics. Methodology is based on two loop transformations: loop tiling and loop reordering. Loop tiling is usually performed automatically by parallel compilers and run-times when a parallel for is executed (there are less processors than iterations and a double loop is created). We use loop tiling technique to classify accesses by sub-matrices. These accesses allow to restrict the amount of data accessed simultaneously by all threads. This gives the change to use the block memory efficiently. Loop reordering technique allows programmer to reorder loops, and to expose required loops for parallelism. It gives a perfect control of executing threads,

and at the same time, this technique allows to fine tune the access to sub-matrices and to reduce the amount of block memory required.

We have presented a parallelisation for large graphs. For the same algorithm there are many parallelisations. In this case, we have presented a parallelisation oriented to scalability based on the number of nodes of the graph. This parallelisation is focused on the premise that the graph does not fit on block memory. We have a very good performance and we also have been executed this algorithm for a very large number of nodes. We have not found any literature showing the graduated assignment graph matching of any graph as large as we present. This parallelisation presents a very good performance and scalability. It only have a poor scalability for small graphs.

We have presented a parallelisation for small graphs. We have seen that previous parallelisation has a poor scalability on small graphs. In this case, we have presented a parallelisation oriented to performance which graphs have a low number of nodes. Under this premise we can assume that graph representation fits on block memory. That means that the computation can be only performed by one block. In order to take advantage of all present hardware, we do multiple small graphs matchings in parallel. This algorithm parallelisation mimics NPB-MZ nested parallelism: outer level has very coarse-grain parallelism with almost no communication, inner level has a very fine-grain parallelism with coupled communication.

Graduated Assignment Common Labelling. *In [15] we present a parallelisation of the graduated assignment common labelling for low power consumption architectures.* We validate previously presented methodology to transform an algorithm into a parallel algorithm. In this case we determine that even small graphs are able to extract good performance if there are enough graphs to create the common labelling. We present the performance and the scalability of the resulting algorithm. The resulting version of the algorithm presents the same results of the serial algorithm.

We validate the previously presented methodology. Although the current algorithm is also based on graduated assignment, the algorithm is different and so its parallelisation. We use the same tools presented on the previous work. In this case we use the methodology to merge and split some parts of the algorithm to create highly coupled kernels. We use loop reordering to find common loops on many parts of the algorithm to merge them. In addition, we split some parts of the

algorithm by loop fission. After all modifications, the resulting parallel algorithm has the same result than serial algorithms.

We have validated previously presented directives. We have used the same semantics from directives presented in [16] to express algorithm parallelism. These annotations have been useful to express parallelism and understand parallelism consequences.

We have presented the graduated assignment common labelling parallel algorithm. We have presented the parallel version of the algorithm and how we have applied the methodology to obtain it. The parallelisation is presented in two kernels. The first kernel computes an auxiliary introduced variable, this kernel parallelisation is very close to the graduated assignment computation. The second kernel computes the common labelling, the exponentiation, and performs the normalisation of the result.

We present performance and scalability results. We have implemented the sequential algorithm and the proposed parallel algorithm. Both algorithms have been tested over a GPGPU parallel architecture and over a generic Intel architecture. We have used two databases to obtain algorithm performance. We execute the algorithm on a low power architecture and we show the performance and scalability for each data set used.

Section 5.6. New Tools

Contributions exposed in this section are available as GPL licensed software. Almost any research or experiment requires a tool to perform required demonstrations. In this thesis, we have used two types of tools: computers and software. We have tested high performance computers (expensive) and common desktop computers (non-expensive). The cost of software depends on licence and its accessibility.

In this section we present a set of tools and software available to the scientific community. Although in the beginning, we do not put a great effort to make our tools publicly available (most of them provide from already existing non free tools), in the lasts stages of this thesis we licensed new created tools with GPL in order to

make them free and publicly available. We only regret not to find/apply a license to force tools modification distribution if a paper is published using such tools.

Some of the tools with no free license constructed are: IBM BlueGene/Cyclops simulator add-ons to obtain cache usage statistics and the simulation of the hardware solution, the port of nested parallelism of NthLib to IBM BlueGene/Cyclops, the port of nested parallelism of NthLib to DSM, and the optimisations over NthLib and DSM for nested parallelism.

Tools created with GPL free license which allows the community to create new research on top of them are: a tracing library, a portable light threads library, the heterogeneous modular simulator, the pure serial FMradio benchmark, acotes prototype compiler and acotes prototype runtime.

Mintaka tracing library. We have developed a library to take traces of programs. This library provides an API to trace the application and generate Paraver traces. To use this library the user must use ibrary definitions and link against it. This library supports the tracing of states, the tracing of events, and the tracing of communications. This library also has support for hardware counters, it is able to flush automatically information related to hardware counters. In addition, the library records flushes of the trace to the disc. Library timing function can be replaced. Library also provides of support to synchronise clocks between multiple machines. This library is fully configurable and it is designed to work simultaneously on shared-memory and distributed-memory.

Portable light threads. We have developed a library in order to provide portable light threads. We have implemented a library based on C calls longjmp, setjmp and alloca to create non kernel threads. This library warps this calls in its API and allows to create very cheap threads with no dependence with assembler. Lonjmp and setjmp are used to save the context and change the contest of the current thread execution, they act as scheduler. Alloca function is used to modify the stack pointer to point to the stack of a new thread.

A validated heterogeneous modular simulator. We have developed a simulator as part of this thesis. All the simulator is available under the GPL license. In this simulator we have taken advantage of GPL license and we have used PowerPC 405 ISA implementation from other GPL project.

Pure serial FMradio benchmark. Previously we have presented the original FMradio. This benchmark was provided by Marco Cornnero of STMicroelectronics extracted from GNUradio. The benchmark has structured as a set of functions and data structures emulating streams. We have adapted this benchmark to create a pure serial application with no emulation of streams. As the original GNUradio project was licensed as GPL, the serial version of the benchmark kept this license.

Acotes streaming programming model prototype compiler. We have developed the OpenMP like streaming programming model prototype compiler over the Mercurium compiler on its first stages of development. The Mercuirum compiler is an infrastructure designed to process C/C++ code with annotations and transform to other C/C++. We have developed the Acotes prototype compiler over this infrastructure. We provide of a compilation phase which transforms all annotations into stand-alone functions and calls to the run-time. Source is designed in two layers to separate stream program model from run-time. It is possible to change easily the run-time to adapt to any other existing run-time.

Acotes streaming prototype run-time library. We have developed a library as a proof of concept to support the acotes streaming programming model. This library is built on top of pthreads and Mintaka library. It emulates distributed tasks with no direct access to shared-memory. This library is able to create multiple tasks, stream connections between them, and even support complex data parallelism.

Chapter 6. Practical Evaluation

Section 6.1. Multi-Processor Tools Over Multi-core

This section shows results supporting contributions exposed on section 5.1. Exposed results and figures are extracted from publications [1] and [3]. In these results we show the cache impact, the performance and the scalability of benchmark programs. Results validate proposed optimisation, expose cache statistics, and nested parallelism behaviour.

In [1] we study the viability and we expose that multi-core architecture can run efficiently programs designed for multiprocessor architectures. Previous work of porting OpenMP to multi-core [60] architecture has shown that multi-core is able to execute OpenMP applications, but performance was not as good as expected.

We develop a set of experiments and metrics based on OpenMP and a multi-core architecture. Our target benchmarks are NPB for OpenMP [40] (described at section 3.2.). Target architecture is IBM BlueGene/Cyclops [45] (described at section 3.3.). The base environment is NthLib [57] and the OpenMP NanosCompiler [55] (described at section 3.3.). We have executed benchmark programs in the original OpenMP and IBM BlueGene/Cyclops environment and we compare its behaviour against two improved environments proposed.

Figure 6.1 shows the cache behaviour for the MG class W program. We can see that the original version has a bottleneck on cache 29, and the number of threads decreases the cache hit % ratio. Proposed optimisations solves the cache problem satisfactory. Software solution has a better cache hit ratio, but threads stacks are

placed far from the executing thread (column 1, highest bars). Hardware solution has good cache hit ratio, but not as good as software solution. In this case we can observe that access to cache 9 to 19 (accesses to global data) are higher, it degrades cache hit ratio. In the other hand, hardware solution stacks are close to its threads, low latencies compensates the loose of cache hit ratio.

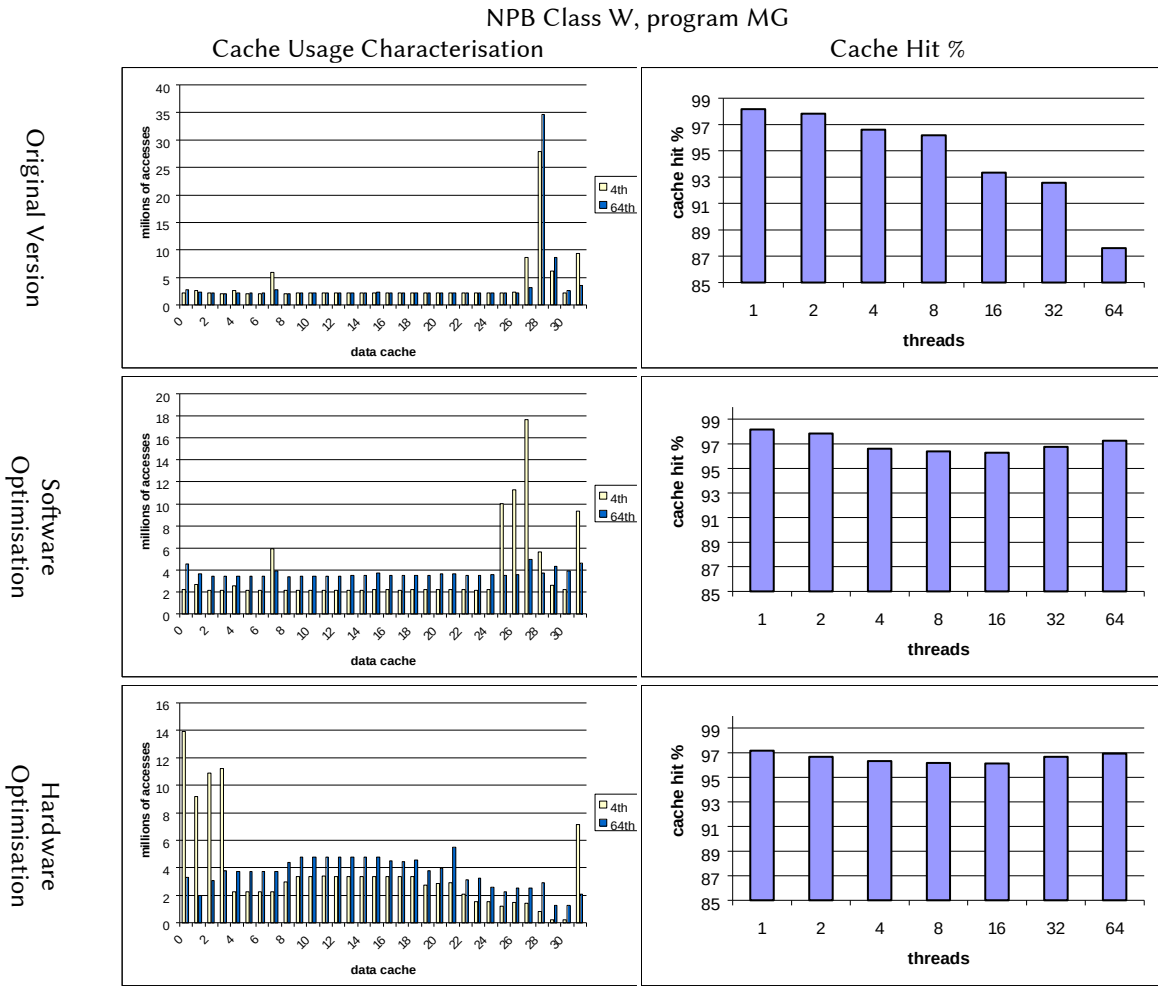


Figure 6.1: Cache behaviour for the MG Class W program in the IBM BlueGene/Cyclops architecture.

Figure 6.2 shows the speedup of the Class W NPB benchmarks. Although the Cyclops has 32 cores (like a multiprocessor of 32 processors) we can observe that original platform performance is far away of expected of 30. The software solution improves the performance of all benchmarks, the gain is between 10% and 70% depending of the application. The hardware solution improves in all programs but

CG, performance improvement is between a 40% and a 100%. CG has best improvement on software solution. BT has an impressive improvement on the hardware solution.

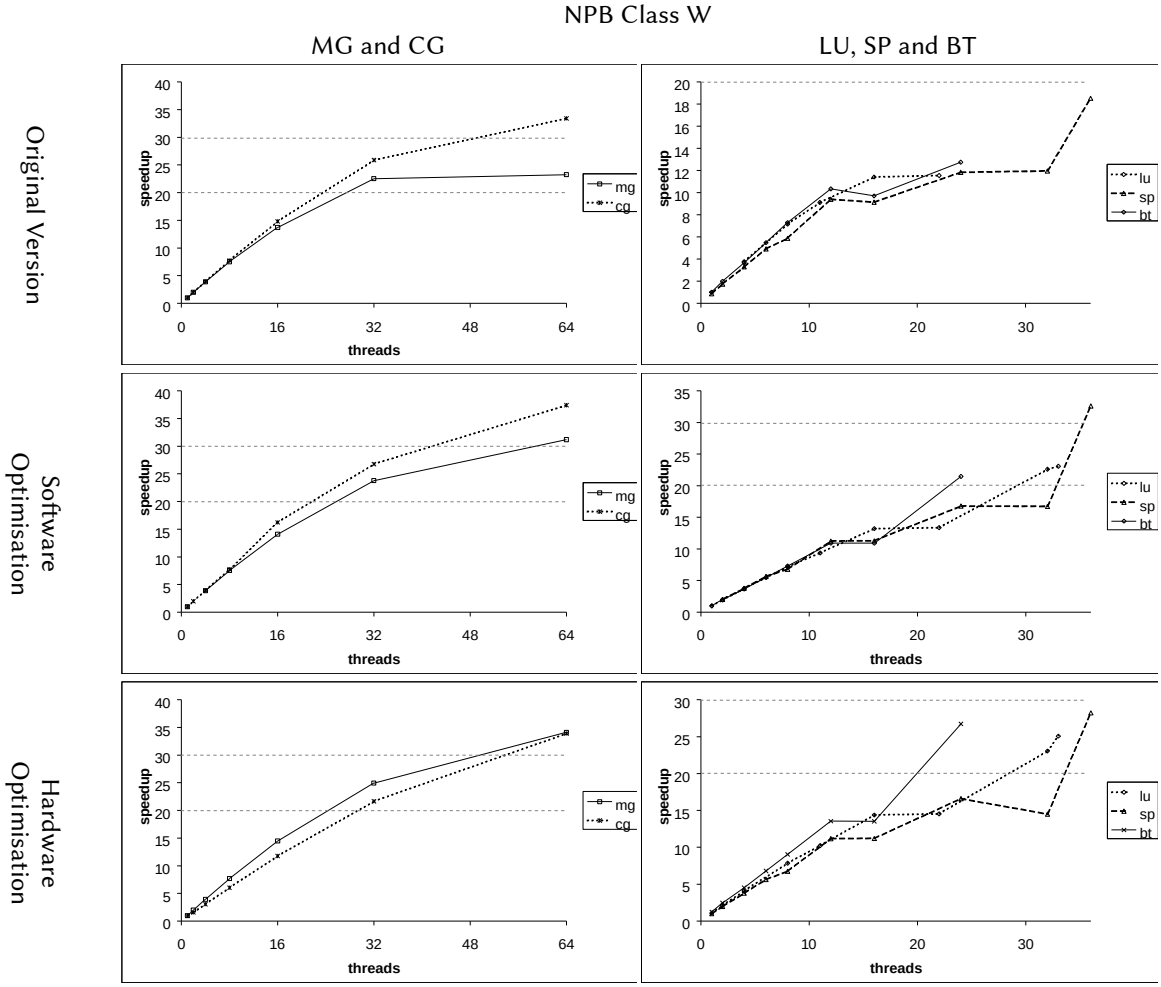


Figure 6.2: Scalability of the NPB programs Class W in the IBM BlueGene/Cyclops architecture.

In [3] we study how to take advantage of multiple threads sharing the same core (and core resources) by using two levels of parallelism: one for coarse-grain parallelism and other for fine-grain parallelism. We expose that this kind of multi-level parallelism can exploit the underlying architecture. We analyse the performance, the cache effect and the effect of coarse-grain parallelism and sharing cores.

We present a set of experiments and metrics based on nested parallelism on OpenMP and a multi-core architecture. Our target benchmarks are NPB-MZ for OpenMP [41] (described at section 3.2.). Target architecture is IBM BlueGene/Cyclops [45] (described at section 3.3.) with the modification of the hardware solution from previous work. The base environment is NthLib [57] and the OpenMP NanosCompiler [55] (described at section 3.3.). NthLib stack allocation policy maps stacks into closer caches. Presented experiments have metrics to evaluate the speedup of many configurations, to study the cache impact, and how to distribute threads across cores.

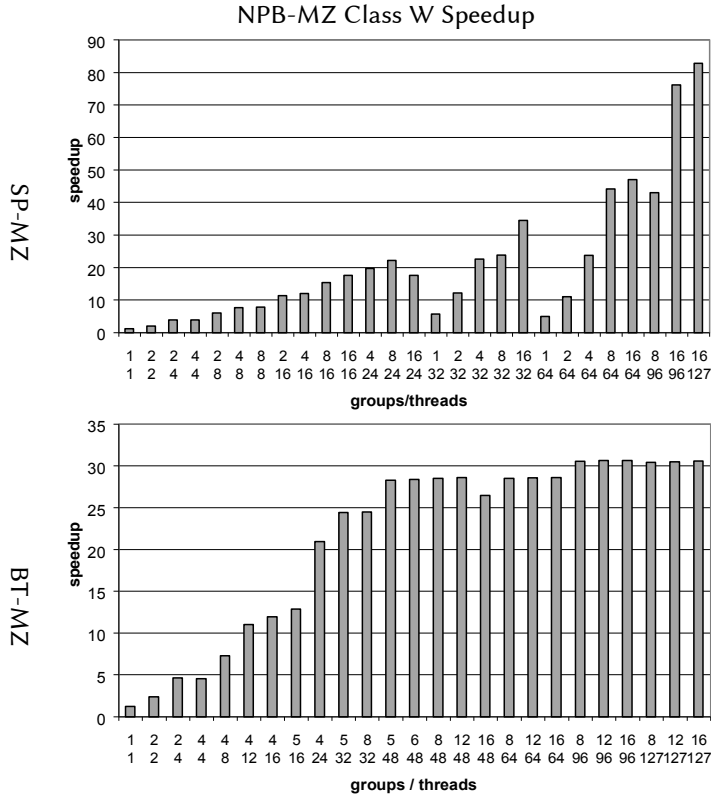


Figure 6.3: Scalability of the NPB-MZ programs class W in the IBM BlueGene/Cyclops.

Figure 6.3 shows the speedup of the Class W NPB-MZ benchmarks. We have focused on SP-MZ and BT-MZ programs. For each program we have executed it from 1 threads up to 127 threads (the 128th hardware thread is reserved for communication). Scalability of SP-MZ is really good, with 16 groups and 127 (almost to 8 threads for each group) it achieves a x80 (three more times than SP no MZ version, shown in the previous work). We can see that increasing the number of groups increases the performance. Only 16/24 execution do not satisfy this condition,

it does not balance well threads between groups. Unfortunately BT-MZ, as we have explained (due to unbalanced tasks) has a maximum theoretical performance of x30.

Figure 6.4 shows the cache access of the SP-MZ varying the number of groups. The first chart shows that the larger is the number of groups the smaller is the number of accesses. This behaviour corresponds to the fact that the external level involves more communications than the internal levels. With more zones, we have smaller groups to coordinate and consequently less overhead. The second chart shows the data distribution for 1 and 16 groups. We can see that both almost have the same pattern of accesses, basically changes the scale.

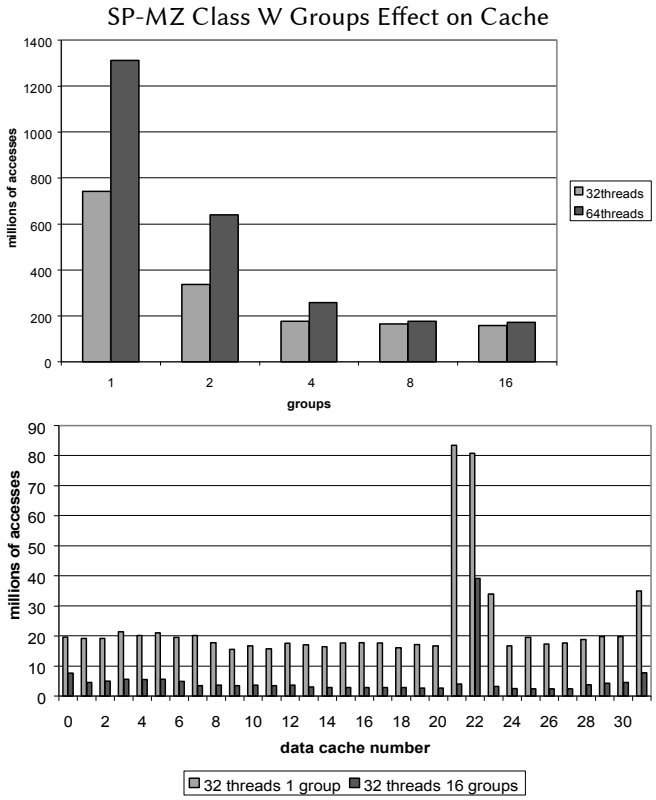


Figure 6.4: SP-MZ groups effect on the cache of the IBM BlueGene/Cyclops.

Figure 6.5 shows the effect of the usage of multiple threads per core. We have used the stride option from NthLib for Cyclops. With stride 1 we use four threads per core, with stride 4 we use 1 thread per core. First chart shows the kind of accesses to the cache given a number of threads and groups. We can see that given a low number of nodes is better to use multiple threads per core. The next chart shows the access pattern to the cache. Stride 1 has more accesses to the same cache, the

number of local accesses are larger. Use stride 4 balances better the number of accesses for each cache. Last chart compares a large execution of stride 1 against to stride 4. Stride 1 concentrates many of the accesses into the firsts chaches, but with stride 4, it is able to take advantage of the whole cache.

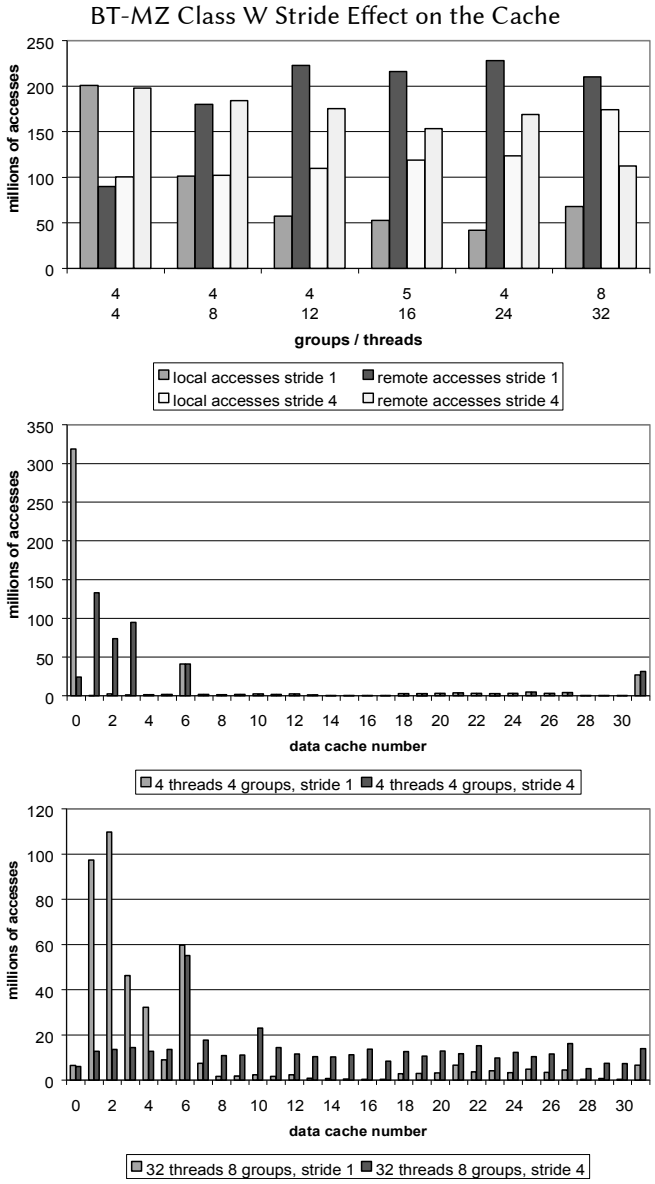


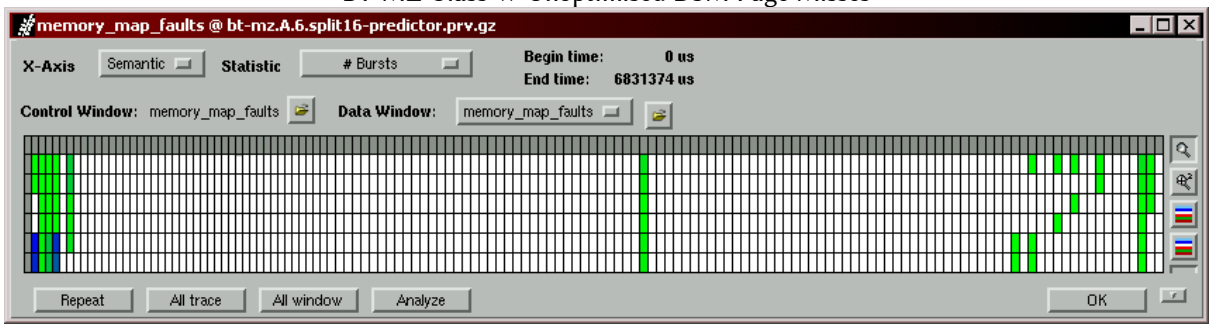
Figure 6.5: BT-MZ sharing threads on the same core effect on the cache of the IBM BlueGene/Cyclops.

Section 6.2. Annotation Programming Model Over Distributed Memory

This section shows results supporting contributions exposed on section 5.2. Exposed results and figures are extracted from publication [2]. In these results we show memory page distribution and access across nodes of a cluster, impact of coarse-grain parallelism on distributed-memory and the performance of the proposed solution. Results validate our proposal to use OpenMP and coarse-grain parallelism on the software-distributed-shared-memory.

We have developed a set of experiments and metrics based on OpenMP and a distributed-memory architectures. Our target benchmarks are NPB for OpenMP [41] (described at section 3.2.), more exactly the BT-MZ Class A. Target architecture is the Kandake machine (a cluster of Pentium computers connected by a Myrinet network, described at section 3.3.). The base environment is NthLib [57], the OpenMP NanosCompiler [55] and the NanosDSM library [63] (described at section 3.3.). In addition we have traced experiments with libFASTparparaver and visualised with Paraver [61].

BT-MZ Class W Unoptimised DSM Page Misses



BT-MZ Class W Optimised DSM Page Misses

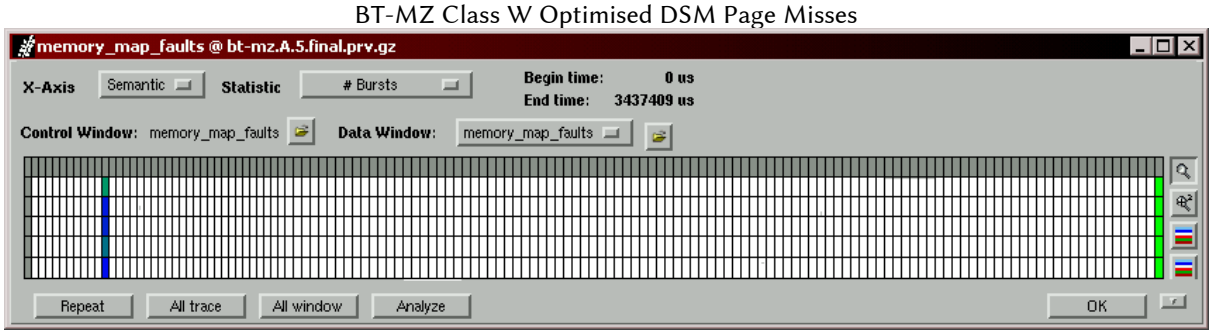
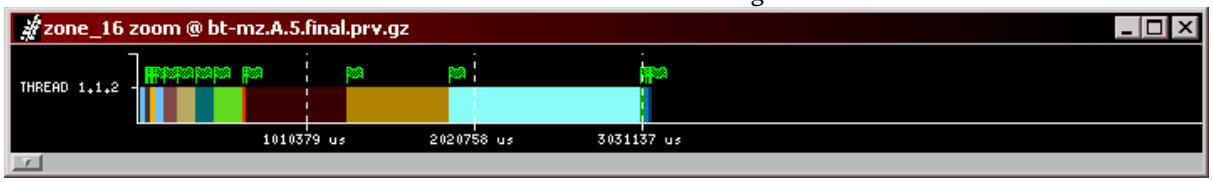


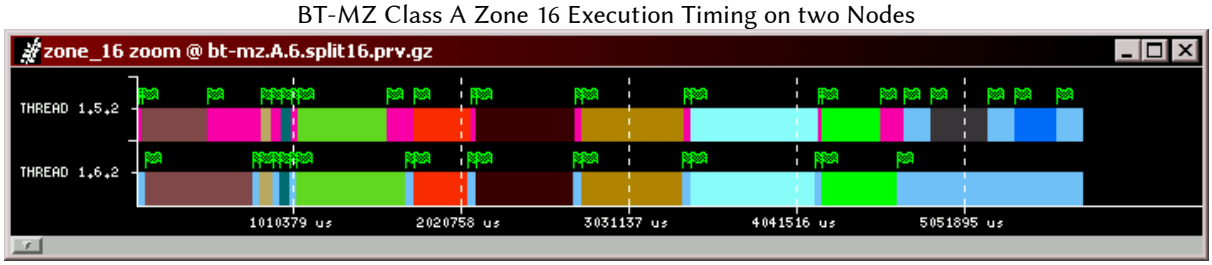
Figure 6.6: BT-MZ Class W memory map of page misses for each node on a SDSM.

Figure 6.6 shows a trace for the page misses of program BT-MZ on a DSM cluster. Each row represents a node, each column a range of program effective address space. Each square shows how many page misses are present on a given node and for a given address range. White squares are no page misses. Light squares presents some page misses, and dark squares presents a large number of page misses. Left squares are the program variables in global memory, central squares are NthLib runtime variables, and right squares are thread stacks. First trace is the corresponding for the unoptimised execution. Second trace is the corresponding for the optimised runtime. We can observe in the second trace that almost all page misses have disappeared. Only remains some page misses in the right (corresponding to basic parameters for all threads, almost negligible), and left misses (corresponding to zone borders synchronisations). We advocate to use streaming like directives to communicate results directly.

BT-MZ Class A Zone 16 Execution Timing on one Node



BT-MZ Class A Zone 16 Execution Timing on two Nodes



BT-MZ Class A Zone 16 Execution Timing on two Nodes with Read/Write Predictor

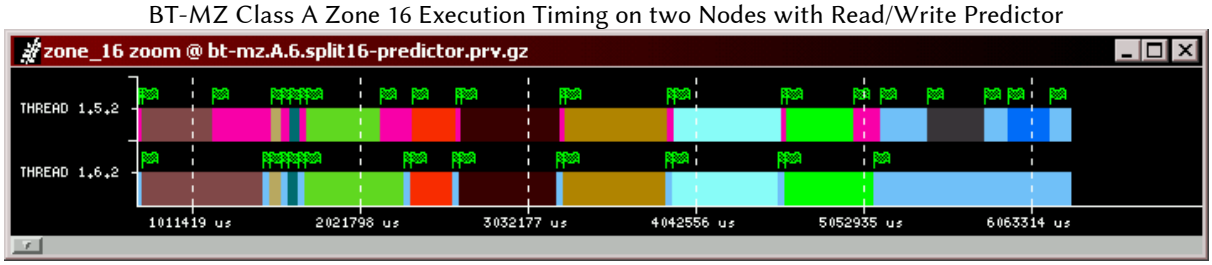


Figure 6.7: BT-MZ Class A zone 16 execution timing.

Figure 6.7 shows the trace of the execution timing of each task of the zone 16 of the BT-MZ program. Each trace has the same time scale and each row is one node. Blocks inside the line represents different tasks for the processing of zone 16. Zone

16 execution time in one node is 3.099 seconds, the same zone, if we split the execution into two zones, it requires 5.696 seconds. Last trace shows the zone splited in two nodes and the action of the read with intention for modification predictor. The execution time is 5.601 seconds. This time is slightly better than the original, but anyway, it is more expensive that execute the zone in one single node.

Figure 6.8 shows performance comparative of the BT-MZ Class A in a cluster. We compare the MPI version against the OpenMP. We have presented results of OpenMP using two versions of the OpenMP, the original SDSM version unoptimised, and the optimised version proposed in our publication [2]. Figure shows that the original SDSM unoptimised is able to scale. Although its bad performance, it has no slowdown. Optimised SDSM version has a very good scalability. In spite of the fact that this solution is not as fast as the MPI version, the difference is acceptable. The OpenMP version is easier to program and maintain.

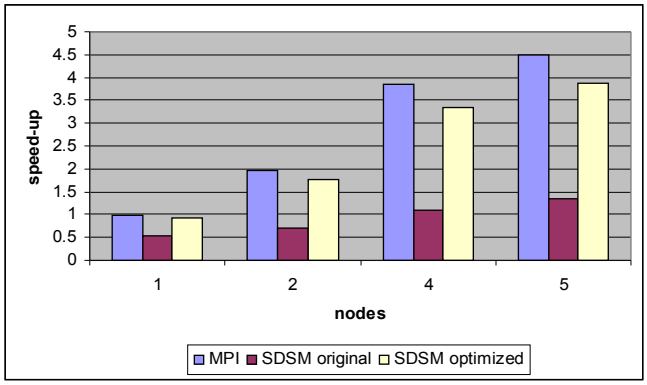


Figure 6.8: BT-MZ Class A performance comparison for MPI and SDSM.

Section 6.3. Heterogeneous Modular Multi-core Simulator

This section shows results supporting contributions exposed on section 5.3. Exposed results and figures are extracted from publications [6] and [7]. In these results, we expose the behaviour of the CellSim simulator. This behaviour is compared to the behaviour of the Cell B.E. processor.

We study the behaviour of the CellSim simulator based on the behaviour of the bus. We have used [93] to replace the original Cell B.E. EIB interconnection bus with a k-

bus. We use benchmarks from [92] in order to compare both executions. We have recompiled with our libSPE in order to be executed inside the CellSim simulator.

Figure 6.9 shows the comparative between Cell B.E. bus performance against CellSim simulator bus simulated performance. The CellSim configuration selected has a k-bus of four buses. Memory transfers are more optimistic in the CellSim, this is because we do not implement the memory interface. Cell B.E. chart shows that the maximum bandwidth for single SPE with memory is limited (following charts shows that the EIB does not have a 8GB/s limitation per one SPE). Although this difference simulator behaviour is correct. The second group of charts shows the transfers for many SPEs in a ring fashion: SPE0 sends to SPE1, which sends data to SPE2, … which sends data to SPE_{N-1}, which sends data to SPE0. This should be the most efficient communication pattern, but it shows some transfer contention. Our k-bus of 4 rings simulates best results for many SPEs. Up to 4 SPEs it performs similar. Last pair of charts represents couples of SPEs communicating point to point one with each other. This is the most efficient communication pattern in Cell B.E. (as shown it its chart). Cell B.E. is able to route multiple communications in the same ring if they do not overlap. Our k-bus design is limited to 4 communications simultaneously, and as a consequence it achieves its maximum bandwith with 4 SPEs. Although behaviours are not exactly the same, performance patterns are close. Many of the limitations presented by the k-bus can be solved by adapting the number of buses involved. We can approximate the correct behaviour to each application. In addition, we can not also rely on the scalability of Cell EIB bus, because, on the one hand EIB details are not public and confidential, and on the other hand, nothing ensures that Cell B.E. EIB behaviour is scalable to other chip configurations.

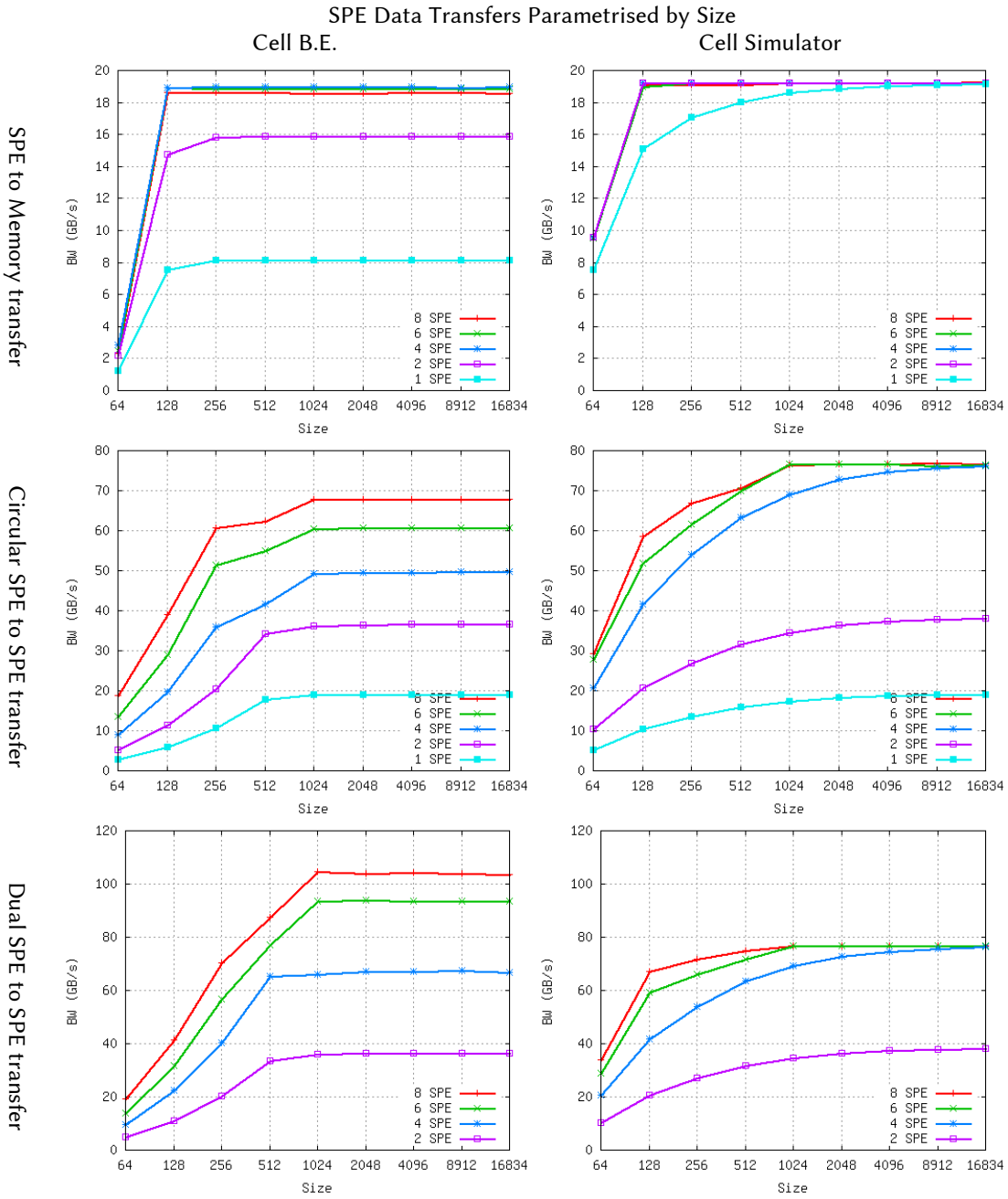


Figure 6.9: Cell B.E. versus Cell Sim SPE interconnection bus behaviour study.

Section 6.4. Annotation Based Programming Model Over Heterogeneous Distributed Memory Streaming Applications

This section shows results supporting contributions exposed on section 5.4. Exposed results and figures are extracted from publication [12] and [8]. In these results, we show how the annotated programming model works, its characteristics, and some performance benchmarks of the prototype (compiler and run-time). Results validate proposed programming model, and exposes the capability of obtaining a good performance exploiting many kinds of parallelism.

In [12] we have summarised all characteristics from the annotation based programming model for streaming applications. Previous works to this programming model have been suggested that annotations are able to extract good performance, and at the same time, they are able to adapt parallelism for existing programmers. In addition we also have seen that the performance can be increased by exploiting coarse-grain parallelism and making explicit some communications.

We have developed a working prototype of the programming model. This prototype is developed in C and C++. It is implemented in shared-memory. Our target architecture is distributed memory, we have introduced limitations over the prototype in order to simulate distributed-memory architecture characteristics. Our compiler prototype is based on the MCXX Mercurium compiler (described at section 3.3.). It converts a C serial program to a streaming program. We also have executed some of the working prototypes on a dual processor IBM Power5 with a dual core, in addition each core has two threads. We have used Acolib prototype library as run-time to execute resulting streaming programs. We have to remark that this library does an emulation of distributed memory. This emulation has an overhead which diminishes the maximum performance.

Figure 6.10 shows the scalability of two benchmarks in the prototype. This figure focuses on task parallelism scalability. The speedup of FMradio is up to 3.5 due to limitations on the FMradio tasks. There is a FFD task with a heavy unbalance which limits the maximum performance. Nokia's Wifi 802.11a presents a good performance. It is able to scale slightly better than the number of present cores. Both programs are initially programmed as plain C, and later transformed into a streaming programs by the compiler automatically. Prototype compiler and runtime

have performed all the transformation automatically. Resulting stream graphs of both applications corresponds to the expected from a manual creation of the streaming application.

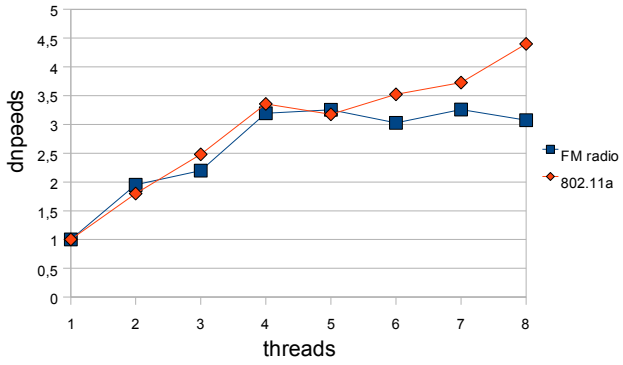


Figure 6.10: Stream programming model prototype scalability of FMradio and Nokia's Wifi 802.11a using only task and pipeline parallelism.

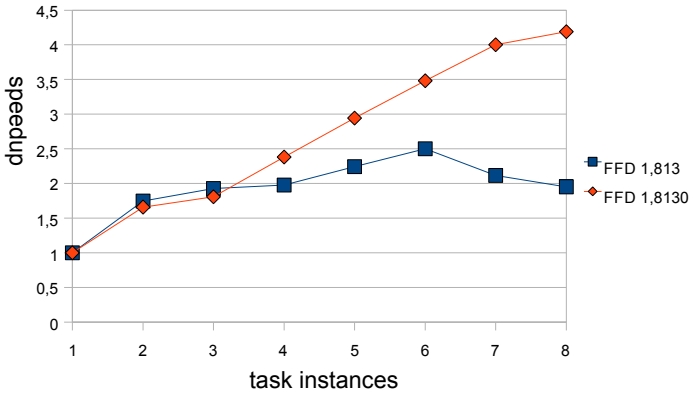


Figure 6.11: Stream programming model prototype scalability of the FFD filter using only data parallelism.

Figure 6.11 shows the scalability of the FFD filter task using data parallelism. FFD filter corresponds to the most expensive task from the FMradio benchmark. We have executed two versions of the FFD filter: 1,813 and 1,8310. These two versions are the same filter but with different parameters. First parameters are the original FMradio FFD filters parameters, second parameters are a more expensive configuration. This figure shows how the task is able to increase its performance by exploiting data parallelism. It increases the number of instances for each task. Each instance

processes a distinct set of elements in parallel. First configuration shows a limited scalability. This poor scalability is given by the overhead introduced by our run-time: the runtime is not designed to execute as fast as possible streaming programs but as a proof of concept. In order to verify that the scalability limitation is imposed by the overhead of the run-time, we increase the size of the filter parameters. Results shows on the second configuration that data parallelism is as effective on our stream programming model than on StreamIt.

Figure 6.12 shows the scalability of the FMradio using task, pipeline and data parallelism. This figure presents that the FMradio with data parallelism is able to use effectively all four available cores. We have enabled data parallelism on the FFD filter task, as a consequence the performance is improved. This figure is a conclusive demonstration that our stream programming model is as flexible as StreamIt and it can exploit the same three kinds of parallelism.

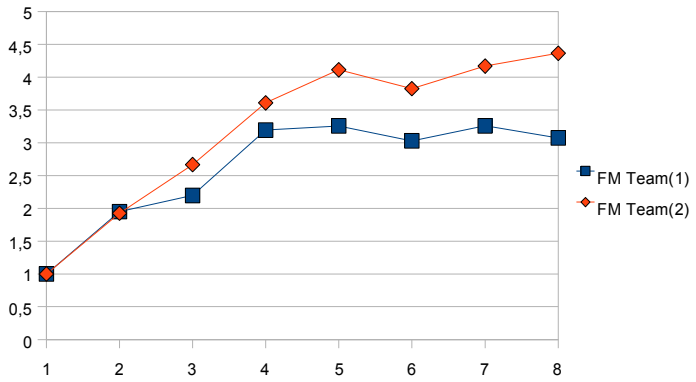


Figure 6.12: Stream programming model prototype scalability of the FMradio without data parallelism (1) and FMradio with data parallelism (2) both using task and pipeline parallelism.

Figure 6.13 shows traces of the execution of a stream program generated by our programming model. All of three traces use the same time scale. Light lines represents communication synchronizations through streams. We show three versions of the same program. The first version represents the unoptimized stream graph version. This version has cyclic communications (this is represented by crossed communication lines). There are some gaps in the execution, they are the operating system scheduler. The second version is the same program but with an optimised stream graph. In this case, communications have no cycles and we can see that tasks finishes as soon as they have computed all input elements. The third

version is a modification of the second. It takes advantage of no cycles and do blocking over communications. The third version sends a block of data instead a single element. It warps kernel invocation inside a for-loop which can be unrolled and vectorised. In addition, it also saves synchronization overhead by operating in groups of data. As a result, the third version has a very good performance.

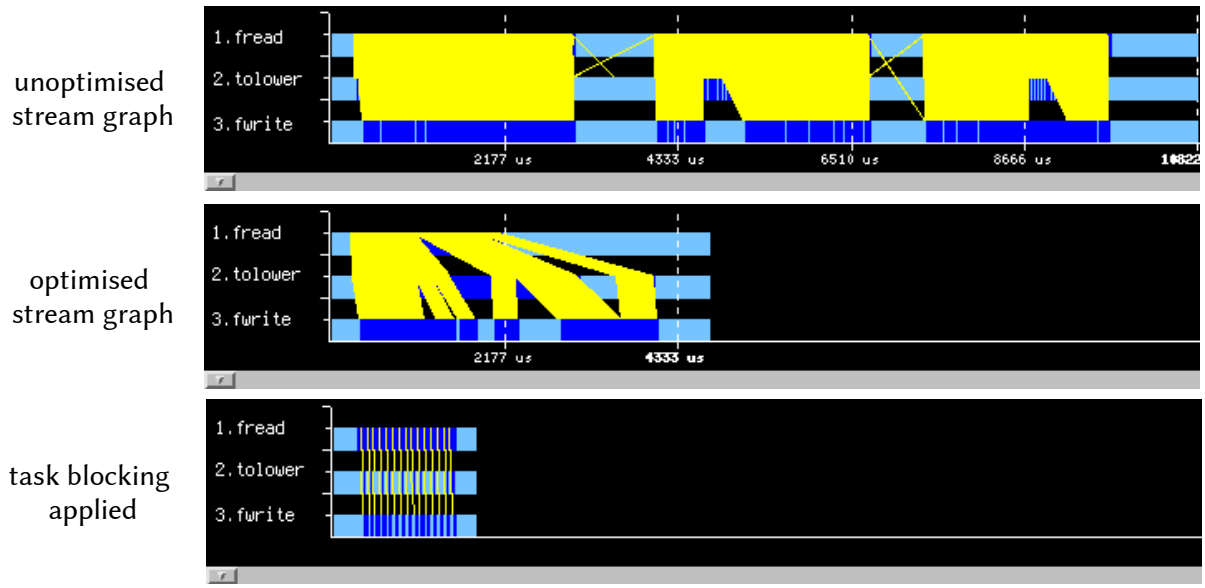


Figure 6.13: Paraver traces of the tolower benchmark as stream program.

Figure 6.14 shows the characteristics of the programming model by example. The first group explains the transformation of a C standard serial program into a streaming program. For each column, it shows the code, the task graph program diagram, and an example of trace obtained in the execution. In this trace, P1 line represents the time line of the main processor execution, and A1 and A2 lines represents the time line of each of available accelerators. Arrows represents communications between processors. Each column introduces a new concept for better understanding the programming model. The last column shows the manually graph optimisation. This step is explained separated by two reasons: 1) our prototype compiler has limited information about variables usage (so method for graph optimisation explained at [8] can not be applied automatically by our prototype) and 2) it allows to explain better how ports are connected and how the graph is optimised. Second part of the figure shows other features of the programming model and some examples. Two of the most important features are

stream peeking and data parallelism. Stream peaking allows to map an existing array into a stream buffer. The array size defines window size and the initial content of the stream buffer. It allows to save memory by reusing existing buffer, but it also helps to stabilise consumer/production ratio: in contrast to StreamIt, it does not require to fill the buffer with data from previous kernel activations. StreamIt by this behaviour breaks consumer/production ratios making them variable, StreamIt elements used to fill the buffer are not used to produce elements. Peek directive orchestrates all peek and pop operations: when the peek directive is reached the last element is effectively popped from the stream. Task fission helps to exploit data parallelism. In this case we also explain how to exploit data parallelism when there are state variables. Teamreplicate directive allows to replicate a computation in all instances of the same task in order to keep the state updated. We also show how two tasks with a different number of instances can distribute elements statically in order to have a good load balancing and not loosing elements order.

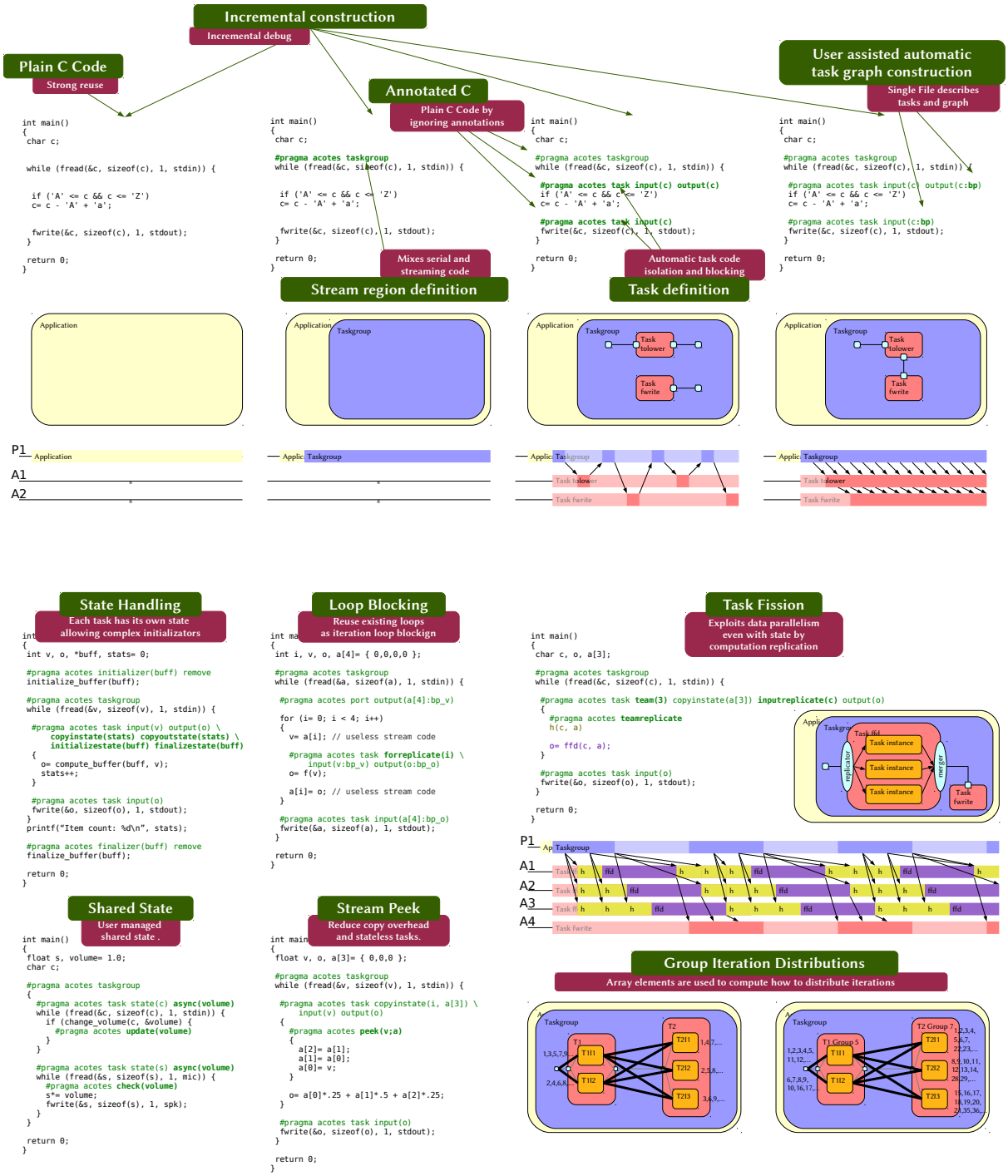


Figure 6.14: Streaming annotated programming model characteristics by example.

Section 6.5. Graph Matching On Current Architectures

This section shows results supporting contributions exposed on section 5.5. Exposed results and figures are extracted from publications [16] and [15]. In these results we show the performance and the scalability of three of our parallel algorithms. Results validate proposed parallel directives, optimisations, and the need of specialised kinds of parallelism depending the objective.

In [16] we study the parallelisation of the graduated assignment graph matching algorithm. In previous works, we have seen the ability to express parallelism on many architectures through OpenMP like annotations. In this work, we define a set of directives to parallelise the algorithm and a methodology to transform a serial algorithms into a parallel algorithms. We present two parallel algorithms of the same serial algorithms. One for large graphs, and another for many small graphs. Results shows that parallelisation for large graphs has a poor performance on small graphs. But the solution for small graphs requires multiple graphs, and it is restricted to small graphs due to limitations on GPGPU block memory.

We present the modification to the algorithm model for large graph parallelisation. We use loop tiling and loop reordering techniques. We apply these transformations to computation of the P and Q matrices presented in equation 3.4. Applied transformations give the same final values but with a different equations. The probability matrix P is computed as follows,

$$\begin{matrix} \forall_{c=0}^{R/B} \forall_{d=1}^{B} \forall_{k=0}^{R/B} \forall_{l=1}^{B} P^{pq}[a,i] = \exp\left(\beta Q_{ai}^{q}\right) \\ a = c \cdot B + d, i = k \cdot B + l \end{matrix} \tag{6.1}$$

we apply loop tiling to a and i loops obtaining loops c, d, k and l. This operation divides the use of matrices P and Q into sub-matrices of sizes $B \times B$.

We want to expose c and k loops for block parallelism, and d and l for thread parallelism. Block parallelism is performed on outer loops, thread parallelism in inner loops. We reorder c and k loops and d and l loops as follows:

$$\begin{matrix} \forall_{c=0}^{R/B} \forall_{k=0}^{R/B} \forall_{d=1}^{B} \forall_{l=1}^{B} P^{pq}[a,i] = \exp\left(\beta Q_{ai}^{q}\right) \\ a = c \cdot B + d, i = k \cdot B + l \end{matrix} \tag{6.2}$$

Now, it is time for matrix Q, we also apply loop tiling,

$$Q_{ai}^{q}=\sum_{e=0}^{R/B}\sum_{f=1}^{B}\sum_{u=0}^{R/B}\sum_{v=1}^{B}P^{pq}[b,j]\cdot C_{aibj}^{pq} \qquad a=c\cdot B+d,\ b=e\cdot B+f,\ i=k\cdot B+l,\ j=u\cdot B+v \tag{6.3}$$

we also reorder loops. In this case we want that inner loops match thread parallelism dimensions, so we move *f* and *v* loops into the inner level:

$$Q_{ai}^{q}=\sum_{e=0}^{R/B}\sum_{u=0}^{R/B}\sum_{f=1}^{B}\sum_{v=1}^{B}P^{pq}[b,j]\cdot C_{aibj}^{pq} \qquad a=c\cdot B+d,\ b=e\cdot B+f,\ i=k\cdot B+l,\ j=u\cdot B+v \tag{6.4}$$

we replace C_{aibj}^{pq} by its definition to obtain the following final expression:

$$Q_{ai}^{q}=\sum_{e=0}^{R/B}\sum_{u=0}^{R/B}\sum_{f=1}^{B}\sum_{v=1}^{B}P^{pq}[b,j]\cdot A_{ab}^{p}\cdot A_{ij}^{q}\cdot C_{ai}^{pq}\cdot C_{bj}^{pq} \qquad a=c\cdot B+d,\ b=e\cdot B+f,\ i=k\cdot B+l,\ j=u\cdot B+v \tag{6.5}$$

We have present a hybrid programming model based on two directives similar to OpenMP able to integrate semantics from OpenMP and CUDA language. As we have stated, CUDA parallel implementations have a duality of strategies: coarse grain parallelism for loosely coupled task, and fine grain parallelism for highly coupled tasks. The former targets block parallelism, the latter targets thread parallelism. Our parallel algorithm notation is designed to share the duality of strategies with CUDA, but using OpenMP like directives. Our hybrid *parallel for* directive can be used to define a coarse grain parallelism or a fine grain parallelism. From equation 6.5 we parallelise *e* and *u* loops in the outer level (coarse grain parallelism) and *f* and *v* loops in the inner level. We also add a *parallel fetch* directive in order to use block memory. We use this directive to accelerate the access tu sub-matrices of *A* and *C* matrices from equation 6.5 by using block memory. Semantics of our directives are the following:

#pragma hy parallel for [into(threads)] [reduction(OP:*r*)]
for $i_0 \le i \le i_f$ **do**
  ··· *for body*
end for

Hybrid p*arallel for* directive executes the following *for* construct in parallel. Each *for body* loop *i* iteration is executed in parallel. If *into(threads)* is not specified it is equivalent to OpenMP *parallel for* directive for the CPU, or it defines a CUDA logical space which *i* iterations are computed across blocks of the current grid (see figure 3.5) for a GPGPU. If *into(threads)* is specified, this directive is ignored on CPU, or it defines a CUDA logical space which *i* iterations across threads of the current block (see figure 3.5). If *reduction* clause is specified, a summarise *OP* operation is performed over *r* variable. Parallel directives can be nested in order to create a multi-dimensional logical execution space.

#pragma hy parallel fetch($m : s_1, \ldots, s_D : o_1, \ldots, o_D : i_1, \ldots, i_D$ [: $i_{\sigma(1)}, \ldots, i_{\sigma(D)}$] **)**
{ *statement* }

Hybrid *parallel fetch* is ignored on CPU and only has effect on a GPGPU. It allows to specify which data is copied from main memory (see figure 3.1) to block memory (see figures 3.5 and 3.4). All threads of the same block copy a sub-matrix of size $s_1 \times s_2 \times \ldots \times s_D$ from matrix *m*, which has D dimensions. This directive is inspired in *peek* directive from ACOTES programming model [28]. The following statement replaces all accesses to matrix *m* from origin indexes { o_1 , o_2 , ... , o_D } plus offset indexes { i_1 , i_2 , ... , i_D } by accesses to sub-matrix stored at block memory. Optionally, a set of permutation of indexes { $i_{\sigma(1)}$, $i_{\sigma(2)}$, ... , $i_{\sigma(D)}$ } can be specified for reordering dimensions of the sub-matrix stored at block memory. Synchronization are added in order to ensure that there is no data hazards. If statement contains any sub-matrix element modification, it flushes the whole sub-matrix to main memory.

We have evaluated the scalability of the parallel large graph matching. We have implemented the corresponding serial version of the algorithm and the CUDA version of the algorithm. We have tested serial algorithms over generic Intel multi-core architectures, and we have tested CUDA version over NVIDIA GPGPUs. We have selected architectures of different power consumptions in order to compare results for embedded computing. Table 6.1 shows detailed characteristics of our experiments (tests) for each architecture and algorithm. Generic processors are multi-core, but serial algorithms are only capable of use one core and one thread.

Table 6.1. List of algorithms and architectures evaluated.

Test	Version	Proc. / GPGPU	GHz	Power	Cores	Threads	Memory Bandwith
SC1	Serial	Intel Atom 330	1.6	8W	2	4	5 GB/s
SC2	CUDA	NVIDIA 9400M	1.1	10W	16	1536	5 GB/s
SC3	CUDA	NVIDIA 320M	0.97	14W	48	4608	10 GB/s
SC4	CUDA	NVIDIA 8800GT	1.67	>50W	112	10752	53 GB/s

For the serial version, we have implemented the original algorithm based on the algorithm 3.1. CUDA version is implemented based on the algorithm 3.1 replacing *Update* function by the algorithm 3.3, and *Normalise* function by the algorithm 3.4. We have tested many B sizes, best results for CUDA have been achieved with B=8. CUDA occupancy ratio is 67%. This is the implementation of NVIDIA hardware used for low power consumption.

We have used a synthetic graph generated test set based on the scalability experiments from [24]. The test set is composed by random graphs with a connectivity percentage between 10% and 50%. Each graph has cardinality from 16 to 1024 vertices. Given a randomly generated graph, the other graph to be compared to is obtained from it, the original is copied and modified by randomly changing the order of nodes, removing and adding edges, changing node values, and removing or adding some nodes.

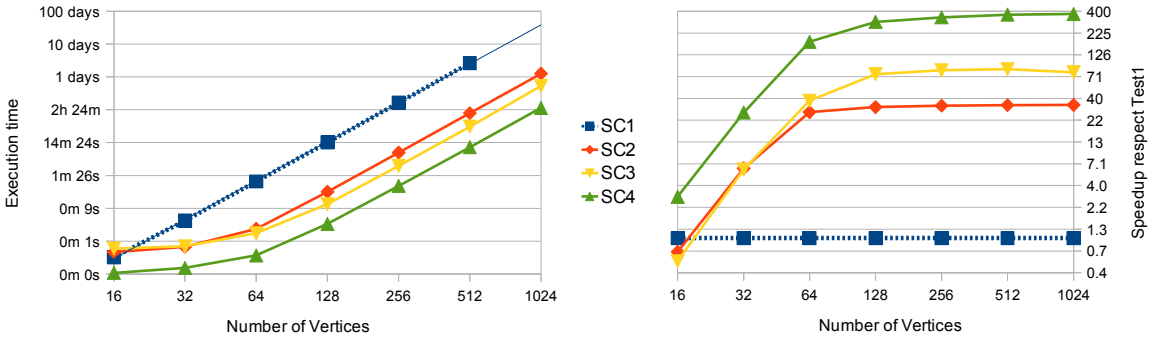


Figure 6.15: Run time of the 4 scalability tests respect to the number of vertices and speed-up of the parallel solutions (SC2, SC3, SC4) respect to the serial solution (SC1). Both plots vertical axis are in logarithmic scale.

Figure 6.15 shows the mean run time and speedup of the four scalability experiments SC1, SC2, SC3 and SC4 given different cardinalities of the graphs. We

have computed up to R=1024 nodes for all experiments but SC1. We have estimated that the execution of SC1 for R=1024 will take about 42 days of intensive computation. The speedup obtained for these benchmarks is 366 times faster for R=512. SC2 and SC3 presents a slowdown, when they compute small graphs, serial version is faster than parallel. SC4 has a speedup but it is not as good as expected for small R sizes. We observe that optimal speedups are achieved from R=64 nodes, below this size, results are far from maximum performance. SC2 and SC3 have a very close power consumption, as a result SC3 has better performance per watt. SC3 deteriorates the speedup with 1024 nodes because machine's operating system has a 5 seconds limitation for the parallel kernel execution time. On this test we have reconfigured kernels to work with smaller sets of data.

We have evaluated the performance of small graph multiple matching algorithm. We have implemented the corresponding serial version of the algorithm and the corresponding OpenMP and CUDA version of the algorithm. We have tested three versions over two machines. Serial and OpenMP are executed on the general purpose Intel processor, CUDA version is executed in the same machine but on the GPGPU processor. Table 6.2 shows detailed characteristics for each test given an architecture and algorithm. Generic processors are multi-core, but serial algorithms are only capable of use one core and one thread.

Table 6.2. List of algorithms and architectures evaluated.

Test	Version	Computer	Proc. / GPGPU	GHz	Power	Cores	Threads	Memory Bandwith
PF1	Serial	ViewSonic VT132	Intel Atom 330	1.6	8W	2	4	5 GB/s
PF2	OpenMP	ViewSonic VT132	Intel Atom 330	1.6	8W	2	4	5 GB/s
PF3	CUDA	ViewSonic VT132	NVIDIA 9400M	1.1	10W	16	1536	5 GB/s
PF4	Serial	NOX	Intel i7 950	3.0	130W	4	8	21 GB/s
PF5	OpenMP	NOX	Intel i7 950	3.0	130W	4	8	21 GB/s
PF6	CUDA	NOX	NVIDIA GT 430	1.4	49W	96	3072	21 GB/s

For the serial version we have implemented the original algorithm based on the algorithm 3.1 (shown at page 53). OpenMP and CUDA versions are implemented based on the serial version, but replacing *Update* function and *Normalise* function by its parrallelised versions using transformations previously presented. OpenMP version is only parallelised at block level, B constant is assigned to B=1. Loops of size B (d, l, f, v and s) are automatically removed by the compiler. CUDA version B

constant is kept to *B*=8. We have adapted executions at PF6 to emulate an NVIDIA with a CUDA compute capability 1.X (the corresponding architecture for low power consumption NVIDIA GPGPU).

We have used two databases in which nodes are defined over a two-dimensional domain that represents its plane position (x,y). Edges have binary attribute that represents the existence of a line between two terminal points. The first dataset is a subset of high noise level of the Letter dataset created the University of Bern [105]. This data set is composed of 15 classes and 150 graphs per class representing the Roman alphabet i.e. A, E, F, ..., X, Y, and Z. The second dataset, called GREC dataset, created at the Universitat Autònoma de Barcelona [105], is composed of 22 classes and 50 graphs per class representing symbols from architectural and electronic drawings. We have selected 3 random sets of elements from GREC dataset with more than 150 elements each plus a random set from LETTER dataset of more than 150 elements. Each set has random elements from all classes with a [8 ... 8], [13 ... 16] and [18 ... 24] nodes for GREC and [7 ... 8] nodes for LETTER. We have matched one graph against N $\in$ [5, 10, 15, 25, 50, 75, 100, 125, 150] random different elements for each set.

Figurs 6.16 shows the execution time and speedup respectively for the mean time of each databases, machine and number of graphs. This results show that OpenMP tests (PF2 and PF5) scaled around the number of cores. CUDA tests (PF3 and PF6) have a good performance even for small *R* values. PF3 execution has better performance that PF5, PF3 has better performance per watt. We also can see that with an *N* (number of graphs) around 15 we can extract almost the maximum performance. On PF6 we can observe that *N*=5, 10 and 15 have the same execution time. This is given because the number of available cores is higher than the number of threads, and, as a consequence, there are not enough parallelism to use all available resources. If we take execution times, the accumulative speedup from serial PF1 to CUDA PF6 for a large N is about x250. New results give a better performance with small *R* if we compare enough graphs. PF3 is able to compare 150 graphs of *R* close to 16 in 2.57 seconds and measured execution time from PF6 takes just 0.65 seconds.

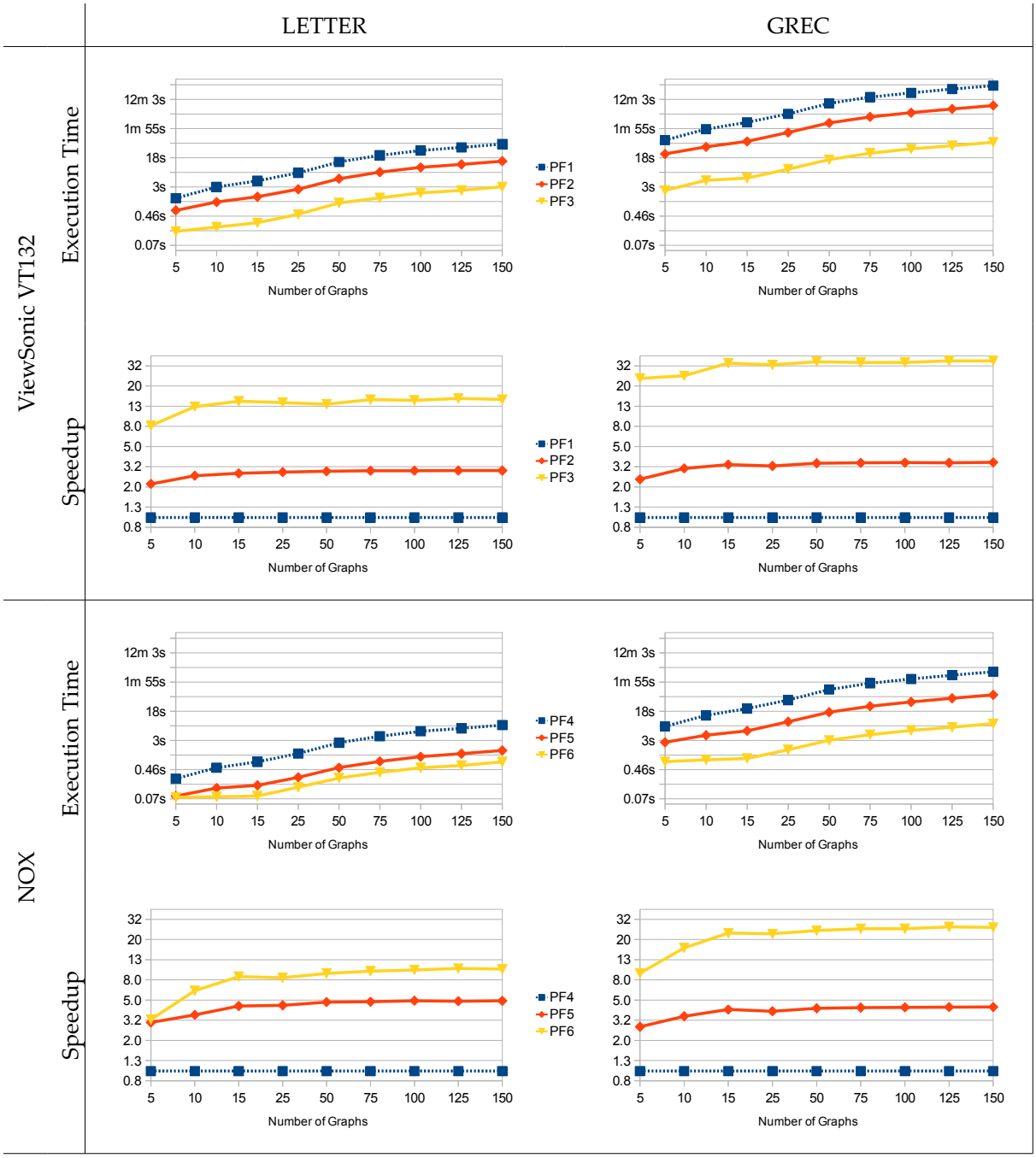


Figure 6.16: Run time and speedup of the small graph multiple matching algorithm given an architecture, a dataset and the number of graphs.

In [15] we study the parallelisation of the graduated assignment graph common labelling algorithm. We apply the same methodology and notation used in the works. In this work, we study the behaviour of a low power computer and its

scalability by using low power GPGPU. In addition to the previously presented methodology we also apply loop splitting in order to increase locality. We test the algorithm with two graph databases. Architecture used is the ViewSonic computer defined in table 3.6 (and table 6.2).

We have used the same two databases presented in the graph matching algorightm. The first dataset is a subset of high noise level of the Letter dataset created the University of Bern [105]. The second dataset is the GREC dataset, created at the Universitat Autònoma de Barcelona [105].

We have selected 5 classes of each dataset to compare execution speed. For each class we have randomly selected a number of graphs for N ∈ [5, 10, 15, 25, 50, 75, 100, 150] for Letter dataset, and N ∈ [5, 10, 15, 25, 50] for GREC dataset. Letter dataset classes selected are {1, 6, 8, 12, 13} each one with a mean number of nodes of {5.3, 5.3, 5.3, 5.4, 4.4}. GREC dataset classes selected are {5, 8, 14, 15, 21} each one with a mean of {19.4, 8.6, 12.7, 20.7, 17.14}.

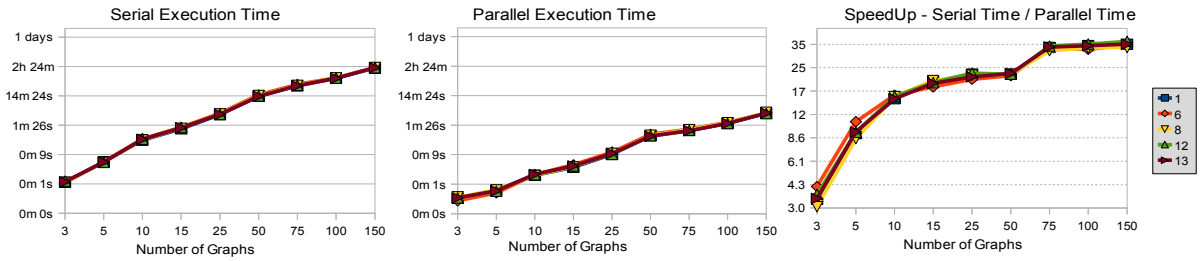


Figure 6.17: Letter run time of Serial and Parallel and speedup respect to the number of graphs for each selected class. Vertical axis are in log. scale.

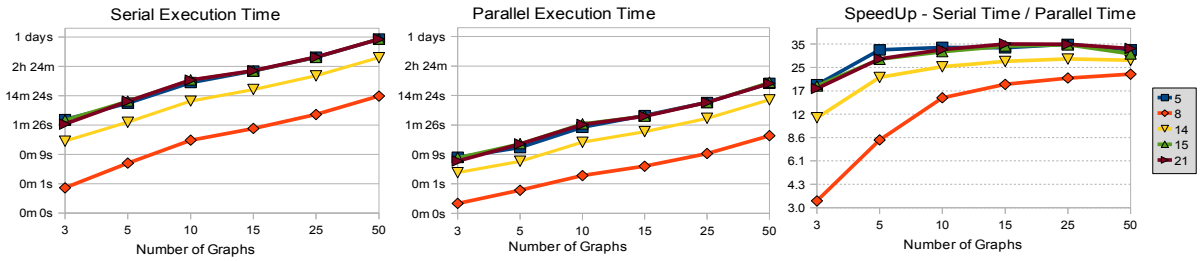


Figure 6.18: GREC run time of Serial and Parallel and speedup respect to the number of graphs for each selected class. Vertical axis are in log. scale.

Figure 6.17 shows the mean run time for each one of the five Letter dataset classes for serial and parallel algorithm experiments for a given different number of the graphs. Figure 6.18 shows the mean run time for each one of the five GREC dataset

classes for serial and parallel algorithm experiments for a given different number of the graphs. The obtained distance is not shown since the sequential and parallel algorithm obtains exactly the same result. It can be observed a clear improvement on the run time when the parallel algorithm is used.

Chapter 7. Community Results Based On This Thesis

The objective of any thesis, beyond to contribute with something new to its science, is to become useful knowledge to create further contributions. In this section, we are proud to present some works performed by the community which used our contributions as part of its basis. We analyse publications of the community, whose works points to our publications and use them as a basis.

Cuvillo et al. present at [106] a port of the OpenMP to IBM Cyclops 64. The IBM Cyclops 64 is the next generation of the IBM Cyclops. Unfortunately our OpenMP port was built on privative software and there was also some major changes between architectures. As a consequence they can not reuse our compiler neither our run-time. After our success with the performance of OpenMP they also decide to research in optimisations in order to reduce OpenMP overheads. They performed optimisations with either hardware or software. The main proposals to optimise were the use of scratch-pads, spin-locks and barriers. Scratch-pads solution emulates our hardware solution, we have placed stacks close to its physical threads to speedup memory accesses. Instead of creating special regions in the cache to map stacks close, they use scratch-pads (core private memory) to have a fast access. Spin-locks and barriers solves one of the problems that we have spotted: the increasing number of threads increases the memory accesses due to synchronizations primitives implemented over memory. They create specific hardware to make such synchronizations and consequently they avoid using memory for such mechanisms. As a result, they claim that all of three optimisations reduce in a 80% the overhead in OpenMP constructs

Meenderinck et al. at [107] use CellSim simulator to present a video decoder designed for a heterogeneous processor. They design a set of specialised assembler instructions in order to speedup video decoding algorithm. Their target is to modify or implement new SPEs. They take advantage of the GPL license of the CellSim and modifies the SPE module for two tasks: 1) to improve an algorithm profiling and 2) to implement and test their new instructions.

There is a work from Giorgi et al. [108] which uses our simulator infrastructure to validate their thesis. They expand the simulator in order to implement a multi-core scheduled data-flow processor. They perform the modification by adding three components: one distributed scheduling element to the CellSim PPE, a local scheduling element to the CellSim MFC, and a frame memory to the CellSim LS. The simulator allowed them to test their theory even without a great effort of implementation.

Azevedo and Juurlink presented at [109] a modification for the Cell B.E. architecture to implement software cache. The idea of the software cache is to emulate a shared-memory environment even if there is no shared-memory. They propose to implement a new instruction to make viable software cache. In order to demonstrate their theory, they modify the CellSim simulator. Their experiments execute against the Cell B.E. with only software support, and against the CellSim, with the experimental modification.

Ramirez et al. at [110] present TaskSim, a new kind of simulator for multi-core processors. When we were developing the CellSim simulator we realised that as more components and more detail we added, more time consuming was the simulation. This problem is common to all multi-core simulators: there is not a simulation of one processor, but many processors and its interconnection networks.

The modular heterogeneous simulator presented on this thesis was designed to work with modules, but with memory-accesses as the only interface. As we have said previously, we do not made any assumption about processors of its behaviour, and almost any kind of implementation was compatible. Our processor implementations were functional processors, and also detailed pipelined processors. But we were not limited. For example, the operating system emulation was not a functional processor neither a detailed pipeline processor.

TaskSim reduces the simulation to traces of accesses and synchronisations. The underlying idea is to acquire traces from working programs, post-process these traces to create tasks, memory accesses, synchronizations and dependences and assign each task to a possible simulator module. Each module uses task information to emulate memory accesses pattern, including waits for processing. Not all TaskSim modules are processors, it also implements buses, interconnection networks, and memories. In their presented work they use the k-bus as the main interconnection network. TaskSim does not use UNISIM due to its overhead.

TaskSim has been demonstrated that to work on memory accesses and base synchronisations on traces, it can create an excellent tool to do architecture design space exploration within a reasonable simulation time.

Task directive introduced in the streaming programming model was almost simultaneously proposed to the OpenMP [111] standard. This directive was effectively introduced in the OpenMP 3.0. Both teams, the one designing tasks for shared-memory and us designing streaming tasks, have been working closer and even sharing the same staff directors. Although its similarities we have two different starting points: they have basically renamed some already existing semantics of OpenMP and we have started from scratch in front of a whiteboard. The main difference between both task models is that OpenMP 3.0 tasks are based on run-time. Their execution model corresponds to one lifetime task. We, in contrast, define tasks as kernels kept alive from invocation to invocation. OpenMP tasks are dynamically created, as a consequence, each invocation needs to be scheduled and suffers for a great overhead. On the other hand, our tasks are designed to be statically created, and reused every time that it is required, which creates a more efficient behaviour. In addition, we decided to stablish the same directive name, because we believed that this difference in the model is only semantic and any compiler is free to choose the best implementation (as we have exposed at [12]).

We have found up to 5 publications following the steps of our stream programming model. Most of them adopt, or even completed, our model and suggest the required modification to the OpenMP standard to support stream programming or heterogeneous processors. As an interesting fact, they also base most of their benchmarks and proofs on the serial FMradio application contributed by this thesis.

The first proposal is performed by Pop and Pop at [112]. This publication is a proposal to the OpenMP organisation to slightly modify the OpenMP standard in order to create a streaming dependence between tasks that we have presented in our programming model. They plan to extend semantics for firstprivate and lastprivate clauses in order to enable stream creation. Presented solution in Pops propossal semantics are exactly the same OpenMP like stream programming model presented by us at [8]. They use the same semantics, graph description, execution time lines and examples. The only difference is to use firstprivate keyword instead of input keyword, and lastprivate keyword instead of output keyword. In our thesis we also claim that we have built our stream programming model intentionally close to OpenMP. Pop brothers suggested just a modification of three lines inside the OpenMP standard will support our programming model.

The same team that have proposed OpenMP 3.0 tasks has also made two proposals for stream-like task dependence [113] and heterogeneous support [114]. Both publications try to approximate OpenMP 3.0 capabilities to the capabilities pursued in our programming model, but they try a different approach using CellSS [115] as an underlying model. In the first work, they extend task directive with input and output clauses, but instead of defining dependences through variables and symbols, it defines dependences through memory addresses. This model is indeed more flexible, but as an important drawback, it forces to compute all dependences into the run-time, with the consequently execution time overhead. The second work proposes to annotate function declarations with architecture dependent information. Their objective is to substitute tasks invocation on the same processor, by invocations on specific processors or cores. This work is a work-around for the limitation of the OpenMP to require shared-memory. With its definition, which is the opposite of our proposals of requires and share clauses, they specify some functions which can be executed on non-shared memory hardware with specific requirements. But it also forces to the programmer to create enough versions of the same function for all possible target processors.

A team from INRIA have published also an extension of the OpenMP programming model [116] and a runtime library [117] for streaming. In many ways presented work is one more attempt to standardise our programming model with some improvements. Their programming model is implemented in a specific branch

of the GCC. Their suggested changes to the OpenMP only affects to the execution model (to follow our proposed task model, as Pops have done at [112]) but in addition they added a special semantic for variable ports description. Their addition is the possibility to use a special variable as a direct access to stream. Its specification is a mix between our peek directive and target port modifier. While our peek directive was designed to co-exists with serial programs, their extension breaks the compatibility with serial code (the program can not be longer compiled as a serial program by ignoring directives) and as a consequence they break the consumer/production ratios protection. We also want to remark that they use the same applications, our FMradio and restricted access Nokia WiFi application, to verify its expressiveness and results.

Chapter 8. Conclusions

Since the beginning of this thesis, desktop computers had become heterogeneous and extremely parallel. This new scenario opens an opportunity for computer vision algorithms. Many of the new complex architectures are focused on giving performance ratios tens or even hundreds of times better. The only gap to fill is what we have focused: the ability of the mainstream programmer to take advantage of existing architectures. We are proud to say that many computer vision algorithms are now able to run almost in real-time on desktop architectures.

This thesis was started with the intuition that multi-core would become common on desktop architectures. Consequently algorithms and programs must be rewritten in order to take advantage of new hardware. Our initial intuition has gone around the productivity concept, in this case, complexity of the design versus performance. Under this premise we had two opposite tendencies: on the one hand chip-makers fighting against chip complexity, and on the other hand algorithm programmers fighting against programs complexity. For a long time, commercial chip-makers focused on the acceleration of serial programming, but, when this strategy cost becomes so expensive, they started to demand parallelism from programmers.

Our target applications have been graph-matching algorithms. These algorithms present many opportunities inside computer vision due to representation. Unfortunately its computation cost was too expensive to be used on real applications. In this thesis, we have effectively reduced the computation execution time of these algorithms, and not only on desktop computers, but also on low power consumption systems. As a consequence we have achieved our main objective that was to make these algorithms available to field applications.

One of our main objectives is to keep a high degree of usability on multi-core architectures. We have focused on annotated programming model, mainly OpenMP adaptations, due to its simplicity and capacity to maintain code readable. Before this thesis, annotated programming models were limited to supercomputing. We have demonstrated that annotated programming models can be effectively used on multi-core processors, distributed-memory architectures and current desktop computers. We also have presented directives and a model able to transform a serial program into a streaming program.

Nowadays multi-core processors are present on most of our desktop computers and also annotated programming models. GCC has implemented support for OpenMP (gcc -fopenmp), and even market performance benchmarks use OpenMP applications for analyse new processors behaviours [118]. In this case, Phoronix reviews the Intel Core i7 990X Extreme Edition, its conclusions is better not to buy it if you cannot exploit all present cores (literally: "*so if you're workload can't efficiently take advantage of six or more threads, you'd be better off with a Core i5 2500K*"). This example shows the necessity to give access to parallelism for programmers.

One of the keys that we have considered to increase the usability is to create hardware in conjunction of programming models. We have developed a simulator in order to prove that small modifications on hardware architecture can help to programmers. Our objective was to show how to increase programmers capacity to obtain performance. The main cornerstone proposed is the capacity to have a global linearly addressable memory (in other words: global pointers), even if we have latency penalisation. Our best contribution was the memory-access protocol to interconnect all simulator modules. We have not demonstrated our initial objective, but the market has sown it to us: architectures like NVIDIA Tesla [49] or the Cell B.E. history [87] has shown that our intuition was correct. NVIDIA uses global memory pointers from its accelerators, and at the same time, it allows to make incremental modifications to take advantage of local memory. On the other hand, Cell B.E. has limited the access to main memory.

We have applied all acquired knowledge to adapt matching graph algorithms to current desktop computers. Current architectures shown a mix of characteristics from multi-processors (with multi-cores), heterogeneity (with GPGPUs with different characteristics), and even distributed-memory. All previous work were

confirmed. Based on our experience we have seen that an annotated programming model and a methodology based on mathematical transformations effectively helps to adapt algorithms. It is possible to exploit private memory of each GPGPU core to speedup executions. We have effectively increased the usability, and at the same time the capacity to extract performance, for programmers.

Section 8.1. Future Work

In this thesis we have focused on the usability of incoming architectures and they applicability over graph-matching algorithms. We have focused on OpenMP programming model due to its flexibility and performance. We have seen that OpenMP is suitable for multi-core, even on massive multi-core architectures. In addition, we also have seen that OpenMP semantics can be extended in order to expand its applicability to other architectures.

Our last work, and community results, have been spotted some core research which should have priority. They are nested parallelism techniques, loop tiling and loop reorder transformations, memory pre-fetching and input and output directives.

Nested parallelism have effectively improved performance. Separation of coarse-grain parallelism, where is a low coupled communication, from fine-grain parallelism (highly coupled communication) has proven to be effective. Even commercial programming models like CUDA have adapted this double parallelism model. We need to do further investigation on this kind of parallelism, it has been demonstrated to be very powerful, but we do not know if there are many applications which can accept them. One of our challenges is to find how to implement the Sinkhorn method [37] but using two levels of parallelism, in the same fashion that NPB-MZ are implemented.

We also have seen some extensions to the OpenMP. Input and output clauses have effectively converted serial programs into streaming programs. We have presented a full programming model, with many features, but we have still some problems to solve. The main problem is how to create many configurations of the same task inside a loop for (we believe that for_distribute clause can help), but also we should design a header definition or/and binary format in order to use and connect tasks embedded in libraries. On the other hand, in our last stages, we have considered

some extensions to the OpenMP to help to develop CUDA programs. We have started to use the Mercurium to implement some parts. We have seen that these directives helped to implement the CUDA versions of the programs, but also reduced the number of bugs introduced on the manual construction. We believe that a finished programming model can save a lot of time and headaches.

Introduced OpenMP like directives on our CUDA algorithms have been based in two concepts and memory management. Two concepts are map operation and reduction operation. Both operations are applied over loop constructions and basically provides information to the compiler about data dependences in the loop: if there is not data dependence, parallelism can be applied. The other directive is fetch, this directive reports to the compiler which variables would be accessed, and which ranges. We want to explore how to simplify the annotation of this information and how the memory directive can be converted into pre-fetching operations, and even stream operations between tasks, whether applicable. We also have seen that it is possible to modify automatically the dimensions of the local fetched variable in order to enhance data locality and coalescence. We have to study and detect under which circumstances it is possible to automatise and release the programmer from this responsibility.

Some of our algorithms transformations have used two specific techniques: loop tiling and loop reorder. There are many loop techniques applied automatically on mainstream compilers, even loop tiling. Loop tiling is usually applied automatically by parallelism directives: there are usually less processors than iterations, so each processor has a loop that executes a sub-range of iterations. This technique and loop reorder technique have been proven to be very efficient, and almost indispensable. We want to study how to mix previous directives with these transformations in order to let to the compiler make the transformation achieving a double objective: first to release the programmer from the manual transformation and bug introduction and second to preserve original algorithm structure and comprehension.

There is still a large path to follow and directives have been proved to be very useful. We expect that by studying better annotations and progressing in their applicability we will be able to transform better existing algorithms.

Chapter 9. References

[1] D. Ródenas et al., "Optimizing NANOS OpenMP for the IBM Cyclops multithreaded architecture," *19TH IEEE International Parallel & Distributed Processing Symposium (IPDPS 2005)*, 2005.

[2] D. Rodenas, X. Martorell, J. Costa, T. Cortes, and J. Labarta, "Running BT Multi-Zone on non-shared memory machines with OpenMP SDSM instead of MPI," *Proceedings of the XVI Jornadas de Paralelismo*, Sep. 2005.

[3] D. Ródenas et al., "Exploiting multilevel parallelism using OpenMP on a massive multithreaded architecture," *Journal of Embedded Computing*, vol. 2, p. 141–155, Apr. 2006.

[4] P. Carpenter, D. Rodenas, X. Martorell, A. Ramirez, and E. Ayguade, "Code generation for streaming applications based on an abstract machine description," *Universitat Politécnica de Catalunya, UPC-DAC-RR-CAP-2007-3*, 2007.

[5] F. Cabarcas, A. Rico, D. Rodenas, X. Martorell, A. Ramirez, and E. Ayguade, "A module-based cell processor simulator," *3rd HiPEAC Advanced Computer Architecture and Compilation for Embedded Systems. ISBN: 978-90-382-1127-5*, 2007.

[6] A. Rico, F. Cabarcas, D. Rodenas, X. Martorell, A. Ramirez, and E. Ayguade, "Implementation and validation of a Cell simulator using UNISIM," *3rd HiPEAC Industrial Workshop, IBM Haifa, Israel*, 2007.

[7] F. Cabarcas, A. Rico, D. Rodenas, X. Martorell, and A. Ramirez, "CellSim: A Validated Modular Heterogeneous Multiprocessor Simulator," *Proceedings of the XVII Jornadas de paralelismo*, Apr. 2007.

[8] P. Carpenter, D. Rodenas, X. Martorell, A. Ramirez, and E. Ayguadé, "A streaming machine description and programming model," in *Proceedings of the 7th international conference on Embedded computer systems: architectures, modeling, and simulation*, Berlin, Heidelberg, 2007, p. 107–116.

[9] P. Carpenter, D. Rodenas, A. Ramirez, X. Martorell, and E. Ayguade, "Code generation for streaming applications based on an abstract machine description." IST ACOTES Project Deliverable D2.2, May-2007.

[10] P. Carpenter, A. Ramirez, X. Martorell, D. Rodenas, and R. Ferrer, "Report on Streaming Programming Model and Abstract Streaming Machine Desription 1st version." IST ACOTES Project Deliverable D2.1, Sep-2007.

[11] F. Cabarcas, A. Rico, D. Rodenas, X. Martorell, A. Ramirez, and E. Ayguade, "CellSim: A Cell Processor Simulation Infrastructure," *4th HiPEAC Advanced Computer Architecture and Compilation for Embedded Systems. ISBN: 978-90-382-1288-3*, 2008.

[12] D. Rodenas, R. Ferrer, X. Martorell, and E. Ayguade, "ACOTES Stream Programming Model," *4th HiPEAC Advanced Computer Architecture and Compilation for Embedded Systems. ISBN: 978-90-382-1288-3*, pp. 19-22, Jul. 2008.

[13] P. Carpenter, A. Ramirez, X. Martorell, D. Rodenas, and R. Ferrer, "Report on Streaming Programming Model and Abstract Streaming Machine Desription Final version." IST ACOTES Project Deliverable D2.2, Sep-2008.

[14] D. Rodenas, F. Serratosa, and A. Solé-Ribalta, "Graph Matching on a Low-cost & Parallel Architecture," *Iberian Conference on Pattern Recognition and Image Analysis, IbPRIA 2011, LNCS 6669*, vol. 2011, p. 508–515, 2011.

[15] D. Rodenas, F. Serratosa, and A. Solé-Ribalta, "Parallel Graduated Assignment Algorithm for Multiple Graph Matching based on a Common Labelling," *Graph based Representations, GbR2011, Münster, Germany, LNCS 6658*, pp. 164-174.

[16] D. Rodenas, F. Serratosa, and A. Solé-Ribalta, "Massive Parallel Graduated Assignment Graph Matching Experiences on Low Power Architectures," *Submited to IJPRAI*, May. 2011.

[17] K. Asanovic et al., "The Landscape of Parallel Computing Research: A View from Berkeley." EECS Department, University of California, 2006.

[18] D. Geer, "Industry Trends: Chip Makers Turn to Multicore Processors," *Computer*, vol. 38, p. 11–13, May. 2005.

[19] E. Rotem et al., "Power and Thermal Management in the Intel Core Duo Processor," *Intel Technology Journal*, vol. 10, no. 2, pp. 109-122.

[20] J. Owens, "Streaming architectures and technology trends," in *ACM SIGGRAPH 2005 Courses*, New York, NY, USA, 2005.

[21] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM Journal of Research and Development*, vol. 49, p. 589–604, Jul. 2005.

[22] C. R. Johns and D. A. Brokenshire, "Introduction to the cell broadband engine architecture," *IBM Journal of Research and Development*, vol. 51, p. 503–519, Sep. 2007.

[23] Sergey Melnik, Hector Garcia-molina, and Erhard Rahm, "Similarity flooding: A versatile graph matching algorithm." 2002.

[24] S. Gold and A. Rangarajan, "A Graduated Assignment Algorithm for Graph Matching," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 18, no. 4, pp. 377-388, 1996.

[25] B. Bonev, F. Escolano, M. A. Lozano, P. Suau, M. A. Cazorla, and W. Aguilar, "Constellations and the unsupervised learning of graphs," in *Proceedings of the 6th IAPR-TC-15 international conference on Graph-based representations in pattern recognition*, Berlin, Heidelberg, 2007, p. 340–350.

[26] A. Mendelson, "How many cores are too many cores?," *3rd HiPEAC Industrial Workshop, IBM Haifa, Israel*, 2007.

[27] D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface*. Morgan Kaufmann, 2007.

[28] B. Armstrong and R. Eigenmann, "Application of Automatic Parallelization to Modern Challenges of Scientific Computing Industries," in *Proceedings of the 2008 37th International Conference on Parallel Processing*, Washington, DC, USA, 2008, p. 279–286.

[29] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: a compiler framework for automatic translation and optimization," *ACM SIGPLAN Notices*, vol. 44, p. 101–110, Feb. 2009.

[30] T. A. Johnson, S.-I. Lee, S.-J. Min, and R. Eigenmann, "Can transactions enhance parallel programs?," in *Proceedings of the 19th international conference on Languages and compilers for parallel computing*, Berlin, Heidelberg, 2007, p. 2–16.

[31] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, "Cell broadband engine architecture and its first implementation: a performance view," *IBM Journal of Research and Development*, vol. 51, p. 559–572, Sep. 2007.

[32] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," *Computer*, vol. 41, p. 33–38, Jul. 2008.

[33] R. Strzodka, M. Droske, and M. Rumpf, "Image Registration by a Regularized Gradient Flow. A Streaming Implementation in DX9 Graphics Hardware," *Computing*, vol. 73, p. 373–389, Nov. 2004.

[34] J. Dongarra, T. Sterling, H. Simon, and E. Strohmaier, "High-Performance Computing: Clusters, Constellations, MPPs, and Future Directions," *Computing in Science and Engineering*, vol. 7, no. 2, pp. 51-59, 2005.

[35] A. Solé-Ribalta and F. Serratosa, "On the Computation of the Common Labelling of a Set of Attributed Graphs," in *Proceedings of the 14th Iberoamerican Conference on Pattern Recognition: Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, Berlin, Heidelberg, 2009, p. 137–144.

[36] A. Solé-Ribalta and F. Serratosa, "Graduated Assignment Algorithm for Multiple Graph Matching based on a Common Labelling. Structural," *Syntactic, and Statistical Pattern Recognition LNCS*, vol. 6218, pp. 180-190, 2010.

[37] R. Sinkhorn, "A Relationship Between Arbitrary Positive Matrices and Doubly Stochastic Matrices," *The Annals of Mathematical Statistics*, vol. 35, no. 2, pp. 876-879, Jun. 1964.

[38] H. W. Kuhn, "The Hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, no. 1-2, pp. 83-97, Mar. 1955.

[39] D. H. Bailey et al., "The Nas Parallel Benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63 -73, 1991.

[40] H. Jin, H. Jin, M. Frumkin, M. Frumkin, J. Yan, and J. Yan, "The OpenMP Implementation of NAS Parallel Benchmarks and its Performance," 1999.

[41] R. F. van der Wijngaart and J. Haopiang, *NAS Parallel Benchmarks, Multi-Zone Versions*. NASA Ames Research Center, 2003.

[42] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow, "The NAS Parallel Benchmarks 2.0." 1995.

[43] E. Blossom, "GNU Radio: Tools for Exploring the Radio Frequency Spectrum," *Linux Journal*, Jun. 2004.

[44] streamit@lists.csail.mit.edu, "StreamIt Cookbook." .

[45] G. Almasi et al., "Dissecting Cyclops: A Detailed Analysis of a Multithreaded Architecture," *Sigarch Comput. Archit. News*, vol. 31, p. 2003, 2002.

[46] C. C. Jose et al., "Evaluation of a Multithreaded Architecture for Cellular Computing," *In Procidings of the 8th International Symposium on High Performance Computer Architecture*, p. 311--322, 2002.

[47] F. Allen et al., "Blue Gene: a vision for protein science using a petaflop supercomputer," *IBM Systems Journal*, vol. 40, p. 310–327, Feb. 2001.

[48] E. Strohmaier, H. W. Meuer, J. Dongarra, and H. D. Simon, "TOP500 Supercomputers for June 2005." eScholarship Repository, 22-Jun-2005.

[49] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39-55, 2008.

[50] J. V. Last, "Playing the Fool," *Wall Street Journal*, 31-Dec-2008.

[51] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales, "Altivec extension to powerpc accelerates media processing." 22-Jan-2010.

[52] P. Bohrer et al., "Mambo: a full system simulator for the PowerPC architecture," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, p. 8–12, Mar. 2004.

[53] J. Clabes et al., "Design and implementation of the POWER5™ microprocessor," in *Proceedings of the 41st annual Design Automation Conference*, New York, NY, USA, 2004, p. 670–672.

[54] OpenMP Organization, *OpenMP Fortran Application Interface*. 2000.

[55] M. Gonzàlez, E. Ayguadé, X. Martorell, J. Labarta, N. Navarro, and J. Oliver, "NanosCompiler: supporting flexible multilevel parallelism exploitation in OpenMP," *Concurrency: Practice and Experience*, vol. 12, no. 12, pp. 1205-1218, Oct. 2000.

[56] C. D. Polychronopoulos, M. B. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten, "Parafrase-2: an environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors," *International Journal of High Speed Computing*, vol. 1, p. 45–72, Apr. 1989.

[57] X. Martorell, J. Labarta, N. Navarro, and E. Ayguade, "A Library Implementation of the Nano-Threads Programming Model," *IN EURO-PAR'96*, vol. 2, p. 644--649, 1996.

[58] M. Gonzalez, J. Oliver, X. Martorell, E. Ayguade, J. Labarta, and N. Navarro, "OpenMP Extensions for Thread Groups and Their Run-time Support," *In Workshop On Languages And Compilers For Parallel Computing*, p. 317--331.

[59] X. Martorell, E. Ayguadé, N. Navarro, J. Corbalán, M. González, and J. Labarta, "Thread Fork/Join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors," *In Numa MultiProcessors. In 13th Int. Conference On SuperComputing ICS'99, Rhodes*, p. 294--301, 1999.

[60] F. Martinez et al., "Evaluation of OpenMP for the Cyclops Multithreaded Architecture," 2003.

[61] V. Pillet et al., "PARAVER: A Tool to Visualize and Analyze Parallel Code," *IN WOTUG-18*, p. 17--31, 1995.

[62] "MPI 1.1 Standard." .

[63] J. J. Costa, T. Cortes, X. Martorell, E. Ayguade, and J. Labarta, "Running OpenMP applications efficiently on an everything-shared SDSM," *Journal of Parallel and Distributed Computing*, vol. 66, no. 5, pp. 647 - 658, 2006.

[64] David August et al., "UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development." 2007.

[65] "SPE Runtime Management Library." CBEA JSRE Series Cell Broadband Engine Architecture Joint Software Reference Environment Series.

[66] "Compute Unified Device Architecture Programming Guide." NVIDIA: Santa Clara, CA.

[67] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen, "Simultaneous Multithreading: A Platform for Next-Generation Processors," *IEEE Micro*, vol. 17, p. 12–19, Sep. 1997.

[68] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: maximizing on-chip parallelism," in *ACM SIGARCH Computer Architecture News*, New York, NY, USA, 1995, p. 392–403.

[69] N. R. Fredrickson, A. Afsahi, and Y. Qian, "Performance characteristics of openMP constructs, and application benchmarks on a large symmetric multiprocessor," in *Proceedings of the 17th annual international conference on Supercomputing*, New York, NY, USA, 2003, p. 140–149.

[70] W. Yamamoto and M. Nemirovsky, "Increasing superscalar performance through multistreaming," in *Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques*, Manchester, UK, UK, 1995, p. 49–58.

[71] W. Yamamoto, M. J. Serrano, A. R. Talcott, R. C. Wood, and M. Nemirosky, "Performance estimation of multistreamed, superscalar processors," in *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, Wailea, HI, USA, pp. 195-204.

[72] R. Thekkath and S. J. Eggers, "The effectiveness of multiple hardware contexts," in *ACM SIGPLAN Notices*, New York, NY, USA, 1994, vol. 28, p. 328–337.

[73] M. Gulati and N. Bagherzadeh, "Performance Study of a Multithreaded Superscalar Microprocessor," in *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, Washington, DC, USA, 1996, p. 291–.

[74] L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Computational Science & Engineering*, vol. 5, p. 46–55, Jan. 1998.

[75] M. Gonzalez, X. Martorell, E. Ayguade, and G. Jost, "Employing nested OpenMP for the parallelization of multi-zone computational fluid dynamics applications," *Journal of Parallel and Distributed Computing*, vol. 66, no. 5, pp. 686-697, May. 2006.

[76] M. Gonzàlez, E. Ayguadé, X. Martorell, J. Labarta, N. Navarro, and J. Oliver, "NanosCompiler: supporting flexible multilevel parallelism exploitation in OpenMP," *Concurrency: Practice and Experience*, vol. 12, no. 12, pp. 1205-1218, Oct. 2000.

[77] H. W. Meuer, "The TOP500 Project: Looking Back Over 15 Years of Supercomputing Experience," *Informatik-Spektrum*, vol. 31, no. 3, pp. 203-222, Apr. 2008.

[78] H. Lu, Y. C. Hu, and W. Zwaenepoel, "OpenMP on Networks of Workstations," 1998.

[79] M. Sato, M. S. Shigehisa, K. Kusano, and Y. Tanaka, "Design of OpenMP Compiler for an SMP Cluster," *IN EWOMP '99*, p. 32--39, 1999.

[80] C. Amza et al., "TreadMarks: Shared Memory Computing on Networks of Workstations," *IEEE COMPUTER*, vol. 29, p. 18--28, 1996.

[81] J. L. Lo, S. J. Eggers, H. M. Levy, S. S. Parekh, and D. M. Tullsen, "Tuning Compiler Optimizations for Simultaneous Multithreading," *IN INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE*, p. 114--124, 1997.

[82] A. Basumallik and R. Eigenmann, "Towards automatic translation of OpenMP to MPI," in *Proceedings of the 19th annual international conference on Supercomputing*, New York, NY, USA, 2005, p. 189–198.

[83] I. Rodero et al., "eNANOS: Coordinated Scheduling in Grid Environments." 2005.

[84] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes," in *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, Los Alamitos, CA, USA, 2009, vol. 0, pp. 427-436.

[85] F. Cappello and D. Etiemble, "MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks," in *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Washington, DC, USA, 2000.

[86] J. Balart, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta, "Runtime address space computation for SDSM systems," in *Proceedings of the 19th international conference on Languages and compilers for parallel computing*, Berlin, Heidelberg, 2007, p. 330–344.

[87] J. Stokes, "End of the line for IBM's Cell," *ars technica*, 23-Nov-2009.

[88] T. Grötker, S. Liao, G. Martin, and S. Swan, *System design with SystemC*. Springer, 2002.

[89] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 Simulator: Modeling Networked Systems," *IEEE Micro*, vol. 26, p. 52–60, Jul. 2006.

[90] S. Sato, N. Fujieda, A. Moriya, and K. Kise, "SimCell: A Processor Simulator for Multi-Core Architecture Research," *IPSJ Online Transactions*, vol. 2, pp. 81-92, 2009.

[91] B. Black and J. P. Shen, "Calibration of Microprocessor Performance Models," *Computer*, vol. 31, p. 59–65, May. 1998.

[92] D. Jimenez-Gonzalez, X. Martorell, and A. Ramirez, "Performance Analysis of Cell Broadband Engine for High Memory Bandwidth Applications," *Performance Analysis of Systems and Software, IEEE International Symmposium on*, vol. 0, pp. 210-219, 2007.

[93] S. Girona, J. Labarta, and R. M. Badia, "Validation of Dimemas Communication Model for MPI Collective Operations," in *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, London, UK, 2000, p. 39–46.

[94] NXP, STMicroelectronics, and Nokia, "EU-funded project to research advantages of parallel computing in consumer devices."

[95] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow."

[96] A. Pop, S. Pop, H. Jagasia,, J. Sjödin, and P. H. J. Kelly, "Improving GNU Compiler Collection Infrastructure for Streamization Antoniu Pop," *GCC Developers' Summit*, 2008.

[97] Ben Serebrin et al., "A Stream Processor Development Platform." 2002.

[98] U. J. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany, "The Imagine Stream Processor," in *Computer Design, International Conference on*, Los Alamitos, CA, USA, 2002, vol. 0, p. 282.

[99] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *ACM SIGOPS Operating Systems Review*, New York, NY, USA, 2006, vol. 40, p. 151–162.

[100] M. Gonzalez, E. Ayguade, X. Martorell, and J. Labarta, "Complex Pipelined Executions in OpenMP Parallel Applications," *IN: PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING*, p. 295--304.

[101] R. Sirvent, R. M. Badia, J. Labarta, J. M. Pérez, J. M. Cela, and R. Grima, "Programming Grid Applications with GRID Superscalar," *Journal of Grid Computing*, vol. 1, no. 2, pp. 151-170, 2003.

[102] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: a programming model for the cell BE architecture," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 2006.

[103] "ACOTES presentation at HiPEAC 6th General cluster meeting," 16-Oct-2006.

[104] M. Harris, "Optimizing Parallel Reduction in CUDA," *NVIDIA Developer Technology*.

[105] K. Riesen and H. Bunke, "IAM Graph Database Repository for Graph Based Pattern Recognition and Machine Learning," in *Proceedings of the 2008 Joint IAPR International Workshop on Structural, Syntactic, and Statistical Pattern Recognition*, Berlin, Heidelberg, 2008, p. 287–297.

[106] J. del Cuvillo, W. Zhu, and G. Gao, "Landing openMP on cyclops-64: an efficient mapping of openMP to a many-core system-on-a-chip," in *Proceedings of the 3rd conference on Computing frontiers*, New York, NY, USA, 2006, p. 41–50.

[107] C. Meenderinck and B. Juurlink, "A Chip MultiProcessor Accelerator for Video Decoding."

[108] Roberto Giorgi, Nikola Puzovic, and Zdravko Popovic, "Implementing DTA support in CellSim." 04-Nov-2009.

[109] A. Azevedo and B. Juurlink, "An Instruction to Accelerate Software Caches," in *Architecture of Computing Systems - ARCS 2011*, vol. 6566, M. Berekovic, W. Fornaciari, U. Brinkschulte, and C. Silvano, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 158-170.

[110] A. Ramirez et al., "The SARC Architecture," *IEEE Micro*, vol. 30, p. 16–29, Sep. 2010.

[111] E. Ayguadé et al., "A Proposal for Task Parallelism in OpenMP," in *Proceedings of the 3rd international workshop on OpenMP: A Practical Programming Model for the Multi-Core Era*, Berlin, Heidelberg, 2008, p. 1–12.

[112] A. Pop and S. Pop, "A Proposal for lastprivate Clause on OpenMP task Pragma." 2009.

[113] A. Duran, R. Ferrer, E. Ayguadé, R. M. Badia, and J. Labarta, "A proposal to extend the OpenMP tasking model with dependent tasks," *International Journal of Parallel Programming*, vol. 37, p. 292–305, Jun. 2009.

[114] E. Ayguade et al., "A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures," in *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, Berlin, Heidelberg, 2009, p. 154–167.

[115] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: a programming model for the cell BE architecture," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 2006.

[116] A. Pop and A. Cohen, "A stream-computing extension to OpenMP," in *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, New York, NY, USA, 2011, p. 5–14.

[117] C. Miranda, A. Pop, P. Dumont, A. Cohen, and M. Duranton, "Erbium: a deterministic, concurrent intermediate representation to map data-flow tasks to scalable, persistent streaming processes," in *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems*, New York, NY, USA, 2010, p. 11–20.

[118] M. Larabel, "Intel Core i7 990X Extreme Review," 06-Apr-2011.