Towards Resource-Aware Computing for Task-Based Runtimes and Parallel Architectures

Dimitrios Chasapis

Advisors: Dr. Marc Casas Guix
Dr. Miquel Moretó Planas

Department of Computer Architecture
Universitat Politècnica de Catalunya

This dissertation is submitted for the degree of
Doctor of Philosophy

March 2019

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola de Doctorat

Curso académico:

Acta de calificación de tesis doctoral

Nombre y apellidos

___

Programa de doctorado

___

Unidad estructural responsable del programa

___

Resolución del Tribunal

Reunido el Tribunal designado a tal efecto, el doctorando / la doctoranda expone el tema de la su tesis doctoral titulada ____

____.

Acabada la lectura y después de dar respuesta a las cuestiones formuladas por los miembros titulares del tribunal, éste otorga la calificación:

☐ NO APTO ☐ APROBADO ☐ NOTABLE ☐ SOBRESALIENTE

(Nombre, apellidos y firma) Presidente/a		(Nombre, apellidos y firma) Secretario/a
(Nombre, apellidos y firma) Vocal	(Nombre, apellidos y firma) Vocal	(Nombre, apellidos y firma) Vocal

______________________, _______ de __________________ de _______________

El resultado del escrutinio de los votos emitidos por los miembros titulares del tribunal, efectuado por la Escuela de Doctorado, a instancia de la Comisión de Doctorado de la UPC, otorga la MENCIÓN CUM LAUDE:

☐ SÍ ☐ NO

(Nombre, apellidos y firma) Presidente de la Comisión Permanente de la Escuela de Doctorado	(Nombre, apellidos y firma) Secretario de la Comisión Permanente de la Escuela de Doctorado

Barcelona a _______ de ____________________ de __________

Abstract

Current large scale systems show increasing power demands, to the point that it has become a huge strain on facilities and budgets. The increasing restrictions in terms of power consumption of High Performance Computing (HPC) systems and data centers have forced hardware vendors to include power capping capabilities in their commodity processors. Power capping opens up new opportunities for applications to directly manage their power behavior at user level. However, constraining power consumption causes the individual sockets of a parallel system to deliver different performance levels under the same power cap, even when they are equally designed, which is an effect caused by manufacturing variability. Modern chips suffer from heterogeneous power consumption due to manufacturing issues, a problem known as manufacturing or process variability. As a result, systems that do not consider such variability caused by manufacturing issues lead to performance degradations and wasted power. In order to avoid such negative impact, users and system administrators must actively counteract any manufacturing variability.

In this thesis we show that parallel systems benefit from taking into account the consequences of manufacturing variability, in terms of both performance and energy efficiency. In order to evaluate our work we have also implemented our own task-based version of the PARSEC benchmark suite. This allows to test our methodology using state-of-the-art parallelization techniques and real world workloads. We present two approaches to mitigate manufacturing variability, by power redistribution at runtime level and by power- and variability-aware job scheduling at system-wide level. A parallel runtime system can be used to effectively deal with this new kind of performance heterogeneity by compensating the uneven effects of power capping. In the context of a NUMA node composed of several multi-core sockets, our system is able to optimize the energy and concurrency levels assigned to each socket to maximize performance. Applied transparently within the parallel runtime system, it does not require any programmer interaction like changing the application source code or manually reconfiguring the parallel system. We compare our novel runtime analysis with an offline approach and demonstrate that it can achieve equal performance at a fraction of the cost. The next approach presented in this theis, we show that it is possible to predict the impact of this variability on specific applications by using variability-aware power prediction models. Based on these power models, we propose two job scheduling policies that consider the effects of manufacturing variability for each application and that ensures that power consumption stays under a system wide power budget. We evaluate our policies under different power budgets and traffic scenarios, consisting of both single- and multi-node parallel applications.

To my family and friends.

I do not know what I may appear to the world, but to myself I seem to have been only like a boy playing on the sea-shore, and diverting myself in now and then finding a smoother pebble or a prettier shell than ordinary, whilst the great ocean of truth lay all undiscovered before me.

— Isaac Newton

Acknowledgements

I would like to extend my gratitude and appreciation to my advisors Dr. Marc Casas and Dr. Miquel Moretó, without their patience and guidance this thesis would not have been possible. I would also like to thank Dr. Martin Schulz and Dr. Barry Rountree who gave me the opportunity to visit and work at Lawrence Livermore National Laboratory. Their collaboration has been invaluable for the completion of this thesis.

Special thanks to Dr. Xavier Martorell, Dr. Julita Corbala and Dr. Paul Carpenter for reviewing this document and for their invaluable feedback.

Thanks are due of course to my collegues at Barcelona Supercomputing Center, for their support (technical or not), especially to the RoMoL team, Dr. Mateo Valero, Dr. Jesus Labarta, Dr. Eduard Ayguadé, Xubin Tan, Luc Jaulmes, Vladimir Dimic, Emilio Castillo, Kallia Chronaki, Isaac Sanchez and Lluc Alvarez.

This thesis has been supported by the Spanish Government (Severo Ochoa grants SEV2015-0493, SEV-2011-00067), by the Spanish Ministry of Science and Innovation (contracts TIN2015-65316-P), by Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272), by the RoMoL ERC Advanced Grant (GA 321253) and the European HiPEAC Network of Excellence. This work was also partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-689878).

Table of contents

1 Introduction **1**
1.1 Thesis Objectives and Contributions . . . 4
1.1.1 Benchmarking with Realistic Workloads . . . 4
1.1.2 Variability-Aware Load-balancing at Runtime Level . . . 5
1.1.3 Power Variability Prediction-driven Job Scheduling . . . 5
1.2 Thesis Structure . . . 7

2 Background **9**
2.1 Parallel Systems . . . 9
2.1.1 Shared and Distributed Memory Systems . . . 10
2.1.2 HPC Cluster Design . . . 11
2.2 Parallel Programming Models . . . 12
2.2.1 Shared and Distributed Parallel Memory Models . . . 12
2.2.2 Synchronization and Parallel Programming Challenges . . . 13
2.2.3 Asynchronous Tasks and Dataflow Model . . . 14
2.2.4 Parallel Runtime Systems . . . 16
2.3 Managing HPC Clusters . . . 18
2.3.1 Job Scheduling . . . 18
2.3.2 Resource Management . . . 19
2.4 Manufacturing Variability . . . 19
2.4.1 Impact of Manufacturing Variability in HPC Systems . . . 20
2.5 Variability-Aware Power Management in HPC Systems . . . 20
2.5.1 Software-aided Power Constraining and Management . . . 21
2.5.2 Power Prediction Models . . . 22
2.5.3 Power Aware System-Wide Job Scheduling . . . 23

3 Experimental Setup **27**
3.1 Hardware Platforms . . . 27
3.2 Software Stack . . . 28
3.2.1 Runtime System . . . 28

3.2.2 Analysis tools and Power Capping Framework 28
3.3 Workload Manager Simulator 29
3.4 Benchmark Applications 31
3.4.1 Prediction Model Training 31
3.4.2 Runtime and Job Scheduler Evaluation Benchmarks 31
3.4.3 Job Scheduler Workload Generation 32
3.4.4 Configuration Exploration Space 33
3.4.5 General Considerations 33

4 Realistic Task-based Workloads 35
4.1 Benchmarking in HPC 36
4.2 The PARSEC Benchmark Suite 37
4.3 Application Parallelization 39
4.4 Programmability 52
4.4.1 Lines Of Code 52
4.5 Performance 54
4.5.1 Scalability 54
4.5.2 Task Granularity Impact 59
4.5.3 Runtime System Overhead 61
4.5.4 Characterization of the Applications 61
4.6 Summary 63

5 Power-Aware Runtime Scheduling 65
5.1 A New Form of Heterogeneity 66
5.2 Mitigating Manufacturing Variability 67
5.2.1 Example with no Barrier or Synchronizations 68
5.2.2 Example with Barrier Operations 69
5.3 Mitigating Heterogeneity 70
5.3.1 Experimental Setup 71
5.3.2 Evaluation 73
5.4 Runtime Approach 73
5.4.1 Exploiting Application Structure 74
5.4.2 Search Algorithm 75
5.4.3 Training Sets 76
5.5 Evaluation 77
5.5.1 Monitoring Window Sensitivity 77
5.5.2 Performance Improvements of the Selected Configurations 81
5.5.3 Performance Improvements Taking into Account Analysis Costs 81
5.5.4 Energy Consumption Reductions 82

5.6 Summary 83

6 Power-Aware Job Scheduling 85
6.1 Power Variability Prediction Models 88
6.1.1 Power Ratio Model 89
6.1.2 Variability-Trained Prediction Model 93
6.1.3 Model Optimization 95
6.1.4 Predicting Power for Multi-Node Applications 95
6.2 Job Scheduling Policies 95
6.3 Model Validation and Policy Evaluation 97
6.3.1 Experimental Setup 97
6.3.2 Model Validation 98
6.3.3 Variability-Aware Scheduling Evaluation 99
6.4 Summary 102

7 Conclusions 103
7.1 Relevant Benchmarking in HPC 103
7.2 Runtime Mitigation of Manufacturing Variability 104
7.3 Model-driven Scheduling Mitigation of Manufacturing Variability 105
7.4 Future Work 106

Appendix A Publications 109
A.1 Conference Publications 109
A.2 Journal Publications 109
A.3 Workshop Publications 109

Bibliography 111

List of Figures 125

List of Tables 129

List of Abbreviations 131

1

Introduction

According to Moore's Law the number of transistors in integrated circuits doubles every two years [105]. This observation can have different interpretations, since more transistors mean additional or more complex components. Computer architects were able to exploit this by designing performance enhancing components, such as instruction pipelining, branch predictors and memory caches. Out-of-order execution of instructions, known as Instruction Level Parallelism (ILP), is also possible by implementing it at hardware level, which allowed computers to execute multiple instructions per cycle. All these, along with a constantly increasing clock frequency, have steadily led to an increase in performance with every new generation of processors, again roughly every two years. Moore's observation held true for more than three decades, however it reached a halting point in the early 2000's.

The post-Moore's Law era brings further implications to computer system design. Dennard scaling [47], which relates to Moore's Law, states that as the size of transistors reduces, their power density stays the same. This means that the performance per watt rate had also been increasing exponentially. As with Moore's law however, this observation is no longer valid. Intel's fastest single core processor (Intel Pentium 4), running at 3.8GHz, reached 100W in power consumption. The power and thermal dissipation at higher frequencies make faster processors unfeasible with today's technology. As a result, we can no longer rely on transistor technology or increasing clock frequency to deliver higher performance, as we have reached a point of diminishing returns. Furthermore, the increasing power demands of processors and the memory subsystem are not only a significant strain on the budget which often exceeds the cost of purchasing the machine [121], but have also reached a point where it is now arguably the defining limit for further performance enhancements. This is known as the Power Wall [107], which references the obstacle that power and heat pose in modern processor design.

At the beginning of the 21st century, computer architects took a different direction, which improved performance significantly and at a feasible power consumption. The idea was to use multiple simple processors on the same chip, and exploit parallelism that algorithms may have, since ILP is inherently limited [152]. This led to the design of chip multi-processors (CMPs), which can take advantage of parallelism at application level. Moreover, the simpler processors that compose a multi-core chip

require less power to operate, while their net benefit in performance is significantly higher than that of a single more powerful processor. The rise of the multi-core era raises new challenges, such as cores contesting for shared resources and departing from the deterministic nature of sequential algorithmic execution.

Moreover, the complexity of producing integrated circuits with transistors at such a microscopic scale has introduced artifacts in the manufacturing process, also known as manufacturing variability. These artifacts can effect the transistor switching speed and leakage, thus their performance and power consumption. Processor vendors today are unable to guarantee that their products, even processors of the same chip design, stepping and firmware, will consume the same amount of power or perform at the same frequency [123]. This form of variability is not only observed among chips, but also between cores within the same die. Early studies show that manufacturing variability can cause variation in power consumption up to 10% [123] and in performance up to 20% [103], for processors running identical workloads. Manufacturing variation is also observed to increase with every new generation of processors [103], making it an important issue for current and future processor design. As a result, modern CPUs are inherently heterogeneous in terms of either performance or power consumption. This form of heterogeneity further complicates programming CMPs, where parallel workloads cannot be statically distributed.

Even though programming and designing parallel systems requires significant effort, the raw performance offered by such platforms outweighs the problems. The excessive parallelism which is present in a large number of workloads from various domains of computing, offers the opportunity for performance to scale well beyond the limited number of cores found on single processor. Supercomputers, which are used in a variety of fields - from scientific simulations of astronomical models and genome analysis, to industry for aerodynamic modeling etc., are composed of hundreds or even thousands of such commodity processors. An example of a powerful contemporary system is the MareNostrum 4 supercomputer located in Barcelona, which is ranked as the 22nd fastest in the world, and 3rd in Europe [142]. It has 153,216 cores and can reach a peak performance of 10.296TFlops/s at 1,632kW of power consumption. The current fastest supercomputer, according to the TOP500 list [142], is Summit, located at Oak Ridge National Laboratory. It consists of 2,282,544 cores that can reach a peak performance of 187,659.3TFlops/s at 8,806kW power consumption. However, the problems observed in a single chip, also scale exponentially on such enormous systems. Power, in specific, is a major limiting factor from moving to the exascale era, since the cost for cooling and powering a machine consisting of millions of processors, an order of magnitude greater than contemporary supercomputers, is simply forbidding. Manufacturing variability can also become a more serious concern, since it introduces heterogeneous performance or power consumption among the different nodes.

To deal with increased complexity of designing parallel programs, many new programming models and paradigms emerged. They all aim at abstracting architectural details, such as the memory subsystem, and expressing parallelism at application level. When programming a single CMP, the most common and successful approach has been Fork-join models, like Thread level parallelism

(TLP) and the more sophisticated OpenMP [110]. Parallel work is divided by the user into multiple entities and run concurrently as different threads of execution. This paradigm is common in shared memory systems, where all threads can access the same memory. The user however is responsible for synchronizing the memory accesses and avoid racing conditions. Writing and reading memory in the wrong order can produce erroneous results, while poor synchronization strategies can cause bottlenecks and race conditions. In distributed memory systems (e.g. HPC clusters), where an application runs on many different nodes which do not have access to the same memory, programming paradigms like message passing (MPI [109]) are used. Programmers need to explicitly move data from one node to another in order for all processes to have an consistent view of the data, at all stages of an application's execution. Data transfers are costly, in terms of cycles, and can again cause contention and race conditions, if not used properly. In both shared and distributed memory systems, significant effort is required from the user in order to guarantee the correctness of the application's execution, as well as achieve good performance. Often, this task requires a very deep understanding of the programming model and the underlying hardware. More sophisticated programming models try to take the burden of the user by providing more intuitive ways to express parallelism. Task parallel models for example, allow the user to abstract parallel work into task constructs. This task is usually as simple as annotating certain function or code blocks as tasks, which can then be executed concurrently [6, 10, 19, 50, 61, 84, 115, 146] Task models often employ dataflow annotations, which allow the user to express memory accesses and data dependencies between tasks in the form of task arguments [6, 50, 84, 146]. Models such as these, reduce the need for explicit synchronization. The execution of parallel work and memory access synchronization is realized by dedicated runtimes, which are part of the programming models. The underlying runtime can also take care of load balancing parallel work among cores and/or different nodes, remaining transparent to the user. In practice, shared and distributed memory models are often mixed, where they deal with inter- and intra-node parallelization.

Exploiting the immense power offered by HPC clusters, requires the use of the aforementioned programming models, in order to express as much parallelism as possible in one's program. However, machines such as these have multiple user and require dedicated software that manage shared resources (such as cores and memory) and the applications that are queued for execution (typically referred to as jobs). The most commonly used workload manager is SLURM [85]. The resulting ecosystem, which allows us to efficiently use these high-end machines, consists of a mix of sophisticated hardware and software. Multicore processors coupled with high bandwidth interconnection, programming models that allows us to exploit parallelism, runtime systems that take care of parallel execution at application level, and finally a workload manager, which manages jobs, while efficiently allocating the system's resources. In their simplest form, HPC clusters are perceived as homogeneous machines, where each node runs at the same frequency and power requirements. Often more complex setups are used in practice, where for example general purpose CPU work along with GPUs. Even in this scenario though, it is easy to classify nodes in homogeneous clusters. Resource managers and runtime systems are also often used to hide a system's heterogeneity, by load balancing work dynamically. Modern multi-core and multi-node systems, which can be composed of thousands CPUs, can no longer be

perceived as homogeneous systems, neither in terms of CPU frequency nor power consumption, due to manufacturing variability. Vendors do not offer any classification of their processors' variability due to manufacturing issues, making even more difficult for users to identify and take into account this new form of heterogeneity. However its impact can be significant in both power consumption and performance. Runtime systems and system-wide workload management software can act as platforms for developing methodologies that both identify and classify manufacturing variability and mitigating its effects.

1.1 Thesis Objectives and Contributions

In this thesis we study manufacturing variability on modern processors and propose power- and variability-aware scheduling policies at runtime and system wide Job Scheduling levels. Our goal is to mitigate the effects of manufacturing variability in modern processors in order to optimize their performance and power efficiency. Our approach, treats power as a shared resource which needs to be carefully managed and allocated to parallel workloads. We propose two different approaches, one at application runtime level and one at system wide resource management level. The runtime approach deals with power constraint systems, where it monitors the application's performance during execution and redistributes power and work between the sockets of a node, till an optimized setup is reached. In order to evaluate our runtime approach, we needed a benchmark suite representative of modern workloads and implemented using our runtime system. For this we implement our own task-based version of the PARSEC benchmark suite, using OpenMP 4.0 tasks. The system wide approach tries to maximize the throughput of the system while minimizing power consumption. Part of our objective is also to understand how our methods affect applications that are actually being used in modern HPC systems.

1.1.1 Benchmarking with Realistic Workloads

Benchmarking is a vital part of evaluating experimental software and hardware. In HPC, benchmarking is usually restricted to small kernel applications. The reasoning behind this trend however is that larger applications consist of smaller kernel ones. Although this is true, it is not always the case that performance can only be gained by parallelizing or optimizing these kernel applications. Coarser grain parallelism also plays an important factor in improving the performance of applications used in today's computing PARSEC [14]. In this thesis, we aim to evaluate the benefits of task-based parallelism beyond the scope of HPC kernels, focusing on a set of parallel applications representative of a wide range of domains from HPC to desktop and server applications. To do so, we apply task-parallelism strategies to the PARSEC benchmark suite [14] and compare them in terms of programmability and performance with respect to the fork-join versions contained in the suite. The main contributions of are:

- We apply task-based parallelization strategies to 10 PARSEC applications.

- We fully evaluate them in terms of performance, considering different scenarios (from 1 to 16 cores) and achieving average improvements of 13%. In some particular cases, the improvements reach 42%.
- We provide detailed programmability metrics based in lines of code, achieving an average reduction of 28% and reaching a maximum of 81%.

Our superscalar version of the PARSEC benchmark suite (PARSECSs), fills the gap in the evaluation methodology usually used in HPC, offering a set of applications actually used in todays computing. The implementation offered also uses the state-of-the-art concepts and models in parallel programming.

1.1.2 Variability-Aware Load-balancing at Runtime Level

To deal with manufacturing variability, we present a runtime guided hardware/software reconfiguration approach that effectively mitigates the effects of inhomogeneous hardware behavior in low power environments. Further, we demonstrate that classical work stealing and load balancing techniques [19, 20, 119, 153] are insufficient to mitigate this performance issue. In the context of a NUMA node composed of several multi-core sockets, our technique is able to efficiently distribute a total node power budget among the node's different sockets, while also adjusting their corresponding concurrency levels. In order to enable this, our approach dynamically selects the best power/concurrency level for each socket involved in the computation by performing a light weight initial training phase. This initial training phase selects the optimal power/concurrency level to be assigned to each socket to reduce applications' load imbalance induced by power/performance inhomogeneity and thus increase performance.

The contributions are as follows:

- We provide a precise description of the limitations of current, state of the art load balancing techniques when dealing with inhomogeneous hardware behavior under node-level power limits.
- We demonstrate how uneven power and thread assignments to sockets can mitigate the inhomogeneous hardware behavior in dual socket NUMA nodes, resulting in up to 30% increased performance for some applications.
- We describe a dynamic runtime technique to discover the optimal power/concurrency assignment for each application on a given parallel machine that provides up to 22% performance improvements for some applications.

1.1.3 Power Variability Prediction-driven Job Scheduling

We propose two variability-aware job scheduling policies to deal with manufacturing variability at system-wide level, by introducing power- and variability-awareness to the cluster resource and job

manager. Workloads at the HPC system level are managed by job schedulers that allocate resources to dispatched jobs. Such jobs can run on distributed memory scenarios and, in this context, MPI [109] is the most common approach to handle distributed memory communications. It is usually coupled with a shared memory programming model, like OpenMP [110] or similar [7].

Either across nodes or within a shared-memory node, both job and runtime schedulers deal with the resource allocation problem, albeit at different levels, offering opportunities to manage power consumption. Indeed, examples of power-aware systems that offer solutions either at the job scheduling [30, 53, 58, 67] or at runtime system [38, 67, 79, 141, 143] levels already exist in the literature.

This work goes beyond the state-of-the-art by proposing job scheduling policies driven by variability-aware power prediction models. We extend power-aware scheduling and power prediction models to deal with manufacturing variability, producing two novel variability- and power-aware job scheduling policies. We consider the power consumption of the CPU, since it accounts for more than 50% [137] of the total node's power consumption. Our Policies rely on two different models and leverage their power requirement predictions of individual parallel jobs to make scheduling decisions that maximize performance while reducing energy consumption. Many different variability-agnostic power and energy prediction models have been proposed [12, 13, 16, 68, 82] and are often employed to manage power distribution on clusters or mitigate the effects of manufacturing variability [8, 38, 53, 67, 79, 141, 143]. Our work shows how variability-aware power prediction models can be effectively used to guide job scheduling policies and bring significant benefits with respect to the variability-agnostic ones.

In particular, this work makes the following contributions:

- Two new variability-aware power prediction models. The first model assumes power variability to impact all applications equally and it is based on executions of a single benchmark on different sockets to measure the power consumption variability across them. Given the power profile of the targeted application obtained from a previous run on a certain socket, the model applies the measured variability ratios to predict its power consumption on all sockets. The second model extends the Performance Monitoring Counters-based Performance Monitoring Counters (PMC) approach to take power consumption variability into account. PMCs are used to measure the activity of individual architectural components while the targeted application is running. Using a linear model trained off-line by running a reduced set of benchmarks on all sockets, the model predicts the power consumption of each architectural component for a certain application.

- Two power- and variability-aware job scheduling policies that optimize job turnaround time and energy efficiency while respecting a system-wide power budget. prediction model. Unlike previous work that does not consider variability during job scheduling decisions [53, 67, 79, 141], our policies use variability-aware prediction model to guide scheduling.

- A complete evaluation of the two variability-aware policies via a discrete event simulator. We implement additional scheduling policies for our evaluation, which represent traditional and state-of-the-art practices used in today's HPC systems. Our evaluation demonstrates how variability-aware policies achieve energy savings up to 8% and job turnaround time reductions up to 30%, considering different power budgets and two workload traffic scenarios (bursty and heavy).

1.2 Thesis Structure

The remainder of this thesis is organized as follows:

In Chapter 2 we summarize the most widely used designs in modern parallel architectures and runtime systems. We also present the most prominent programming models designed for such systems and focus on the state-of-the-art in the widely used task-based programming model for shared memory architectures. Furthermore, we discuss the shortcommings in the current state of benchmarking in HPC systems. In the end of the Chapter we present an emerging issue in modern HPC system design, the manufacturing variability and how it effects a system's power efficiency. In Chapter 3 we show our experimental platforms, the hardware-software stack and methodology used to obtain the results presented in this thesis. In Chapter 4 we present our task-based implementation of the PARSEC benchmarks suite. We show how tasks and dataflow annotations can be used to improve the performance and programmability of parallel applications, when compared to other commonly used programming paradigms. Moreover, we discuss why the PARSEC benchmark suite and our implementation offers a better and more realistic testbed for HPC systems. In Chapter 5 we further discuss how manufacturing variability can harm performance and the energy efficiency of an HPC system. In this Chapter we focus on a runtime solution to mitigate the effects of manufacturing variability. In Chapter 6 we move from the runtime to a more system-wide approach to dealing with manufacturing variability. We show how variability-aware power prediction models can be employed to guide job scheduling decisions, in order to improve the job throughput and power efficient of an HPC cluster. In Chapter 7 we offer our closing remarks and summarize the contributions and conclusions of this work. We also discuss the future direction of this work.

2

Background

In this Chapter we provide the necessary background context and the state-of-the-art related to this thesis. In Section 2.1 we describe the different architectures found in today's parallel systems. We divide them into two main categories, based on the way memory is viewed by the individual computing units. We then describe how a computing cluster is designed and explain the distinction between homogeneous and heterogeneous clusters. In Section 2.2, we present the different programming models used to program shared and distributed memory machines, as well as the challenges users face when writing parallel applications. We then discuss how more sophisticated programming models can help users write more efficient and maintainable parallel code, when coupled with a runtime system. In Section 4.1, we discuss the limitations of contemporary benchmarking suites and methodologies, and make a case for moving to a benchmarking suite that better represents workloads run on today's HPC systems, implemented in a state-of-the-art parallel programming model. In Section 2.3, we describe the software used to manage the workload and the resources on an HPC cluster. Manufacturing variability, which causes same model processors to run at varying frequencies and power consumption is presented and discussed in Section 2.4. The state-of-the-art in power managing HPC clusters and improving the power efficiency of parallel runtime systems is presented in Section 2.5.

2.1 Parallel Systems

With the stagnation of processor frequency and the inherent limitation of ILP in computer programs, computer architects turned to multi-core processor chip design in order to exploit parallelism at application level. Both hardware design and software implementation for exploiting the parallelism available by the multiple cores brings significant new challenges. Traditionally, computer programs were designed in an sequential algorithmic manner. However, in a parallel environment, the tasks performed by a program need to be divided into smaller ones, which can be concurrently executed. This is known as thread or task level parallelism (TLP). In an ideal scenario, a program can be parallelized in an equal number of concurrent tasks, as the number of available cores. If a system has N cores and the sequential version of the program run in T seconds, then the expected speedup

would be $N * T$. This is known as linear scaling. However, in practice this is rarely the case. Even if an algorithm is embarrassing parallel, meaning that it can easily be divided in N tasks, the speedup may be near-linear because the application may become bound by memory or I/O.

Understanding the underlying hardware, is key in order to achieve good performance. For example, parallel tasks operate in smaller segments of data than their sequential counterpart. This can lead in better utilization of the memory hierarchy and more efficient use of the data replacement policies. In such cases, performance can achieve super-linear speedup, meaning that it is possible to surpass even linear scaling. The importance of efficiently utilizing the memory organization on a parallel system is apparent from the above example. As such, the most common distinction between parallel systems is the way memory is organized and viewed by different cores. There are two main memory schemes used, shared memory and distributed memory.

2.1.1 Shared and Distributed Memory Systems

In shared memory systems, all cores can access the entire memory using the same physical address space. Typically, modern processors feature multiple cores, each with access to a private cache and an interface connection to the DRAM subsystem. A common design for contemporary processor features private instruction and data L1 caches to each core, an additional L2 private cache and a shared L3 cache between all cores. It is also possible to only have two levels of caches, in which case L2 cache is shared and connects to the DRAM, instead of the L3. The shared cache memory (be it L2 or L3) is also referred to as Last Level Cache (LLC).

If all cores can access any arbitrary memory location with the same speed (latency and bandwidth), then such a system is referred to a Uniform Memory Access (UMA). If however, the physical location of a core influences the cost of accessing memory, this system is referred to as Non-Uniform Memory Access (NUMA). This design is most common in practice fore modern processors. A single unit can feature two or more sockets on the motherboard, mounted with a multi-core processor each. Each processor has it's own cache hierarchy, with the LLC connected to the DRAM memory via a cache-coherent network interconnection. Because memory access time depends whether data is present in the local cache hierarchy of each processor, such a system is a NUMA one.

In a distributed memory system, each processor has its own physical memory address space. All processors can communicate with each other via a network interconnection and can exchange data through it. The network bandwidth and latency are of uttermost importance for the systems performance, since it can act as a bottleneck on large scale systems, where multiple processors may transfer data through the same interconnection. As such, the network topology is very important for such systems. Connection between processors can be point-to-point links or use dedicated switching hardware, grouping processors together. For the interconnection network, low latency and high throughput protocol is used, like Infiniband.

2.1.2 HPC Cluster Design

HPC cluster typically consist of hundreds or thousands of processors and cores, offering immense parallelization and computing power. Summit, the current faster supercomputer, according to TOP500 list [142], consists of 2,282,544 cores. MareNostrum, one of Europe's largest supercomputers [142], consists of 19,440 cores. Multiple computing units, referred to as nodes from this point on, are connected together via a network interface, from commodity ones like Ethernet to high-throughput ones like Infiniband. In practice, large HPC clusters feature a combination of shared and distributed memory system design. Each node features multiple UMA or NUMA sockets. Nodes use the network interconnection to transfer data between them.

Homogeneous vs Heterogeneous Systems

An HPC cluster may only feature same type processors. Such a system is known as homogeneous, referring to the uniform computing capacity of all nodes. General purpose processors offer a good option for most problem classes, however specialized hardware, like GPUs, are better candidates for certain problems (e.g. linear algebra and machine learning) and act as accelerators. An alternative to the homogeneous system design is combining different type of computing units. As an example, a system can feature NUMA nodes general purpose processors and additional nodes with multiple GPUs. This design approach is known as heterogeneous.

Apart from accelerating certain problem cases, heterogeneous system design is also considered for power efficiency. Typically, smaller cores are more power efficient than faster ones. The GPU approach is to offer a hundred times more cores than processors. These cores are slower but more power efficient than those of a processor. Emerging processor architectures explore the potential of designing multi-core chips where not all cores have the same computing capacity. Such processors are referred to as Asymmetric Multi-core (AMC) processors. ARM big.LITTLE is an AMC processor design, steered towards power efficiency. *Bigger*,faster cores are combined with *smaller*, slower but more power efficient ones. The reasoning behind this design is to offer fast cores for running tasks in the critical path of the application, while the smaller ones can offer more parallelism at a smaller power cost, for non-critical tasks. Although heterogeneous systems offer benefits in power efficiency and performance, they are more difficult to program compared to homogeneous ones. Since in an heterogeneous system not all processing units perform the same, evenly distributing work among cores is not a good option. Faster cores will finish earlier and remain idle, waiting for the slower ones. Either the programmer needs to explicitly distribute work in a fashion that will not create bottlenecks. Alternatively, dedicated software can act as a scheduler to dynamically load-balance work between cores.

Heterogeneity is not always a deliberate choice. Due to manufacturing issues, processors of the same model, stepping and firmware can still demonstrate variability in both performance and power consumption [103, 122]. As such, a system that is expected to act as homogeneous is in fact heterogeneous. Dynamic load-balancing is again required to mitigate performance issues, while

power needs to also be managed for more efficient use by the available cores. Software solutions, like the ones proposed by this thesis, can be employed to deal with this type of heterogeneity and mitigate its negative effects.

2.2 Parallel Programming Models

Programming large parallel machines is considered a job for expert users. To ease the programming effort and make parallel machines accessible a lot of parallel programming models have been proposed. The main goal of any programming model is to offer the means to express the available parallelism in an application. Additional factors that distinguish a programming model is how the underlying machine is abstracted and how the execution and synchronization of parallel work is managed.

2.2.1 Shared and Distributed Parallel Memory Models

An important factor that programming models need to handle is the underlying memory system. How cores access memory can influence the cost of their communication. In shared memory programming models, the most common programming paradigm is fork-join. Workloads are decomposed into smaller ones and a new thread of execution is spawned for each. Since all cores share the same physical address space, threads can access memory and communicate a very low cost, but synchronizing accesses is very important to maintain a consistent and correct view of the memory among all threads. This threading model is the most prevalent programming model for shared memory machines, while many other shared memory models derive from it [19, 88, 110, 120].

In distributed memory systems, execution threads, usually referred to as processes, have a private memory. Workload decomposition needs to take account that data needs to also be segmented. Each process only works on the data segment it has a copy of, but after work completion, results need to be aggregated. A common approach to achieve communication is through a message passing library, like MPI [109]. Such libraries, offer the communication primitives to transfer send data from one node to another, in a point-to-point fashion or broadcast to everyone.

Note that the distinction between shared and distributed memory programming models concern the way memory is abstracted and presented to the user. For example and MPI application can use the shared memory subsystem to exchange messages, within the same node or processor. Other models, like UCA [52] can offer a unified memory view to the user, where the underlying library implementation can take care of moving data between different address spaces. In the Futures programming model certain values are passed around without actually having been evaluated, until a callback mechanism is called when their value is required by the program. Futures is a widely used shared memory programming model, implemented even in the standard C++ and Boost [130] libraries. However, distributed memory implementations also exist [37, 120]. In practice, hybrid approaches are often used. Shared memory models, like OpenMP, can be coupled with a distributed memory one,

like MPI. Inter- and intra-node communication and parallelization is handled by the corresponding model, exploiting the best both have to offer.

2.2.2 Synchronization and Parallel Programming Challenges

Parallel execution brings new challenges for users. In sequential execution the order commands run is well defined by the user. Contrary, in a parallel environment commands may run concurrently and different threads of execution race for the shared resources (e.g. memory) on the system. Competing for system resources needs to be managed by the user or dedicated software, such as a runtime system. If not synchronized properly, parallel execution threads may starve or in the case of memory, wrong access order may violate RAW dependencies. These situations are referred to as /emphrace conditions. Different programming models offer varying solutions to the synchronization problem. Parallel programming models offer solutions like barrier primitives to define these synchronization points. In a shared memory environment, where system resources are shared between threads, synchronization is achieved by using primitives like locks and semaphores. These primitives guarantee that a resource will be accessed by only one thread at a time. However, which thread and in which order a resource is accessed needs to be defined by the user. The order and manner of access is especially important for the memory system, since racing threads may produce wrong results or evict memory which is in use by another thread. In microarchitectural level, where memory instructions require multiple cycles to complete, atomic instructions guarantee that data will not be accessed before the instruction completes. The fork-join model, followed by many shared memory models (Pthreads,OpenMP), also dictates that threads need to synchronize when joining.

In distributed memory environments, each process is typically operating on a separate copy of the data. Occasions may rise where a process needs to broadcast its results or share them with a different process. Synchronization is achieved by transferring the data through the network, with the use of a library like MPI. Again, the responsibility for synchronizing processes is the user's. In a distributed memory environment, where data is transferred typically over a network interface, the user needs to also consider the cost of doing so. Decomposing the workload into large processes is general preferable, since communication among processes incurs significant overheads. Although distributed memory models can easily abstract and be used with shared memory machines, it is generally preferable to combine them with shared memory programming models. The shared memory model paradigm is more efficient in a shared memory environment, since it does not require explicitly moving data between cores, causing less overhead. As already stated, this is a hybrid approach, where shared memory models are used for inter-node communication and distributed ones for the intra-node interactions.

Synchronization is not the only challenge users have to face in parallel environments. Decomposing the workload into smaller ones, which can be run concurrently, is also critical. This step defines the available parallelism in the application, as well as the data movement and synchronization between different threads and processes. Moreover, the homogeneity of the threads/processes workload is also

very important. The user must consider the distribution of the workload and the architecture of the underlying machine if he is to fully utilize the available parallelism and the machine's full processing power. Statically and evenly distributing inhomogeneous workload will cause certain cores or nodes to finish before others and remain idle. Alternative decomposition strategies can be more efficient, but harder to implement.

2.2.3 Asynchronous Tasks and Dataflow Model

Different parallel programming models compete in providing intuitive and novel ways to express parallelism. A very successful parallel programming paradigm is task-based parallelism. Tasks offer an easy and abstract way to express parallelism. The OpenMP 4.0 [110], a widely used programming standard for shared memory machines, allows the user to annotate functions that can be run asynchronously. Other programming models, like Cilk [93], allow the user to implement parallel functions, through library calls. Task-based models require the programmer to synchronize data accesses between competing parallel tasks, with synchronization primitives. They are also typically coupled with a runtime system, implemented as a library, which deals with task creation, synchronization and load balancing. More sophisticated task-based models also offer tools to simplify task synchronization, and allow the runtime to be aware of the order tasks need to execute, while respecting data dependencies between them. Explicitly expressing the execution order of tasks enables the runtime to make less conservative decisions when scheduling tasks, essentially making dynamic load balancing more efficient. It also supports dataflow annotations that describe data dependencies among tasks. This information can be used by the runtime system to synchronize task execution.

An intuitive way to express task data dependencies is the dataflow model, which is implemented by the most widely used programming models, like OpenMP 4.0[110]. In the dataflow model, the user is able to express the data footprint of a task, typically in the form of task arguments. The user also needs to specify whether an argument is going to be read as input or written to as output, or both as input-output. Task arguments are translated into a memory addresses at runtime and the dataflow relations, as defined by the user, can be used to construct a task dependency graph (TDG), which is an acyclic directed graph describing the dataflow relations between all available tasks. The nodes on such a graph represent the tasks queued for execution and the directed edges represent the data dependencies between the tasks. A task is ready to execute when there are no more input dependencies (input edges). When a task finished, it's outgoing edges are removed and the graph is checked again for tasks that may now be free of dependencies.

Figure 2.1 shows a simplified version of the `ferret` benchmark implemented in OmpSs [50]. OmpSs is an extension to the OpenMP 4.0 model with similar syntax and some additional features like socket-aware scheduling for NUMA architectures. The `ferret` application is parallelized with a pipeline model, where each task is a pipeline stage and the dataflow relations direct the order of execution of these stages. The user can use pragma directives to identify functions that should run asynchronously. These task pragmas can have dataflow relations expressed with the use of `in, out`

```
void load() {
  int i = 0;
  while( load_image(image[i]) ) {
    #pragma omp task in(image[i])
                     out(seg_images[i])
    seg_images[i] = t_seg(image[i]);
    #pragma omp task in(seg_images[i])
                     out(extract_data[i])
    extract_data[i] = t_extract(seg_images[i]);
    #pragma omp task in(extract_data[i])
                     out(vectoriz_data[i])
    vectoriz_data[i] = t_vec(extract_data[i]);
    #pragma omp task in(vectoriz_data[i])
                     out(rank_results[i])
    rank_results[i] = t_rank(vectoriz_data[i]);
    #pragma omp task in(rank_data[i])
                     out(outstream)
    t_out(rank_data[i], outstream);
    i++;
  }
  #pragma omp taskwait
}
```

Fig. 2.1 `Ferret` implementation in OpenMP 4.0/OmpSs

and `inout` annotations. These declare whether a variable is going to be read, written or both by the task. An underlying runtime system is responsible for scheduling tasks, track dependencies, balance the load among available threads and ensure correct order of execution, as dictated by the dataflow relations. In our example data dependencies will force tasks spawned in the same iteration to run in sequential order, while tasks from different iterations can run concurrently. An exception is `t_out` which shares a common output between all instances, `outstream`, to store the final results of `ferret`.

Few studies exist that examine the performance of task parallelism compared to other models. [7] evaluate OpenMP tasks by implementing a few small kernel applications using the new OpenMP task construct. Their evaluation tests the model's expressiveness and flexibility as well as performance. [116] compare three models that implement task parallelism, Wool, Cilk++ and OpenMP. They compare their performance using small kernels, as well as some microbenchmarks aimed to measure task creation and synchronization costs. They show that Cilk++ and Wool have similar performance, while they outperform OpenMP tasks for fine grain workloads. On coarser grain loads, all models have matching performance with OpenMP gaining in one case, due to superior task scheduling.

BDDT [146] is a task-based parallel model, very similar to OmpSs, that also uses a runtime to track data dependencies among tasks. BDDT uses block-level argument dependency tracking, where task arguments are processed into blocks of arbitrary size, which is defined by the user. This offers some flexibility when tracking dependencies of arrays, without the need to modify the memory layout, while also maintaining precision (depending on the chosen block-size). Moreover, it offers additional syntax semantics to exclude certain arguments from the dependency analysis, further reducing the overhead of online dependency tracking by reducing the size of the dependency graph. BDDT is shown to outperform loop constructs implemented using OpenMP.

2.2.4 Parallel Runtime Systems

In Section 2.2.2 we discuss the challenges users need to face when programming parallel machines. Most parallel programming models today, such as the task-based dataflow model presented in Section 2.2.3, implement runtime systems that deal with some of these challenges in different ways. Some runtime systems may only offer synchronization primitives, like barriers and locks, or implement data transfer primitives, like broadcasting. It is often the case though that a runtime system offers additional functionality to take some of the burden of the user's shoulders. A common key feature is managing the parallel workload, distribute it among cores, instead of depending on the user to statically divide and distribute it. Dynamically handling a parallel workload is an easy task for a runtime, which can redistribute work to idle cores or nodes. This feature is often referred to as dynamic load balancing, and allows the user to ignore some of the underlying architectural details, like the heterogeneity of the cores or nodes and the manufacturing variability of CPUs. Apart from relieving the programming effort, these features make the code more portable and maintainable across different machines.

The versatility of how a runtime system can be exploited to improve performance, energy consumption or expand a model's functionality is demonstrated by the sheer number of different approaches and techniques developed by research centers and industry. Chronaki et al. [41] improve performance of task-based models on asymmetric multi-core architectures by identifying task that are in the critical path, and scheduling such tasks on the faster processing units of the machine. A critical task is a task that delaying its execution will prevent the completion of the whole application. Castillo et al. [34] propose a minimal extension to hardware design that allows dynamic reconfiguration of multi-processors' per core computational power. By identifying critical tasks, the runtime is used to guide the dynamic reconfiguration of hardware, so that tasks in the critical path are given more computational power. Myrmics [99] is a runtime system with task dependency tracking, designed to scale on heterogeneous architectures. Brumar et al. [22] minimize redundant execution by exploiting repetitive patterns in parallel workloads. In their approach, a parallel task may not be executed if a *similar* one has already been executed before. In this case, the same result is reused. They define a methodology for measuring task *similarity*. This approach sacrifices result precision in order to improve performance. Vassiliadis et al. [150] propose a task based model that aids the runtime system to identify less *significant* tasks and then decide whether it should execute them accurately or approximately. A less *significant* task is defined as a task that has small impact on the accuracy of the final result of the application. They report reduction in energy consumption up to 83%, when compared to a fully accurate execution. Jaulmes et al. [83] expand the OmpSs programming model and its underlying runtime with error detection and protection, for iterative solvers, in a transparent manner from the user's perspective. The effectiveness of automatic compared to manual vectorization in task-parallel models is studied by Caminal et al. [29].

However, having a dedicated runtime aids in managing task creation, synchronization, load balancing, data transfers, etc is not free. Significant overhead can be incurred by such a software system, which in many occasions can limit the scalability of a parallel code or even perform worse

than the sequential version of the same code. A typical way to deal with this overhead is to avoid decomposing a workload into very small, fine-grained tasks or processes, so that the actual work is always more than the execution time required for the runtime system. This can limit the available parallelism significantly. Thus, the user often needs to find a *sweet spot* for the decomposition size of the workload. Any software approach however, is an order of magnitude slower than the equivalent hardware implementation.

Future architectures should be designed in a way that they can use direct information from the runtime system and also provide an infrastructure for basic runtime functionalities (such as task creation and data tracking) to eliminate any related overhead [32, 147]. Tan et al. [140] demonstrate the feasibility of the approach by implementing a hardware accelerator on an FPGA, Picos++, which deals with tracking and managing task data dependencies (e.g. OpenMP, OmpSs, IntelTBB). This hardware approach delivers 1.8× performance speedup and up to 40% less power consumption. Etsion et al. [59] propose an abstraction to out-of-order pipeline that operates at task granularity, instead of ILP. Castillo et al. [33] propose a hybrid software/hardware mechanism, where data dependence tracking is offloaded to hardware, but task scheduling is still managed at software level. They report average speedup of 4.2% over a hardware implemented runtime and require 7.3x less area, while the software scheduler is more flexible than a hardware implemented one.

A significant body of work focuses on exploiting information available to the runtime in order to guide and improve data management in memory. This is often achieved with hardware extensions that allow the runtime system to communicate with the memory subsystem. Sanchez et al. [127] extract control and data dependencies information from the runtime's task dependency graph in order to reduce data transfers. RADAR [101] uses data dependencies of tasks to track their memory footprint and find dead blocks in last level cache memory. Dead blocks can be then evicted from the cache. Pan et al. [111] exploit the input annotations of tasks to identify data blocks that will be reused in future tasks and use this information to guide cache partitioning. Álvarez et al. [2] present a proposal for managing stacked DRAM memories in HPC systems. Stacked DRAM memories combine the benefits of high-bandwidth DRAM with the large space of conventional off-chip memory. In their approach, the runtime is used to manage data transfers between memories using idle workers, keeping all this functionality transparent to the user. Papaefstathiou et al. [112] uses tasks' lifetime to guide prefetching and data replacement in cache memories. In the same spirit, Dimic et al. [48] improve cache replacement policy to reduce the miss ratio of last level caches. Álvarez et al. [3] propose using data dependencies from the runtime system to manage scratchpad memories. Caheny et al. [26, 28] propose adding another cache layer in the directory protocol and exploit information available in the runtime system to reduce coherence traffic in NUMA nodes. The same authors [27] present a hardware/software hybrid system, which uses task-based and dataflow model semantics to identify data that does not need memory coherence. By disabling coherence for such data, the system improves performance and energy efficiency.

Part of this thesis is inspired by the aforementioned runtime approaches and exploits the underlying runtime to identify and mitigate the effects of manufacturing variability (see Sections 2.4 and 5) present in multi-socket NUMA nodes.

2.3 Managing HPC Clusters

In Section 2.1.2 we discuss how an HPC cluster is designed. Typically, an HPC cluster consists of multiple computing nodes, not necessarily of the same design and computing capacity. Building and maintaining an HPC cluster is a considerable investment. As such, the idle time of an HPC cluster should be minimized by any institution or organization owning one, to make the most out of the machine. It is not uncommon to share an HPC cluster between multiple users, since few applications require its full computing power. Moreover, nodes may break but the system should still operate with the rest, while a workload may require specific nodes, in an heterogeneous environment. The different workloads themselves may have different priorities and/or dependencies between them. All these aspects need to carefully managed to maximize a cluster's job throughput. All HPC cluster's today use dedicated software to manage their workload and resources, such as SLURM [85] and PBS [64].

In a typical cluster environment, the user writes a special script which runs the actual parallel application and a few additional directives. These directives describe parameters related to the execution of the application, like the number of nodes and time the application requires in order to run. The script is then submitted to a queue, which is maintained by the workload manager. The submitted script and the application are referred to as a job. A user, or multiple users, may submit multiple jobs. The workload manager decides when and on which nodes a job will run. The order that jobs are run is dictated by the scheduling policy and the resources available. As such, a workload manager has two main functions: job scheduling and resource management.

2.3.1 Job Scheduling

Simple FIFO queues are not the most efficient way to run jobs on an HPC cluster. System administrators may specify the scheduling parameters or implement their own policies to match their needs. Even in the simplest setups however, a FIFO queue is not sufficient. Since jobs have varying execution time and may run on multiple or a single node. Consider the following example: We have a cluster of 126 nodes, while job A needs to run for 30 minutes on 64 nodes and job B for 20 minutes on 100 nodes. If job A is running, job B must wait for A to complete, since there are not enough nodes. Now consider that there are more jobs after B. Job C requires 16 nodes and 10 minutes to run. In a simple FIFO scheduling policy job C will need to wait for both jobs A and B to complete, which is waiting for 50 minutes. An alternative scenario, is for the workload manager to look forward in the queue, since B cannot be scheduled, for jobs that require less nodes and will complete within 30 minutes, so that running them before job B will not delay its execution more than job A would. This policy is called *backfilling* and is the most common one in contemporary workload managers [62].

2.3.2 Resource Management

Resources are shared between jobs, so the workload manager needs to take care of which job gets which resource. It is not necessary that a node will run only a single job, so the workload manager must take care of how it allocates cores and memory as well as the nodes themselves. Another important factor that a workload manager must consider is the topology of the network and the distance of the nodes. A multi-node job will perform better if nodes are closer together, as synchronization and data movement would cost less. In this work we argue that power, which is limited, should also be managed by the workload manager. In Section 2.5.3 we discuss the matter further and provide related work which explores the potential benefits of power-aware scheduling policies. In Chapter 6, we present our approach, which employs power prediction models to drive scheduling decisions, considering the available power on the cluster.

2.4 Manufacturing Variability

In the recent past, significant performance benefits were obtained by increasing number of transistors on processor die, a phenomenon which became known as Moore's Law. A different interpretation of Moore's law is that of Dennard Scaling, which states that performance per watt is doubling every two years, roughly. To achieve this, engineers had to shrink down transistors, which also means that the threshold voltage and current had to also be scaled down. As we reached a point of diminishing returns, neither Moore's nor Dennard's observations hold true. Shrinking transistors below a certain size leads to increased sub-threshold conduction, leakage currents and heat dissipation. It is not possible to deal with the above issues and increase performance, without affecting a chips power consumption and reliability [55]. Moreover, decreasing transistor size makes lithography extremely challenging causing artifacts that affect transistor parameters during the manufacturing process, such as distortions in film thickness and channel length. The most important parameters affected are threshold voltage (V_th) and the effective gate length (L_{eff}), which can directly affect a transistor's switching speed. V_th impacts the power leakage of transistors, while the switching speed of transistors directly affects the chip's performance and power consumption [21, 73].

Since the manufacturing process cannot guarantee that transistors will operate at nominal parameter values, processors of the same production line (stepping and firmware) manifest variation in both their performance (frequency) and power consumption. Most vendors use frequency binning, meaning that processors with the same performance characteristics are placed in the same group. The same method is not employed however for binning together processors with the same power consumption characteristics. As such, modules processors in current HPC systems are already inhomogeneous from the point of view of power. Early results on 64 processors have shown a 10% power variation for identical workloads at equivalent performance [41]. Despite the advances in fabrication process and power gating, the impact of manufacturing variability is expected to worsen in future processor generations [103, 126].

2.4.1 Impact of Manufacturing Variability in HPC Systems

In Section 2.1.2 we compared the homogeneous and heterogeneous cluster designs and discussed the challenges users face when programming for the latter. Typically HPC system operators obtain processors from the same bin, as by vendor characterization, so that processors operate at he same frequencies. This way, homogeneity is guaranteed, at least for the units that it's intended, avoiding having to deal with the unpredicted variability caused by the manufacturing process. However, as power is becoming a major concerned, an HPC systems can longer be perceived as homogeneous in terms of power consumption. Since vendors do not offer any classification of processor variability, administrators of HPC systems ignore power consumption variability and treat the system as homogeneous. In this work, as with a number of recent related work found in literature [8, 53, 67, 79, 141, 143], it is demonstrated that it is important to consider power consumption variability in order to improve the system's power efficiency and performance. Furthermore, computer architects have employed statistical model's [60, 95, 104, 128] to measure variation of processor's, since vendors do not release any relevant information on their models.

It is not uncommon for HPC clusters to operate under system-wide power constrains. However, since not all processors consume the same amount of power, it is not sufficient to evenly power cap them. Moreover, power capping a processor will translates power consumption variability into frequency variability and thus, performance heterogeneity [122]. As such, a previously homogeneous system is now heterogeneous, either from its power consumption or performance points of view, which in turn implies that efficient use of the machine requires additional programming effort and/or software support for mitigating its effects. In this work we propose two different approaches for providing software support: First at runtime level, where we propose methodology for improving dynamic load balancing on power constraint sockets. Secondly, we propose a new analytical method for predicting power consumption variability, and then use the model to guide job scheduling decisions.

2.5 Variability-Aware Power Management in HPC Systems

As we head towards the exascale era, power is increasingly becoming a serious constraint. full capacity. According to a report from the US Department of Energy [138], energy efficiency is considered to be among the top ten most serious research challenges we face today, in order to achieve exascale computing capacity. The same report also points out that apart from the advances expected in hardware, in order to reduce the power consumption of HPC systems, it is also necessary to design software which is able to manage thousands of nodes in a power efficient manner. Large HPC systems currently are expected to meet their power demands at all times, even when all cores are operating at full capacity. This is rarely the case and poses a huge strain on the power budget, provisioning the system for the worst case scenario, no matter how unlikely it is. A software solution could manage the underlying hardware and make sure that such a scenario never takes place. Moreover, modern clusters suffer from manufacturing variability. Any software solution should consider the heterogeneity

in power consumption in order to be effective. It is a combination of energy efficient hardware architectures and power-aware parallel runtimes and system software (e.g. workload managers) that will make exascale computing feasible.

2.5.1 Software-aided Power Constraining and Management

The most direct way of managing the power a processor consumes, is with DVFS. Vendors typically allow their processors to operate at different frequency levels. Lower frequencies offer lower power consumption and DVFS can be set and managed by software. Different cores can also operate at different frequencies. However, processor vendors, realizing the importance of managing power in future and current HPC systems, are starting to offer greater flexibility by allowing users to set the exact power limit a processor can reach.

The ability to set up power bounds in many-core systems is becoming a common feature. For example, Intel introduced a set of machine-specific registers (MSRs) [132] on their Sandy/Ivy Bridge processors to explicitly constrain on-chip power consumption. Although this seems as a straight-forward solution to managing power, system administrators must consider manufacturing variability. Processors do not consume the same power, even though they are designed to do so. In order to provide homogeneous performance, chips of the same architecture must hide frequency variability, which can only be achieved via variations in their power consumption. To abide to this user-set constraint, CPU cores resort to reducing their frequency. Under a power constraint however, different chips operate under different frequencies. Since the release of commodity chips with such capabilities, several studies have shown the impact power capping can have. In particular, work by Rountree et al. [122] motivates the research presented in this thesis on how processor performance variability due to power capping can be addressed.

In a power constrained environment where all chips need to operate under a certain power cap, this frequency variability can no longer be hidden [123], leading to heterogeneous performance. As a result, a theoretically homogeneous system turns into a heterogeneous one with performance variations of up to 64% [79]. While ignoring this manufacturing variability leads to performance and energy inefficiencies, there are opportunities for achieving improvements at the power budgeting or parallel runtime system levels when variability is properly managed [38, 67, 79, 141, 143].

Inadomi et al. [79] also study the performance variability on a number production clusters and propose a variation-aware power budgeting framework. Their approach requires specific single core executions for profiling the HPC applications plus a once-per-system profiling to build a reference table containing performance variability information for all nodes. This table and the single core profiling is used to make decisions using a model. Compared to their method, our runtime approach does not require dedicated profiling runs or system wide reference tables containing performance variations. Instead, we use profiling information obtained at runtime to adjust power distribution and concurrency levels, which reduces the analysis costs and increases its benefits.

Bailey et al. [8] propose a linear programming formulation for MPI+OpenMP programs for maximizing performance under job-level power constraints. While this approach provides a good approximation of the upper bound of possible performance in dynamic runtime systems, the use of a linear programming solver is too slow to be practical for optimizing applications at runtime. The same group also introduced Conductor [102], a dynamic runtime system that directs power to the critical path of the computation to minimize overall execution time under a power cap. Conductor, however, does not deal with the hardware manufacturing variability we describe in this work. As such, our approach is orthogonal and can be combined to maximize parallel applications performance.

On a single node, Cochran et al. [43] classify the PARSEC benchmark suite applications for their power, temperature and performance characteristics. Using these results, they maximize performance while meeting power constraints by using thread packing and DVFS. In contrast to our runtime approach, though, they rely on their priori characterization, while our approach can work without prior information.

There is a significant body of work focused on job scheduling for power constrained systems. Etinski et al. [57] propose an LP-based job scheduling policy; Sarood et al. [129] use performance modeling to make job scheduling decisions in power constraint system to improve job throughput; and Ellsworth et al. [54] discuss a dynamic job scheduling algorithm, which when running under a system-wide power limit, detects unused power and redistributes it to nodes that can make use of it.

The impact of manufacturing quality on power consumption variability of processor chips has been studied in a significant number of works as well. The power leakage of processors is directly connected to our work, since by setting a power limit on the socket, we impair its ability to adjust power consumption to maintain the proper frequency level. Davis et al. [45] study the effect of inter-node variability on power model characterization, in the context of homogeneous clusters. Herbert et al. [76] show that exposing the power leakage variability of processors to the DVFS control algorithm to shift work to the less leaky processors, can reduce overall system power consumption. Further, several projects study the on-die power variation to improve DVFS scheduling [77, 96, 141]. As an additional concern, modern processors require transistors to shrink to a level that introduces significant power and reliability variations among processors, a phenomena explored in detail by a variety of groups [21, 73, 145]. Overall, most studies conclude that power variation is expected to become worse in the future [73, 126], which will make the effects of power budgeting more apparent. Thus, a variability-aware software solution is imperative in managing complex parallel applications and improving both performance and energy efficiency.

2.5.2 Power Prediction Models

Knowing an applications power requirements can be invaluable information when making decisions on when an where to run it in a large HPC cluster. Relying on user supplied information is not enough, since this information can be both difficult to obtain but also unreliable. An alternative is to train analytical prediction models to get the information. Power prediction models have been extensively

studied over the years. In particular, models based on PMC have been very successful in predicting power consumption [12, 13, 16, 68, 82]. However, none of the models found in the literature consider manufacturing variability. In this work, we demonstrate that it is possible to make PMC prediction models aware of the manufacturing variability and accurately predict the socket and application specific power consumption, insteadof making the same prediction for all sockets. Our PMC-based model is based on the work of Bertran et al. [12, 13], which aims at providing insight into the way individual architectural components influence power consumption. Our PMC-based model extends Bertran's model to account for manufacturing variability in terms of power consumption. The original model requires carefully crafting micro-benchmarks that isolate activity per architectural component, in order to train it. We show that comparable results can be obtained by training the model with a small set of kernel applications and a microbenchmark that stresses the memory unit. The benefit of using the suggested set of applications for training the model, is that it is a more portable solution, compared to Beltran's original set of microbenchmarks. In Beltran's work, the microbenchmarks need to be designed so that only certain architectural components are stessed by each microbenchmark. This is achieved by carefully implementing an assembly code and considering all the architectural details of the underlying machine. Although our approach will suffer a small penalty in precision, there is not need of modification when run on a different machine. Our approach intends to predict power consumption variability while remaining easy to deploy on any system. The implementation, application and evaluation of our proposed analytical analytical prediction model is discussed in Chapter 6.

2.5.3 Power Aware System-Wide Job Scheduling

Handling power has become an important factor when managing and designing HPC systems. However, contemporary workload managers deployed on such systems (e.g. SLURM [85]) do not consider power as a resource and do not manage their workloads in an energy efficient way. Yet, researchers have identified the need to make workload management software power-aware and have already published various experimental approaches [5, 56, 58, 66, 66, 78, 90, 92, 106, 114, 129].

A survey on the techniques developed in nine of the TOP500 HPC centers for improving energy efficient is presented by Maiterth et al. [100]. They identify several emerging techniques, some with common characteristics. Over-provisioning [129] considers building a system where it is not possible to run all the nodes at full capacity. Instead, the system operates under a certain power budget and dynamically distributes the available power among nodes. Nodes can operate under different power caps. For example, a few nodes may operate at full capacity, while the rest are disabled or constrained. Other approaches [106, 114] take advantage of applications that can be considered *moldable*, meaning that these applications can run at different configurations (e.g. number of threads).

A significant body of work examines approaches on how to optimally use DVFS or hardware imposed power constrains (e.g. RAPL) in order to save energy but also optimize performance [5, 56, 58, 66, 78, 114]. A different approach is also identified, where instead of using hardware imposed

power caps, energy efficiency is achieved only by job scheduling [66, 90, 92]. Manufacturing variability is also considered in some studies, which exploit the variance in power and performance among nodes to improve energy efficiency [114, 133]. Although the identified techniques are not used in production in any of the HPC centers, they provide some insight on future trends in energy efficient HPC computing.

Etinksi et al. [56] present a practical approach to apply DVFS on an HPC cluster, exploiting periods of low activity. With DVFS, scaling down the frequency to save power causes significant performance degradation. Their approach manages to reduce the negative impact DVFS by applying it when overall activity is low on the cluster. Moreover, Etinksi et al. [58] present a power-aware job scheduling policy, MaxJobPerf. Their policy considers two types of resource that need to be allocated to new jobs, processors and power. To decide which job should be scheduled next and how power should be distributed among jobs, they use integer linear programming. Sarood et al. [129] use performance modeling to increase job throughput in power constrained systems. A power management of overprovisioned systems has also been studied by Patki et al. [30, 113]. Unlike our work, all sockets in a cluster are viewed as homogeneous in terms of power consumption, which can lead to suboptimal scheduling decisions.

More recent work identifies the need to consider manufacturing variability when making scheduling decisions or managing a system's power budget [8, 53, 67, 79, 141, 143]. Inadomi et al. [79] extensively study the impact of manufacturing variability on a number of production clusters and propose a variation-aware power budgeting framework. They introduce variability to their prediction model by statically measuring power variability on each socket and then apply it to their original, variability agnostic, predictions. However, they base their approach on the assumption that variability is application independent. Teodorescu et al. [141] study the impact of manufacturing variability and propose a linear programming algorithm to find the best parameters for power budgeting with DVFS. Ellsworth et al. [53] propose a power distribution framework that optimizes an HPC cluster's power consumption under a certain system wide power budget. In contrast to our work, jobs are scheduled without considering power consumption, but power is redistributed, favoring more power intensive jobs. A two level solution for overprovisioned clusters is presented by Gholkar et al. [67], where a job scheduler is used at system level to allocate nodes and distribute power. The job scheduler predicts the total energy consumption in order to make a scheduling decision. Individual sockets may run under power constrains, in which case, a second runtime scheduler decides the optimal configuration of active processors and the power distribution among them in order to mitigate the power variability. Adagio [124] detects the critical path of MPI applications and uses DVFS to reduce power consumption of non-critical pieces of work, hiding performance variability from the scheduler.

In this thesis we present two job scheduling policies that consider manufacturing variability to optimize performance and energy efficiency, in power constrained clusters. We use two different analytical models to predict power consumption and manufacturing variability, in order to guide scheduling decisions. The first model is similar to [79], making the same assumption that variability is application dependent, but used in a different context. The second model offers a more robust

approach, eliminating the aforementioned assumption. In contrast to hardware imposed power caps, our approach aims to maintain power consumption below a certain budget, only by optimizing job scheduling. This way we avoid the implications of degraded and varying performance, of power constrained sockets.

3

Experimental Setup

In this Chapter we describe the hardware and software platforms used for the experimental evaluation of this work. In this work we assume that we work with parallel workloads that are implemented with a programming model that allows dynamically controlling its concurrency level. Applications should also be able to be decomposed into concurrent tasks. The underlying hardware platform should offer the ability to impose user defined power caps, at least per each socket (such as Intel CPU models that are equally or older than the Sandy Bridge family of processors). Moreover, older CPU models do not demonstrate notable manufacturing variability. We also assume the presence of a workload manager to manage running applications on an HPC cluster.

3.1 Hardware Platforms

For our experimental evaluation we use two distinct HPC clusters, the MareNostrum III supercomputer at Barcelona Supercomputing Center and the Catalyst and Quartz clusters at Lawrence Livermore National Lab. Marenostrum III is one of Europe's largest supercomputers and represents the state-of-the-art in production environments in HPC. Our initial evaluation of the PARSECSs is conducted on MareNostrum III, however, since it's a large production machine, access to specific MSR registers, required for monitoring and capping power on Intel chips, is restricted. For this reason we also use the Catalyst and Quartz cluster, where MSRs are accessible by the user through special kernel modules.

- *MareNostrum III*: It consists of 3,056 compute nodes in total. Each node is IBM System X server iDataPlex dx360 M4, composed of two 8-core Intel Sandy Bridge processors E5-2.60Hz, 20MB of shared last-level cache. There are eight 4GB DDR3 DIMM's running at 1.6GHz (a total of 32GB per node and 2GB per core).
- *Catalyst*: The Catalyst cluster [97] consists of 324 NUMA nodes, each with two 12-core Intel Xeon E5-2695v2 sockets and equipped with 128GB of main memory. It can reach a peak performance of 149.3 PFLOPS. Access to MSR counters is granted to normal users through a

kernel module. We use 128 of these nodes (256 sockets) in our experiments, which totals to 3,072 cores.

- *Quartz*: The Quartz cluster [97] consists of 2,634 NUMA nodes, each with two 16-core Intel Xeon E5-2695v4 sockets and equipped with 128GB of main memory. It can reach a peak performance of 3,251.4 PFLOPS. For our experiments we use We use 128 of these nodes (256 sockets) in our experiments (4,096 cores in total). As with the case of Catalyst, access to the RAPL interface is granted through a kernel module. This is a larger production machine we use to gather application execution traces and later use in our workload manager simulator.

3.2 Software Stack

3.2.1 Runtime System

We use the OpenMP 4.0 standard to implement the PARSECSs benchmark suite. We make use of the Nanos++ OpenMP runtime system (version 0.8a). Because of it's modular design it is ideal to expand it's functionality and offers all OpenMP 4.0 features along with some experimental ones. Note that we only use the standard OpenMP 4.0 features in our implementation. We also use Nano++ for developing our power-aware runtime approach. Nanos++ is coupled with the Mercurium source-to-source compiler (version 1.99), and gcc 4.7 as the back-end compiler.

3.2.2 Analysis tools and Power Capping Framework

We use a number of tools to perform various types of analyses on our benchmarks. We use the Extrae instrumentation package [91] (version 2.5) and the Paraver trace viewer [91] (version 4.5) to analyze and compare the PARSEC and PARSECSs benchmark suites.

Performance and Power Monitoring Our methodology requires to monitor performance counters plus power consumption rates. We use *perf* version 3.10 and *mpstat* version 10.5.1 for monitoring architectural and core component activity. For measuring power and enforcing power limits we use Intel's RAPL registers, which expand typical hardware counters, offering precise readings on power consumption and temperature, as well as offering the functionality to constrain core power consumption to a certain limit. On Sandy and Ivy Bridge CPUs, these special registers are accessible per socket, but on newer architectures like Haskwell, they offer the same features per core. We implement a daemon built on top of *libmsr* [132], which is a user friendly framework for accessing RAPL registers safely from user space, through a special kernel module. Our study focuses on the variability on processors. In our experiments we measured variance in DRAM power consumption of less than 1% (measured using the RAPL interface), among different sockets when running the same benchmark. For this reason, we only report the power consumption of processors. We use the same framework for power capping sockets on Catalyst and Quartz. These power caps are enforced at hardware level by reducing the effective frequency of all cores, to match the requested power

budget. The user can specify a time window and a maximum average power for that window. The processor guarantees that it will not exceed this average. Intuitively, longer windows may allow better performance for applications that utilize the CPU in bursts; if the burst exceeds the window size, the processor will have to be throttled.

The sample rate is 100ms for power monitoring and 1s for performance counters. Although we are able to monitor an applications real power consumption on a finer grain, our predictions are limited to 1s granularity, since they depend on the performance counter data collected at the coarser granularity.

3.3 Workload Manager Simulator

For testing our scheduling policies, we implement a discrete event simulator, and implement our scheduling policies on top of it. Although there already exist workload manager simulators for SLURM [136] and Flux [65], they do not model power. Moreover, these simulators also require tracing the applications. We found it to be more practical to implement our own to better control what traces need to be generated and how power is to be handled and modeled.

The simulator requires all jobs to be first executed on the physical hardware to gather performance and power profiles, which are then used to simulate their execution under different scheduling schemes. The performance and power profiles are referred to as traces in this document. They track performance and power information over time, throughout a job's execution. These traces are different to job queues' traces, which contain information on the time a job is issued, scheduled and completed, as well as which resources were allocated for its execution. Idle power is modeled as the average power consumed by each socket, as measured on the actual hardware. Using a simulator allows us to rapidly test and evaluate new policies without any accuracy loss since the simulations are led by power traces obtained from real parallel executions. Our design allows the user to implement scheduling policies using python scripts. In our setup we implement SLURM's [85] default job scheduling policy, since it's the de-facto resource management tool on production clusters. The additional policies suggested by this work expand the functionality of SLURM's default scheduler with power- and variability-awareness.

Validation of Workload Manager Simulator

To validate the simulator we run a small scale experiment using 8 nodes (16 sockets) on Quartz. The workload we use consists of a mix of a 100 instances of the PARSECSs benchmarks. Each instance is a single socket job and is randomly chosen out of the total 10 PARSECSs benchmarks. We run a bash script that periodically issues 10 instances (every 60s) until all the job instances are issued. Then, it waits for all jobs to finish execution. We generate a trace with the timestamp of when every job was issued and on which node it was run on. We also keep track of the total time it takes for all the jobs to complete along with the total energy the sockets consumed (including idle time).

Table 3.1 Benchmark training set for PMC-based power prediction model.

Benchmark	**Description**
cholesky	cholesky factorization kernel
knn	K-nearest neighbours kernel
matmul	Floating point matrix multiplication kernel
md5	MD5 message-digest algorithm
prk2_stencil	Tests the efficiency with which a space-invariant symmetric filter (stencil) applies to images
qr_tile	Tiled QR factorization kernel
sparseLU	Sparse LU factorization kernel
stap	Space-Time Adaptive Processing for radar detection of an objects position
symmatinv	Symmetric matrix inversion kernel
vector-redu	Computes the sum of the elements of a vector
mem_bench	A micro-benchmark that stretches different memory levels

We then use the simulator to repeat the experiment and reproduce the results measured on the actual machine. We use the same set of nodes and 100 instances. We run all PARSECSs applications on all 16 sockets to gather the traces and power profiles (on a different run from the original experiment described in the previous paragraph). Then we issue the same 100 jobs again, this time on the simulator, at the same intervals as with the original run on the actual machine. The simulator will also use the workload trace to get the socket each job run on, and try to replicate the same socket to job allocation, if possible. This way, it tries to essentially recreating the same scheduling decisions SLURM took on the actual machine. When all jobs finish execution, we measure the total execution time and energy consumption.

Comparing total execution time and energy consumption between the two experiments shows that the simulator is 1.6% slower than the actual execution on the cluster. Moreover, the jobs' total power consumption is higher on the simulator by 1.1%. These results show that our simulator has very good accuracy and the results discussed in Section 6 are representative of the impact our scheduling policies would have on the actual machine.

Table 3.2 NAS Multi-Zone benchmarks are multinode applications that use both MPI and OpenMP to express parallelism. MPI is used for inter-node communication and OpenMP is used for intra-node parallelizations of loops.

Benchmark	Description
BT-MZ	Block Tri-diagonal solver. The workload consists of a mesh, unevenly divided among processes.
LU-MZ	Lower-Upper Gauss-Seidel solver. The workload consists of a mesh, evenly divided among process.
SP-MZ	Scalar Penta-diagonal solver. The workload consists of a mesh, evenly divided among processes.

3.4 Benchmark Applications

3.4.1 Prediction Model Training

To train our models we use a set of small kernel and micro-benchmarks, listed in Table 6.2, that capture different behaviors. In addition to these kernels, we design a microbenchmark that stresses each level of the memory hierarchy, in order to measure the impact that each cache level has on power consumption. Our set consists of kernel applications which are representative HPC workloads, however often larger and more exhaustive set of benchmarks are selected for training [12]. Although larger sets can give better results, by capturing a wider variety of application behaviors, we demonstrate that our set provides comparable results, while it's easy to deploy. In comparison, Bertran et. al [12] use a large set of 100 micro-benchmarks, fine tuned to the underlying architecture. This set would be ideal for our prediction models as well, however it is not portable and requires significant effort and understanding of the underlying architecture, which makes its deployment challenging.

3.4.2 Runtime and Job Scheduler Evaluation Benchmarks

To evaluate the runtime and job scheduling policies considered in this thesis, we use the PARSECSs benchmark suite, which we developed after the PARSEC benchmark suite (further described in Section 4.2). The PARSECSs benchmark suite consists of emerging workloads for shared memory architectures, representative of applications run on typical HPC systems. Our implementations use the OmpSs/OpenMP 4.0 programming model, which allows us to use current and emerging realistic workloads under a sophisticated programming environment. This is essential for evaluating our runtime solution for mitigating manufacturing variability and improving the energy efficiency of the programming model, as the original PARSEC suite is implemented in Pthreads, and only a couple of them use OpenMP 3.0 constructs, such as parallel loops. Using a model like OmpSs/OpenMP 4.0, allows us to easily modify and test our runtime approach without the need to re-implement our methodology for every benchmark using the Pthreads model. Moreover, using tasks and dataflow

relations allows us to express more complex parallelization strategies, that are not possible to implement with the typical fork-join model of Pthreads and OpenMP 3.0. This allows us to evaluate our proposal using applications that better represent contemporary parallel workloads. We discuss our implementations of the PARSECSs in more detail in Chapter 4.

In the case of the job scheduling techniques, we use the PARSECSs as our set of single socket parallel jobs. However, multi-node jobs are also common in HPC environment. For this reason we expand our benchmark set with the MPI+OpenMP versions of the NAS multi-zone benchmarks [86] (NAS-MZ), posing as our multi-node jobs. The NAS-MZ benchmarks are described in table 3.2. The multi-node jobs run with different configurations for 8, 16 and 64 MPI processes, where each process runs on a single socket. All instances and processes run on 16 cores, with the exception of *facesim* (8 cores), *fluidanimate* (8 cores), *lu-mz_D.8* and *lu-mz_D.8* (1 core per MPI process). Our diverse set of applications can run from a single core up to 768 cores, for the larger MPI codes. We run all benchmarks 5 times and report median values. This is done to minimize the impact of noise or unrelated to manufacturing variability inference in our experiments. However, note that variation in performance and power observed in all of our experiments were below 2% (when repeating the same run on the same socket).

3.4.3 Job Scheduler Workload Generation

Typically workload manager schedulers are evaluated using workload traces from the job queues of actual HPC clusters [58, 63]. However, in our case this is not applicable, since these type of traces do not contain information on power and manufacturing variability. Moreover, we are not able to create a job queue trace out of the clusters we have access to, since reading performance counters such as the RAPL interface requires root access. This is not an option for us on these production machines. For these reasons we generate our own cluster workload combining single- and multi-node applications, so that we can measure the performance and power profiles of the workload. A similar methodology is used in other power and manufacturing variability related studies [53, 114], but in our approach we use a wider number and range of applications. The applications used are described in Section 3.4.2.

We generate two random job distributions as our workload on the cluster, corresponding to bursty and heavy loads. The bursty scenario consists of 763, periodically creating a heavy load that requires a large number of sockets to be served, even exceeding the systems total capacity, having jobs wait. However, there are also time periods that the system may be idle or have only a few jobs to serve. The heavy load scenario consists of 2286 jobs, where there are always enough jobs to occupy the whole system, for 98% of the total execution (2% corresponds to initial submission when the whole system is idle and the few last jobs remaining at finalization, before all jobs complete and system returns to idle state). In the rest of this document, we use the term traffic when referring to the cluster's load.

3.4.4 Configuration Exploration Space

Our runtime approach needs to try different configurations of power distribution and number of active cores in order to find a favorable one. Exhaustively exploring all possible configurations is not feasible, so we describe how we choose our configuration space. We consider power bounds of 80W, 100W and 120W for total node power. If we allow a power limit of 80W, we consider 5 different ways of distributing the power among the two sockets of the NUMA node: 30W:50W, 35W:45W, 40W:40W, 45W:35W and 50W:30W as well as 36 ways of specifying the maximum concurrency allowed in each 2-socket NUMA node: 2-2, 4-2, 6-2, 8-2, 10-2, 12-2, 2-4, etc. up to 12-12. In total, this leads to a total of 180 combinations. Similarly, when allowing a power limit of 100W there are 8 ways of distributing it, which combined with the 36 possible ways of distributing the concurrency, leads us to a total of 324 combinations. Similarly, when the total power budget reaches 120W, the total number of combinations is 468. Overall, for each particular application we have 972 different combinations. Other Considerations: The results of these experiments are machine dependent since each particular 12-core socket reacts in a different way when a power limit is set. Ideally, all 972 configurations per application should be executed on many NUMA nodes to really account for many possible hardware reactions when a power limit is set. However, due to the size of our experimental campaign, we randomly chose a single 2-socket NUMA node for each considered application and run all 972 combinations on it. Although this random choice can slightly influence the relative results between the benchmarks, the general conclusions we extract from them remain unchanged.

3.4.5 General Considerations

Special care is needed when conducting our experiments in order to ensure that we minimize interference not related to manufacturing variability (such as OS noise and NUMA effects). To deal with such random effects, we run our benchmarks 5 times on each socket and observe the variability within the same node. Our benchmarks demonstrate inter-node variability is very low compared to the one observed when running the same application on different nodes (inter-node variability is less than 2%, while the intra-node one can be over 15%). If in any case we observe higher than normal inter-node performance variability, we discard the results and repeat the experiment. A similar strategy is used to deal with power consumption variability due to temperature variations. We always measure temperature and discard results that are not within the range of 38-42 °C. TurboBoost has also been disabled to make sure that the hardware mechanisms that can alter the effective frequency of cores, other than throttling to maintain the power cap, do not interfere.

4

Realistic Task-based Workloads

In the last few years processor clock frequencies have stagnated, while exploiting Instruction-Level Parallelism (ILP) has already reached the point of diminishing returns. Multi-core designs arose as a solution to overcome some of the technological constraints that uniprocessor chips have, but they exacerbated some others as a counterpart. Multi-core architectures can potentially provide the desired performance by exploiting Thread Level Parallelism (TLP) of large scale parallel workloads on chip. Such large amount of parallelism is managed by the software, which means that the programmer needs to implement highly efficient and architecture-aware parallel codes to achieve the expected performance. This is obviously much harder than programming a uniprocessor chip, which is commonly referred as the *Programmability Wall* [35]. Moreover, dealing with this wall will be even harder in the near future with the arrival of many-core systems with tens or hundreds of heterogeneous cores and accelerators on-chip.

Threading is the most common way to program multicore processors. POSIX threads (Pthreads) [25] and OpenMP [36] are two of the most common programming models to implement threading schemes. Additionally, MPI [109] can be incorporated to threading codes to handle parallelism in a distributed memory environment. However, to develop efficient threading codes can be a really hard job due to the increasing amount of concurrency handled by many-core processors and the current trend towards more heterogeneity within the chip. Synchronization points are often needed in threading codes to control the data flow and to enforce correctness. However, the cost of these schemes increases with the amount of parallelism handled on chip, seriously hurting performance due to issues like load imbalance or NUMA effects. Also, relaxing synchronization costs often involves significant programming efforts as it requires the deployment of complex and application specific mechanism like thread pools.

Task parallelism [6, 10, 19, 50, 61, 84, 115, 146] is an alternative parallel paradigm where the load is organized into tasks that can be asynchronously executed. Also, some task-based programming models allow the programmer to specify data or control dependencies between the different tasks, which allows synchronization points relaxation by explicitly specifying the data involved in the operation [6, 50, 84, 146].

The task-based execution model requires to track the dependencies among tasks, which can be explicitly specified by the programmer [84, 154] or dynamically handled by an underlying runtime system [50, 51, 146]. When dependencies are detected among tasks, a deterministic execution order is applied by the runtime system to enforce correctness. In this way, all the potential parallelism of the code is exposed to the runtime system, which can exploit it depending on the available hardware. Additional optimizations like load balancing or work stealing [19, 50] can be applied at the runtime system layer without requiring any platform-specific consideration from the programmer.

The potential of task-based programming models is expected to be significant in a wide range of areas. In this Chapter we show our task-based implementation of the PARSEC benchmarks (PARSECSs). Our objective is to provide an evaluation framework for task-based parallel models with data dependence tracking. This combination allows programmers to exploit parallelism in applications that is not feasible, or require tremendous effort from the programmers part, when using other parallel models. The PARSEC benchmark suite is a suitable test best, since the applications include are not restricted to small kernels. Instead, the diverse set of workloads and computing domains covered offers the opportunities for task parallel and dataflow based models to exploit such parallelism.

These emerging parallelization paradigms offer a more diverse test bed than typical fork-join models with barrier synchronization. This allows us to better understand the impact of manufacturing variability on modern parallel workloads. Moreover, task-based programming models are coupled with runtime systems, which deal with load balancing, dependence tracking, thread synchronization and data allocation. These are all necessary tools to deliver good performance and should already be able to deal with manufacturing variability to some extend. In this work we don't aim to simply expose manufacturing variability, our goal is to effectively mitigate it. By studying its impact on a state-of-the-art runtime system, we can offer a solution that is relevant and significant by today's standards.

4.1 Benchmarking in HPC

Other studies exist that compare parallel programming models in the literature. Although these studies do not focus on task parallelism, they employ benchmarks and similar methodology to evaluate their target models. [42] study and compare the performance of UPC and Co-array Fortran, two PGAS languages. They use select benchmarks from the NAS benchmark suite. [4] use microbenchmarks to measure and compare the performance of 11 context-oriented languages. Their study shows that they all often manifest high overheads.

Although all the works we mention try to evaluate various programming models, in terms of performance, and some times on usability and versatility, they are all limited to small kernels or even just micro-benchmarks. We find that this approach is not sufficient to give us an insight on how a model will impact actual large-scale applications. [89] use a proxy application in their work to evaluate a number of different programming models (OpenMP, MPI, MPI+OpenMP, CUDA, Chapel, Charm++, Liszt, Loci). Their approach however is limited to only one application. Different

application domains can be very different, and may require different parallelization techniques to get good scalability and performance. A programming model could fail to even provide a way to express a parallelization scheme, let alone deliver performance. It is important to have an in depth understanding of a models behavior and limitation in order to make an educated decision whether research should direct its efforts to adopt and further expand it.

Pipeline parallelism has been the subject of study in some recent studies. This programming idiom is found often in streaming and server applications and goes far beyond the HPC domain. [93] propose an extension to the Cilk model, for expressing pipeline parallelism on-the-fly, without constructing the pipeline stages at their dependencies a priori. It offers a performance comparison between the proposed model, Pthreads and Thread Building Blocks (TTB) for three PARSEC benchmarks, ferret, dedup and x264.

This trend of using microbenchmarks and kernel application is also followed when evaluating other aspects of HPC, apart from parallel programming models, such as emerging microarchitectures, novel load-balancing techniques and scheduling policies, etc. The SPEC CPU2006 [75] and SPEC CPU2017 [24] are benchmark suites designed to evaluate processor architectures. However, although the included workloads are fitting for processor design evaluation, they are not representative of larger, more complex applications that are run in today's large computer systems. The PARSEC benchmark suite [15] on the other hand is composed by applications from varying computing domains, but are also common problems run on HPC systems. Both SPEC and the PARSEC however, are implemented using the most basic of parallel programming models, like Pthreads. Such programming models, although expressive enough to exploit the available parallelism, offer little insight into how these applications interact with more sophisticated programming models, which may have a dedicated runtime system to deal with workload management and synchronization. In this work we implement a variation of the PARSEC benchmarks suite, the PARSECSs, using OMPSs/OpenMP 4.0 task directives and dataflow relations. Our implementation uses the most common features between contemporary task-based models, so they can be easily ported. Using task parallelism allows us to implement more complex and efficient parallel programming paradigms, like pipelines. In this work, we will be using the PARSECSs to evaluate our runtime and job management solutions to mitigating the manufacturing variability.

4.2 The PARSEC Benchmark Suite

With the prevalence of many-core processors and the increasing relevance of application domains that do not belong to the traditional HPC field, comes the need for programs representative of current and future parallel workloads. The PARSEC [14] features state-of-the art, computationally intensive algorithms and very diverse workloads from different areas of computing. PARSEC is comprised of 13 benchmark programs. The original suite makes use of the Pthreads parallelization model for all these benchmarks, except for `freqmine`, which is only available in OpenMP. The suite includes input sets for native machine execution, which are real input sets. Table 4.1 describes the different benchmarks

Table 4.1 PARSEC Benchmark Suite

Benchmark	Description	Native input	LOC
blackscholes	Intel RMS benchmark. It calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation (PDE).	10,000,000 options	404
bodytrack	Computer vision application which tracks a 3D pose of a marker-less human body with multiple cameras through an image sequence.	4 cameras, 261 frames, 4,000 particles, 5 annealing layers	6,968
canneal	Simulated cache-aware annealing to optimize routing cost of a chip design.	2,500,000 elements, 6,000 temperature steps	3,040
dedup	Compresses a data stream with a combination of global compression and local compression in order to achieve high compression ratios.	672 MB data	3,401
facesim	Intel RMS workload which takes a model of a human face and a time sequence of muscle activation and computes a visually realistic animation of the modeled face.	100 frames, 372,126 tetrahedra	34,134
ferret	Content-based similarity search of feature-rich data such as audio, images, video, 3D shapes, etc.	3,500 queries, 59,695 images database, find top 50 images	10,552
fluidanimate	Intel RMS application uses an extension of the Smoothed Particle Hydrodynamics (SPH) method to simulate an incompressible fluid for interactive animation purposes.	500 frames, 500,000 particles	2,348
freqmine	Intel RMS application which employs an array-based version of the FP-growth (Frequent Pattern-growth) method for Frequent Itemset Mining (FIMI).	250,000 HTML documents, minimum support 11,000	2,231
raytrace	Intel RMS workload which renders an animated 3D scene.	200 frames, 1,920×1,080 pixels, 10 million polygons	13,751
streamcluster	Solves the online clustering problem.	200,000 points per block, 5 block	1,769
swaptions	Intel RMS workload which uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions.	128 swaptions, 1,000,000 simulations	1,225
vips	VASARI Image Processing System (VIPS), which includes fundamental image processing operations.	18,000×18,000 pixels	127,957
x264	H.264/AVC (Advanced Video Coding) video encoder.	512 frames, 1,920×1,080 pixels	29,329

included in the suite along with their respective native input and the lines of code (LOC) of each application. We apply tasking parallelization strategies to 11 out of its 13 applications: `blackscholes`, `bodytrack`, `canneal`, `dedup`, `facesim`, `ferret`, `fluidanimate`, `freqmine`, `streamcluster` and `swaptions` and `x264`. We leave 2 applications out of this study: `raytrace` and `vips`. `Vips` is a domain specific runtime system for image manipulation. Since vips is a runtime itself, it is not reasonable to implement it on top of another runtime system. Therefore we do not include this code in our evaluations. `Raytrace` code has the same extension as `ferret`, `facesim` and `bodytrack` and the same parallel model as `blackscholes` [44]. Therefore, since it does not offer any new insight, we do not consider the `Raytrace` code in this work.

We have a preliminary task-based implementation of the `x264` encoder, which scales up to 14x on a 16-core machine, the same as the Pthreads version. Since we just emulate the same parallel model

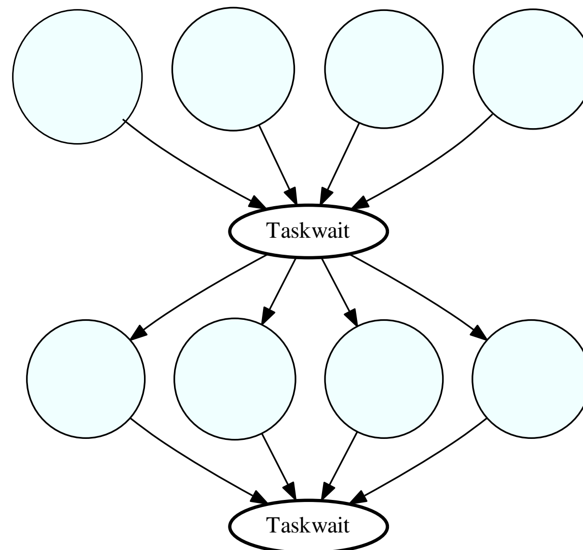


Fig. 4.1 Task-graph of blackscholes application. No dependencies exit between tasks, only barrier synchronization between iterations.

as the original Pthreads version and obtain the same performance, we do not include this code in the results Section as it provides no insight.

4.3 Application Parallelization

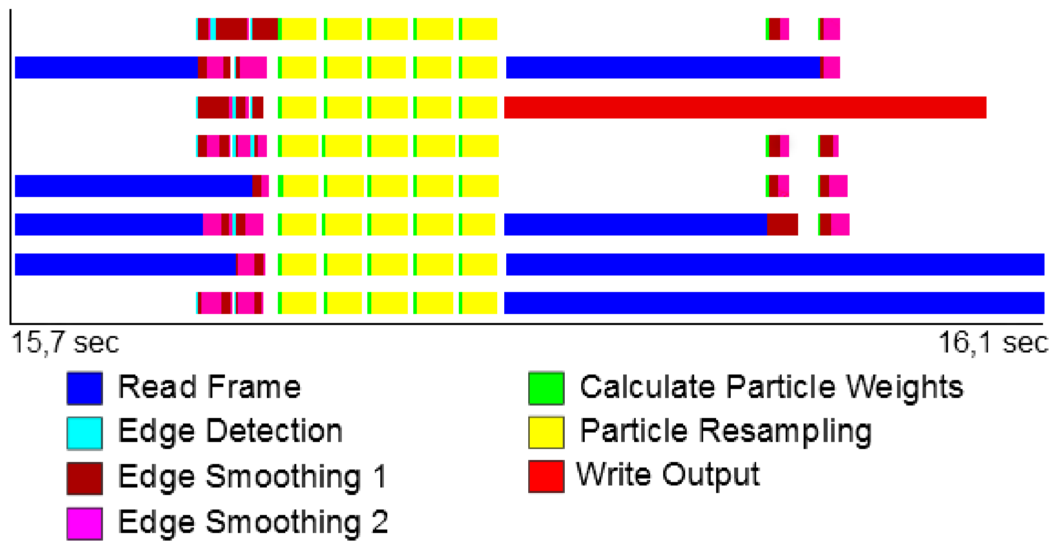
In this Section we discuss how the PARSEC applications are parallelized in Pthreads/OpenMP2.0 and how they can be implemented efficiently using a task-based approach. When possible, we exploit dataflow relations in order to take advantage of implicit synchronization (as described in Section 2.2.3). If it is not possible we use conventional synchronization primitives such as locks, atomics and barriers.

Blackscholes This application solves a Black-Scholes Partial Differential Equation [18] to calculate the prices for a portfolio of ten million European options.

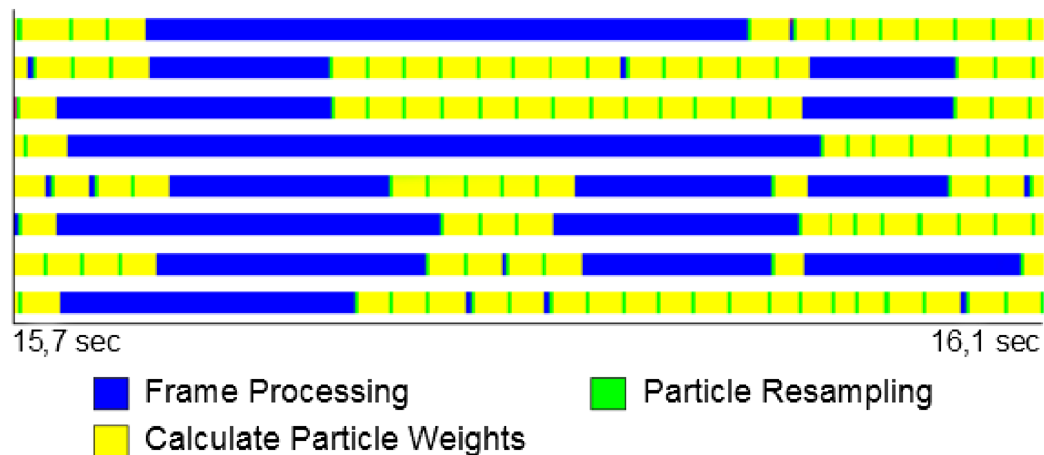
Pthreads This version simply divides the portfolio into work units by the number of available threads, and stores them into the `numOptions` array. Each thread calculates the prices for its corresponding options and waits in a barrier until all the threads have finished executing. The algorithm is run multiple times to obtain the final estimation of the portfolio.

Task-based In the case of the task-based version, we divide the work into units of a predefined block size. This block size allows having much more task instances than threads, which implies a much better load balance, as this is an embarrassingly parallel application with no dependencies among tasks in the same run. A task graph of the task-based implementation is shown in figure 4.1. We

can see that this is an embarrassing parallel application with only barrier synchronization between iterations of the main loop. No data dependencies exist between tasks. For all the task graphs shown in this document we use smaller workloads than the ones used for evaluation. This way it's easier to read the task graphs. For example in figure 4.1 there are four tasks per iteration, but this is easily configurable to hundreds or thousands of tasks. Such is the case of the native workloads used for the evaluation of our implementations.



(a) Trivial Task-based Strategy



(b) Optimal Task-based Strategy

Fig. 4.2 Parallel execution of Pthreads and task-based versions of `bodytrack` on an 8-core machine and native input size. Different parallel regions correspond to different colors. White gaps in the figure, represent idle time.

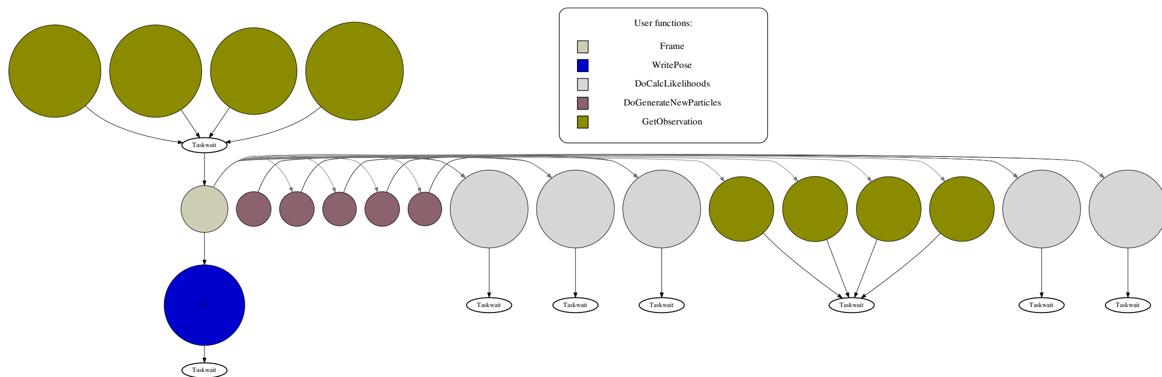


Fig. 4.3 Task-graph of bodytrack application. Edges show task data dependencies.

Bodytrack Computer vision application that tracks a marker-less human body using multiple cameras through an image sequence. The application employs an annealed particle filter to track the body using edges and the foreground silhouette as features of interest.

Pthreads `Bodytrack` applies the same algorithm on each frame of the image sequence to track the different poses of the body. The human body is modeled as a tree-based structure, consisting of 7 conic cylinders. It reads 4 images taken from several cameras to capture a scene from 4 different angles, thus each frame consists of these 4 images. These images are read and encoded to a single data structure. For each frame, `bodytrack` extracts the edges and silhouette features for each of these 4 images. In this feature extraction stage we have 3 different kernels.

1. **Edge detection**: Gradient based edge detection.
2. **Edge smoothing (phase 1)**: Gaussian filter used to smooth edges applied on array rows.
3. **Edge smoothing (phase 2)**: Gaussian filter used to smooth edges applied on array columns.

Afterwards, `bodytrack` goes through an annealed particle filter stage, which consists of M annealing layers over a set of N particles. The particles are multi-variate configurations of the state and location of the tracked body. Given the image features, the particles are assigned weights, which increase or decrease the chance that a particle represents a body part. N particles are then chosen, depending on the probability dictated by their weights. Random noise is added to this set of particles, creating a new set. This process is repeated for all annealing layers. `Bodytrack` then picks one of the M configurations, the one which has the highest weighted average. This process has two parallel kernels.

4. **Calculate particle weights**: Computes weights for the particles, using the edges and silhouette produced from the previous stages.
5. **Particle Resampling**: Adds Gaussian random noise to the particles, thus creating a new set of particles.

In the case of Pthreads, the 4 images of a frame are read and processed in parallel using one thread per image. The Pthreads implantation is limited by the 4 images it can process concurrently, while there is no other candidate work at this point. A specific asynchronous I/O implementation is required to read the files in parallel. Then, the features extraction stage is executed using all the available threads, with a synchronization barrier at the end of each phase. The same structure is followed in the annealed particle filter stage, with two barriers at the end of each phase. Between the two stages, serial code has to be executed, which leaves only one thread busy and the rest idle. Finally, the output results are written sequentially in one file.

Task-based In the case of the task-based version, we adopt a more coarse grain approach. We do not parallelize the feature extraction stage, instead we taskify the whole frame processing, allowing concurrent execution of all frames. The parallel kernels of the annealed particle filter stage are taskified in our version, and synchronization is achieved by dataflow annotations. Figure 4.3 shows a task graph of our implementation. The directed edges show the dependencies between the different tasks, which dictate the execution order of the tasks. Each frame needs to be written when calculations are completed. In our version we can do this asynchronously while the threads are busy with the processing stage of another frame. Thus, output I/O is effectively overlapped with computation stages.

Figure 4.2 shows parallel executions of two different task-based implementations: The first one just mimics the Pthreads behavior (4.2a) and the second is an optimal task-based implementation (4.2b). Different colored boxes represent different task types, as well the duration of that task type on each core. In both cases, the white gaps denote the time each thread spends idle. Both figures show the same duration for each execution. In the optimal version, all functionality is implemented within the frame-processing task, thus execution time for read-frame, edge-detection and edge-smoothing is represented with blue color (frame-processing). Tasks particle-resampling and calculate-particle-weights are also implemented as nested tasks. They are displayed with different colors (green and yellow respectively). We can observe that the Pthreads-like version suffers from greater idle time compared to its optimal task-based counterpart. Work is distributed more efficiently in the optimal implementation by processing different frames concurrently. This allows us to overlap I/O and serial code segments of one with available work from another one.

Canneal This kernel uses a cache-aware simulated annealing [9] to optimize routing cost of a chip design. `Canneal` progressively swaps elements that need to be placed in a large space, eventually converging to an optimal solution. The problem is stored as a list with routing costs between nodes.

Pthreads This version compares random element pairs of the graph concurrently and swaps them until it converges to an optimal solution. No locks are used to protect the list from concurrent accesses/writes, but swaps are done atomically instead. However, the evaluation of the elements to be swapped is not atomic. This means that disadvantageous swaps may occur, which will require the

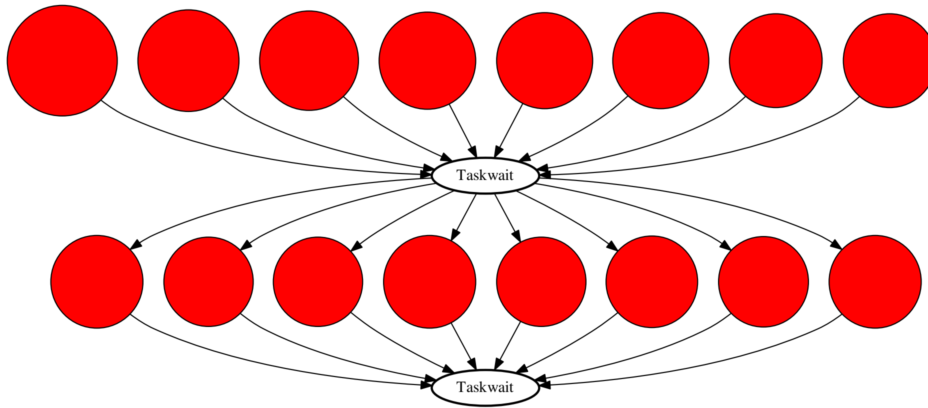


Fig. 4.4 Task-graph of canneal application. Only barrier synchronization among tasks.

algorithm to eventually swap them again. This method has provided better results than the alternative algorithm with locks [15].

Task-based Our task-based version follows the same paradigm. Several tasks are spawned without any dependencies between themselves. We use the same atomics as with the Pthreads version. Since tasks work with an arbitrary number of list elements, it is not possible to describe which elements of the list a task is going to randomly access. In the task graph in figure 4.4 it is clear that there are no data dependencies. Synchronization is only achieved through the use of barriers.

We also try an alternative fine grain implementation, where a task is spawned for each random pair of list elements. This would allow the runtime to know if two tasks are working on the same list of elements. However this implementation implied fine-grain tasks. Each task would merely do a single swap between two list elements. The overhead of the dynamic scheduling is a problem in this scenario. A more complex but more efficient solution is suggested by Symeonidou et al. [139] with the use of memory regions. Adopting this method in a task-based model would allow the programmer to describe parts of the list (or other pointer based data structures) and express dataflow relations as abstract memory regions. This solution also implies fine-grain tasking and is not evaluated at the Symeonidou's work.

Dedup The `dedup` kernel is used to compress data streams using local and global compression to achieve higher compression rates. This method is called deduplication [117].

Pthreads: Dedup is parallelized using a pipeline model with the following stages:

- **Fragment**: First, the data-stream is read and partitioned at fixed positions into coarse grain data chunks. Each chunk can be processed individually by the rest of the stages. This stage is executed on a single thread.

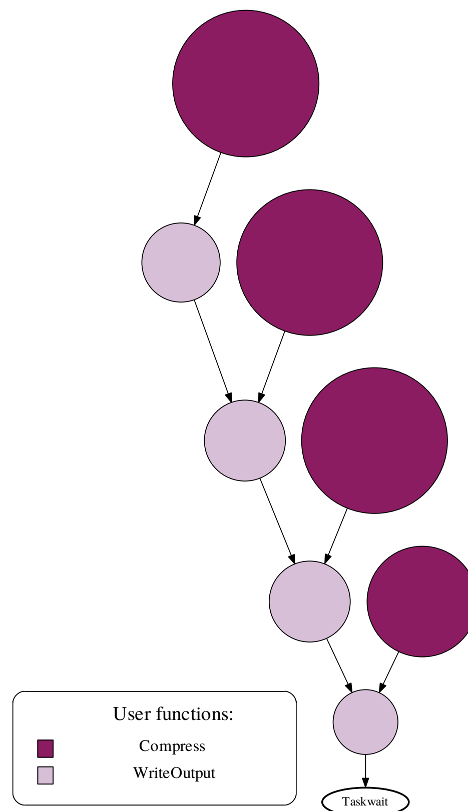


Fig. 4.5 Task-graph of dedup application. Data dependency edges force the correct order of writting the data chunks to the output.

- **Fragment Refine**: A new data chunk initiates the second pipeline stage, where it is further partitioned into smaller fine-grain chunks. The portioning is done by using the Rolling-fingerprint algorithm.
- **Deduplication**: This stage eliminates duplicate fine-grain chunks. Unique chunks are stored in a hash-table. Locks are used here to protect each bucket from concurrent accesses.
- **Compress**: At this stage chunks are compressed in parallel. Identical chunks are compressed only once as duplicates are removed at the deduplication stage.
- **Reorder**: This stage writes the final compressed output data to a file. It writes only unique chunks' compressed data and for the duplicates it stores their hash values. However this stage needs to reorder the data chunks as they are received to match the original order of the uncompressed data.

The Pthreads version maintains a queue and a thread pool dedicated to each stage. When a chunk becomes available at one stage, it is moved to the queue of the next stage. Each stage polls at its queue for available chunks to process. The reorder stage is done sequentially with a devoted thread

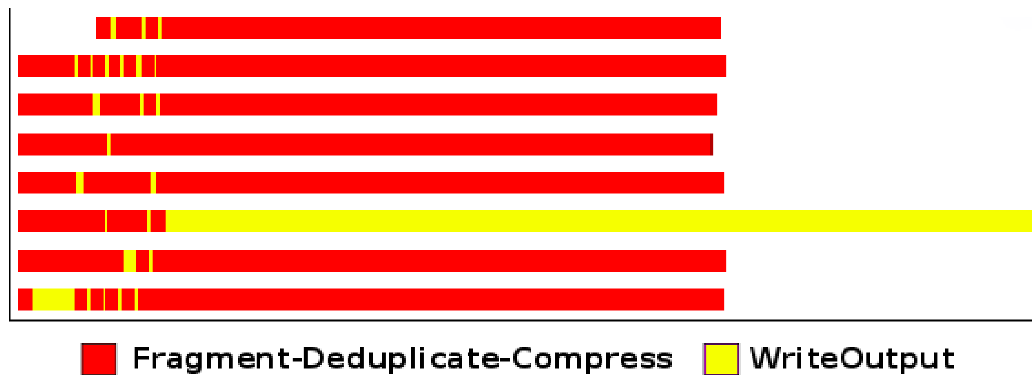


Fig. 4.6 Parallel execution of the task-based version of `dedup` on an 8-core machine and native input size. Different task types correspond to different colors.

that can be in an idle loop waiting for previous stages to finish. Each thread pool comprises by a number of threads equal to the number of available cores. The only exceptions are `Fragment` and `Reorder` stages, which are served by a single thread each.

Task-based In our implementation we taskify each pipeline stage and express data dependencies using static arrays and dataflow relations, one for each pipeline stage. `FragmentRefine` however partitions the data chunks into very fine grain segments, ranging from a few hundreds to thousands. For such granularity, our approach suffers from high overheads due to dynamic schedulnig overhead. The same is observed in [148], where an alternative approach is adopted. In their approach, two pipelines are identified: The outer pipeline, consisting of stages `Fragment`, `InnerPipeline` and `Reorder`. The inner pipeline consists of `FragmentRefine`, `Deduplicate` and `Compress`. To reduce the dynamic scheduling overhead, they merge together `Deduplicate` and `Compress`. By doing so, the available parallelism is limited, but still there is enough work not to harm performance and scalability. In our approach, we merge together the inner pipeline, creating one sequential function, exploiting only the parallelism available in the outer pipeline. Even in this scenario, the available parallelism is still abundant, since the application is bound by the writing of the output file, which is sequential. Figure 4.6 shows a trace of the task-based version. We can see that communication stage (in yellow) is effectively overlapped with the computation stage (in red), however, there is not enough work to keep all the threads finish, until the end of the execution.

Furthermore, we modify the `Reorder` stage, by replacing it with a simple stage where the chunk is simply written to file (`WriteOutput`). Using dataflow relations and a shared output resource between the `WriteOutput` tasks, we ensure that chunk `N-1` will be written before chunk `N`. Thus, we do not need to reorder data chunks in this task type. Moreover, the scheduler makes sure chunks are written as soon as they become available by the `InnerPipeline` task, an improvement over the Pthreads version, where `Reorder` instances need to idle wait until all previous chunks ones have been written. Figure 4.5 shows the dependencies among tasks. We can observe that `WriteOutput` tasks will be

run in the correct order, as soon as their dependencies are resolved. Another difference between the two versions, is that Pthreads oversubscribe threads to cores for each pipeline stage, while in our implementation we only assign one thread to each core.

Facesim Computes a visually realistic human face animation by simulating the underlying physics. As input it uses a 3D model of a human face containing both a tetrahedra mesh and triangulated surfaces for the flesh and bones, respectively. Additionally it uses a time sequence of muscle movement [134].

Pthreads The application statically decomposes the original tetrahedron mesh into smaller partitions, equal to the number of available threads. There are three main parallel kernels:

- **Update State**: Calculates the steady properties of the mesh, constrains like stress and stiffness.
- **Add Forces**: Computes the force contribution between vertices acting on the 3D model.
- **Conjugate Gradient (CG)**: An iterative method that solves the linear system produced by the other two previous kernels and find the final displacement of the vertices for the current frame.

`Update_State` and `Add_Forces` kernels consist of one and two parallel loops respectively, while `CG` has three. Synchronization between loops and kernels is achieved by barriers. The corresponding force computations from the skeleton are also done in `Update_State` and `Add_Forces`, but after the parallel computations on the tetrahedra mesh have been made. In Pthreads a master thread is assigning work to all threads in a round-robin fashion through a queuing system. Each thread maintains its private queue, which is protected by locks.

Task-based In the task-based version, the application level queuing system is completely replaced by the OmpSs runtime. In our initial implementation all parallel loops are taskified. Additionally, in `Update_State` there is a sequential code segment, `Update_Collision_Penalty_Forces`. This code segment operates on the bones, while the parallel loop of `Update State` does so on the tetrahedra. By taskifying it and adding dataflow relations between this Section and the following `Add_Forces` kernel, we can overlap `Update_Collision_Penalty_Forces` with the rest of `Update_State`.

To improve performance we refactor tasks' creation in `CG` by nesting the first task creation loop inside another task. This enables us to overlap task creation time with computation, which contributes to increase `Facesim's` task-based implementation performance. Although we achieve better scalability than the original code, task creation still imposed overheads. To address this issue we replace tasks in `CG` with the OmpSs parallel loops construct (equivalent to the OpenMP one), which implements loop worksharing with a task. Even though this approach limits the available parallelism (barrier synchronization, no dataflow annotations), the overhead associated to task creation and scheduling is greatly reduced and overall performance improved.

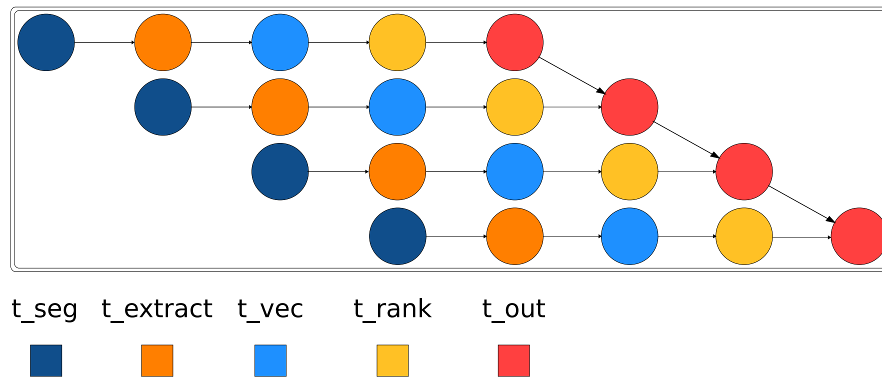


Fig. 4.7 Task-graph of ferret showing the pipelined execution model. Edges show data dependencies among different tasks.

Ferret Content similarity search server for feature-rich data [98] like audio, video, images, etc. The benchmark application is configured for image similarity search.

Pthreads Ferret is parallelized using a pipeline model. A serial query is broken down into 6 pipeline stages:

- **Load**: This stage loads an image that is going to be used as a query.
- **Segmentation**: At this stage the image is decomposed into the different objects displayed on it. Different weight vectors are to be assigned on each object to achieve better results.
- **Extract**: At this stage a 14-dimensional vector is computed for each object from the segmentation stage, describing features such as color, area, state, etc.
- **Vectorization**: This is the indexing stage that tries to find a set of candidate images in the database.
- **Rank**: This stage ranks the results found, using the EMD metric for each query-object's vector and the database's image vectors.
- **Output**: Outputs the result of the ranking stage. Multiple instances of this stage need to run serially, since they all share the same output stream.

In the Pthreads version every stage is served by a dedicated thread-pool of N threads each, where N is the number of available cores. The only exceptions are the `Load` and `Output` stages that are executed by a respective single thread. Each stage polls on its corresponding queue for available work. When a stage finishes, it pushes the results to the next stage's queue.

Task-based In this version, we implement a variation of this pipeline model. As soon as the first stage, `Load`, finds a new image, it spawns all stages of a pipeline for that image, thus reducing the pipeline to five stages. We model the dataflow relations between different stages as simple one dimension arrays, as shown in Figure 2.1. Tasks working on different image queries do not share any dependencies. An exception is task `t_out` which shares the same output file between all pipelines, thus sequential execution is forced between all instances of this task. The pipeline stages and dependencies are constructed a priori, which is good enough for this application, but this is not always the case. [93] proposes a system that can handle dynamic pipeline creation by constructing a DAG with the stages using indexes and the `cilk_continue` and `cilk_wait` keywords. Indexes are used to define the different pipeline stages, while `cilk_continue` creates a stage that can run once all previous stages in the same pipeline iteration are done, and `cilk_wait` creates a stage that will wait for its stage counterpart of the previous iteration to finish. A strategy based on versioning the dependency objects between the stages has been proposed [149]. Output dependencies are renamed and privatized, thus the static array for privatization is not required.

Figure 4.7 shows the task-graph of the `ferret` application. Colored nodes denote the concurrent tasks (each color matches a specific task type). Tasks that have data dependencies are connected by directed edges. By inspecting the task-graph we can see a pipeline pattern of execution. Despite the fact that the task-based approach does not significantly improve the overall performance, as we can see in Section 4.5, it significantly reduces the effort required to express the pipeline parallelism, compared its Pthreads counterpart, as it is shown in Section 4.4.1 in detail.

Fluidanimate This application simulates incompressible fluid interactive animation, using the Smoothed Particle Hydrodynamics (SPH) method [108].

Pthreads `Fluidanimate` uses five special kernels which are responsible for rebuilding the spatial index, computing fluid densities and forces at given points, handling fluid collisions with the scene geometry and finally updating particle locations The fluid surface is partitioned and each thread works on its own grid segment. The kernels are parallelized as do-all loops, separated by barriers. Moreover, there are cases where these threads need to update values beyond their partition, which are handled using locks.

Task-based The task-based implementation follows the same approach, we apply a loop tiling transformation, for each parallel loop, and taskified each iteration. Figure 4.8 show how dependencies form among between tasks. Tasks from different loops can run concurrently, as soon as their dependencies are met. For example we see that the first task of each loop, only requires that the first two tasks from the previous loop finish their execution to have their dependencies resolved. We maintain the same barrier and lock synchronization scheme, using the OmpSs synchronization primitives.

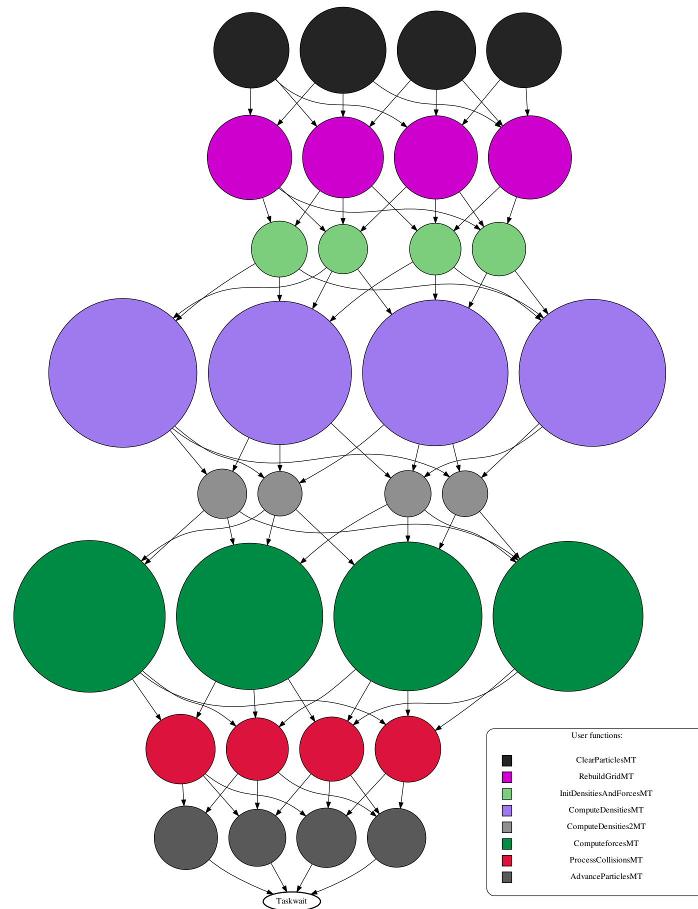


Fig. 4.8 Task-graph of fluidanimate application. Edges represent data dependencies among tasks

Freqmine Data mining application that makes use of an array-based version of the Frequent Pattern (FP) growth method for Frequent Itemset Mining [69].

Pthreads The application uses a compact tree data structure, denoted *FP-tree* [72], to store information about frequent patterns of the transaction database. The FP-tree is coupled with a header table, which is a list of database items, sorted by decreasing order of occurrences. The FP-growth algorithm traverses the FP-tree structure recursively, constructing new FP-trees until the complete set of frequent itemsets is generated. There are three parallel kernels. The `Build_FP-tree_header_table` kernel performs a database scan and counts the number of occurrences of each item. The result is the FP-tree header table. `Build_Prefix_tree` kernel performs a second database scan required to build the prefix tree and the `Data_Mining` kernel obtains the frequent itemset information by using the previous two structures. It creates an additional lookup table, which allows faster traversals on sparse itemsets. The original PARSEC benchmark uses OpenMP2.0 for loop parallelization inside each kernel.

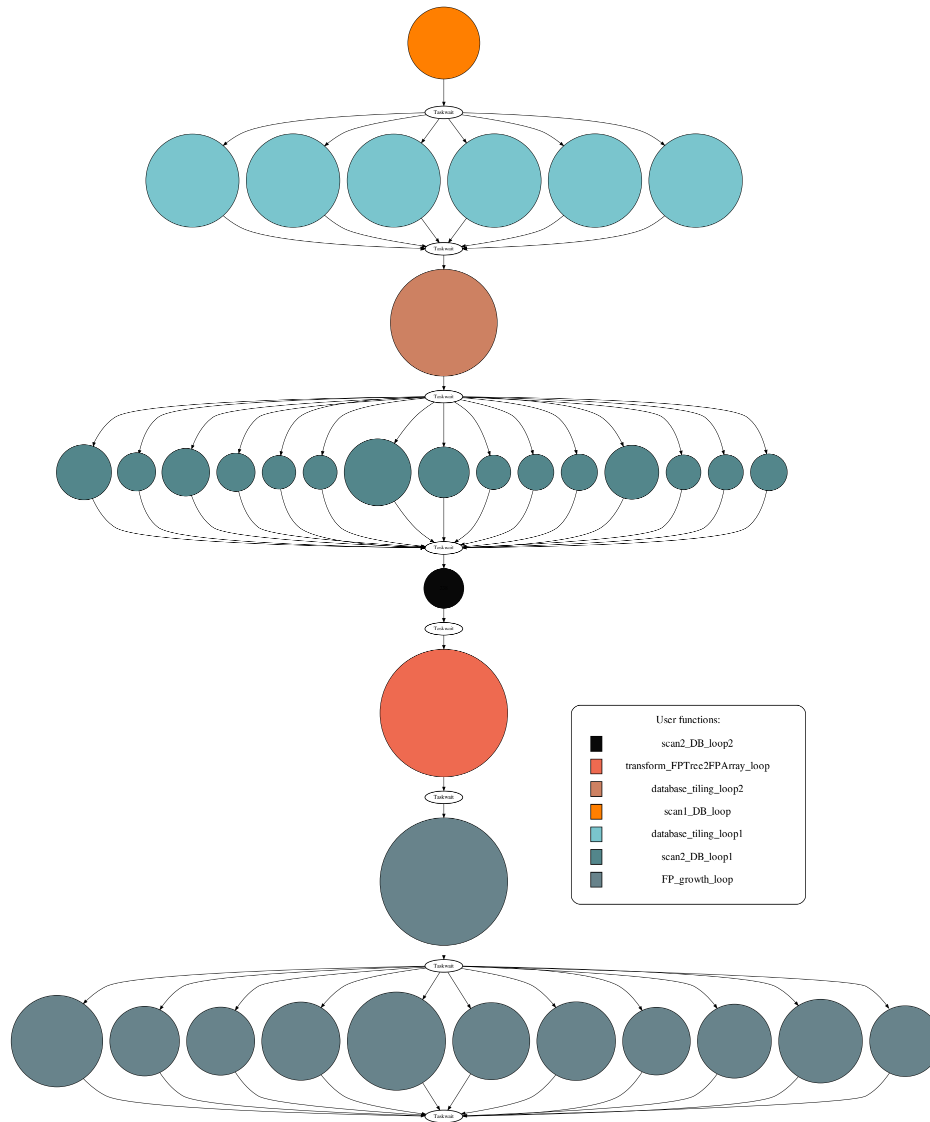


Fig. 4.9 Task-graph of freqmine application. Edges represent data dependencies among tasks.

Task-based In our implementation we taskify each iteration. We do not use any dataflow relations in this application, and resolve to adopt the locking and barrier synchronization used in the original OpenMP version. The barrier synchronization is shown in the task graph in figure 4.9.

Streamcluster Streamcluster is a kernel that solves the online clustering problem. It takes a stream of points and then groups them in a predetermined number of clusters with their respective centers.

Pthreads Up to 90% of total execution time is spent in function `pgain`, computing whether opening a new center is advantageous or not. For every new point, function `pgain` calculates the cost of

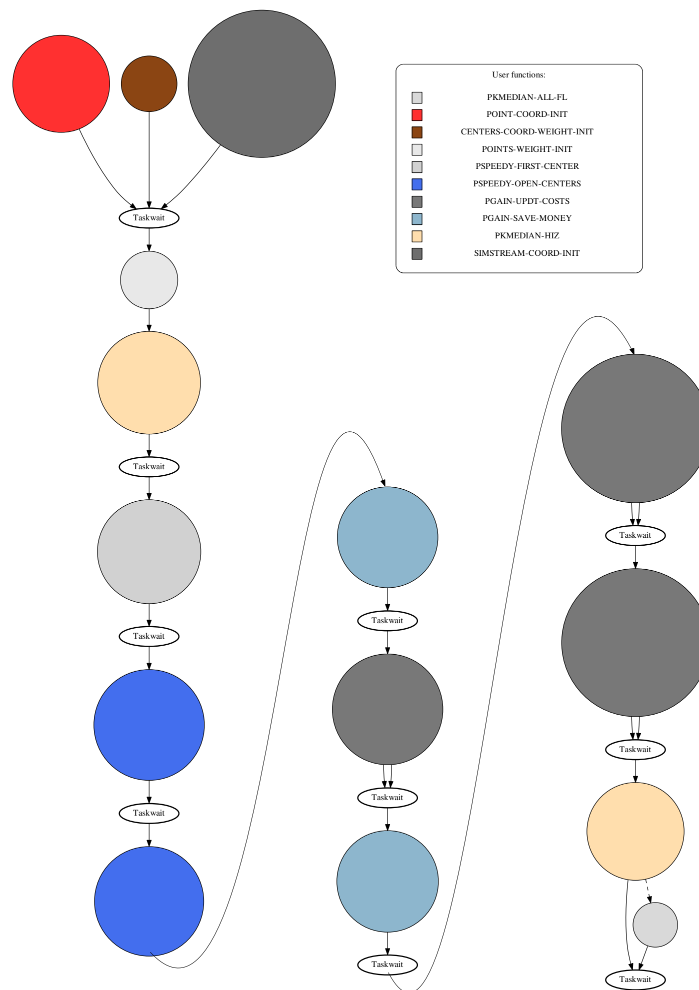


Fig. 4.10 Task-graph of streamcluster application. Edges represent data dependencies among tasks.

making it a new center by reassigning some points to it and comparing it to the minimum distance $d(x,y) = |x-y|^2$ between all points x and y. The result is accepted if found to favor the new center. Data points are statically partitioned by a given block size, which determines the level of parallelism in the application. In the Pthreads version this is equal to the number of threads.

Task-based In our implementation we follow a different decomposition strategy, making the number of tasks independent of the number of partitions. Barriers are employed to synchronize accesses to a partition in both Pthreads and the task-based implementation. In the case of Pthreads, an additional user implemented library is used for the barriers. This library is not required in the case of the OmpSs implementation, as the runtime already has a generic barrier implementation. A task graph of our task-based implementation and the dependencies among tasks is shown in figure 4.10

Swaptions Economics application that uses the Heath-Jarrow-Morton (HJM)[74] for pricing of a portfolio of swaptions. To calculate prices it employs the Monte Carlo simulation.

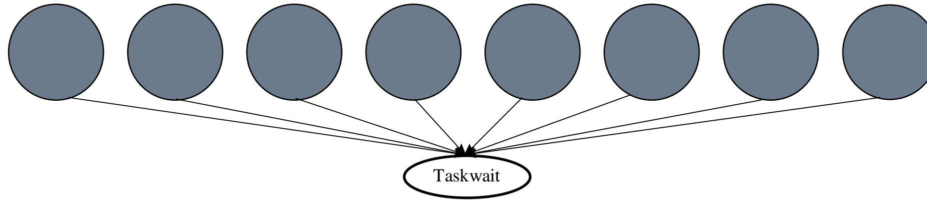


Fig. 4.11 Task-graph of swaptions application.

Pthreads The application stores the portfolio into an array. In the Pthreads version, this array is divided by the number of available threads, each thread working on its own part of the array.

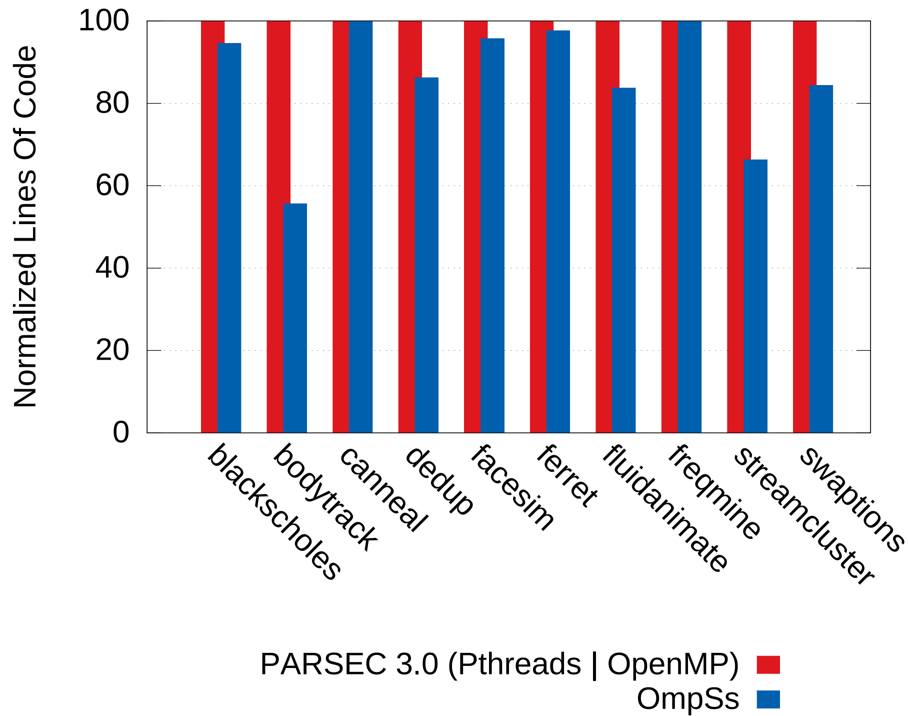
Task-based We use the exact same strategy, where each task works on a part of the array. No data dependencies exist between the tasks. Figure 4.11 shows the corresponding task graph for the swaptions application. No dependencies exist between tasks, synchronization is only achieved through barriers.

4.4 Programmability

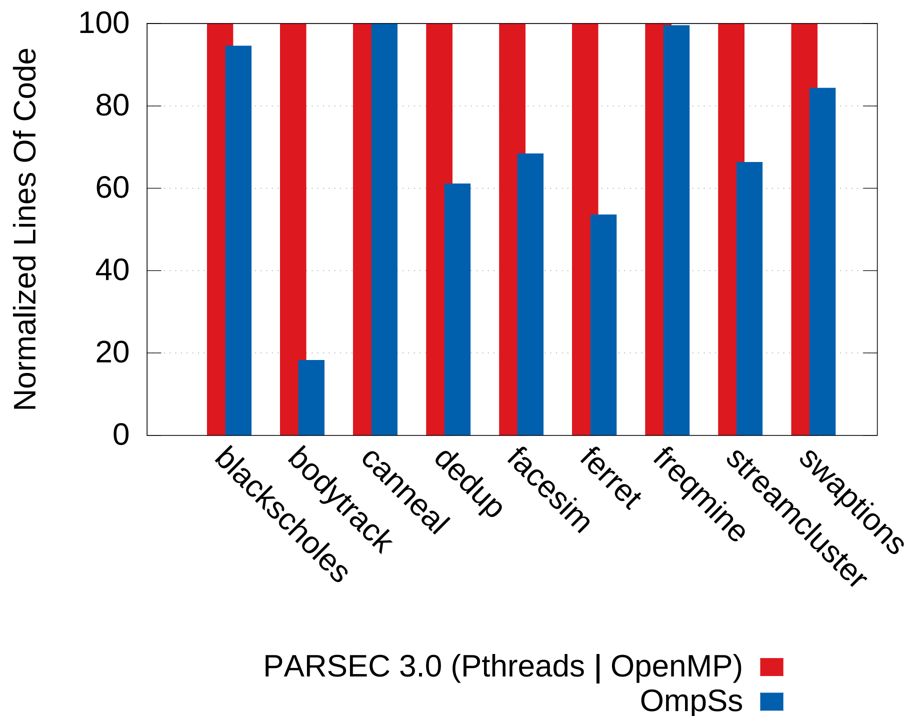
Different models and languages offer diverse ways to express concepts, such as parallelism or asynchrony. In this section we evaluate how successful and easy it is to express parallelism using task-based models. A good proxy to evaluate how complex a particular implementation is the number of lines of code it takes. Despite being a metric proposed some decades ago, comparing different programming models in terms of the total number of code lines is still a valid metric. Indeed, recent publications make an extensive use of it [49, 149].

4.4.1 Lines Of Code

The reduction of the lines of code (LOC) attests to a more compact and readable code. In some of our PARSEC task-based implementations, a simple pragma directive replaced application specific schedulers, scheduling queues, thread pooling mechanisms and lock synchronization. We do not change the algorithm in any of the task-based parallel strategy implemented in the PARSEC suite. Figure 4.12a shows a normalized comparison between the lines of code of our task-based implementations and the original Pthreads/OpenMP implementations of the PARSEC 3.0 distribution. The PARSEC 3.0 versions we refer to are always the Pthreads versions, except in the case of `freqmine` where, since there is no Pthread version available, the OpenMP2.0 version is taken as reference. We preprocess all



(a) Comparison between all source files.



(b) Comparison between only source files containing parallel code.

Fig. 4.12 Comparison of lines of code between our task-based implementations and the original Pthreads or OpenMP versions.

source files so that they only contain lines of code relevant to the respective programming model[1]. Figure 4.12b shows the total lines of code comparison when we only consider files that are relevant to the parallel implementation, that is, files that contain calls to Pthreads or task invocations, asynchronous I/O implementations, atomic primitives, etc. In this graph we see that the reductions in terms of lines of code of our task-based strategies are significant. In case of `bodytrack`, we are able to remove 81% of the code lines. Since `Bodytrack` implements its own scheduler to deal with load balancing, there is much room for code reductions by replacing this ad-hoc mechanisms for a few pragma annotations.

By using tasks and dataflow relations, it is very easy to implement pipelines. We adopt this approach for both `dedup` and `ferret`, which result in a significant decrease in LOC (38% and 46%, respectively). Figure 2.1 shows the pipeline code for `ferret`. All that is required is to taskify the different pipeline stages and make sure that dataflow relations force in-order execution of tasks in the same pipeline instance. The Pthreads version requires the implementation of queues between each stage, which must also be safe to use by multiple threads and concurrent accesses. `Streamcluster` and `fluidanimate` task-based versions also reduce lines of code by 33% and 21% respectively, by removing the need for an additional, user implemented, barrier library. `Blackscholes` and `swaptions` are relatively simple applications, containing only one do-all parallel loop each. In these cases the LOC difference is minimal (0.5% and 15%, respectively). In the cases of `canneal` and `freqmine` we see no difference in LOC. `Canneal` is not a data parallel application and in both cases Pthreads and tasks are used merely as thread launching mechanism, while the synchronization effort is essentially the same.

It is worth noting that conventional synchronization primitives can still be used with tasks, without penalizing the programmer. `Freqmine` is implemented in OpenMP, which excels at parallelizing loops with very little effort from the programmer and is the ideal programming model for this application. In our implementation we simply taskify the loops, essentially not affecting LOC. `Facesim` also benefits from the tasks-based approach by 37%, as the queues required to schedule work have been completely removed. Overall, we see that the task-based model reduces code size and by 28% on average.

4.5 Performance

In this section we compare our task-based implementations to the original PARSEC implementations in Pthreads or OpenMP.

4.5.1 Scalability

Figures 4.13,4.14,4.15,4.16 shows how the task-based codes scale compared to the PARSEC Pthread-/OpenMP versions. Results are shown individually per benchmark as we increase the number of cores

[1]PARSEC benchmarks contain mixed serial, Pthreads, OpenMP and TBB source code, and make use of macros to enable conditional compilation for only one programming model at a time.

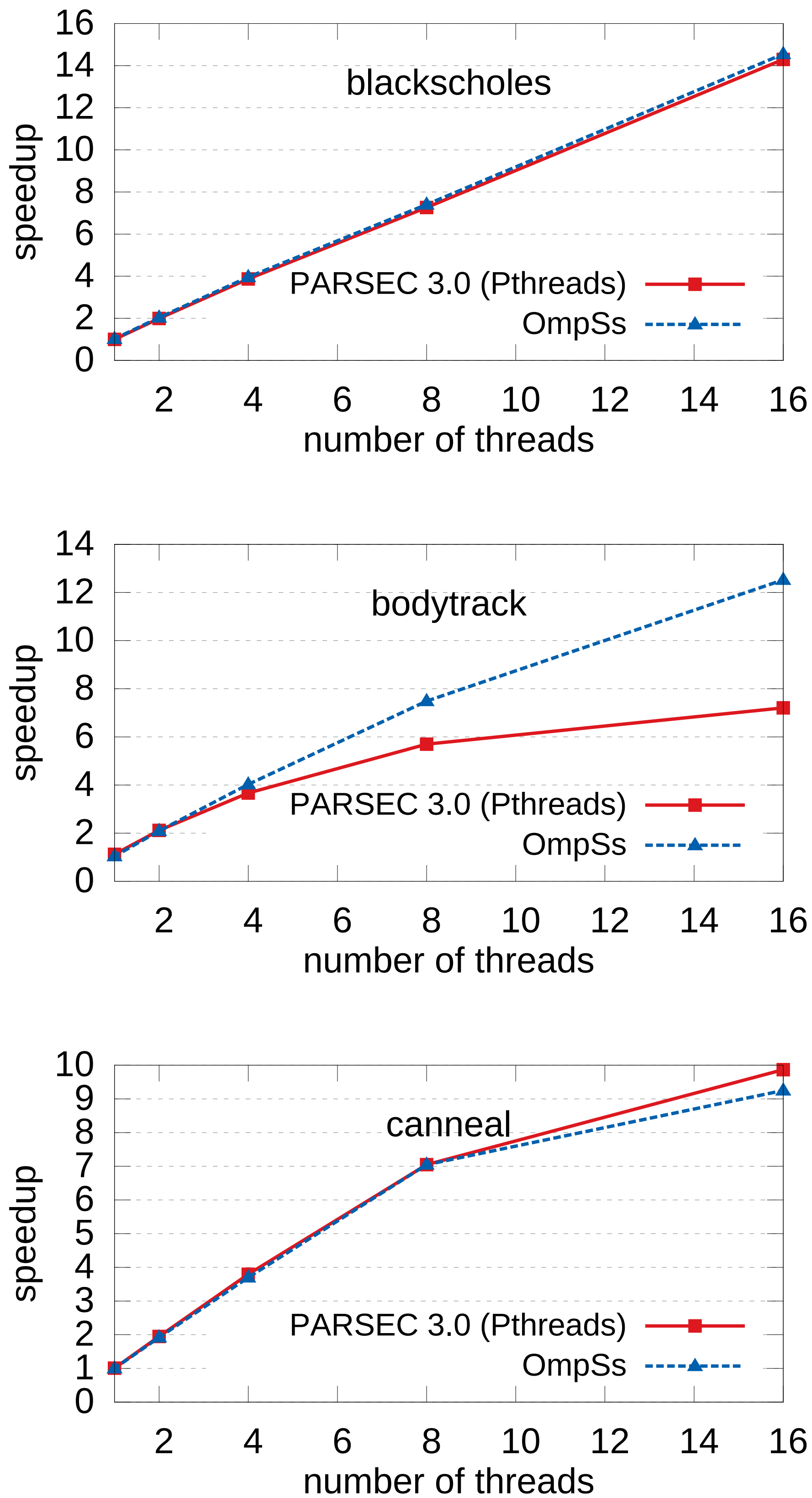


Fig. 4.13 Comparison of scalability between the task-based implementations and the original (Pthreads/OpenMP) versions.

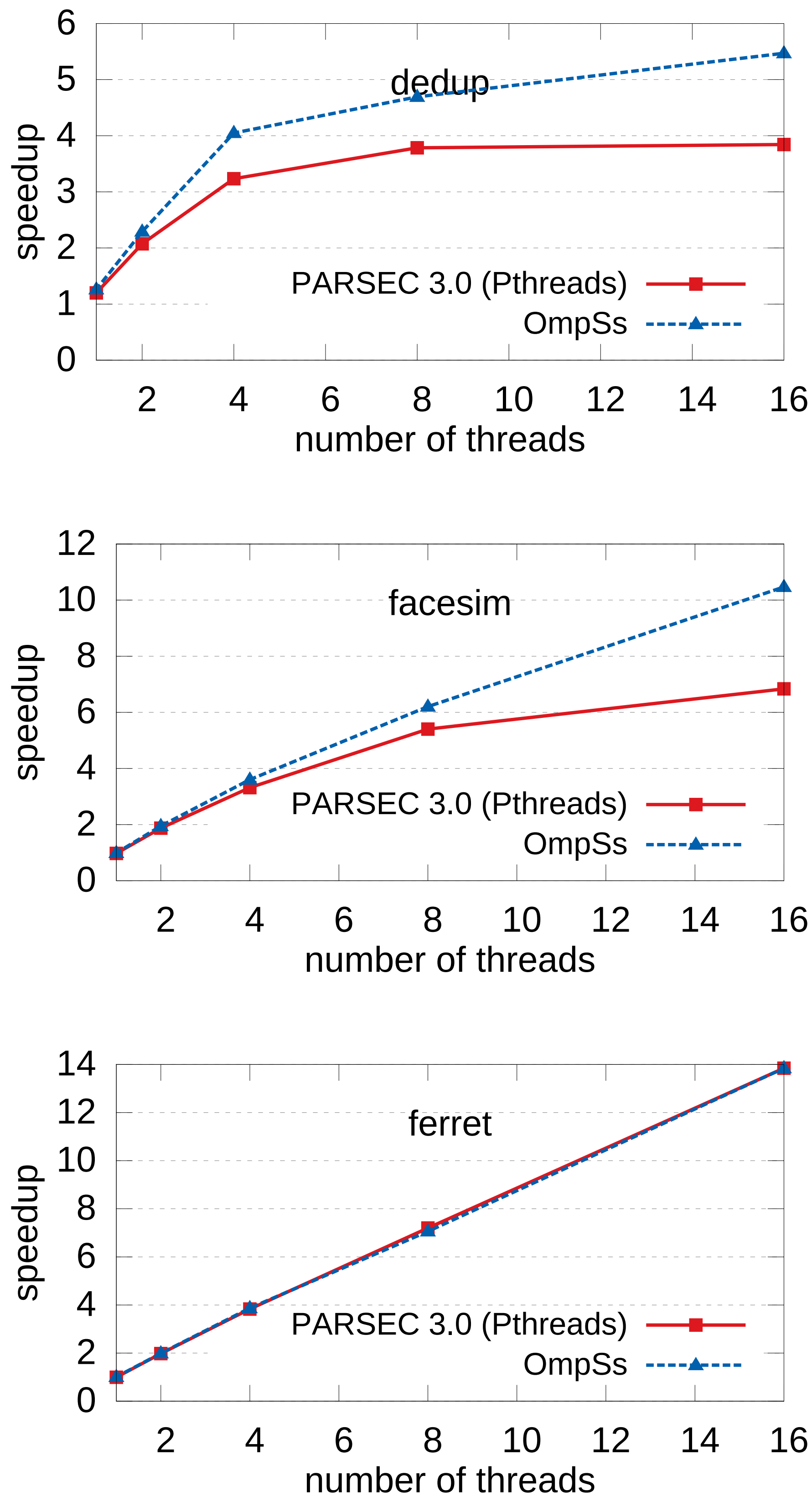


Fig. 4.14 Comparison of scalability between the task-based implementations and the original (Pthreads/OpenMP) versions.

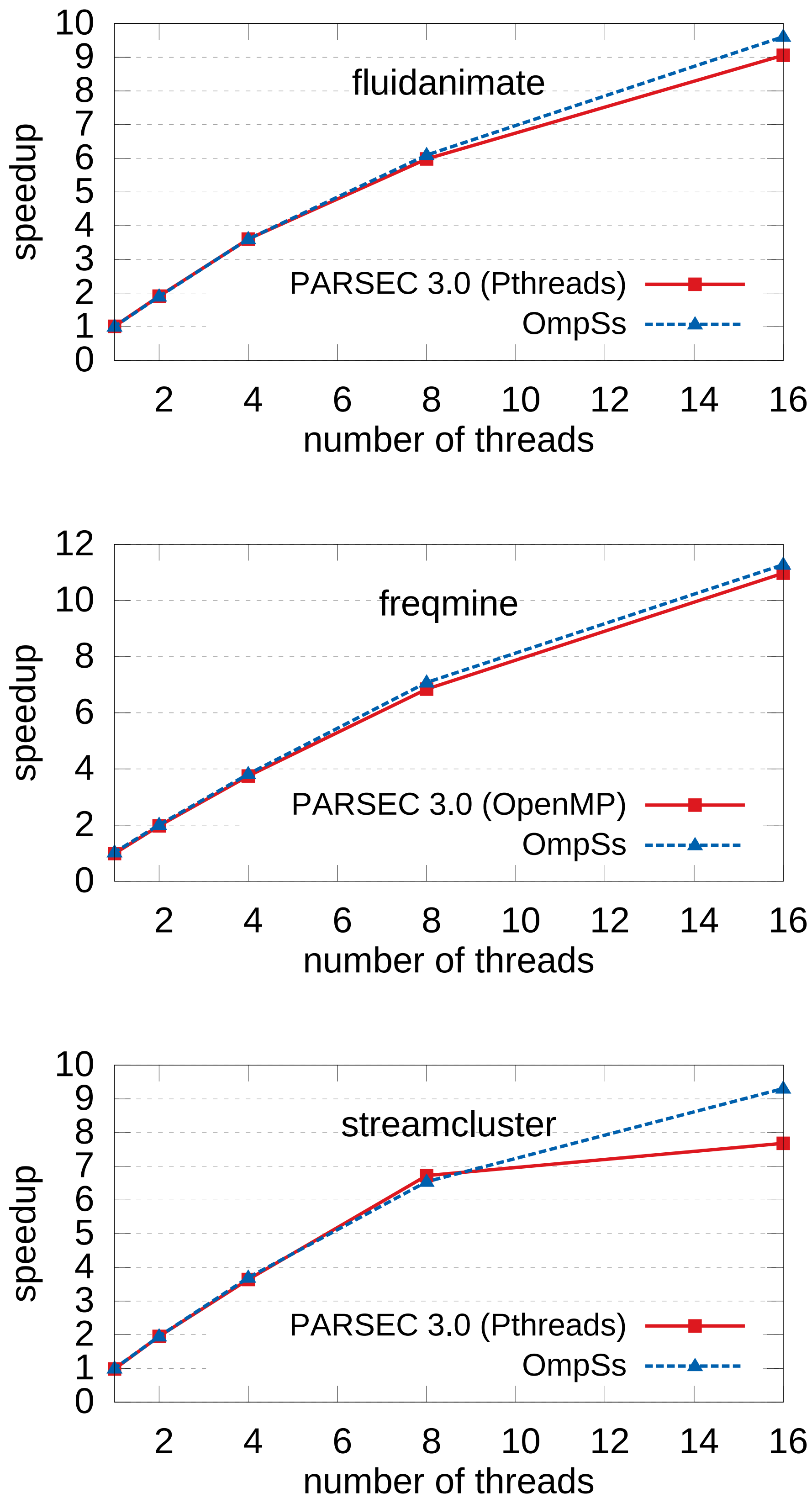


Fig. 4.15 Comparison of scalability between the task-based implementations and the original (Pthreads/OpenMP) versions.

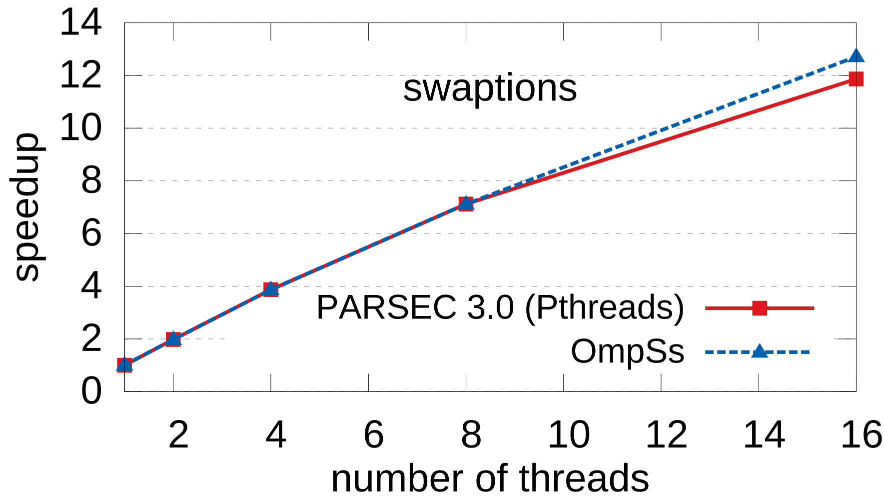


Fig. 4.16 Comparison of scalability between the task-based implementations and the original (Pthreads/OpenMP) versions.

assigned to the application and normalized to the execution time of the serial implementation [2] of the application. Nearly all applications scale linearly up to 4 cores.

In the case of `bodytrack`, as described in Section 4.2, by concurrently executing different frames, there is always enough work for all threads, while by taskifying the output stage of each frame, we overlap this I/O bottleneck with other computation stages. The speedup when run on 16 cores is 12.1x, while the Pthreads implementation reaches a poor 6.8x speedup when run on 16 cores. The `dedup` application has a very expensive stage that writes the compressed data to the output file. Our task-based implementation is very effective in overlapping this time with computation from the compression stage. Also, the task-based version does not have to reorder the data chunks, since the I/O execution takes place in-order as dictated by dataflow relations. This results in an impressive 30% performance improvement of the OmpSs version with respect to Pthreads when run on 16 cores. The Pthreads `facesim` implementation is burdened by barriers that limit available parallelism. By using dataflow relations we taskify sequential segments of significant cost we effectively synchronize them with parallel sections preceding and following it. The performance improvements comes from the overlap of sequential computations with parallel sections. The task-based parallelization of `facesim` reaches a speedup of 10.2x when run on 16 cores, while the PARSEC code only reaches a 6.4x speedup.

In the cases of `blackscholes`, `canneal`, `ferret`, `fluidanimate`, `freqmine` and `swaptions`, the Pthreads/OpenMP versions already achieve good scalability results. With the exception of `ferret`,

[2] The PARSEC benchmark suite provides a serial implementation for `blackscholes`, `bodytrack`, `dedup`, `ferret`, `freqmine` and `swaptions`. For the other benchmarks, the original Pthreads parallel implementation executed on a single core is considered as the baseline.

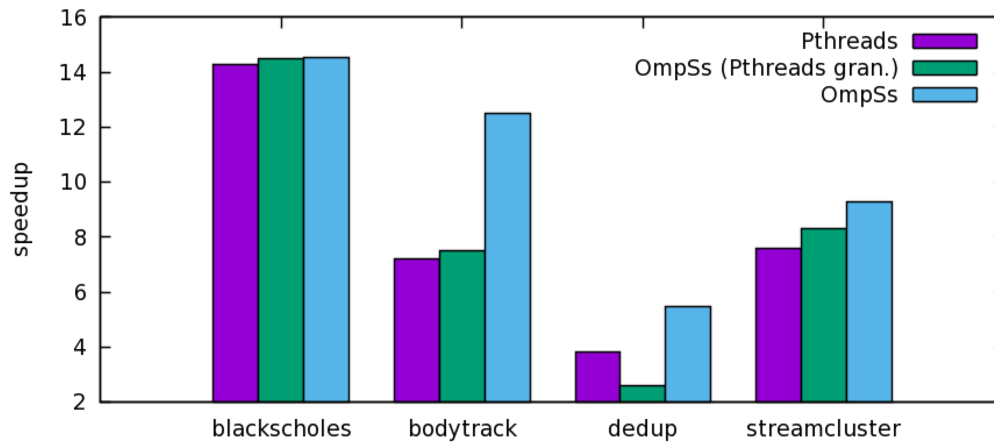


Fig. 4.17 Speedup comparison for Pthreads and OmpSs with the same granularity, as well as our optimized OmpSs version, when run on 16 cores. Results are normalized to the sequential version of the original code.

the task-based codes have very close resemblance to their Pthreads/OpenMP counterparts, and have offered reduced opportunities for OmpSs to dynamically exploit additional parallelism. The parallel implementation in these applications, with the exception of `ferret`, is limited to parallel do-all loops with barrier synchronization, essentially exploiting the same amount of parallelism among all versions (OmpSs/Pthreads/OpenMP).

In the case of `ferret`, although the code is substantially different, both versions employ the same pipeline model and deliver the same level of parallelism, which is already high in the Pthread version. We express a bit of extra parallelism by extending the pipeline with multiple stages, which write to the output file, effectively overlapping some communication with computation. However, the final impact in the total execution time is limited as the time needed to write the output file is a very small fraction of the total execution time. Finally, we observe performance gain (18%) in `streamcluster`, which can be partly attributed to the more efficient barrier implementation of OmpSs, when compared to the user implemented barriers of the Pthreads version. However, the most important performance drawback that the original Pthreads implementation suffers from, is the negative NUMA effects. This issue is observed when we run our experiments on a two socket system. The Pthreads code partitions the working set by the number of available cores. We employ a different partition scheme to counter the NUMA effects. Through experimentation we observe that the best results can be obtained when using 80 blocks.

4.5.2 Task Granularity Impact

The granularity of individual tasks is an important factor that needs to be considered when parallelizing an application. Small task granularity can reduce load imbalance but such performance benefits can be

neglected by the overhead of the runtime system, as it has to create and schedule more tasks. Results in Section 4.5.1 show how tuning the task granularity brings performance benefits in some cases (`blackscholes`, `bodytrack`, `dedup` and `streamcluster`) while in others it is better to keep the same parallel granularity as the PARSEC distribution codes (`canneal`, `facesim`, `ferret`, `freqmine` and `swaptions`).

In order to provide a more comprehensive comparison, this section examines the performance of `blackscholes`, `bodytrack`, `dedup` and `streamcluster` when using exactly the same granularity as in the PARSEC distribution code. Figure 4.17 shows the speedup of these benchmarks when run on 16 cores. The purple bar shows the speedup of the Pthreads version, the green one shows the speedup of a task-based implementation that has the same parallel granularity as its Pthreads counterpart. Finally, the light blue bar shows the speedup of the optimal task-based implementations discussed in Sections 4.2 and 4.5.1.

For the cases of `blackscholes` and `streamcluster` the parallelization scheme followed in the three codes (Pthreads and the two OmpSs versions) is the same. The difference between the two OmpSs versions is the granularity of the block sizes that are processed per task. In case of `blackscholes`, the OmpSs implementation with the same granularity as Pthreads does not perform better since the parallelism of this benchmark follows a fork-join model. In the case of `streamcluster`, the task-based implementations always improve the Pthreads performance, even if they operate following the same parallelization scheme and granularity as the Pthreads version. These improvements come from the NUMA effects correction that the OmpSs versions carry out.

In the case of `bodytrack` the optimal OmpSs implementation follows a quite different parallelization scheme than the original Pthreads code, as explained in Section 4.2. We consider a trivial implementation in OmpSs where we follow the same parallelization strategy as in Pthreads. As shown in Figure 4.17, we do not observe any significant difference in performance among Pthreads and the equivalent OmpSs implementation. However, the new parallelization scheme is not applicable to Pthreads as it requires to synchronize the workload by explicit dependencies, which are not available in the Pthreads API.

In the case of `Dedup`, the trivial Pthreads-like implementation performs poorly, achieving a speedup of 2.6x. In this implementation, each pipeline stage is taskified following the Pthreads approach. Each large chunk is partitioned into smaller chunks, that will spawn three new tasks (`Compress`, `Deduplicate` and `WriteOutput`). This level of granularity creates hundreds of thousands of tasks, increasing the runtime's overhead significantly. In contrast, the optimized task-based version operates at the granularity of the large chunks, creating only a few hundreds of tasks, effectively reducing the runtime overhead.

In some cases, OmpSs can over-perform Pthreads even if the same parallelization scheme and granularity is followed, like the `streamcluster` results demonstrate. In some other cases (`dedup` and `bodytrack`), the performance improvements come from an optimized parallelization scheme. Such new schemes could be hardly implemented in Pthreads since they require a direct synchronization via explicit dependencies, which is not available in the Pthreads API. Finally, in case of simple fork-join

applications (i. e. `blackscholes`) our performance benefits just come from further optimizing the parallel granularity.

4.5.3 Runtime System Overhead

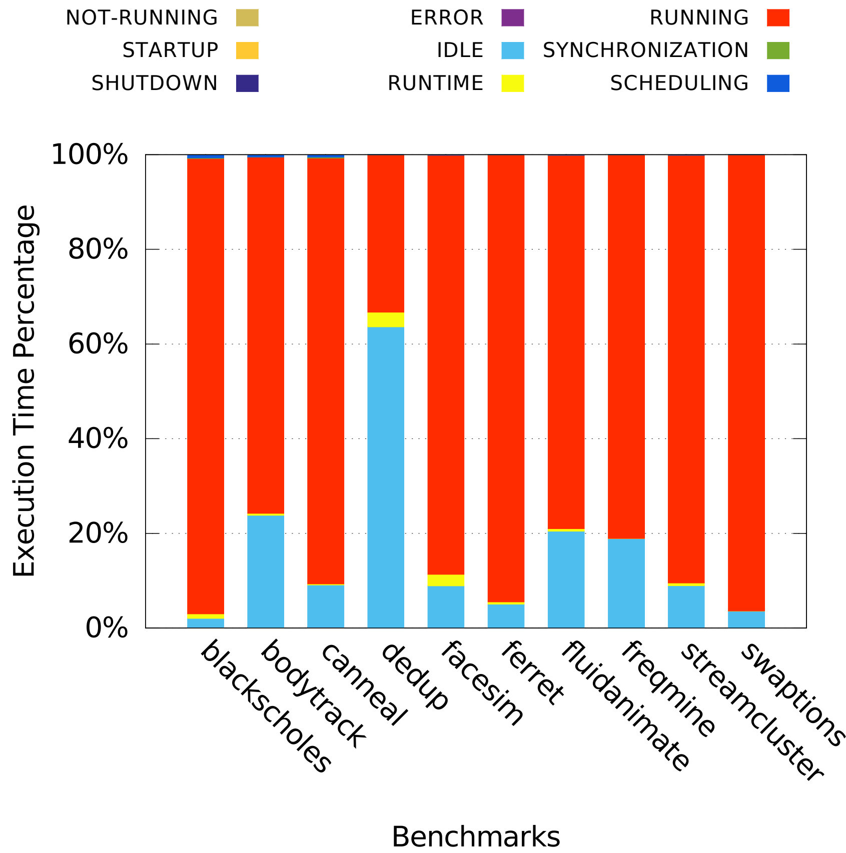
Task creation, scheduling and data dependencies tracking are all handled by the OmpSs runtime system. In this section, we evaluate the impact of these activities over the final parallel performance. Figure 4.18a shows a breakdown of the total execution time of each application. Each bar shows the breakdown of one application after averaging the values over eight concurrently executing threads. The red color represents the portion of time dedicated in running tasks, that is, in running user code. All applications, excluding dedup, spend more than 75% of time doing useful work. The cyan bar represents idle time, which corresponds to the time a thread is waiting for some work to become available and is caused by load imbalance and sequential code phases. In most cases this time is low, except for `dedup`, where it reaches 60% of the total execution time. In Figure 4.18a we also represent the time spent in other activities like synchronization, scheduling, etc. None of these activities represent more than 5% of the total execution time.

Figure 4.18b shows the same breakdown of execution time but only for the main thread of execution, which is the one that runs serial parts of the code besides parallel tasks. `Dedup` has significantly lower idle time in the main thread, which indicates that there is not enough parallel work to keep all threads busy. This issue has been previously reported [148]. In general we see that the overhead of the runtime system is low, with only a few cases that show some time spent in synchronization, scheduling, and miscellaneous runtime overhead (in light yellow). Synchronization time can be time spent waiting a barrier or acquiring/releasing locks. Scheduling includes time needed to resolve dependencies and make scheduling decisions, while other runtime overheads are related to activity that cannot be associated with task scheduling and creation. Overall, we have seen that our implementations improve scalability considerably (by 13% on average), while runtime overhead remains low.

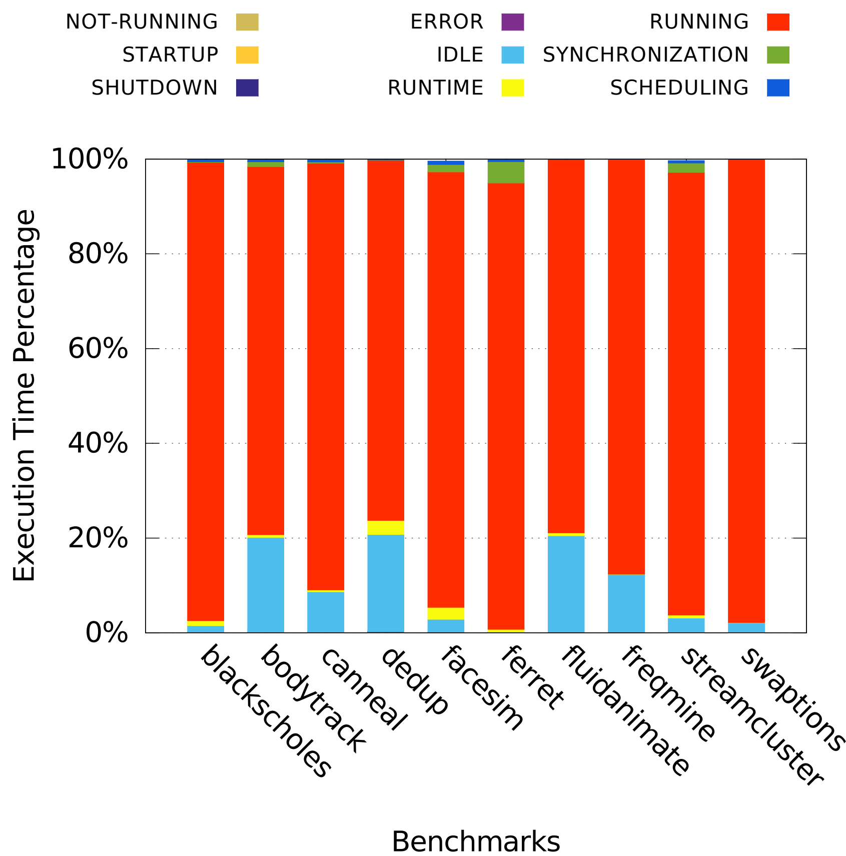
4.5.4 Characterization of the Applications

In Table 4.2 we characterize the considered applications in terms of parallelization model, I/O intensity and synchronization scheme. The table also shows code reductions and performance improvements achieved on a 16-core Sandy Bridge system. This table summarizes the properties of applications that make them good candidates for adopting a task-based model.

Applications characterized as data-parallel are limited to loop parallelism, where tasks are merely emulating an OpenMP loop construct. In these cases there is no performance gain, and the programming effort involved either with Pthreads, OpenMP or tasks, is similar. Pipeline applications are better candidates since they separate the application into discrete abstract stages. Implementing this paradigm with tasks implies taskifying the functionality of each stage and describing the data or control dependencies between them. In Pthreads, the programmer has to implement application specific



(a) Average time over 8 threads



(b) Main thread runtime breakdowns

Fig. 4.18 Runtime breakdowns when running on an 8-core configuration.

Table 4.2 PARSEC parallelization model and properties characterization.

Benchmark	Parallel Model	I/O Heavy	Synchronization	LOC Reduction	Perf. Impr.
blackscholes	data-parallel	✗	dataflow	5.4%	0%
bodytrack	pipeline	✓	dataflow	81%	42%
canneal	unstructured	✗	locks/atomics	0%	-6.2%
dedup	pipeline	✓	dataflow/locks	38%	30%
facesim	pipeline	✗	dataflow/barrier	31%	34%
ferret	pipeline	✗	dataflow	46%	0%
fluidanimate	data-parallel	✗	dataflow/barrier	21%	5.7%
freqmine	data-parallel	✗	barrier/locks	0%	2.7%
streamcluster	data-parallel	✗	dataflow/barrier/atomics	33%	18%
swaptions	data-parallel	✗	dataflow	15%	6.6%

thread pools and queuing systems to achieve the same performance. Also, task-based models offer in many cases an opportunity to easily expand the pipeline stages of the application with sequential and I/O intensive codes (e.g. `facesim` and `bodytrack` respectively). Indeed, by replacing locks and barriers, the runtime can discover additional dynamic parallelism and eliminate the cost of acquiring locks. Our task-based parallelization strategies successfully scale up the pipeline applications with a poorly scaling Pthread version (`bodytrack`, `dedup` and `facesim`) while reducing the code complexity in all of them. In case of `ferret`, the task based version does not perform better than the Pthreads counterpart since its scalability is already very good (14x on a 16-core machine). The reduction in terms of lines of code is however dramatic: 46%. In the case of unstructured programs, e.g. `canneal`, task based programming does not offer any advantage over threading approaches.

Overall, we conclude that task-based parallelism can be effectively used to reduce the effort required to implement pipeline parallelism, while there are also important performance benefits to be gained if the application has no specific thread pooling mechanisms or I/0 intensive serial regions. In this scenario, the pipeline can be easily expanded to include the I/0 region and overlap it with a computation stage of the pipeline.

4.6 Summary

In this Chapter we evaluate the benefits of task-based parallelisim by applying it to the PARSEC benchmark suite. We discuss and compare our implementations to their PARSEC Pthreads/OpenMP counterparts. We show how task parallelism can be applied on a wide range of applications from different domains. In fact, by comparing the lines of code between our implementations and the original versions, we make a strong case that task-based models are actually easier to use. The asynchronous nature of task-based parallelism, along with data dependency tracking through dataflow

annotations, allows us to overlap computation with I/O phases. The underlying runtime system can take care of issues like scheduling and load balancing without significant overhead.

Our experimental results demonstrate that the task model can be easily applied on a wide range of applications beyond the HPC domain. Although, not all applications can benefit from a task-based approach, there are cases where it can greatly improve scalability. The programs that benefit most are those that present pipeline execution model, where different stages of the application can run concurrently. The proposed benchmark suite is expected to be of great use in evaluating experimental software and hardware system designs, offering a more mature testbed compared to the typically used small kernel applications.

5

Power-Aware Runtime Scheduling

One major constraint of future High-Performance Computing (HPC) systems is their power consumption. Agencies have set strict targets for building an exascale machine — e.g., the US Department of Energy has set the limit to 20MW [125] — while others, like the European Union, are investing in novel approaches leveraging mobile technologies to build low-power HPC infrastructures [118]. The resulting need for reducing hardware power consumption has started to force computer architects and vendors to include power capping capabilities in their hardware designs. This allows applications to more efficiently exploit their entire power envelope of a system, while guarding the system against intermediate power spikes. Prior work has shown that this can lead to significant performance benefits [102, 113].

Manufacturing variability, however, causes processors and DRAM memories to react inhomogeneously to power constraints enforced by the system. While already present in current systems, such variability has so far been hidden by varying power consumption to achieve homogeneous performance. In fact, existing studies show a variation of up to 10% in power consumption to deliver the same performance [122]. With the ability to vary power removed by imposing a particular power limit, this variation becomes visible in realized performance [79]. Further, this uneven distribution of delivered performance is specific to each single hardware component, since two nominally identical processors can suffer from different degrees of manufacturing issues. From the HPC applications perspective, this can cause load imbalances, even if the workload is perfectly balanced, resulting in significantly degraded performance.

To make this problem even worse, since such degradations are hardware specific, it is not possible to design static or hardware agnostic techniques to mitigate this induced new type of load imbalance. In this Chapter we propose an application agnostic, runtime-based approach to mitigate the effects of manufacturing variability on NUMA nodes. Our approach tries different configurations of power distribution and number of active cores on each socket on a NUMA node, while monitoring the performance of the application periodically, for small fraction of the overall execution. This way, the different configurations are rated for their performance and the best one chosen for the rest of the execution.

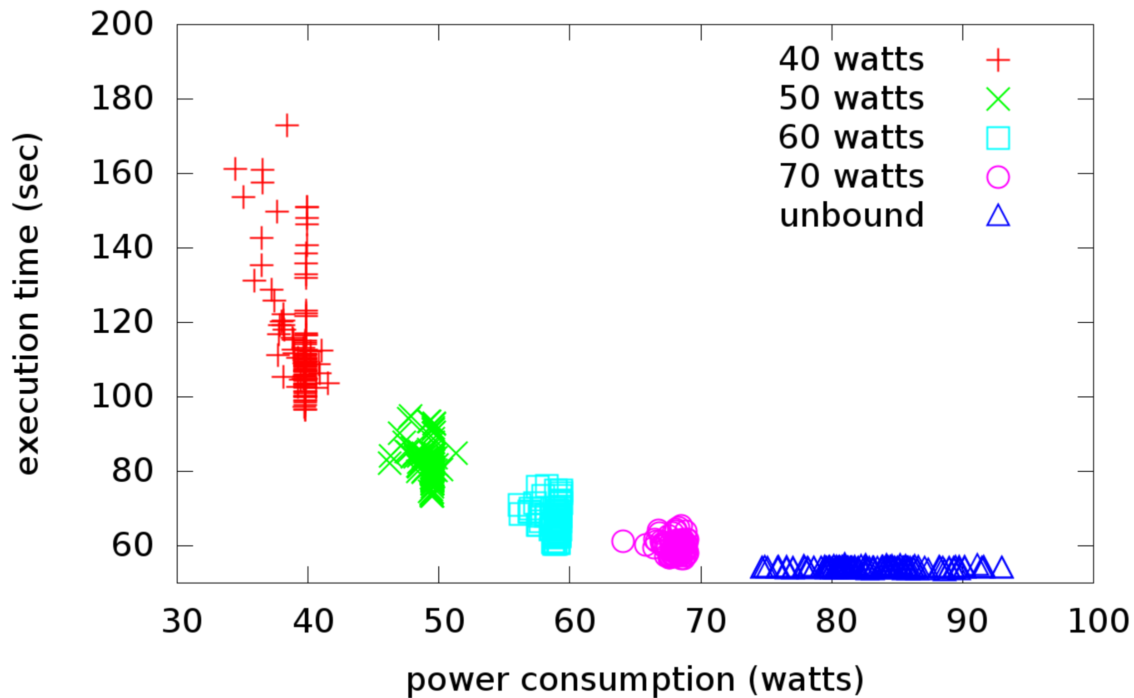


Fig. 5.1 Performance obtained when the `freqmine` application is run on 64 different 12-core Intel Xeon E5-2695v2 sockets under different power budgets.

5.1 A New Form of Heterogeneity

HPC systems are becoming increasingly power hungry, as we keep pushing the boundaries of performance on the road to exascale and beyond. While significant advances have been made in increasing the power efficiency of each single hardware component, i. e., flop/watt ratios continue to decrease driven by significant architecture advances, these savings are not enough to compensate for the growth in terms of computational elements required to realize the needed performance advances. Consequently, we need to build systems that use power more efficiently and ensure that any power provisioned for a system is also used and turned into realized performance, i.e., we need systems that can dynamically manage their power budgets among the available hardware components to direct power where it is needed.

Current machines are “worst-case provisioned”, i.e., all components of a system can be powered at the same time without reaching the system’s power limit. Since applications rarely keep all components occupied[1], this conservative approach leads to “wasted power”, i.e., provisioned power that is not used. Prior studies show that this wasted power can be up to 30% of a system’s power rating [102, 113]. One solution is to reduce the provisioned power to the expected average power consumption, or even lower, allowing systems to exploit all available power, and, consequentially,

[1] It’s a well known fact that many applications only run at a fraction of peak performance — often way below 10%

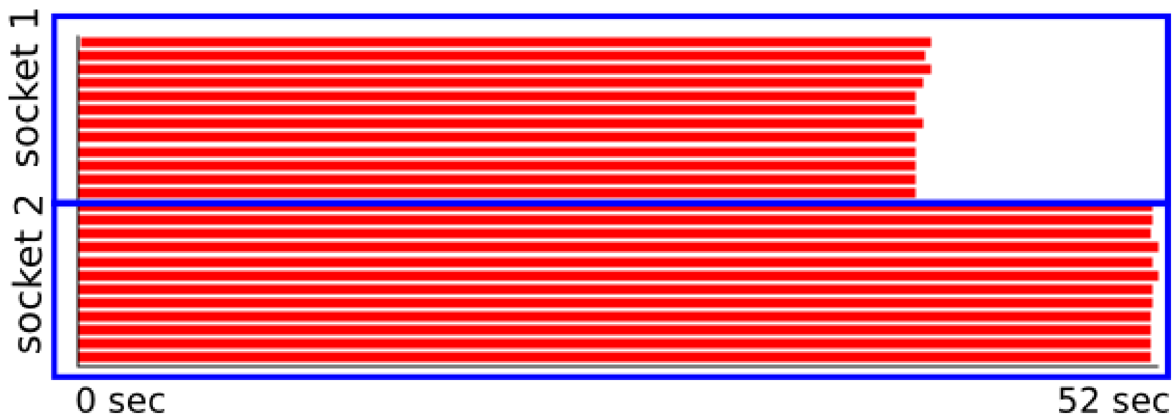
allowing for larger systems at the same total provisioned power. In such systems, which we refer to as “overprovisioned systems”, though, we must cap power to avoid power spikes caused by intermittent phases to exceed the provisioned power and with that endanger the operation of the entire system.

Many current architectures either already provide such power capping mechanisms or have them on their near term road map. However, such capping doesn’t come for free: it impacts performance, as show by the results of some initial experiments in Figure 5.1. The graph shows timing and power consumption of multiple runs of the `freqmine` code from the PARSEC benchmark suite [15] on 64 nodes of the Lawrence Livermore National Laboratory (LLNL) Catalyst cluster [97], using 12 threads per execution. `Freqmine` has been adapted to use OpenMP-like task-based parallelism [40] and runs in the top of the Nanos++ (v0.7a) parallel runtime system [23]. Since each node is composed of two 12-core Intel Xeon E5-2695v2 sockets, our experiments involve 128 different 12-core sockets and each run is limited to one of these 128 sockets. We consider five different power bounds: 40, 50, 60, 70W and unlimited per socket. The TDP for each socket was 115W. The x-axis shows the measured power consumption for each execution of the `Freqmine` benchmark while in the y-axis we show the corresponding execution time. We can see that running without a power bound results in almost no performance variation, but exhibits a wide spread of power consumption. Under a power bound, this power variation is no longer possible and we can see a drastic impact on performance variation instead. Further, we can see that lower bounds result in higher variations.

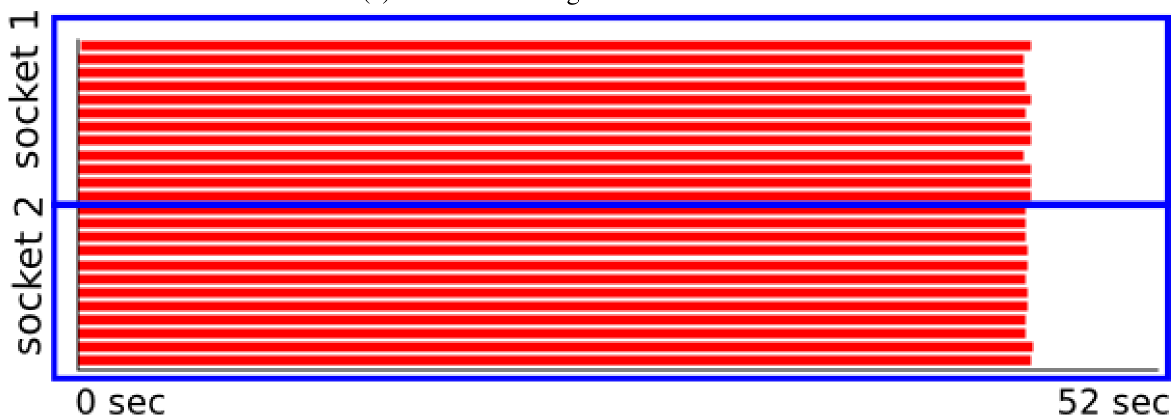
This behavior can be explained by manufacturing variability, which causes different processors to exhibit different efficiencies. The consequence of this phenomena is, though, that nominally homogeneous NUMA turn into heterogeneous systems when operated under a power cap equally applied to each socket. Since such irregular responses are due to uncontrollable manufacturing issues, there is no way to now how a particular software component while behave if run on several particular nodes unless it has been run there before.

5.2 Mitigating Manufacturing Variability

In order to mitigate this performance inhomogeneity caused by manufacturing variability, current static load balancing and scheduling mechanisms are insufficient, since they are only based on workloads and do not take into account dynamic variability. Classical dynamic work stealing and load balancing techniques may mitigate [19, 20, 119, 153] this problem under certain circumstances, but they are not enough when dealing with complex codes with frequent synchronization points. In the following, we illustrate the limitations of dynamic load balancing techniques on power limited scenarios by means of two examples: one considering a code with no barriers (Section 5.2.1) and another one with many (Section 5.2.2). In both examples, the considered applications belong to the PARSEC benchmark suite, but have been adapted to use OpenMP task-based parallelism [40] and executed on top of the Nanos++ (v0.7a) runtime system [23].



(a) Static scheduling and 12 cores enabled.



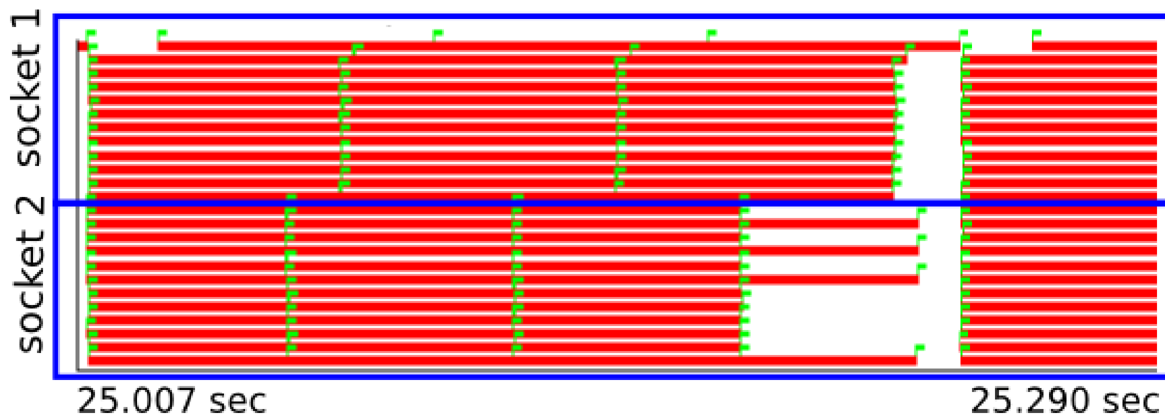
(b) Dynamic scheduling and 12 cores enabled.

Fig. 5.2 Executions of `swaptions` under 40 W power capping.

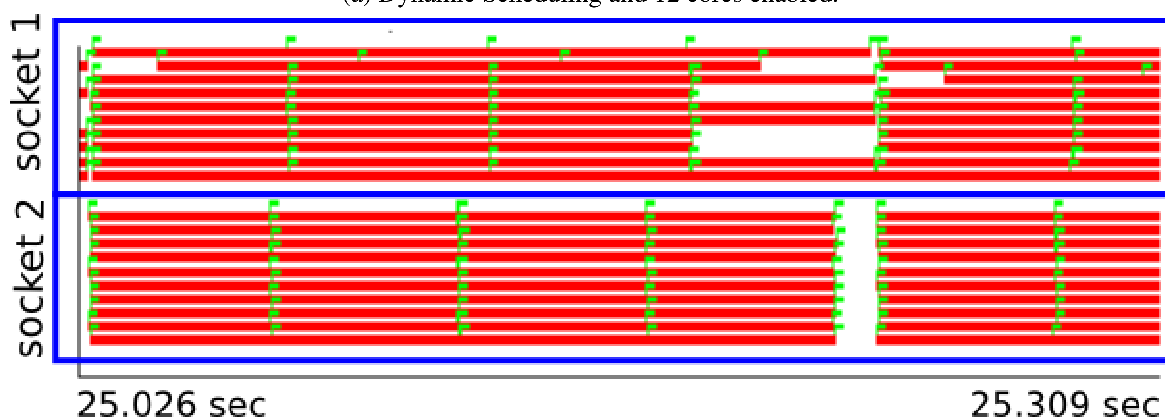
5.2.1 Example with no Barrier or Synchronizations

Figure 5.2 compares two executions of the `swaptions` benchmark [40] on a NUMA node composed of two 12-core Intel Xeon E5-2695v2 sockets, each run with 24 threads and with power capped at 40W. The x-axis of the figures show time and the y-axis the activity of each of the 24 threads involved in the parallel execution. When the activity of a particular thread i appears in red on time t, the thread is doing useful work; if it appears in white, the thread is idling. The scale of the x-axis is the same in both figures and covers 0s to 52.8s. In the run shown in Figure 5.2a the load is evenly distributed among all threads statically using a naive distribution. The reaction of the two sockets involved in the parallel run is different, which makes the threads running on the faster socket (Threads 1-8) finish much earlier than the threads mapped to the slow socket (Threads 9-15). As a result, threads from 1 to 8 are idle for 26% of the execution time.

In Figure 5.2b we show a second parallel execution of the same code, performed in the same NUMA node as above, but with dynamic scheduling. For this, we have over-decomposed the parallel execution into more tasks than cores and let the parallel runtime system assign tasks to cores once



(a) Dynamic Scheduling and 12 cores enabled.



(b) Dynamic Scheduling and 10 cores enabled.

Fig. 5.3 Executions of `blackscholes` near a synchronization point under 40 W.

they were idle. In this way, the cores on the fast socket executed some of the tasks that were assigned to cores on the slower socket in the static case, which allows the whole parallel execution to achieve a 1.13x speedup over static scheduling. This dynamic task assignment technique is equivalent to the numerous work stealing approaches described in the literature [19, 20, 119, 153]: it is able to deal with uneven hardware responses under restricted power budgets in the absence of barriers or synchronization points. Note however, that in order for conventional work stealing to be effective, finer grain parallelism is prefered. Introducing synchronization and coarser parallel work unit limit the effectiveness of this method.

5.2.2 Example with Barrier Operations

Figure 5.3 compares the behavior of two parallel executions of the `blackscholes` benchmark [40] on the same NUMA node as the one used above, again limited to 40W per socket. In this case we show the behavior of the parallel run around a barrier operation instead of the whole execution. The x-axis represents time and the y-axis shows the threads involved in the parallel execution. Green flags

mark the separation between the different pieces of sequential work in which the parallel execution is split or, in other words, the tasks.

Figure 5.3a shows the behavior of the dynamic task scheduling using the same policy as above, with idle time shown in white. While in the absence of barriers this technique properly balances the load between the two 12-core sockets, the results in this case clearly show that they fail in case of barriers: the green flags show that the same tasks exhibit differences in execution time depending on the socket they run on: around 74μs on average when run on the slow socket and 58 when run on the fast one, despite each task executing the same computational workload. Having tasks with smaller granularity could offer more flexibility to the runtime's scheduler to better distribute the tasks among the sockets and cores, but it is not always possible to decompose work into smaller units. Furthermore, this requires alterations to the source code of the application. In this Chapter we propose a different approach to solve this issue by redistributing power among sockets and finding the optimal concurrency level, suitable for the specific socket and application.

Figure 5.3b shows a second execution with the number of threads per socket reduced to 10, i.e., 2 cores per socket or 4 cores total are left unused during the execution. In this example power is evenly distributed with 40W per socket. The average execution time of the tasks mapped to the slow socket gets reduced to 54μs, while the average time of those mapped to the fast socket takes 48μs to run. This improved execution time is caused by the fact that the socket power budget is now distributed among 10 cores instead of 12. More importantly, the heterogeneous character of the socket's response to the imposed power limit seems to be reduced by leaving 2 cores idle. This better balance between the two sockets significantly reduces the impact of barriers and, therefore, their idle time. Clearly, this is a much more balanced execution than the one shown in Figure 5.3a. Overall, the parallel run considering 10 cores per socket and dynamic task assignment shows a 1.21x speedup with respect to the execution with 12 cores per socket combined with a dynamic assignment.

This last example clearly shows that, under restricted power budgets and uneven hardware reactions, operating with the maximum possible concurrency while dynamically balancing load is insufficient since barrier points can introduce significant idling effects. In this cases, it can be better to restrict concurrency levels in order to homogenize the hardware reaction to low power budgets. Alternatively, it can also be helpful to unevenly distribute the total power budget assigned to the multi-core sockets of a NUMA node in order to compensate for varying processor efficiency, as we demonstrate in Section 5.3.

5.3 Mitigating Heterogeneity

Following Sections 5.2.1 and 5.2.2, which illustrate the negative impact of heterogeneity introduced by power capping, we now provide a general evaluation of the benefits of heterogeneity mitigation. We consider a wide range of parallel applications coming from many areas and we test their performance considering a large range of power and concurrency configurations. For each application and power bound, we select the best configuration and compare its performance with the performance obtained

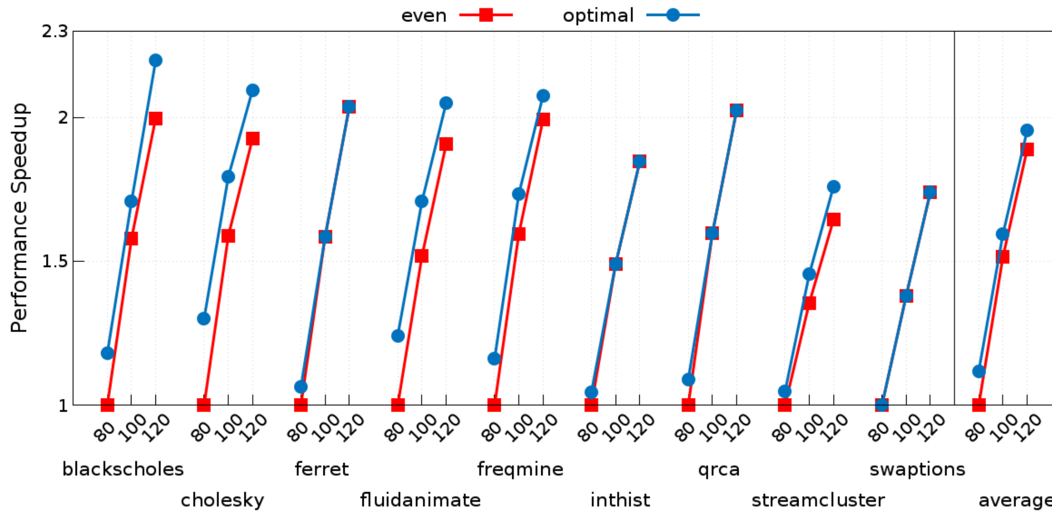


Fig. 5.4 Comparison between the even and the best configuration observed by application profiling. The speedup is computed over the execution of the even resource distribution for 80 W.

by deploying the naive even configuration — assign half the power to each thread and use all the available cores — combined with traditional task scheduler and balancer.

5.3.1 Experimental Setup

Applications: we utilize nine OpenMP codes: six of them come from the PARSEC benchmark suite [15, 40] (`black-scholes`, `ferret`, `fluidanimate`, `freqmine`, `streamcluster` and `swaptions`). Two of them are dense linear algebra routines (a cholesky matrix factorization, `cholesky`, and a QR communication-avoiding code, `qrca` [46]) and another one builds a histogram from a set of data points (`inthist`). All of these codes exploit task-based parallelism. The benefits of using a programming model which is coupled with a runtime system is that only the runtime needs to be modified to accomodate our power budget and active core balancing algorithm, as well as the online monitoring methodology. Individual applications remain untouched, and the runtime handles everything in a transparent manner.

Hardware and System Software: NUMA nodes of the Catalyst supercomputer [97] are composed of two 12-core Intel Xeon E5-2695v2 sockets each. The applications run in top of the Nanos++ (v0.7a) parallel runtime system [23]. We map one thread per active core. To set power constraints and measure power consumption on each socket, we use Intel's RAPL [80] . These registers are accessed by our modified version of the Nanos++ runtime using the libMSR library [132].

Configurations: We consider power bounds of 80W, 100W and 120W for total node power. This leaves 40W, 50W and 60W respectively per socket, which is between 35% and 52% of each socket's 115W TDP. Although limiting socket power to 50% or less may seem aggressive, studies have shown that typical HPC workloads only use 60%-85% of the available power on the socket

Table 5.1 Optimal configurations per application and power bound in terms or Watts and active cores per socket

	80 W	100 W	120 W
blackscholes	40-40 W, 10-10 cores	55-45 W, 10-12 cores	70-50 W, 12-10 cores
cholesky	30-50 W, 2-12 cores	35-65 W, 2-12 cores	30-90 W, 10-10 cores
ferret	40-40 W, 10-10 cores	50-50 W, 12-12 cores	60-60 W, 12-12 cores
fluidanimate	45-35 W, 10-6 cores	55-45 W, 10-6 cores	65-35 W, 10-6 cores
freqmine	45-35 W, 12-6 cores	55-45 W, 10-12 cores	65-55 W, 12-12 cores
inthist	40-40 W, 10-10 cores	45-55 W, 12-12 cores	60-60 W, 12-12 cores
qrca	45-35 W, 12-6 cores	50-50 W, 12-12 cores	60-60 W, 12-12 cores
streamcluster	35-45 W, 2-12 cores	60-40 W, 12-12 cores	65-55 W, 12-12 cores
swaptions	40-40 W, 12-12 cores	50-50 W, 12-12 cores	60-60 W, 12-12 cores

[114]. Moreover, actual applications, such as the PARSECSs benchmarks, exhibit different behavior at different execution stages. As a result, an application may reach its power peak for only a portion of its total execution. Power limits above 50% would have minimal impact on overall performance. Furthermore, there is a well established trend of an increase in manufacturing variability in more modern processor models [103] and is expected to keep rising. In order for our study to be relevant for future processors, we choose a configuration that creates enough variability between two sockets. If we allow a power limit of 80W, we consider 5 different ways of distributing the power among the two sockets of the NUMA node: 30W:50W, 35W:45W, 40W:40W, 45W:35W and 50W:30W as well as 36 ways of specifying the maximum concurrency allowed in each 2-socket NUMA node: 2-2, 4-2, 6-2, 8-2, 10-2, 12-2, 2-4, etc. up to 12-12. In total, this leads to a total of 180 combinations. Similarly, when allowing a power limit of 100W there are 8 ways of distributing it, which combined with the 36 possible ways of distributing the concurrency, leads us to a total of 324 combinations. Similarly, when the total power budget reaches 120W, the total number of combinations is 468. Overall, for each particular application we have 972 different combinations.

Other Considerations: The results of these experiments are machine dependent since each particular 12-core socket reacts in a different way when a power limit is set. Ideally, all 972 configurations per application should be executed on many NUMA nodes to really account for many possible hardware reactions when a power limit is set. However, due to the size of our experimental campaign, we randomly chose a single 2-socket NUMA node for each considered application and run all 972 combinations on it. Although this random choice can slightly influence the relative results between the benchmarks, the general conclusions we extract from them remain unchanged.

5.3.2 Evaluation

In Figure 5.4 we show our experimental results. On the x-axis we represent all the considered applications and the three power bounds we consider: 80W, 100W and 120W. In the y-axis we represent, for each particular application, the speedup achieved over evenly distributing 80W among two sockets (40W per socket) and keeping 12 active cores per socket. On average, the optimal configurations outperforms the totally even distribution (50% of the power and 12 cores per socket) by 11.8% (80W), 7.3% (100W) and 7.6% (120W).

Not surprisingly, the more restrictive the power capping is, the more beneficial the optimal configuration becomes in terms of performance. The uneven hardware reaction gets exacerbated by restrictive power bounds, which gives more room for improvement when the hardware is rebalanced by changing the power and concurrency assignation per socket. Application-wise, the benefits are much larger for applications composed of several execution phases separated by barriers, like `fluidanimate` (24% improvement) or `cholesky` (30%). On the other side, `swaptions` does not get any benefit from our power rebalancing techniques since its lack of barriers enables simple load balancing schemes to mitigate the hardware heterogeneous response, as described in Section 5.2.1.

In Table 5.1 we list the optimal configuration for each application and power bound. As expected, for applications without barriers (`swaptions` and `inthist`) the most balanced configurations (40W:40W and 12-12 active cores, 40W:40W and 10-10 cores respectively) are optimal. Since for these applications the parallel runtime system successfully manages the load, there is no need for system balancing by means of power or concurrency reassignment among the involved threads. On the other side, applications like `cholesky` or `fluidanimate` do really benefit from leaving significant parts of the cores idle and rebalancing the power accordingly. Clearly, the hardware heterogeneity induced by setting a power bound is not compensated by load balancing schemes delivered at the parallel runtime system side and some concurrency and power rebalancing must be done to maximize the performance of these applications.

This evaluation demonstrates that classical work stealing and load balancing techniques are not able to compensate the heterogeneity induced by power capping, except for trivial situations where a parallel code has no global barriers or synchronization points and the size of the parallel work unit is small enough to allows versatile alternative scheduling scenarios. Since the potential benefits of power and concurrency rebalancing is up to 30%, there is a need for developing techniques able to figure out the optimal configuration within a single execution run.

5.4 Runtime Approach

While the previous experiments demonstrate the potential benefits of unevenly setting up the power caps and the number of active cores per socket in a power-constrained NUMA node, these benefits are obtained under the huge cost of running each application multiple times in the targeted NUMA node, each time with a particular power and active cores limit. Further, the results obtained using extensive

search on a particular NUMA node are not applicable to another one since the hardware response to low power bounds are driven by manufacturing variability and cannot be known in advance. Also, deriving a particular performance ratio per power bound among the sockets contained in a NUMA node is not enough as different applications react in a different way to such variability. It is thus necessary to develop techniques able to quickly determine the optimal power-# of cores distribution for a particular software component and NUMA node.

We implement our method at the runtime level, since such systems offer load-balancing and can be easily extended with additional functionality. They are also widely used and expected to play a significant role in future parallel architectures [32, 147].

5.4.1 Exploiting Application Structure

Parallel codes often decompose loops or segments of serial code into multiple work units that run in parallel. While (at least to date) many codes follow the Single Program Multiple Data (SPMD) approach where multiple cores execute the same code several times, even more complex patterns, different repetitions of loops that iterate over similar sets of data several times produce similar execution patterns. As a consequence, codes almost always exhibit a certain degree of repetitive behavior that can be observed either over time or by considering the logical execution structure, which is composed of event sequences [31, 81, 144]. This iterative nature of parallel applications allows us to effectively guide the whole application behavior by observing only small but significant portions of the parallel execution. Our approach considers execution segments and associates each with a particular power and number of active core assignation per socket. To identify such a segment, the runtime tracks and identifies task instances of the same task type. The intuition behind our approach is that same type tasks should behave in similar fashion. However, if tasks of the same type behave in an non-deterministic manner, then our approach will fail to find a better configuration. For example, in the case of *blackscholes* we only have one task type, thus this case is trivial. However, in other applications, such as *ferret*, there are a few different task types. For *ferret* these are *t_seg*, *t_extract*, *t_vec*, *t_rank* and *t_out*. The runtime can identify the type of an instance by the call site of the taskified functions or user provided labels. Then, for a given monitoring window, the runtime will identify all the task types running, but compare the execution time of the tasks with the same type. There are cases however that this may not be possible. For example if no tasks of the same type are found running on both sockets or if the tasks of the same type run different workloads (their execution time varies beyond a threshold, thus they are not comparable), then the current monitoring window is discarded and move to the next configuration. This may cause the runtime to miss a good configuration, or even the optimal one.

Identifying representative segments of an application has been extensively studied, especially in the context of hardware simulation, where running the entire application is too slow. Techniques such as the one presented by Sherwood et. al [131] and the SimPoint 3.0 framework [71] could be employed for a more robust analysis and deal with the aforementioned issues. However, our

approach exploits information already available to the runtime that can be accessed fast, minimizing the analysis' overhead.

5.4.2 Search Algorithm

The search algorithm aims to find the optimal power and total number of active cores balance among the different sockets of a NUMA node. It starts with evenly distributing power and activating all cores and then progressively iterates over a set of power/#cores configurations and selects the best one. Per each configuration, the targeted application runs for a certain amount of time. The particular amount of time each configuration runs for is a parameter we call *monitoring window*. The smaller this parameter, the shorter the exploration, but the more chances of getting a non-optimal configuration since the amount of time it has been trained for may not be representative of the whole execution. On the other hand, large window sizes significantly increase the chance of finding the right configuration, but make the algorithmic search phase larger.

To characterize the performance achieved by each configuration we use a *throughput metric* defined as the number of tasks executed during the monitoring window each configuration runs for. This metric is well defined for all applications we consider in this work (see Section 5.3.1) and is particularly well-suited since it also implicitly captures the amount of idle time spent by the active cores. Further, it does not imply a significant amount of measurement overhead if the task granularity is kept over the tens of μs threshold. Although this metric is specific for task-based codes, any other light-weight metric able to capture the amount of time spent doing useful work would provide similar results for other kinds of applications or programming models.

During the first monitoring window, the runtime system measures the throughput of evenly distributing power among the sockets and using all the available cores. This is considered the best candidate until a configuration providing larger throughput is observed. After each iteration we then compare the throughput for the current profile with the best one. If the current one is better, it becomes the new best and is used for the subsequent comparisons. This analysis continues until the search space is exhausted, which may require more than one application run. At the end of each run, if we have not yet exhausted the search space, the runtime saves a checkpoint of the analysis state and resumes it in a succeeding run.

Special care must be taken to make sure that we are considering monitoring windows that constitute representative execution segments. If two windows capture different task types comparing them is not fair since different tasks have different execution times. To address this issue, we keep a set of task types for each different window. If the task sets collected during the best and current windows are not equal, it could mean that the two configurations were run at a different stages of the application's execution. We call these incompatible profile results *mismatching windows*. When this occurs, we ignore the current configuration without comparing it to the best and continue by checking another configuration over the next monitoring window. Special care need to be taken for the first monitoring window. If the first and second windows mismatch, we discard both and retrain

Fig. 5.5 Comparison of best configuration found by exhaustive and scoped online analyses for different *monitoring window* size, when running under a 80W power constraint. The size of the monitoring window can influence the precision of the analysis.

the first configuration. This will continue until we capture a representative segment of the application, meaning that two consecutive windows will be matching.

Note that different alternatives are available when dealing with mismatching windows. However, for this work we employ the simplest case, which is to discard it.

The search algorithm looks for the best configuration after trying several ones and measuring their throughput over their corresponding monitoring windows. As explained, the monitoring windows size is an input parameter of our search algorithm. The algorithm's sensitivity to the windows size and its optimal value are explored in detail in Section 5.5.1. Also, the set of configurations the search algorithm iterates over is a key choice. Large sets increase the chances of getting the optimal power/#active cores balance per socket, but also increases the cost of running the search. Alternatively, reduced sets may produce cheaper searches but also be unable to find configurations that significantly improve performance.

5.4.3 Training Sets

We have implemented four variations of our analysis, based on the size of the configurations sets:

Exhaustive Search: We use the different configurations defined in Section 5.3.1. As discussed above, in case we target a 80W power bound, the exhaustive search considers 180 different configurations. This is a conservative, but expensive analysis.

Naive Scoped Search: The scoped search does not consider extremely unbalanced configurations since they rarely produce the most optimal results. As a general rule we focus the search on a small area around the default balanced configuration. The reasoning here is that just slightly providing more power or reducing the concurrency in the slower socket will mitigate the imbalance between the sockets. The scoped search considers 80 different configurations for the 80W bound. They are composed of five different power configurations (30W:50W, 35W:45W, 40W:40W, 45W:35W and 50W:30W) deployed for each one of the 16 active core distributions: 6-6, 6-8, 6-10, 6-12, 8-6, ... , 12-12.

Scoped Search 1: This training set aims to further reduce the search space, but considers both balanced and unbalanced configurations. It avoids irrational distributions like assigning more than half of the power but less than half of the active cores to one of the sockets. This training set considers the even power configuration (40W:40W) and 9 different active cores distributions for it: 8-8, 8-10, 8-12, 8-10, 10-10, 12-10, 8-12, 10-12 and 12-12. It also takes into account assigning 35W to the first socket and 45W to the second one with active core counts 6-10, 6-12, 8-10 and 8-12 and its counterpart, that is, 45W to the first socket and 35W to the second with active cores counts of 10-6, 12-6, 10-8 and 12-8. Finally, this training set considers two unbalanced configurations: 30W:50W assigned to the sockets and 2-12 active core counts per socket, and 50W:30W with 12-2 active cores. This leaves us with 19 configurations when operating under the 80W power bound.

Scoped Search 2: This training set contains the same distributions as Scoped Search 1 except the two unbalanced configurations 50W:30W and 30W:50W, reducing the set to 17 configurations. Very unbalanced configurations can produce large performance improvements, but also increase the search costs since they significantly slowdown the execution in certain cases. This training set avoids the dangers of such unbalanced configurations by not considering them.

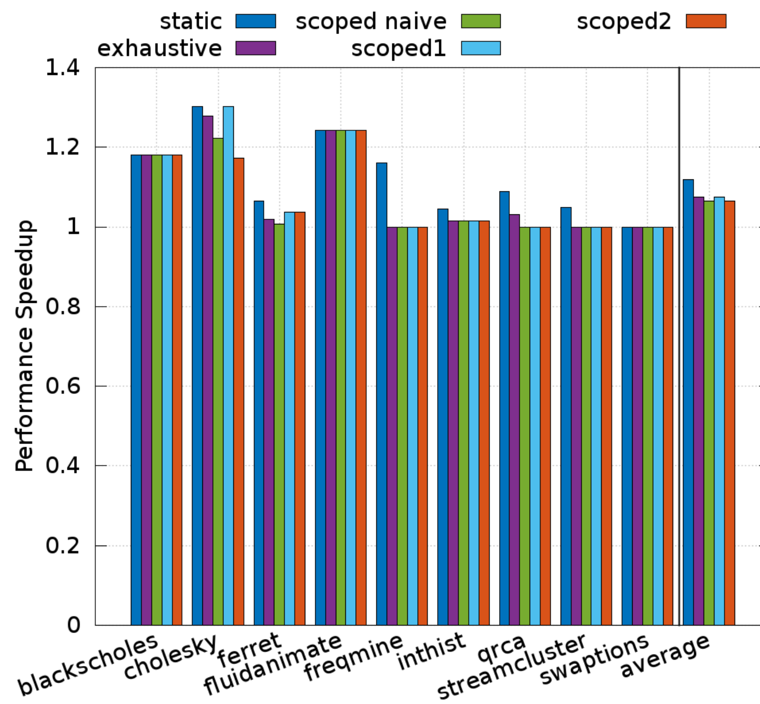
The reasoning here is that extreme configurations that greatly favor one socket or severely reduce available parallelism are not likely to benefit an application. Our goal is to reduce the effect of the frequency imbalance between the sockets on a node, extreme configurations would only benefit applications make sub-optimal use of the available parallelism (e.g. dedup).

5.5 Evaluation

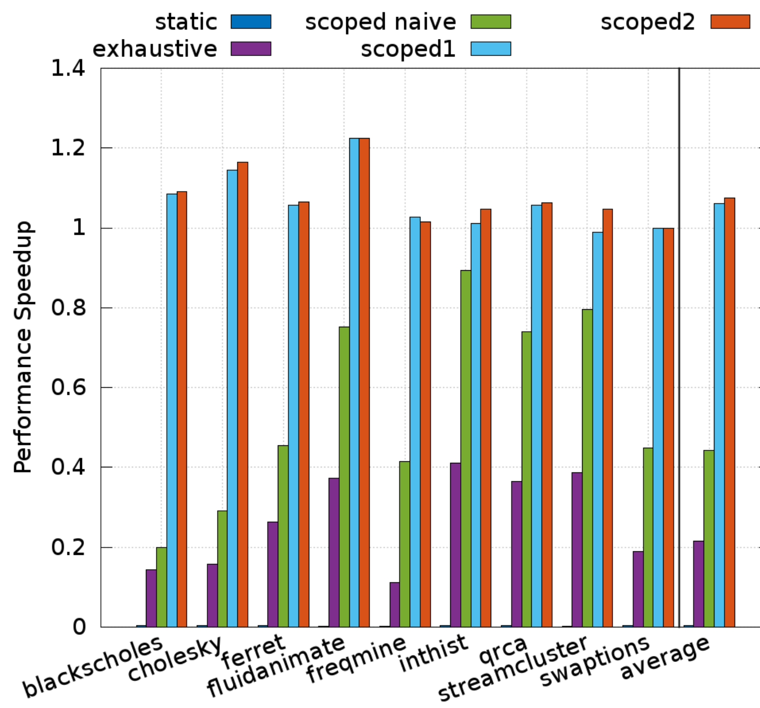
This section shows the results in applying our optimization technique. The experimental setup in terms of applications, system software and hardware is the same as in section 5.3.1. For our evaluation we focus on an 80W power budget (see *Configurations* in Section 5.3.1).

5.5.1 Monitoring Window Sensitivity

This first section shows how an optimal window size is obtained by obtaining a detailed sensitivity study. This optimal size is leveraged in the following general evaluation of the search algorithm in terms of its costs and benefits depending on the training set.



(a) Performance benefits of the selected configurations without accounting for the cost of the search.



(b) Performance benefits of the selected configurations taking into account the cost of the search.

Fig. 5.6 Performance benefits using monitoring windows of 1 second under 80 W power limit.

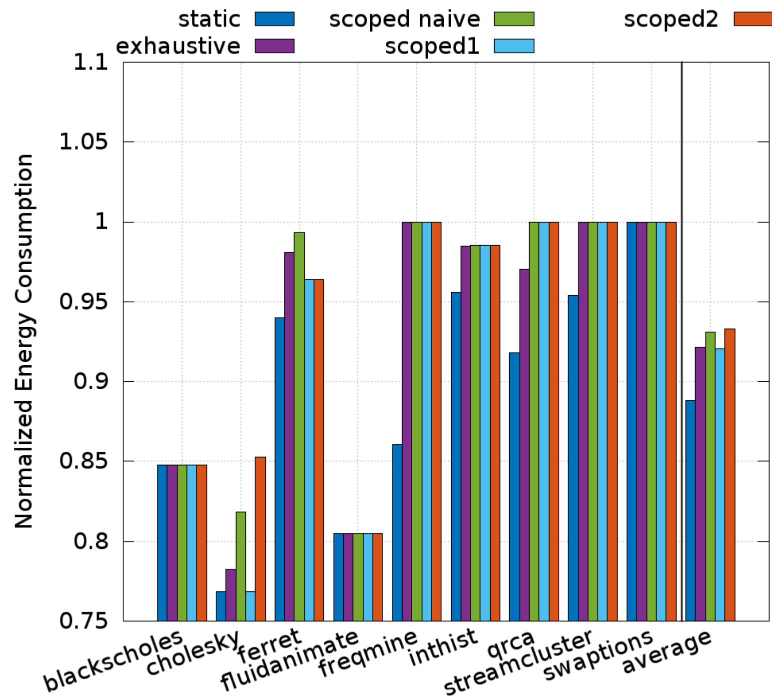
Figure 5.5 shows how window sizes of 0.5, 1, 2 and 5 seconds influence the effectiveness of the algorithmic search under an 80W power constraint. The x-axis represents the considered windows sizes for each application, while the y-axis shows the speedups obtained over the trivially balanced configuration (40W and 12 active cores per socket), represented in the figure with red horizontal lines. The blue horizontal line represents the speedups achieved by the optimal configuration found using the multi-execution analysis presented in Section 5.3 . Purple and green lines show results considering the exhaustive and scoped naive search spaces described in section 5.4.3, while the blue and the orange lines represent the scoped1 and scoped2 search spaces. These results do not consider the cost of the search algorithm, just the benefit of the optimal configurations found when using different windows sizes and training sets.

Results shown in Figure 5.5 clearly show that a windows size of 0.5 seconds on average does not provide any gain when using the scoped naive training set and only marginal gains when using the exhaustive search. Indeed, the exhaustive and scope naive searches bring significant performance degradations in cases like `ferret` and `qrca` and fail in providing a configuration that delivers the potential performance gains in case of `fluidanimate`. When considering the scoped1 and scoped2 training sets, 0.5 seconds window sizes do not provide significant benefits in case of `ferret` and `qrca`. On average, 0.5 seconds large windows provide average speedups of 1.02x, 0.98x, 1.07x and 1.06x when exhaustive, naive scope, scope1 and scope2 training sets are used.

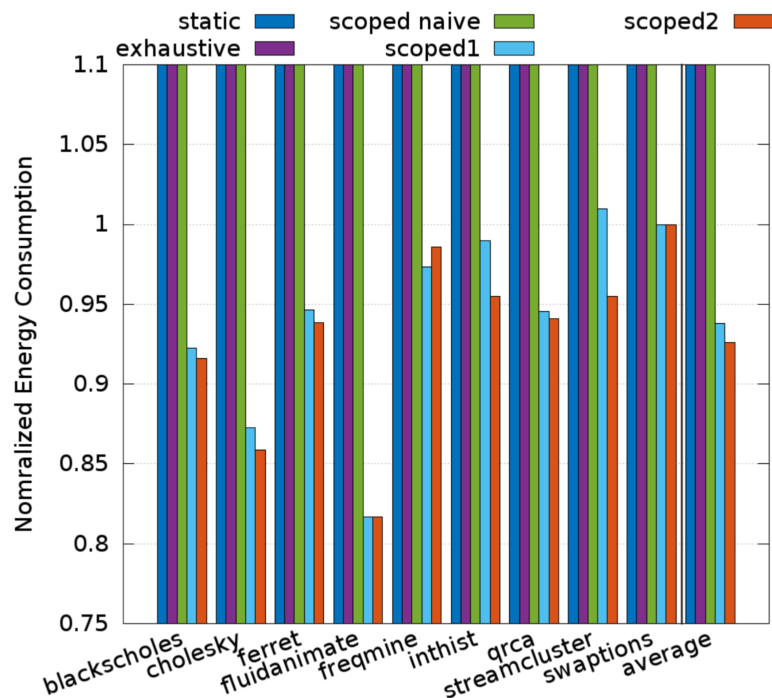
Increasing the windows size from 0.5 to 1 second improves the quality of the configurations selected.

Indeed, it provides speedups of 1.27x, 1.22x, 1.30x and 1.17x for `cholesky` or 1.24x, 1.24x, 1.24x and 1.24x for `fluidanimate` when exhaustive, trivial scoped, scoped1 and scoped1 searches are used, respectively. On average, the 1 second window size provides benefits of 1.08x when exhaustive search is used and 1.07x when the trivial scope set is considered, 1.08x when the scope1 is considered and 1.07x for the scope2. As a reference, when running a whole execution per each configuration we get optimum power and active core distribution that bring average speedups of 1.12x. Increasing the windows sizes to 2 and 5 seconds does not significantly improve the results quality although they asymptotically get closer to the ones obtained using the multi-execution analysis. In conclusion, the 1 second window size is the optimal one since it provides similar benefits for all the considered training sets as the 2 and 5 second window sizes under a lower cost.

The search algorithm works very well for applications with regular computations separated by barriers (`blackscholes`, `fluidanimate` or `cholesky`) since each monitoring window can capture different iterations of the same behavior. In case of `ferret` or `qrca` the scarcity of barriers or synchronization points reduces the potential gains of our techniques. When computations are more irregular, it is more challenging to have consistent monitoring windows, which reduces the effectiveness of our scheme. In the particular case of `freqmine` the task type that accounts for more than 90% of the execution is input dependent and actually a single instance of this task type can take up to half of the total execution time. As a result the vast majority of the considered configurations are dismissed since their corresponding monitoring windows either mismatch or fail to capture any information.



(a) Energy reductions of the selected configurations without accounting for the cost of the search.



(b) Energy reductions of the selected configurations taking into account the cost of the search.

Fig. 5.7 Energy consumption reduction using 1 second monitoring windows under a 80 W power bound.

5.5.2 Performance Improvements of the Selected Configurations

In Figure 5.6a we show in detail the performance benefits provided by the optimal configurations found by each one of the four training sets considering a 80W power bound and 1 second long monitoring windows. The results are expressed in terms of speedup with respect to the execution time when using the naive even distribution (40W:40W and 12-12 active cores). The static technique consists of entirely running the applications for each one of the 180 configurations defined in Section 5.3.1. The results of the static technique have already been presented in Section 5.3.2. This technique, while prohibitively expensive in practice as it requires 180 runs per application, always finds the best possible configuration and hence provides an upper bound of the speedup possible. We represent its results in Figure 5.6a.

In case of `blackscholes` and `fluidanimate`, all the training sets (exhaustive, naive scoped, scoped 1 and scoped 2) find configurations that provide the same speedup as the static technique, 1.17x and 1.24x respectively. In case of `cholesky`, the exhaustive and scoped 1 training sets allow the system to find configurations that provide speedups very close to 1.3x, the best possible one. The naive scope and scoped 2 techniques provide speedups close to 1.2x. Although these benefits are significant, they are far from the ones achieved by the other techniques. The reason is that the `cholesky` application benefits a lot from unbalanced distributions (Table 5.1), which are neither considered by the naive scope nor by the scoped 2 training sets. In case of `freqmine`, although the optimal identified configuration by the static analysis does provide significant benefits, the 4 training sets considered by the searching algorithm fail in finding this optimal configuration, since tasks are input dependent and can take up to half of the execution time, as a result most windows fail to capture task throughput since not tasks finish execution. In case of `ferret`, `inthist`, `qrca` and `swaptions`, the potential benefits of power and active cores balancing are very limited, since these applications do not have a significant number of barrier synchronizations. As we have explained in Section 5.2.1, when the overall number of barriers is not significant, classical load balancing mechanisms are enough to maximize performance under low power scenarios.

On average, all training sets provide benefits of around 1.07x, while the static technique provides an average speedup of 1.11x. The training costs of the five approaches are not considered in Figure 5.6a.

5.5.3 Performance Improvements Taking into Account Analysis Costs

All considered techniques require an analysis to find a power/active cores balance that optimally improves the trivially balanced distribution. This analysis starts once the execution of the parallel code begins and finishes when all the considered configurations have been tested. If a single application run is not sufficent to test all the configurations of the training set, the application is run again and again until the training is complete.

Figure 5.6b shows the speedups achieved by all considered techniques including the training phase costs. In case of the static analysis it is required to run the application multiple times, one per each of the 180 different power/active cores distributions. Consequently, the overall speedup is

0.003x, much smaller than 1x. The exhaustive training set considered 180 distributions and checks their performance over monitoring windows that are 1 second long. Therefore, more than one run is required to test all the configurations for those applications with execution times smaller than 180 seconds. Since this is the case of all the considered parallel codes, the average speedup achieved by the exhaustive training set is 0.21x. Similarly, the trivial scoped training set obtains a speedup of 0.44x. These three techniques do not improve the trivial approach which consists in just evenly distributing the total available 80W power budget and using all the cores available in the 2-socket NUMA node.

The scoped 1 and 2 training sets consider much fewer configurations than all previously mention approaches and, therefore, their training costs are significantly smaller. Indeed, they are able to test all configurations for 1 second, select the best one and then run the rest of the application using this optimal configuration. Of course, the cost of the training phase can reduce the overall benefits of the optimal configuration, as it is the case of `blackscholes`, where the benefits of the scoped 1 and 2 training sets are reduced from 1.17x to 1.08x and 1.09x respectively. `Cholesky` and `fluidanimate` have larger execution times than `blackscholes`, which allows them to compensate the cost of the training phase when scoped 1 and 2 training sets are considered and to keep almost the same performance gains as if the training costs were not considered (1.15x and 1.22x, respectively).

Finally, Figure 5.6b reports marginal performance benefits for some applications for which the search algorithm does not find any distribution significantly better than the trivial. For example, in case of `qrca` there are speedups of exactly 1x in Figure 5.6a, but of 1.05x and 1.06x for scoped1 and scoped2 in Figure5.6b. The explanation of this behavior is that, although the search algorithm fails to find any configuration that is significantly better than the evenly distributed, there are indeed many configurations that perform slightly faster than the even one, which accelerate the execution as they are used during the training phase.

Overall, the static technique and the exhaustive and trivial scope training sets produce an overall performance slowdown, which makes these approaches useless in practice. On the other hand, the scoped 1 and 2 training sets provide important performance benefits even when the search phase is taken into consideration, which makes these approaches very useful to maximize performance in power constrained scenarios.

5.5.4 Energy Consumption Reductions

Figure 5.7a shows the energy consumption reductions achieved when using configurations found by the five different techniques if training costs are not considered. The baseline is the energy spent by the trivially balanced configuration (12 active cores and a maximum of 40W per socket) and all results are normalized to this baseline. The configurations selected by the static analysis provide energy reductions of 11% with respect to the even distribution since the normalized energy gets reduced from 1 to 0.89. The reductions provided by the search algorithm are between 7% and 8%, depending on the training set.

Once the cost of the exploration phase is considered, the energy consumed by the static analysis, the exhaustive and the scope naive training sets are 295, 3.7 and 1.8 times larger than the energy consumed by a single run with the even configuration. In Figure 5.7b we just represent normalized energy consumptions below 1.1 for readability purposes. As observed before, the scoped 1 and 2 training sets are able to compensate the cost of the training set and deliver improvements over the even configurations. The energy consumption reductions are 0.94 and 0.93 for the scoped 1 and 2 training sets respectively.

5.6 Summary

In this Chapter we demonstrate how state-of-the-art parallel runtime systems can mitigate the performance imbalance between sockets on the same node when operating under strict power constraints. We establish that load-balancing, although improving performance, is not sufficient for a wide set of applications. In our study we use six applications from the PARSEC benchmark suite and three additional applications, all implemented with OpenMP 4.0 tasks. The OpenMP runtime offers us a perfect platform for developing our methodology, without the need to any additional modification for each application. By performing profiling runs of the applications with different power distributions and active number of cores on each socket we demonstrate that it is possible to achieve speedups up to 1.30x over naively spreading evenly the power budget and using all possible cores. We also propose and implement an online analysis that monitors only a segment of the application's execution and is able to switch between different power and concurrency configurations at runtime, reducing the overhead of profiling. Our evaluation shows that it is possible to carefully compose the configuration search space by eliminating candidates that are unlikely to give a good result (such as reducing power but increasing concurrency). The online analysis achieves speedups up to 1.22x over the naive case.

This work focused on figuring out the optimal power-concurrency balance on machines with 2 sockets per NUMA node, which is a common setup in current systems. Future configurations will have many more sockets on a single node. In respect to our method this trend will likely require increasing the training set sizes, thus the cost and its complexity. However, the benefits of our technique will also increase: more sockets imply an even more varied response to low power scenarios within the same NUMA node. Adding accelerators will further increase the number of frequency/power capping domains. By restricting our searches to well-balanced configurations, as shown in this work, we can avoid a combinatorial explosion in the training set sizes, which keeps training costs within reasonable margins and enables larger performance improvements on multi-socket NUMA nodes. The results on 2 a socket system, as described in the thesis, therefore cover the worst case scenario.

6

Power-Aware Job Scheduling

Power is becoming a major financial and environmental concern, restricting the compute capacity of High Performance Computing (HPC) systems. Today's most power efficient supercomputer operates at 14.1GFLOPS/W [70], but even if we had a system able to operate at the 50GFLOPS/W rate, which is the limit that some funding agencies have set up for building an exascale machine, the full system would consume several tens of MWatts of power, which constitutes a large economic burden. Consequently, a report from the US Department of Energy (DOE) [138] identifies energy efficiency as one of the top ten research challenges on the road to exascale. For similar reasons, the European Union has set up an HPC program low-power systems based on mobile technology [118].

An emerging design practice for HPC systems, known as *overprovisioning* [113], is to have more nodes than the maximum power budget could feed if run at peak capacity, in contrast to traditional approaches, which are focused in having enough power even when all nodes run at their peak. Overprovisioning is driven by the observation that most applications in practice never reach peak power and hence do not fully utilize the available power envelope. In such overprovisioned systems, we can lower the average power provisioned to each node, allowing us to power more nodes within the same power budget. This approach is made possible by recent developments in hardware design that enable power management and power capping from user space, as this is necessary to efficiently manage power as a limited and shared resource. As an alternative to restricting power to nodes, we can choose to only operate fewer but at full power. Their total power consumption should not exceed that of the total power budget. This second approach restricts the available parallelism in the system, but allows for faster execution and does not force us to deal with any complications related slower then expected execution of the system's workload. Which approach is preferable (or a combination of both) should depend on system and workload characteristics. For example, whether the workload would benefit from extra processing units or limitations in the completion time set by the user or administrator.

Workloads at the HPC system level are managed by job schedulers that allocate resources to dispatched jobs. Such jobs can run on distributed memory scenarios and, in this context, MPI [109] is the most common approach to handle distributed memory communications. It is usually coupled

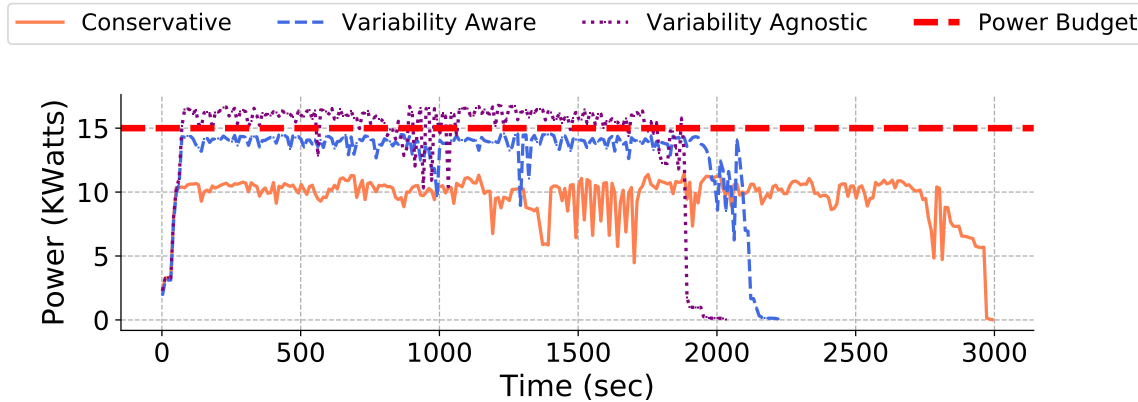


Fig. 6.1 Total power consumption trace for Conservative, Variability agnostic and Variability aware scheduling policies, when running the same workload. Considering socket variability maximizes performance and meets the power budget.

with a shared memory programming model, like OpenMP [110] or similar [7]. Either across nodes or within a shared-memory node, both job and runtime schedulers deal with the resource allocation problem, albeit at different levels, offering opportunities to manage power consumption. Indeed, examples of power-aware systems that offer solutions either at the job scheduling [30, 53, 58, 67] or at runtime system [38, 67, 79, 141, 143] levels already exist in the literature.

Manufacturing variability or process variation refers to the power and frequency heterogeneity observed across chips implementing the exact same architecture as a consequence of uncontrollable material differences in the manufacturing process [123]. In order to provide homogeneous performance, chips of the same architecture must hide frequency variability, which can only be achieved via variations in their power consumption. However, in a power constrained environment where all chips need to operate under a certain power cap, this frequency variability can no longer be hidden [123], leading to heterogeneous performance. As a result, a theoretically homogeneous system turns into a heterogeneous one with performance variations of up to 64% [79]. While ignoring this manufacturing variability leads to performance and energy inefficiencies, there are opportunities for achieving improvements at the power budgeting or parallel runtime system levels when variability is properly managed [38, 67, 79, 141, 143].

This thesis goes beyond the state-of-the-art by proposing job scheduling policies driven by variability-aware power prediction models. We consider two different approaches to predicting manufacturing variability. The first model assumes that all applications are affected the same way by manufacturing variability. This assumption is not correct, as demonstrated in Section 6.3.2. The second model offers a more robust approach, where the model uses a training set of applications to identify how different application behavior is impacted by manufacturing variability. We extend power-aware scheduling and power prediction models to deal with manufacturing variabity, producing two novel variability- and power-aware job scheduling policies. The goal of these policies is to maximize the utilization of the cluster without exceeding the available power budget and without restricting per

socket power consumption. Instead, by using the power prediction, the policies can find the maximum number of concurrent jobs that can run within the power budget. This offers an alternative to per node power capping, which in combination with manufacturing variability, would create an heterogeneous cluster. Current workload managers do not account for this type of heterogeneity. We consider the power consumption of the CPU, since it accounts for more than 50% [137] of the total node's power consumption. Our Policies rely on two different models and leverage their power requirement predictions of individual parallel jobs to make scheduling decisions that maximize performance while reducing energy consumption. Many different variability-agnostic power and energy prediction models have been proposed [12, 13, 16, 68, 82] and are often employed to manage power distribution on clusters to mitigate the effects of manufacturing variability [8, 38, 53, 67, 79, 141, 143].

As a motivation we show Figure 6.1, where three different scheduling policies are compared. The Conservative simply considers that all jobs consume the same power on all sockets. The Variability agnostic predicts accurately the power consumption of individual jobs, but does not consider socket variability. On the contrary, the Variability aware policy does also consider socket variability, making a different prediction per socket. As displayed in Figure 6.1, the Conservative policy is the one providing the worse performance. The Variability agnostic improves performance by making more accurate predictions but fails to account for the more power consuming sockets, exceeding the 15KWatts power budget. Finally, the Variability aware policy manages to improve performance, while respecting the power budget. Figure 6.1 illustrates how accounting for manufacturing variability while scheduling parallel jobs provides performance benefits. Section 6.3.1 describes the experimental setup we consider to generate Figure 6.1.

This thesis shows how variability-aware power prediction models can be effectively used to guide job scheduling policies and bring significant benefits with respect to the variability-agnostic ones. In particular, this thesis makes the following contributions beyond the state-of-the-art:

- Two new variability-aware power prediction models. Both models use Performance Monitoring Counters (PMC) to predict an application's power consumption on a specific socket. PMCs are used to measure the activity of individual architectural components while the targeted application is running and a linear model is then used to find their contribution to power consumption. The first model assumes power variability to impact all applications equally. It uses a single benchmark to measure the power consumption variability across sockets and apply it to the variability agnostic PMC-based model. The second model extends the PMC-based approach to take power consumption variability into account, as part of the model. It trains the model for each individual socket, using a reduced set of benchmarks.

- Two power- and variability-aware job scheduling policies that optimize job turnaround time and energy efficiency while respecting a system-wide power budget. Unlike previous work that does not consider variability during job scheduling decisions [53, 67, 79, 141], our policies use variability-aware predicton model to guide scheduling.

Table 6.1 Architectural component activity ratios formulas for Intel Broadwell Architecture, inferred from Intel's 64 and IA-32 Architectures Software Developer's Manual [1]

Power Component	Component Activity Formula
Fetch	UOPS_RETIRED.ALL / CPU_CLK_UNHALTED.THREAD_P
Branch Prediction Unit	BR_INSTR_RETIRED.ALL_BRANCHES / CPU_CLK_UNHALTED.THREAD_P
Arithmetic & Logic Unit	(UOPS_DISPATCHED_PORT.PORT_0 + UOPS_DISPATCHED_PORT.PORT_1 + UOPS_DISPATCHED_PORT.PORT_5) / CPU_CLK_UNHALTED.THREAD_P
Floating Point	FP_COMP_OPS_EXE.X87 / CPU_CLK_UNHALTED.THREAD_P
L1 cache	L1D_ALL.REF / CPU_CLK_UNHALTED.THREAD_P
L2 cache	(L2_RQSTS.ALL_RFO + L2_RQSTS.ALL_DEMAND_DATA_RD) / CPU_CLK_UNHALTED.THREAD_P
L3 cache	LLC.References / CPU_CLK_UNHALTED.THREAD_P
Memory	LLC.Misses / CPU_CLK_UNHALTED.THREAD_P

- A complete evaluation of the two variability-aware policies via a discrete event simulator. We implement additional scheduling policies for our evaluation, which represent traditional and state-of-the-art practices used in today's HPC systems. Our evaluation demonstrates how variability-aware policies achieve energy savings up to 8% and job turnaround time reductions up to 30%, considering different power budgets and two workload traffic scenarios (bursty and heavy).

The remainder of this Chapter is organized as follows. Section 6.1 presents the two variability-aware power prediction models. Section 6.2 introduces the variability and power-aware scheduling policies we propose. A validation of the models and evaluation of our job scheduling policies are presented in Section 6.3.3 and, finally, Section 6.4 provides summarizes the ideas and results presented in the Chapter.

6.1 Power Variability Prediction Models

Power modeling has received a lot of attention from researchers and developers as it provides a quick and robust way to understand the power behavior of a system. A common approach for predicting power consumption consists in the usage of Performance Monitoring Counters (PMC) [11, 13, 16, 17, 94, 135], since sampling PMC does not introduce significant power interference [82, 87] and PMC-based prediction models decompose a chip into several components in terms of power consumption [13]. While power prediction models are employed to find the best tradeoff between

power and performance [67], we are not aware of any previous work that uses variability-aware power models to guide job scheduling decisions.

6.1.1 Power Ratio Model

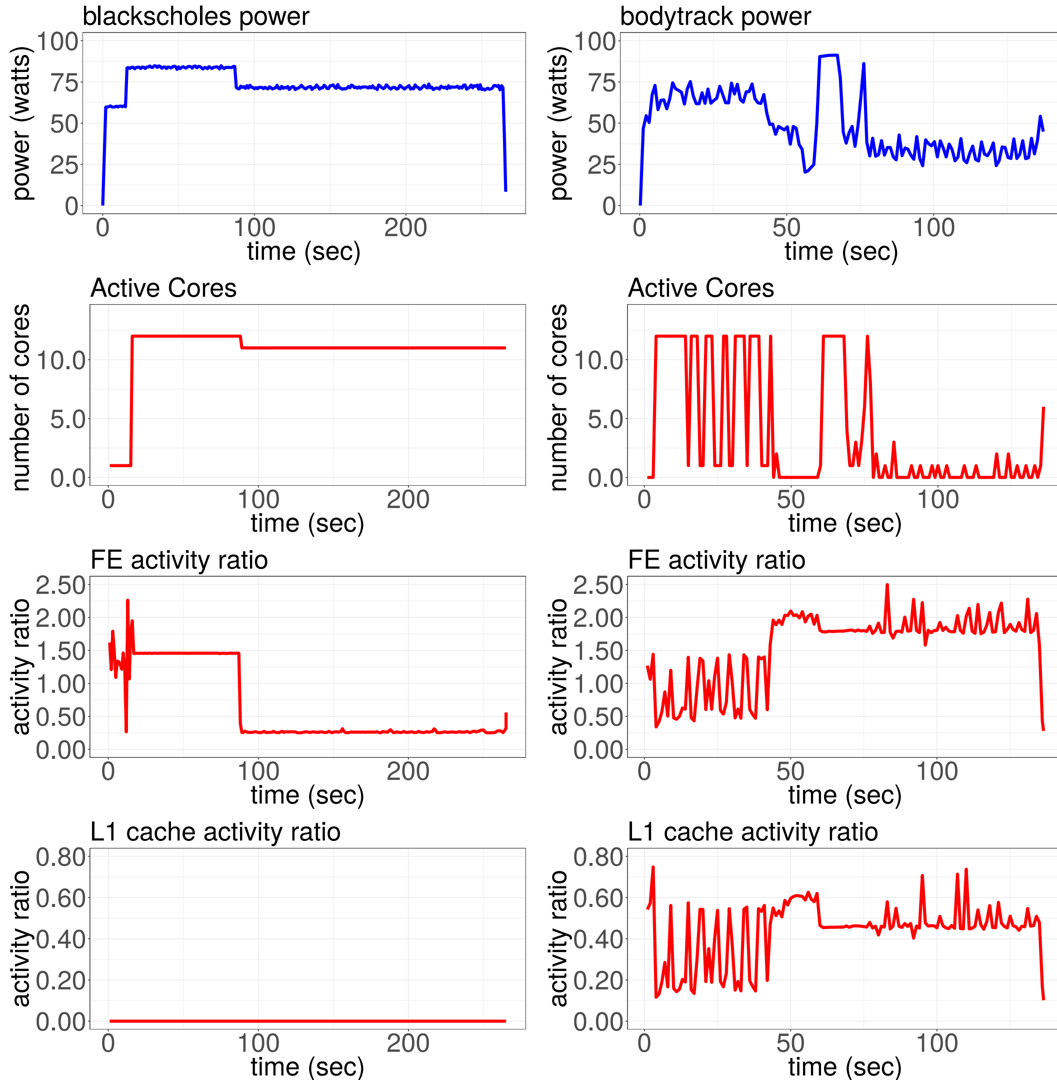


Fig. 6.2 Power, active cores and component activity ratio traces when running on 12 cores of a single socket. Architectural components shown are the fetch unit (FE) and L1 cache. The activity ratios are the number of retired micro operations per unhalted cycle, relevant to each architectural component. In the case of cores, activity ratio is the number of active cores. For memory the activity ratio is measured as the number of references (for caches) or LLC misses (for main memory) per cycle.

Our first (baseline) model attempts to circumvent the relative complexity of dealing with manufacturing variability by assuming that all applications are impacted the same by it. It uses a PMC-based model to predict an application's power consumption, without considering variability. Then it uses a

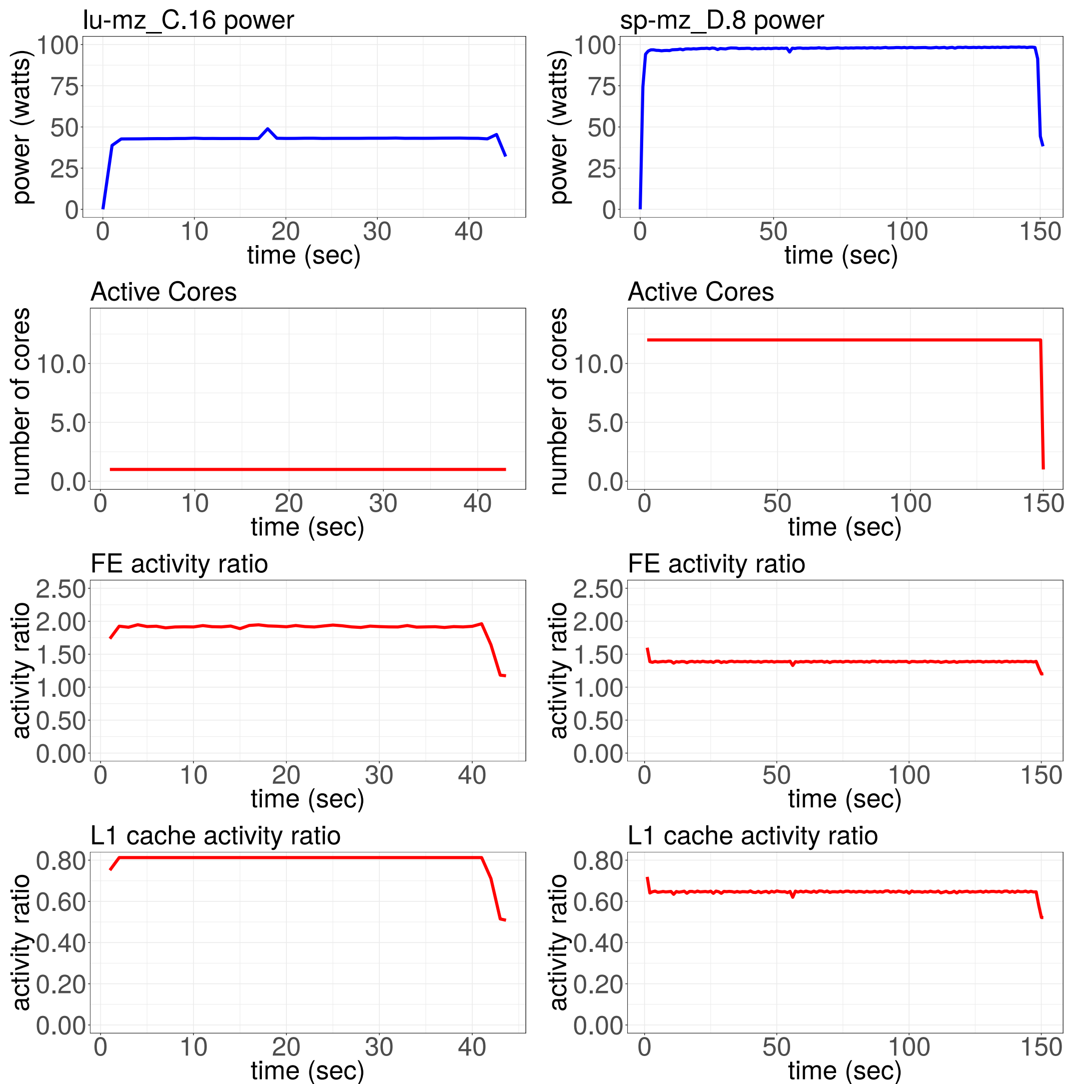


Fig. 6.3 Power, active cores and component activity ratio traces of MPI, multinode applications when running on 12 cores per socket. Results show only activity on one socket, but the other sockets demonstrate similar behavior. Architectural components shown are the fetch unit (FE) and L1 cache. The activity ratios are the number of retired micro operations per unhalted cycle, relevant to each architectural component. In the case of cores, activity ratio is the number of active cores. For memory the activity ratio is measured as the number of references (for caches) or LLC misses (for main memory) per cycle. For multi-node applications PMC data is collected for all the processes and individual predictions are made for each socket.

single benchmark, in our case an OpenMP implementation of the *cholesky* decomposition of a dense 65 MB matrix, and measures its average power consumption on each socket we want to generate a model for. We chose *cholesky* because it is a dense linear algebra kernel, which stresses both CPU and cache memory accounting for power consumption variability across the sockets. Once this information is obtained, we characterize the variability between sockets in terms of power ratios and

we apply them to power prediction of a target application (denoted as *app*), made by using the PMC profiles obtained from an execution on a single reference socket (denoted as $socket_{ref}$).

The PMC-based prediction model uses PMC values to capture the contribution of each chip's architectural components to power consumption and then models each component using activity ratios. These ratios are defined as the number of retired micro operations relevant for the targeted architectural component per active cycle. For example, for main memory and caches, activity ratios are the number of references or misses per cycle, respectively. Activity ratios reflect the usage of the corresponding component for a given application. The model assumes that a component's contribution to power consumption is proportionate to its usage (activity ratio). The granularity at which we can decompose a chip into architectural components depends on the underlying architecture and the available PMC. Table 6.1 shows the different components and their corresponding PMC formulas for the Intel's Broadwell architecture. We infer the formulas from the Intel 64 and IA-32 Architectures Software Developer's Manual [1].

Figures 6.2 and 6.3 show power and activity ratio profiles for the active cores (CORES), fetch unit (FE) and L1 cache, for the *blackscholes*, *bodytrack*, *lu-mz_C.16* and *sp-mz_D.8* parallel codes. For multi-node applications, *lu-mz_C.16* and *sp-mz_D.8*, we show the activity ratios and power consumption of one of the processes, running on one of the sockets.

Each process has its own set of activity ratios that result in individual predictions, per socket that the individual process run on. The specific experimental setup to obtain these measurements is detailed in Section 6.3.1. All applications have a unique power profile that is the result of the different component activity ratios. For example, *blackscholes* and *sp-mz_D.8* have high CORES activity that contribute to the power consumption. However, *blackscholes* has minimal activity in L1 cache, which results in lower overall power consumption, when compared to *sp-mz_D.8*. Moreover, *lu-mz_C.16* has similar activity ratios to *sp-mz_D.8* but significantly lower activity in cores (only uses 1 core per process), which results in lower power consumption. Finally, we can observe how changes in component activity influences power consumption in the cases of *blackscholes* and *bodytrack*.

We then align the activity ratios with measured power data, which allows us to express the power of an application on a particular socket as:

$$P = AC * W_{cores} + \sum_{c=1}^{N_{comp}} (AR_c * W_c) \tag{6.1}$$

where AR_i is the activity ratio of power component i, AC is the average number of active cores and P is the power consumption, at a given moment. These values are known for any given application in our training set. However, we need to find the contribution of each component to the total power consumption P. This is expressed using a set of weights, denoted as W_i for architectural component i and W_{cores} for active cores.

We determine the weights using a training stage during which we monitor power along with the architectural component activity for a small set of kernels. The choice of training benchmarks should

Table 6.2 Benchmark training set for PMC-based power prediction model.

Benchmark	**Description**
cholesky	cholesky factorazation kernel
knn	K-nearest neighbours kernel
matmul	Floating point matrix multiplication kernel
md5	MD5 message-digest algorithm
prk2_stencil	Tests the efficiency with which a space-invariant symmetric filter (stencil) applies to images
qr_tile	Tiled QR factorization kernel
sparseLU	Sparse LU factorization kernel
stap	Space-Time Adaptive RProcessing for radar detection of an objects position
symmatinv	Symmetric matrix inversion kernel
vector-redu	Computes the sum of the elements of a vector
mem_bench	A micro-benchmark that stretches different memory levels

reflect different application behaviors (e.g., computation vs. memory bound) and stress different architectural components, such as integer or floating point units, in addition to the different memory levels. The list of applications used for training can be found in Table 6.2. To better account for the power contribution of the memory subsystem, this list includes an additional synthetic code, *mem_bench*, a microbenchmark that causes misses on different levels of the memory hierarchy.

With the data measured in this training stage, we then use linear regression to determine the values of W_i and W_{cores} that best fit in Equation 6.1 on a given socket. The resulting linear model is socket-specific and can predict the power consumption of a generic application assuming its activity ratios for all architectural components and cores are known.

We obtain the values of W_i and W_{cores} for a specific socket we use as reference. To account for manufacturing variability we use the power ratios computed using the *cholesky* benchmark. The power consumption of app on any $socket_i$ is then obtained by the following formula:

$$P^{app}_{socket_i} = P^{app}_{socket_{ref}} * \frac{P^{choleksy}_{socket_i}}{P^{cholesky}_{socket_{ref}}} \tag{6.2}$$

In the case of multi-node applications, we predict the power consumption of each socket an MPI process run on, by applying the power ratio corresponding to that socket.

The PR model's precision is subject to the benchmark application used for measuring the sockets' manufacturing variability. Figure 6.4 shows the manufacturing variability for two distinct benchmarks, *cholesky* and *sparseLU*, as measured by running them on all sockets. As observer, the two benchmarks produce different variability ratios. In the case of *sparseLU*, which solves the LU factorization problem on a sparse matrix, the observed variability ratio is on many occasions 1. This means that this benchmark fails to measure manufacturing variability effectively and would be able to adjust the original prediction to fit a socket's variability. On the other hand, *cholesky*, which is a computation

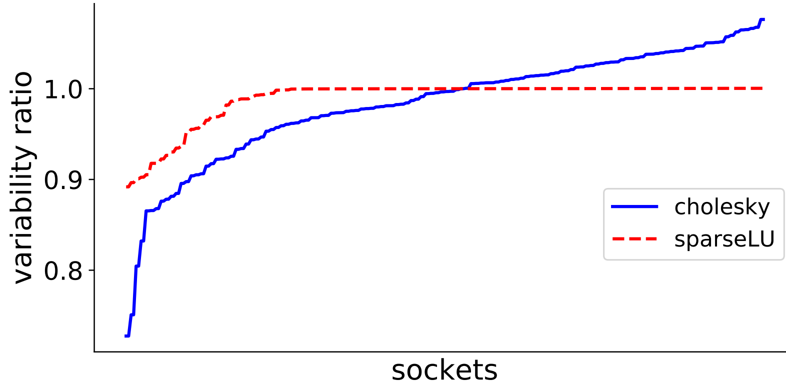


Fig. 6.4 Comparison between the variability ratios over all 256 sockets as observed for *cholesky* and *sparseLU* benchmarks. Variability ratio here is the fraction the Power consumption of the benchmark on a given socket, divided by the power consumption of the same benchmark on a reference socket. The cpu bound *cholesky* detects variability more precisely than *sparseLU*.

bound application and stresses the processor more than *sparseLU*, produces a more precise view of the system's heterogeneity due to manufacturing variability.

6.1.2 Variability-Trained Prediction Model

Our second model does not assume the impact of manufacturing variability to be independent of the parallel code. Instead, it aims at capturing the impact of manufacturing variability on each specific application. Due to manufacturing variability, power consumption differs between sockets, which means that W_c and W_{cores} are socket-specific and obtained by solving Equation 6.1 individually for each socket. In terms of the activity ratio values per application, we assume them to be invariant across all sockets featuring the same architectural design. Consequently, the socket-specific Formula 6.1 can be extended to integrate all sockets featuring the same architectural design:

$$P^{app}_{socket_i} = AC^{app} * W^{socket_i}_{cores} + \sum_{c=1}^{N_{comp}} (AR^{app}_c * W^{socket_i}_c) \tag{6.3}$$

From this formula we can obtain a power prediction of an application running on any chosen socket, which is characterized in terms of its weights. For each parallel code we just need a single run on a generic socket to compute AR_c and AC, which are socket-independent as they are determined by the architectural design.

Figure 6.5 shows power profiles of the *blackscholes* code (solid line) running on three different sockets, together with the corresponding predicted power consumption (dashed line). Details on the machine and execution setup can be found in Section 6.3.1. Figure 6.5 displays how the power

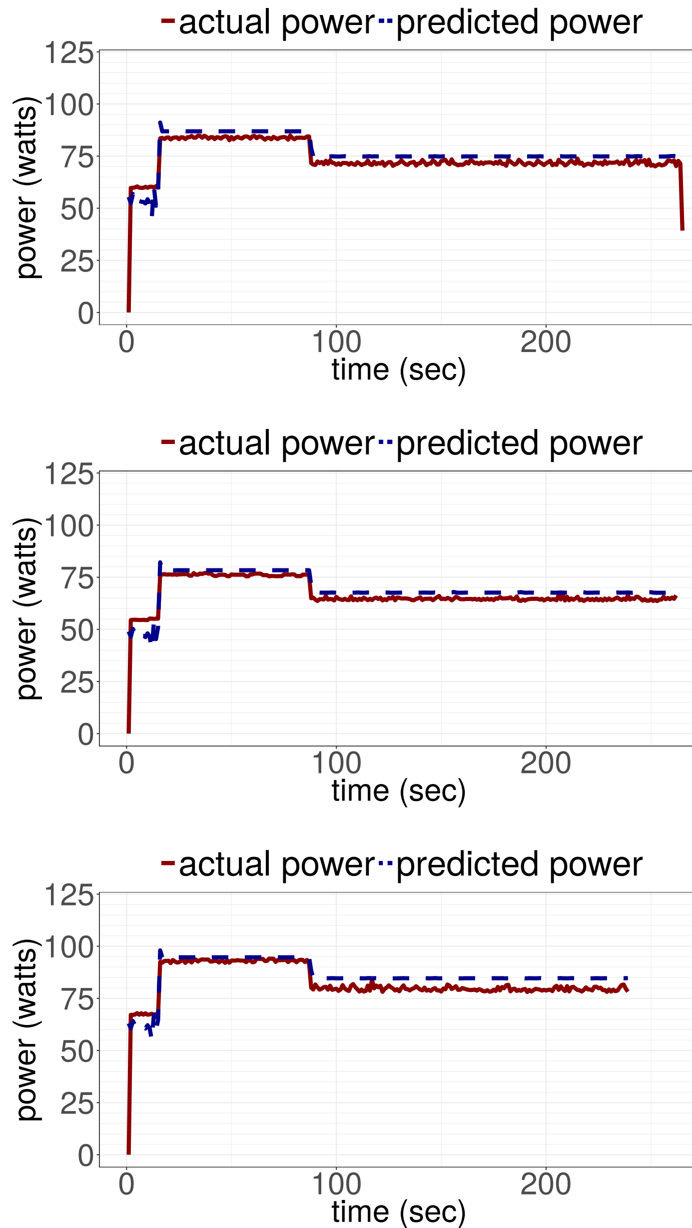


Fig. 6.5 Actual and predicted power consumption, using the PMC-based (Optimized PMC) model, for *blackscholes* under three distinct sockets. Same CPU chip model is mounted on all three sockets, utilizing all 12 available cores.

consumption varies up to 19% for the same application (from 76W to 96W peak power), depending on the socket it runs on. The predicted values capture the power consumption variability for all cases.

6.1.3 Model Optimization

To improve the model's precision, per each application, we consider using only a subset of architectural components that provides the most accurate results. This way we mediate any biasing the training set may have. All possible combinations of architectural components are considered and a prediction error is computed for each one. We use the Mean Absolute Percentage Error (MAPE) formula for computing this prediction error

$$M = \frac{100}{n} \sum_{t=1}^{n} \left| \frac{A_t - F_t}{A_t} \right| \tag{6.4}$$

where A_t and F_t are the actual and predicted values for observation t, and n is the total number of observations. The model with the lowest error value is chosen for all future predictions. This tuning process can be done offline per each targeted application using data obtained from a single parallel execution. The same optimization is also applied to the PR model. From this point on, any mention to the PR and VT models, references the optimized versions of the models. The models without the optimization mentioned in this paragraph, will be referred to as unoptimized PR and VT models. Section 6.3.2 shows a detailed evaluation and validation of both models.

6.1.4 Predicting Power for Multi-Node Applications

For multi-node applications, individual predictions need to be made for each MPI process. A corresponding model needs to be applied to each socket and process, which has been trained for each socket individually, producing a different set of weights. The result is a prediction of the power consumption of each process and each socket. For example, if we have an application with N processes and a system with M sockets, then we make $N \times M$ predictions.

6.2 Job Scheduling Policies

Next, we propose two new variability-aware job scheduling policies, the *Power Ratio Variability Prediction* and the *Variability-Trained Prediction*. These schedulers make use of the power variability prediction models introduced in Section 6.1. With these models, the schedulers predict the job's power consumption on all available sockets and schedule the job on the most efficient one. Before scheduling the job, the scheduler checks whether the system wide power budget would be respected once the job starts running. If this is not the case, the job will wait until other jobs finish and more power is available. In the case of multi-node jobs, we consider the total power consumption of all its processes. If possible, sockets on the same node are preferred, but intra-node topology is not considered. Furthermore, we consider three job scheduling policies representative of the state-of-the-art and with increasing complexity: *SLURM extended*, *Power Estimation*, and *Power Estimation+Variability Aware*. The first two are variability-agnostic, while the third is variability-aware. Finally, we also consider an *Ideal Variability Prediction*, which is based on an oracle power variability predictor that knows exactly how much power a job will consume on any processor in the

system. We extend the SLURM's logic [85] to implement the various power- and variability-aware job scheduling policies presented in this section. We chose SLURM as our reference because it is widely used on HPC production systems and well studied in the literature. All job scheduling policies are described in the following:

SLURM Extended: This policy implements SLURM scheduler's logic. We extend the default behavior to not exceed the global power budget, by considering the worst case scenario, which is that each job can consume the maximum power budget allowed per socket. Additionally, we extend the scheduler to initiate backfilling for power as well [114]. Traditionally, if a job requests more sockets than currently available, the scheduler will try to schedule a different job without causing delays. The same will happen if a job requests more power than the system can allocate.

Power Estimation (*SLURM+PE*): This policy extends further *SLURM extended*'s behavior by using a user provided estimation of a job's power consumption. For precision we obtain power profiles of previous execution of the jobs to estimate the power consumption. This is the equivalent of using a variability-agnostic prediction model. This scheduler does not consider manufacturing variability as it assumes all sockets consume the same power for a given job.

Power Estimation+Variability Aware (*SLURM+PEVA*): This policy implements elements from the state-of-the-art practices in power-aware job scheduling [67, 79]. Similarly to *SLURM+PE*, it estimates the power requirements of a job using a power trace from a previous execution, same as *SLURM+PE*. It also orders sockets and allocates first the most power efficient ones to minimize the system's net power consumption. The socket ordering is obtained by running a simple benchmark, the *cholesky* kernel, on all sockets and observing their power consumption. A drawback of this approach is that it assumes all parallel jobs to be influenced by manufacturing variability in the very same way. Another drawback is that a job's power estimation depends on the socket used for profiling and thus it is possible to under or over-estimate the final power.

Power Ratio Variability Prediction (*SLURM+PRVP*): This is the first new policy we propose. It relies on our Power Ratio Model, presented in Section 6.1.1, to guide scheduling decisions. A single power and performance profile of a given job (or one for each socket a process was run on, in the case of multi-node jobs) is required in order to compute the activity ratios, which can be performed on any set of sockets in the system. Running the single benchmark a priori on all sockets is also required.

If the predicted power for a new job makes the system budget to go over its limit, then the job waits until resources are released. The backfilling scheme is the same as with the previous policies. This policy's framework is shown in Figure 6.6. Note that two training processes are shown in the figure, for the different power prediction models proposed in this work. The lower box shows corresponds to the *SLURM+PRVP* policy.

Variability-Trained Prediction (*SLURM+PMCVP*): Our second proposed policy is similar to the PRVS policy, but it uses our VT prediction model, presented in Section 6.1.2, to obtain power consumption predictions. This policy requires running the training benchmark set from Table 6.2 on all sockets in order to train the model. Using the variability aware power predictions, scheduling decisions are made in the same manner as with the *SLURM+PRVP* approach. The framework for

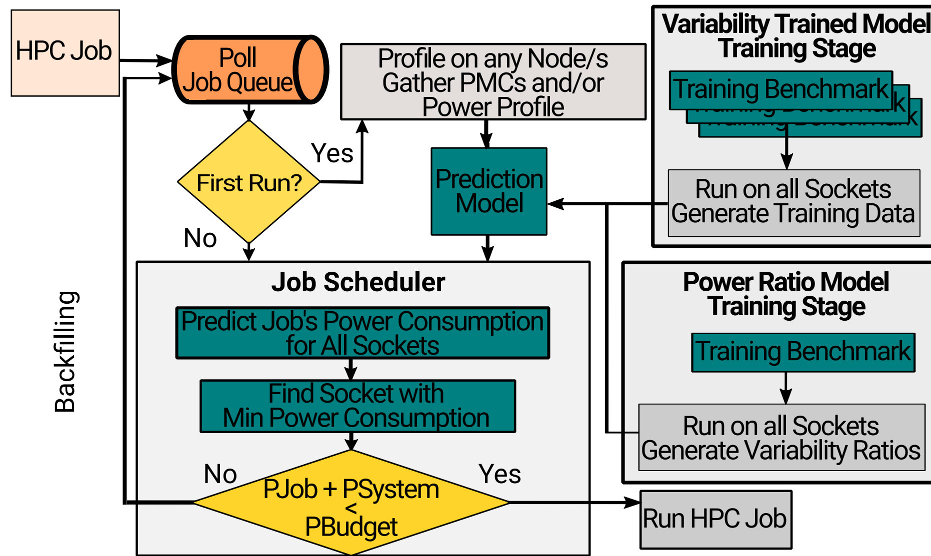


Fig. 6.6 Framework for the *SLURM+PRVP* and *SLURM+PMCVP*. The upper right box shows the steps for training the PMC model, while Power Ratio training stage is shown in the box below.

this policy is described in Figure 6.6. The box on the upper right corner corresponds to the training process of the *PMC-based Prediction Models*, used by *SLURM+PMCVP*.

Ideal Variability Prediction (*SLURM+IVP*): Identical to *SLURM+PRVP* and *SLURM+PMCVP*, but using an oracle power predictor to drive job scheduling decisions. This policy is aimed at showing the impact of using a 100% accurate model to guide scheduling decisions. Thus, *SLURM+IVP* is used for comparison purposes to show the maximum benefits that can be achieved by power- and variability-aware job scheduling policies.

6.3 Model Validation and Policy Evaluation

6.3.1 Experimental Setup

To evaluate the proposed models and job scheduling policies, we have access to 128 nodes of the Quartz cluster, as described in Section 3.1. We use an in-house simulator to mimic the behavior of a job scheduling system on a production platform like Quartz. The simulator is described in more detail in Section 3.3. For training the prediction models, we use the set of kernel and micro-benchmark applications described in Section 3.4.2. Our benchmark applications used as the cluster's workload are also described in Section 3.4.2. They consist of a mix of single and multi-node applications. Single node ones can run on a single node, while multi-node ones require a number of sockets. We append the number of processes used at the end of the name of each multi-node application. They range from 8 to 64 processes and each process is scheduled on a single socket. For example, an 8 process application requires 8 sockets (4 nodes) to run. We maintain socket temperature between 38-42 °C, in order to only observe the power consumption variation relevant to manufacturing variability. Based

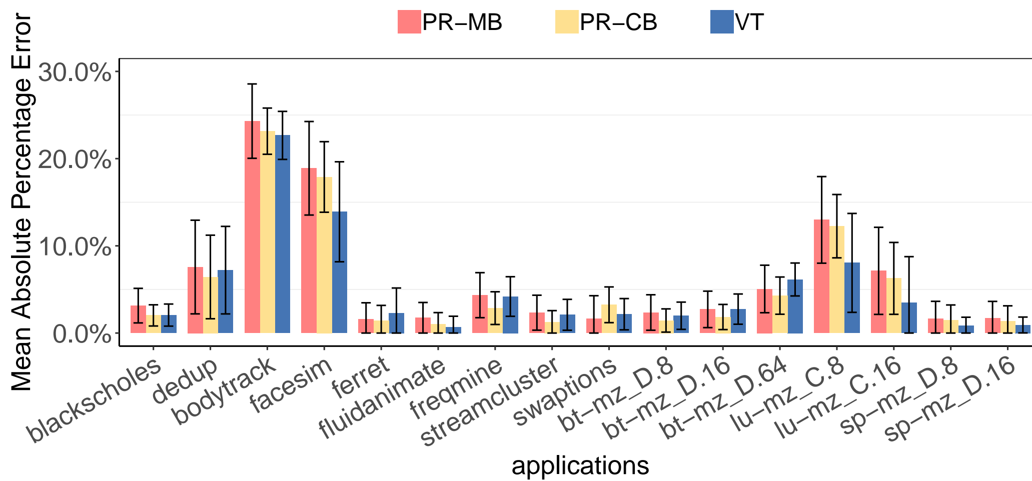


Fig. 6.7 Comparison of average power prediction error for all models over all sockets. The error bars show the standard error deviation across all sockets for the corresponding application.

on the data and independent nature of jobs run on HPC clusters, we expect the observed trends in this work to scale on larger systems.

6.3.2 Model Validation

In this section we experimentally validate the prediction models presented in Section 6.1. We analyze the models in terms of Mean Absolute Percentage Error (MAPE) (see Equation 6.4) between predicted and real values. Figure 6.7 shows the MAPE values of the average power predictions for all applications over all sockets. The error bars show the standard deviation of the MAPE metric, computed over all the 256 sockets. These results correspond to the optimized versions of the Power Ratio (PR) prediction model, presented in Section 6.1.1, and the Variability-Trained (VT), which is presented in Section 6.1.2. Since the PR model incorrectly assumes that all applications are affected by manufacturing variability the same way, we show two versions of the PR. Models are denoted as PR-MB and PR-CB, using by a memory bound (*sparseLU*) and a computation bound benchmark (*cholesky*) for computing the variability ratios, respectively. Overall, all models performs well, achieving an average error below 10%. The VT model outperforms the PR models, while PR models varies depending on the benchmark used to compute the variability ratios. PR-MB consistently performs worse than PR-CB, since the memory bound benchmark detects up to 15% less variability than the computation bound one. This disparity among results for the PR model can become a more serious problem in the future, as variability is expected to increase [103]. The more robust VT model will be better suited, since it does not falsely assume that variability is application independent. The unoptimized versions (see Section 6.1) of the same models perform again similarly among themselves, but significantly worse that their optimized counterparts shown in figure 6.7. On average, all unoptimized model versions' error reach up to 16% (results not shown).

Our results show that all three models are able to predict the power consumption variability in some cases. However, it is possible to miss-predict if an application's behavior is not well represented by the benchmarks used for training.Two such cases are *bodytrack* and *facesim*, that although they are effectively using more than 8 cores, their power consumption remains below 60 Watts. Moreover, PR-MB and PR-CB additionally fail to produce accurate predictions for *lu-mz_C.8*, which use only a single core on each socket. Higher prediction errors can impact the effectiveness of the scheduling policies that rely on them, causing them to over-estimate a job's power consumption. Over-estimating power can lead to underutilization of the cluster's resources. Even high error values though, such as in the case of *bodytrack*, are better and more robust approximations of the actual power consumption than user estimations, which comes with no guarantees.

6.3.3 Variability-Aware Scheduling Evaluation

In this Section we evaluate the two novel scheduling policies, *SLURM+PRVP* and *SLURM+PMCVP* , which are presented in Section 6.2. We compare them to four other scheduling policies, *SLURM extended* , *SLURM+PE* , *SLURM+PEVA* and *SLURM+IVP* , which are also described in 6.2. *SLURM extended* policy implements SLURM's logic in our simulator, with the addition of power-awareness and power backfilling. *SLURM+PE* and *SLURM+PEVA* employ state-of-the-art features of power-aware policies [30, 67, 113], while *SLURM+IVP* demonstrates the ideal scenario, where prediction is 100% accurate. In the case of *SLURM+PRVP* we use *cholesky* to produce the variability ratios required by the PR model, since it produces more accurate predictions than *sparseLU* (see Section 6.3.2). The evaluation is done based on a simulator, as described in Section 6.3.1, by feeding it performance and power traces from actual executions on the Quartz cluster. For our experiments we generate random job workloads composed of nine applications from the PARSECSs suite and seven multi-node jobs from NAS-MZ, simulating both bursty and heavy traffic scenarios (see Section 6.3.1). The main objective of each policy is to improve the cluster's performance and energy consumption, while keeping the total energy consumption below a certain global power budget. All policies treat power as a limited resource and depending on a predicted or estimated power peak for each job, they restrict the number of running jobs to only those that can be accommodated by the given global power budget.

Figure 6.8 compares the different policies in terms of average job turnaround time reduction (x axis) and their maximum power consumption (y axis). The turnaround time reduction shown in percentages on x axis is over the *SLURM extended* policy, for the corresponding power budget and traffic. We define job turnaround time as the time a job waits to be scheduled, including scheduler overhead, plus its execution time. The average job turnaround time is computed as the sum of turnaround time of all jobs, divided by the number these jobs. Results consider four different system-wide power budgets, 5K, 7.5K, 10K, 15K and 20KWatts and the bursty and heavy traffic scenarios described in Section 6.3.1. Our workloads require 25K Watts to run using the whole cluster without any power restrictions. In the 5KW case, the *SLURM extended* is forced to drop jobs that use 64

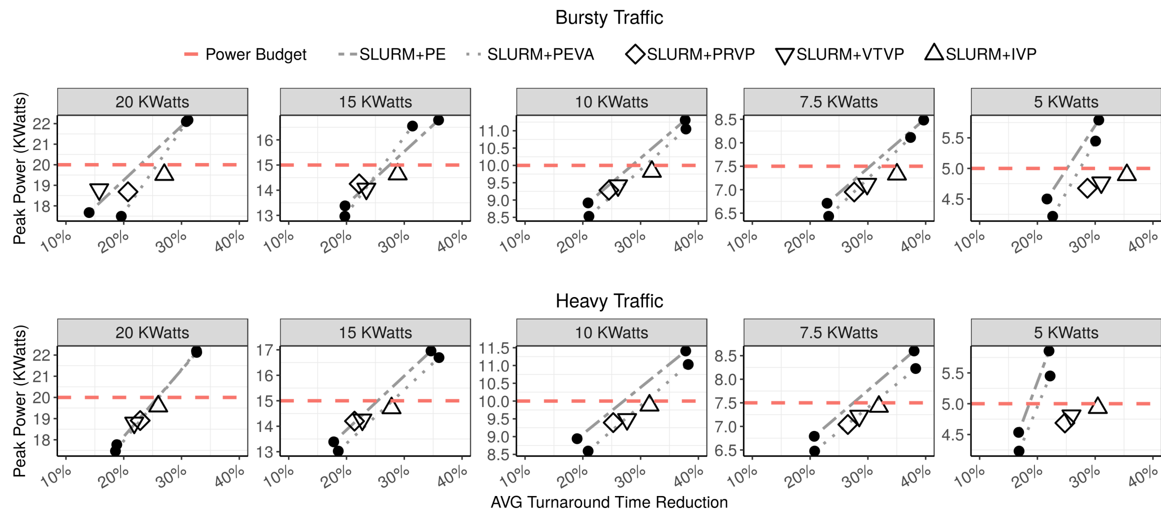


Fig. 6.8 Power and Average turnaround time reduction over SLURM extended. Results SLURM+PE and SLURM+PEVA values range as shown by the corresponding lines, depending on the estimation provided.

sockets, since it estimates that these jobs require more power than available to the system. The minimum budget that allows *SLURM extended* to run jobs that demand 64 sockets is 7.5KW. Since the case of 5KW *SLURM extended* runs a lighter load (dropped 64 socket jobs), results are slightly biased towards *SLURM extended* (just for the 5KW case).

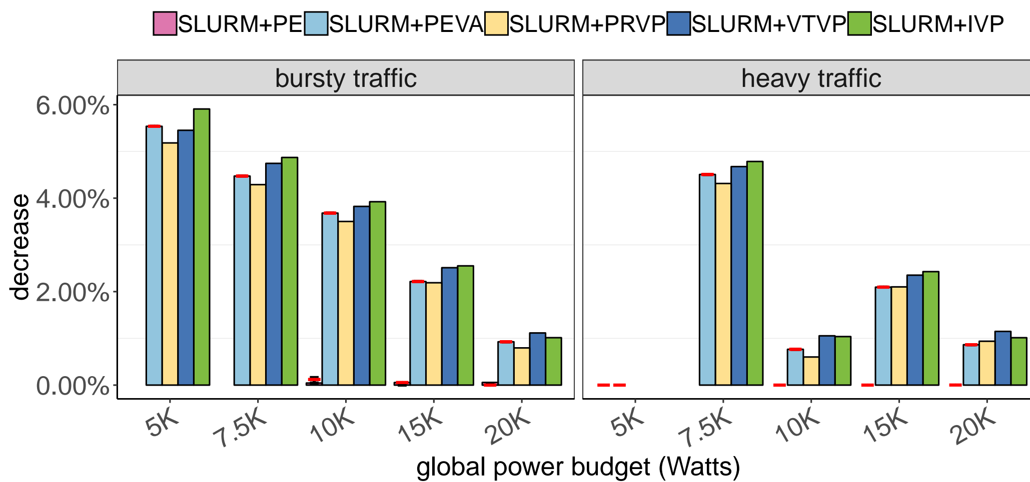


Fig. 6.9 Energy consumption Reduction over *SLURM extended*, under different budgets and traffic scenarios. SLURM+PE, which is variability agnostic fail to save any energy, while SLURM+VTVP performs best.

SLURM+PRVP, *SLURM+PMCVP* and *SLURM+IVP* policies are denoted with different symbols. The two considered traffic scenarios produce similar results. Except for the budget of 20KWatts,

SLURM+PMCVP performs marginally better (up to 2%) than *SLURM+PRVP*, which reflects the better precision of its model. The lowest reduction in terms of job turnaround time, 12%, is observed in the case of the bursty traffic scenario, under a power budget of 20KWatts, for the *SLURM+PMCVP* policy. The largest improvement, 33%, is obtained by the *SLURM+PMCVP* policy when managing the bursty traffic under a power budget of 5KWatts. The benefits of the *SLURM+PRVP*, and the *SLURM+PMCVP* policies reach an average of 24%, considering all power budgets and traffic scenarios. *SLURM+IVP*, which is the ideal scenario, performs slightly better than *SLURM+PRVP*and *SLURM+PMCVP*under all power budgets, although the benefits achieved by our two proposed policies are very close to the best possible scheduling. *SLURM+PE* and *SLURM+PEVA* are shown as lines, where their performance and maximum power consumption can lie on any point on the corresponding line. This is because unlike our proposed policies, which use prediction models to compute the power consumption of each job on each socket, the power-estimation-based policies, *SLURM+PE* and *SLURM+PEVA*, use a single power estimation obtained from power profiling on a single socket. Due to the power variability of each socket, it is possible that the estimation varies according to the variability of the socket used for profiling. This variance impacts the policies' efficiency. Underestimating power by using a power efficient socket allows more sockets to get allocated, but the net power consumption of the system can exceed the system-wide power budget. Contrary, overestimating power can lead to significant performance degradation, since the policy becomes more conservative, underutilizing the available power budget. Using a socket with moderate power consumption to get the power trace used for the estimation can achieve comparable results to *SLURM+PRVP*, *SLURM+PMCVP*. However, all points on the lines showing the range of possible results for *SLURM+PE* and *SLURM+PEVA* show worse reduction than *SLURM+PRVP*, *SLURM+PMCVP* and *SLURM+IVP*, since variability is not considered.

Figure 6.9 shows the reduction in the system-wide energy consumption. Both traffic scenarios' benefits increase as the system wide power budget is reduced. When less power is available, considering variability is important for energy saving, since we can choose to always use the most power efficient sockets. Note that under heavy traffic, the 5K scenario offers no benefit. This is because a significant number of multi-node jobs are dropped by the *SLURM extended* since they appear to require more power than available to the system. As a result, *SLURM extended* runs a lighter workload than he rest of the policies. This also happens in the bursty scenario, but since the workload only contains a few 64 socket jobs that are dropped, we still observe significant benefits. *SLURM+PE*, which is variability-agnostic, offers no benefit over the *SLURM extended* policy. *SLURM+PEVA*, which prioritizes allocation of power efficient sockets, matches the energy savings of the *SLURM+PRVP* and *SLURM+PMCVP* policies. Accounting for power variability can have a significant impact on energy efficiency reaching 8% on the most energy-restricted scenarios.

Results shown across Section 6.3.3 prove that our proposed *SLURM+PRVP* and *SLURM+PMCVP* policies can improve energy efficiency up to 8% (4% on average) over simple solutions commonly used. Moreover, job turnaround time is reduced up to 30% (24% on average). Compared to the *SLURM+PEVA* policy, which is variability-aware, our method is more robust as none of the

SLURM+PE and *SLURM+PEVA* policies can guarantee that the system-wide power budgets are respected.

6.4 Summary

In this work, we demonstrate that taking into account manufacturing variability to drive job scheduling policies provides significant benefits in terms of performance and power consumption. We propose two job scheduling policies, each one using a different power prediction model: the first assumes that power variability impacts all application equally, while the second one aims at obtaining the power variability impact per application by training the model for each individual socket. We compare both approaches with a range of state-of-the-art approaches as well as an approach using an oracle model.

We examine the benefits of our policies under bursty and heavy traffic scenarios and different power budgets. We observe significant improvements on job turnaround time (up to 30% and 24% on average) and energy consumption (reducing it up to 8% and 4% on average) when compared to state-of-the-art approaches that do not consider appropriately the manufacturing variability in existing processors. Moreover, the model-driven policies proposed in this work accurately predict the variability per socket and, as a result, they guarantee that power consumption always remains below the system-wide budget, while the policies that rely on user estimations or prior power profiling fail to do so.

7

Conclusions

In this thesis, we presented two approaches that mitigate the inherent heterogeneity found even in processors of the same design model, due to manufacturing variability. Manufacturing variability refers to the variability in power consumption and frequency of processors because of artifacts during the manufacturing process of transistors, which result in variations in transistor parameters, like V_{th} and L_{eff}. Our first approach is implemented at runtime level and redistributes power among socket on NUMA nodes, to mitigate their performance variability, essentially improving dynamic load-balancing. Our second approach is implemented at workload manager level, where we develop and use a variability-aware power prediction model to guide system-wide scheduling decisions and improve the system's throughput and energy efficiency. Furthermore, in order to evaluate both approaches, we have implemented the PARSECSs benchmark suite, using a task-based programming model (OpenMP 4.0), based on the original PARSEC benchmarks. In this Chapter we present the conclusions and possible future directions this research line can head to, based on the results and ideas presented in this thesis.

7.1 Relevant Benchmarking in HPC

Benchmarking is key component for researchers to evaluate their ideas and proposals, both at software and hardware level. Choosing the benchmark applications carefully is vital for an evaluation to be meaningful and not biased. Benchmark applications should ideally capture all application behaviors and scenarios that can occur when running actual HPC workloads. Typically in HPC, small kernel applications are the backbone of every benchmark suite. The reasoning behind this choice, is that HPC applications are usually composed by smaller kernels, which can be efficiently parallelized. This approach however, is not representative of many workloads that are run today on HPC systems. As HPC computing has become more accessible today and more powerful programming models allow users to parallelize a more diverse set of applications, these kernels are no longer representative of the applications or programming paradigms of the workload running on HPC systems. In Section 4, we present PARSECSs, a task-based implementation of the PARSEC benchmark suite. The PARSECSs

benchmark suite is composed by applications from a large number of different domains, used in todays computing. Our implementation use a state-of-the-art task-based model and employ emerging programming paradigms, such as pipeline parallelism, in addition to traditional ones. We also evaluate our task-based implementations to the original PARSEC Pthreads/OpenMP 2.0 versions. We show that task-parallelism can be applied on varied domains of computing and that it is actually easier to use. We show that although in many cases, such as the smaller kernel applications, task-parallelism is as efficient as traditional models, there are many occasions where it can improve an applications performance. The asynchronous nature of tasks with the addition of dataflow relation, allow the user to easily overlap I/O phases with computation. Moreover, the underlying runtime is able to efficiently synchronize and load-balance parallel applications. In comparison, the original PARSEC applications require the user to implement such load-balancing and synchronization mechanisms from scratch, resulting in more complex and less portable code. Moreover, using a benchmark suite which is implemented with a state-of-the-art runtime, allows researchers to implement their ideas at runtime level, without the need to modify each application in the suite.

7.2 Runtime Mitigation of Manufacturing Variability

Advanced programming models are typically coupled with dedicated runtime systems, which deal with synchronization and load-balancing in parallel applications. These runtimes offer ideal platforms for developing experimental solutions for any kind of performance problems parallel applications may face. Moreover, runtimes can hide architectural details from the user and allow the user to design applications without considering code portability across different machines, or particularities of specific hardware. This creates an ideal platform for developing a framework capable of dealing with the issues created by manufacturing variability, since the user can let the runtime deal measure and deal with variability at execution time, in a transparent manner.

Static solutions are not practical when dealing with manufacturing variability. First, processors are not characterized by vendors regarding their power variability. This power variability translates also to performance variability, since when power constraining a processor, it can no longer consume additional power to reach the nominal frequency set by the manufacturer. Moreover, applications are affected differently by manufacturing variability. For example, computational bound applications suffer more variability compared to memory bound ones, since they stretch the cpu more. An application using the ALU unit may experience different levels of variability than another one using the FPU, since they stretch different components of the processor (thus different transistors). Any static approach, like statically distributing the parallel workload among processors will be inefficient, since we cannot have a priori knowledge of how the variability on a certain processor will affect a specific application. Runtimes that employ dynamic load-balancing on the other hand, can react to the effects of manufacturing variability during execution. Although oblivious to the underlying heterogeneity of the system, a runtime will redistribute work from busy to idle processors. However, in this work we show that even dynamic load-balancing can be further improved. In Chapter 5 we present a runtime

solution, which extends dynamic load-balancing with dynamic resource redistribution in order to improve the performance of parallel applications when running on power constrained NUMA nodes. We profile the task-based PASRSECSs benchmarks under different power distributions among the sockets on the same node and different number of active cores. Our analysis shows that performance can be improved up to 1.30x, when compared to evenly distributing power and having all cores active. We also extend the OpenMP 4.0 runtime system to transparently profile an application during execution and then choose an optimized power and core distribution. Our implementation starts off with an even distribution of power and cores among the sockets and monitors the performance of tasks for small fraction of time. It consecutively tries and monitor different configurations to find a better candidate than the starting configuration. We also demonstrate that it is possible to achieve up to 1.22x speedup (optimal solution by static analysis is 1.3x speedup), by carefully designing the configuration exploration space. Eliminating configurations that are unlikely to give good results, such as allocating more power to a socket with only a couple of cores active.

7.3 Model-driven Scheduling Mitigation of Manufacturing Variability

Large HPC clusters are typically managed by dedicated software referred to as workload managers. This type of software manages user submitted workloads and allocates the necessary resources, such as cores and memory. Since these machines operate with hundreds or thousands of cores, their power demands to operate are very high. Moreover, current HPC systems are power provisioned considering the worst case scenario, which is that all nodes may need to operate at maximum capacity. This scenario is not likely, as clusters are typically running diverse type of applications. Memory bound workloads stress the processor less than computation bound ones, requiring less power while still making use of the nodes they run on. An emerging cluster design, which aims to make more efficient power use and is known as overprovisioning, is to not abide by the aforementioned rule, that the system must be provisioned with enough power to operate even at full capacity. All nodes may be in use, but not at the upper limit of their power consumption. If nodes require more power, then less nodes should be used.

For a system such as this to operate, the workload manager must consider power as a finite resource, as it does for cores and memory. The workload manager should try to find candidates that can run together without exceeding the power budget. Current workload managers do not implement this functionality, but possible solutions already exist in the literature [8, 30, 53, 67, 79, 113, 141, 143]. However, most consider redistributing power where it is needed more, instead of considering the power requirements of a workload at scheduling time. Moreover, only a few consider manufacturing variability. In Section 6, we present an alternative approach, which indeed considers an application's power requirements at scheduling time. It decides which among multiple candidates should run together, given a system-wide power budget which must not be exceeded. Instead of relying on user estimation, we employ an analytical model, which relies on PMC to predict an application's power requirements. We also extend the prediction model to consider manufacturing variability. Our

evaluation against contemporary workload managers shows that scheduling jobs, while considering their power requirements, can improve job throughput up to 30% or 24%, for heavy and bursty traffic scenarios respectively. In terms of energy efficiency, we observe improvements up to 8% and 4% on average, for the same heavy and bursty traffic scenarios. When compared to manufacturing variability agnostic approaches, our approach can alway guarantee that the power budget will not be exceeded. Contrary, scheduling policies that do not consider variability, may over- or under-estimate power the consumption of an application. This means that they either do not manage to get optimal job throughput and energy efficiency or that the power budget may be exceeded. We also compare our variability-aware prediction model, to the current state-of-the-art [79], which relies on a single benchmark to measure variability and then adjust the original variability-agnostic prediction. In contrast to our model, this approach relies on the unsound assumption that all applications are impacted by manufacturing variability equally. As already discussed, this is not correct, a computation bound application will experience more variability than a memory bound one, for example. As a result, our prediction model is more reliable, since it does not rely on the variability observed using a single benchmark.

7.4 Future Work

The proposed methodologies and their evaluation results, presented in this thesis, lay the foundations for further research topics. These are the main research directions we believe this research should head to:

- Study impact of our runtime approach on emerging architectures: Currently we tested our methodology on a two socket machine. A higher number of sockets on a single NUMA node is possible and most likely newer systems will feature a higher number of NUMA sockets on a single node. Furthermore, vendors are enabling finer grain of control over the individual cores, on newer generation of processors. Currently, we were able to enable/disable cores but not control each cores power consumption individually. Instead, the user can control and measure the power consumption of the sockets as a whole. Broadwell and Haskwell family of processors already enable the user to control and monitor the power of each core. Our runtime approach could benefit from such fine control over the processor, since it could identify the less power efficient cores, when choosing candidates to disable, and achieve higher performance. Moreover, A higher count of sockets allows even more flexibility on how the power should be optimally distributed among them. However, it is of great interest to evaluate the benefits of this finer control over the processors experimentally. A challenge in for this line of research, is to deal with the increased size of the exploration space, since higher count of cores and sockets also means a higher number of configurations. The exploration space needs to be carefully designed, so that the additional cost of the runtime profiling does not result in high overheads.

- Apply our proposed methodologies on emerging generations of processors: Manufacturing variability is observed and expected to increase with newer generations of processors [103]. The results of this thesis were obtained on machines that experienced variability of 10% for power and 20% for performance. As this percentages increase so will the benefits of both our proposals. Moreover, the variability agnostic approaches also presented in this thesis, for comparison with our proposals, will perform even worse. It is of interest to conduct a thorough investigation and quantify the benefits of any variability-aware methodology, on these newer processors.

- Predicting variability in performance. Currently, our prediction models work very well for predicting the power variability among sockets. This information is essential when trying to maintain the net power consumption of a system, below a certain budget. However, alternative approaches achieve this by simply power constraining individual components and sockets. Although this may be effectively for not exceeding the power budget, not considering the manufacturing variability will result in sub-optimal scheduling of tasks (at runtime) and/or parallel workloads (at workload manager level). Our runtime approach deals with this issue by monitoring an applications performance, while trying different resource allocation on a NUMA node. A workload manager typically does not have such fine grain control or access over the individual application. A possible solution that could be adopted by a workload manager, would be to predict the performance variability, thus identifying the heterogeneity of the system per application. Predicting performance on power constrained processors is a challenging task, since all performance counters are likely to be affected under different power consumption. A PMC-based model, such as the one presented in this thesis, would need to adapt to that reality and researchers should find an approach that can deal with the varying monitoring counter values.

Appendix A

Publications

A.1 Conference Publications

- Casas, M., Moreto, M., Alvarez, L., Castillo, E., Chasapis, D., Hayes, T., Jaulmes, L., Palomar, O., Unsal, O., Cristal, A., Ayguade, E., Labarta, J., and Valero, M. (2015). *Euro-Par 2015*, chapter Runtime-Aware Architectures, pages 16–27
- Chasapis, D., Casas, M., Moretó, M., Schulz, M., Ayguadé, E., Labarta, J., and Valero, M. (2016). Runtime-guided mitigation of manufacturing variability in power-constrained multi-socket numa nodes. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 5:1–5:12
- Chasapis, D., Casas, M., Moretó, M., Schulz, M., Rountree, B., and Valero, M. (2019). Power efficient job scheduling by predicting the impact of processor manufacturing variability. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS'19 (currently under review)

A.2 Journal Publications

- Chasapis, D., Casas, M., Moretó, M., Vidal, R., Ayguadé, E., Labarta, J., and Valero, M. (2015). Parsecss: Evaluating the impact of task parallelism in the parsec benchmark suite. *ACM Trans. Archit. Code Optim.*, 12(4):41:1–41:22 (Also presented in HiPEAC'16 conference)

A.3 Workshop Publications

- Vidal, R., Casas, M., Moretó, M., Chasapis, D., Ferrer, R., Martorell, X., Ayguadé, E., Labarta, J., and Valero, M. (2015). Evaluating the impact of openmp 4.0 extensions on relevant parallel workloads. In *OpenMP: Heterogenous Execution and Data Movements - 11th International*

Workshop on OpenMP, IWOMP 2015, Aachen, Germany, October 1-2, 2015, Proceedings, pages 60–72

Bibliography

[1] (2010). *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*. Intel.

[2] Alvarez, L., Casas, M., Labarta, J., Ayguade, E., Valero, M., and Moreto, M. (2018). Runtime-guided management of stacked dram memories in task parallel programs. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, pages 218–228.

[3] Alvarez, L., Moreto, M., Casas, M., Castillo, E., Martorell, X., Labarta, J., Ayguade, E., and Valero, M. (2015). Runtime-guided management of scratchpad memories in multicore architectures. In *International Conference on Parallel Architectures and Compilation*, PACT '15, pages 379–391.

[4] Appeltauer, M., Hirschfeld, R., Haupt, M., Lincke, J., and Perscheid, M. (2009). A comparison of context-oriented programming languages. In *International Workshop on Context-Oriented Programming*, COP '09, pages 6:1–6:6.

[5] Auweter, A., Bode, A., Brehm, M., Brochard, L., Hammer, N., Huber, H., Panda, R., Thomas, F., and Wilde, T. (2014). A case study of energy aware scheduling on supermuc. In Kunkel, J. M., Ludwig, T., and Meuer, H. W., editors, *Supercomputing*, pages 394–409.

[6] Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., and Zhang, G. (2009). The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418.

[7] Ayguadé, E., Duran, A., Hoeflinger, J., Massaioli, F., and Teruel, X. (2008). An experimental evaluation of the new openmp tasking model. In Adve, V., Garzarán, M. J., and Petersen, P., editors, *Languages and Compilers for Parallel Computing*, LCPC'08, pages 63–77.

[8] Bailey, P. E., Marathe, A., Lowenthal, D. K., Rountree, B., and Schulz, M. (2015). Finding the limits of power-constrained application performance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 79:1–79:12.

[9] Banerjee, P. (1994). *Parallel Algorithms for VLSI Computer-aided Design*.

[10] Bellens, P., Perez, J. M., Badia, R. M., and Labarta, J. (2006). CellSs: a programming model for the cell be architecture. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC'06, pages 5–5.

[11] Bellosa, F. (2000). The benefits of event: Driven energy accounting in power-sensitive systems. In *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*, EW 9, pages 37–42.

[12] Bertran, R., Buyuktosunoglu, A., Gupta, M. S., Gonzalez, M., and Bose, P. (2012). Systematic energy characterization of cmp/smt processor systems via automated micro-benchmarks. In

Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-45, pages 199–211.

[13] Bertran, R., Gonzalez, M., Martorell, X., Navarro, N., and Ayguade, E. (2010). Decomposable and responsive power models for multicore processors using performance counters. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 147–158.

[14] Bienia, C. (2011). *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University.

[15] Bienia, C., Kumar, S., Singh, J. P., and Li, K. (2008). The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, pages 72–81.

[16] Bircher, W. L. and John, L. K. (2012). Complete system power estimation using processor performance events. *IEEE Transactions on Computers*, 61(4):563–577.

[17] Bircher, W. L., Valluri, M., Law, J., and John, L. K. (2005). Runtime identification of microprocessor energy saving opportunities. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, ISLPED '05, pages 275–280.

[18] Black, F. and Scholes, M. S. (1973). The Pricing of Options and Corporate Liabilities. *J. Pol. Economy, University of Chicago Press*, 81(3):637–54.

[19] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1995). Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216.

[20] Blumofe, R. D. and Leiserson, C. E. (1999). Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748.

[21] Borkar, S., Karnik, T., Narendra, S., Tschanz, J., Keshavarzi, A., and De, V. (2003). Parameter variations and impact on circuits and microarchitecture. In *Proceedings of the 40th Annual Design Automation Conference*, DAC '03, pages 338–342. ACM.

[22] Brumar, I., Casas, M., Moreto, M., Valero, M., and Sohi, G. S. (2017). Atm: Approximate task memoization in the runtime system. In *2017 IEEE International Parallel and Distributed Processing Symposium*, IPDPS'17, pages 1140–1150.

[23] BSC (2015). Programming models group. the Nanos++ parallel runtime. https://pm.bsc.es/nanox.

[24] Bucek, J., Lange, K.-D., and v. Kistowski, J. (2018). Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 41–42.

[25] Butenhof, D. R. (1997). *Programming with POSIX Threads*.

[26] Caheny, P., Alvarez, L., Derradji, S., Valero, M., Moretó, M., and Casas, M. (2018a). Reducing cache coherence traffic with a numa-aware runtime approach. *IEEE Transactions on Parallel and Distributed Systems*, 29(5):1174–1187.

[27] Caheny, P., Alvarez, L., Valero, M., Moretó, M., and Casas, M. (2018b). Runtime-assisted cache coherence deactivation in task parallel programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 35:1–35:12.

[28] Caheny, P., Casas, M., Moretó, M., Gloaguen, H., Saintes, M., Ayguadé, E., Labarta, J., and Valero, M. (2016). Reducing cache coherence traffic with hierarchical directory cache and numa-aware runtime scheduling. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, pages 275–286.

[29] Caminal, H., Caballero, D., Cebrian, J., Ferrer, R., Casas, M., Moretó, M., Martorell, X., and Valero, M. (2018). Performance and energy effects on task-based parallelized applications: User-directed versus manual vectorization. *The Journal of Supercomputing*, 74.

[30] Cao, T., He, Y., and Kondo, M. (2016). Demand-aware power management for power-constrained hpc systems. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 21–31.

[31] Casas, M., Badia, R. M., and Labarta, J. (2010). Automatic phase detection and structure extraction of MPI applications. *International Journal on High Performance Computing Applications*, 24(3):335–360.

[32] Casas, M., Moreto, M., Alvarez, L., Castillo, E., Chasapis, D., Hayes, T., Jaulmes, L., Palomar, O., Unsal, O., Cristal, A., Ayguade, E., Labarta, J., and Valero, M. (2015). *Euro-Par 2015*, chapter Runtime-Aware Architectures, pages 16–27.

[33] Castillo, E., Alvarez, L., Moreto, M., Casas, M., Vallejo, E., Bosque, J. L., Beivide, R., and Valero, M. (2018). Architectural support for task dependence management with flexible software scheduling. In *2018 IEEE International Symposium on High Performance Computer Architecture*, HPCA'18, pages 283–295.

[34] Castillo, E., Moreto, M., Casas, M., Alvarez, L., Vallejo, E., Chronaki, K., Badia, R., Bosque, J. L., Beivide, R., Ayguade, E., Labarta, J., and Valero, M. (2016). Cata: Criticality aware task acceleration for multicore processors. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 413–422.

[35] Chapman, B. (2007). The multicore programming challenge. In Xu, M., Zhan, Y., Cao, J., and Liu, Y., editors, *Advanced Parallel Processing Technologies*, APPT'07, pages 3–3.

[36] Chapman, B., Jost, G., and Pas, R. v. d. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*.

[37] Chasapis, D. (2013). Distributed Futures. Master's thesis, Université Paris-Sud.

[38] Chasapis, D., Casas, M., Moretó, M., Schulz, M., Ayguadé, E., Labarta, J., and Valero, M. (2016). Runtime-guided mitigation of manufacturing variability in power-constrained multi-socket numa nodes. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 5:1–5:12.

[39] Chasapis, D., Casas, M., Moretó, M., Schulz, M., Rountree, B., and Valero, M. (2019). Power efficient job scheduling by predicting the impact of processor manufacturing variability. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS'19.

[40] Chasapis, D., Casas, M., Moretó, M., Vidal, R., Ayguadé, E., Labarta, J., and Valero, M. (2015). Parsecss: Evaluating the impact of task parallelism in the parsec benchmark suite. *ACM Trans. Archit. Code Optim.*, 12(4):41:1–41:22.

[41] Chronaki, K., Rico, A., Casas, M., Moretó, M., Badia, R. M., Ayguadé, E., Labarta, J., and Valero, M. (2017). Task scheduling techniques for asymmetric multi-core systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(7):2074–2087.

[42] Coarfa, C., Dotsenko, Y., Mellor-Crummey, J., Cantonnet, F., El-Ghazawi, T., Mohanti, A., Yao, Y., and Chavarría-Miranda, D. (2005). An evaluation of global address space languages: Co-array fortran and unified parallel c. In *Principles and Practice of Parallel Programming*, PPoPP'05, pages 36–47.

[43] Cochran, R., Hankendi, C., Coskun, A. K., and Reda, S. (2011). Pack & cap: Adaptive dvfs and thread packing under power caps. In *IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 175–185.

[44] Cook, H., Moreto, M., Bird, S., Dao, K., Patterson, D. A., and Asanovic, K. (2013). A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. *SIGARCH Computer Architecture News*, 41(3):308–319.

[45] Davis, J. D., Rivoire, S., Goldszmidt, M., and Ardestani, E. K. (2011). Accounting for Variability in Large-Scale Cluster Power Models. In *Exascale Evaluation and Research Techniques Workshop*, EXERT'11.

[46] Demmel, J. W. (1997). *Applied Numerical Linear Algebra*.

[47] Dennard, R. H., Gaensslen, F. H., Rideout, V. L., Bassous, E., and LeBlanc, A. R. (1974). Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268.

[48] Dimic, V., Moretó, M., Casas, M., and Valero, M. (2017). Runtime-assisted shared cache insertion policies based on re-reference intervals. In *Euro-Par 2017: Parallel Processing - 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28 - September 1, 2017, Proceedings*, pages 247–259.

[49] Dongarra, J., Graybill, R., Harrod, W., Lucas, R. F., Lusk, E. L., Luszczek, P., McMahon, J., Snavely, A., Vetter, J. S., Yelick, K. A., Alam, S. R., Campbell, R. L., Carrington, L., Chen, T., Khalili, O., Meredith, J. S., and Tikir, M. M. (2008). Darpa's HPCS program- history, models, tools, languages. *Advances in Computers*, 72:1–100.

[50] Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J. (2011). OmpSs: a Proposal for Programming Heterogeneous Multi-Core architectures. *Parall. Proc. Lett.*, 21(2):173–193.

[51] Duran, A., Ferrer, R., Ayguadé, E., Badia, R., and Labarta, J. (2009). A proposal to extend the OpenMP tasking model with dependent tasks. *Internation Journal on Parallel Programming*, 37(3):292–305.

[52] El-Ghazawi, T. and Smith, L. (2006). UPC: Unified parallel c. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06.

[53] Ellsworth, D. A., Malony, A. D., Rountree, B., and Schulz, M. (2015a). Dynamic power sharing for higher job throughput. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 80:1–80:11.

[54] Ellsworth, D. A., Malony, A. D., Rountree, B., and Schulz, M. (2015b). POW: System-wide Dynamic Reallocation of Limited Power in HPC. In *Proceedings of the 2015 International Symposium on High-Performance Parallel and Distributed Computing*, HPDC'15, pages 145–148.

[55] Esmaeilzadeh, H., Blem, E., Amant, R. S., Sankaralingam, K., and Burger, D. (2013). Power challenges may end the multicore era. *Commun. ACM*, 56(2):93–102.

[56] Etinski, M., Corbalan, J., Labarta, J., and Valero, M. (2010). Utilization driven power-aware parallel job scheduling. *Computer Science - Research and Development*, 25(3):207–216.

[57] Etinski, M., Corbalan, J., Labarta, J., and Valero, M. (2011). Linear programming based parallel job scheduling for power constrained systems. In *High Performance Computing & Simulations*, HPCS'11, pages 72–80.

[58] Etinski, M., Corbalan, J., Labarta, J., and Valero, M. (2012). Parallel job scheduling for power constrained {HPC} systems. *Parallel Computing*, 38(12):615 – 630.

[59] Etsion, Y., Cabarcas, F., Rico, A., Ramírez, A., Badia, R. M., Ayguadé, E., Labarta, J., and Valero, M. (2010). Task superscalar: An out-of-order task pipeline. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 89–100.

[60] F. Romanescu, B., Ozev, S., and Sorin, D. (2018). Quantifying the impact of process variability on microprocessor behavior.

[61] Fatahalian, K., Horn, D. R., Knight, T. J., Leem, L., Houston, M., Park, J. Y., Erez, M., Ren, M., Aiken, A., Dally, W. J., and Hanrahan, P. (2006). Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06.

[62] Feitelson, D. G., Rudolph, L., and Schwiegelshohn, U. (2005). Parallel job scheduling — a status report. In Feitelson, D. G., Rudolph, L., and Schwiegelshohn, U., editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–16.

[63] Feitelson, D. G., Tsafrir, D., and Krakov, D. (2014). Experience with using the parallel workloads archive. *Journal of Parallel and Distributed Computing*, 74(10):2967 – 2982.

[64] Feng, H., Misra, V., and Rubenstein, D. (2007). Pbs: A unified priority-based scheduler. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07, pages 203–214.

[65] fluxsim (2018). Flux simulator. http://confluence.sammeth.net/display/SIM/Home.

[66] Fraternali, F., Bartolini, A., Cavazzoni, C., and Benini, L. (2018). Quantifying the impact of variability and heterogeneity on the energy efficiency for a next-generation ultra-green supercomputer. *IEEE Transactions on Parallel and Distributed Systems*, 29(7):1575–1588.

[67] Gholkar, N., Mueller, F., and Rountree, B. (2016). Power tuning hpc jobs on power-constrained systems. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, pages 179–191.

[68] Goel, B., McKee, S. A., Gioiosa, R., Singh, K., Bhadauria, M., and Cesati, M. (2010). Portable, scalable, per-core power estimation for intelligent resource management. In *Proceedings of the International Conference on Green Computing*, GREENCOMP '10, pages 135–146.

[69] Grahne, G. and Zhu, J. (2003). Efficiently using prefix-trees in mining frequent itemsets. In *FIMI*, volume 90.

[70] green500 (2017). The Green500 list. http://www.green500.org.

[71] Hamerly, G., Perelman, E., Lau, J., and Calder, B. (2005). Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction-Level Parallelism*, 7:1–28.

[72] Han, J., Pei, J., and Yin, Y. (2000). Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29(2):1–12.

[73] Harriott, L. R. (2001). Limits of lithography. *Proceedings of the IEEE*, 89(3):366–374.

[74] Heath, D., Jarrow, R., and Morton, A. (1992). Bond Pricing and the Term Structure of Interest Rates: A New Methodology for Contingent Claims Valuation. *Econometrica*, 60(1):77–105.

[75] Henning, J. L. (2006). SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17.

[76] Herbert, S., Garg, S., and Marculescu, D. (2012). Exploiting process variability in voltage/frequency control. *IEEE Trans. Very Large Scale Integr. Syst.*, 20(8):1392–1404.

[77] Herbert, S. and Marculescu, D. (2009). Variation-aware dynamic voltage/frequency scaling. In *High Performance Computer Architectures*, HPCA'09, pages 301–312.

[78] Hsu, C. and chun Feng, W. (2005). A power-aware run-time system for high-performance computing. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC'05, pages 1–1.

[79] Inadomi, Y., Patki, T., Inoue, K., Aoyagi, M., Rountree, B., Schulz, M., Lowenthal, D., Wada, Y., Fukazawa, K., Ueda, M., Kondo, M., and Miyoshi, I. (2015). Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 78:1–78:12.

[80] Intel (2011). *Intel-64 and IA-32 Architectures Software Developer's Manual*. Intel.

[81] Isaacs, K. E., Bhatele, A., Lifflander, J., Böhme, D., Gamblin, T., Schulz, M., Hamann, B., and Bremer, P.-T. (2015). Recovering logical structure from charm++ event traces. In *Proceedings of the 2015 ACM/IEEE Conference on Supercomputing*, SC'15, pages 49:1–49:12.

[82] Isci, C. and Martonosi, M. (2003). Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 93–.

[83] Jaulmes, L., Casas, M., Moretó, M., Ayguadé, E., Labarta, J., and Valero, M. (2015). Exploiting asynchrony from exact forward recovery for due in iterative solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'15, pages 1–12.

[84] Jenista, J. C., Eom, Y. h., and Demsky, B. C. (2011). OoOJava: Software out-of-order execution. *SIGPLAN Notices*, 46(8):57–68.

[85] Jette, M. A., Yoo, A. B., and Grondona, M. (2002). SLURM: Simple Linux Utility for Resource Management. In *In Lecture Notices in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*, pages 44–60.

[86] Jin, H. and Van der Wijngaart, R. F. (2006). Performance characteristics of the multi-zone nas parallel benchmarks. *Journal of Parallel and Distributed Computing*, 66(5):674–685.

[87] Joseph, R. and Martonosi, M. (2001). Run-time power estimation in high performance microprocessors. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, ISLPED '01, pages 135–140.

[88] Kale, L. V. and Krishnan, S. (1993). Charm++: A portable concurrent object oriented system based on c++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 91–108.

[89] Karlin, I., Bhatele, A., Keasler, J., Chamberlain, B. L., Cohen, J., Devito, Z., Haque, R., Laney, D., Luke, E., Wang, F., Richards, D., Schulz, M., and Still, C. H. (2013). Exploring traditional and emerging parallel programming models using a proxy application. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 919–932.

[90] Khemka, B., Friese, R., Pasricha, S., Maciejewski, A. A., Siegel, H. J., Koenig, G. A., Powers, S., Hilton, M., Rambharos, R., and Poole, S. (2015). Utility maximizing dynamic resource management in an oversubscribed energy-constrained heterogeneous computing system. *Sustainable Computing: Informatics and Systems*, 5:14 – 30.

[91] Labarta, J. and Gimenez, J. (2006). *Performance Analysis: From Art to Science*, chapter 2, pages 9–32.

[92] Leal, K. (2016). Energy efficient scheduling strategies in federated grids. *Sustainable Computing: Informatics and Systems*, 9:33 – 41.

[93] Lee, I.-T. A., Leiserson, C. E., Schardl, T. B., Sukha, J., and Zhang, Z. (2013). On-the-fly pipeline parallelism. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 140–151.

[94] Li, T. and John, L. K. (2003). Run-time modeling and estimation of operating system power consumption. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, pages 160–171.

[95] Liang, X. and Brooks, D. (2006). Mitigating the impact of process variations on processor register files and execution units. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 504–514.

[96] Lin, B., Mallik, A., Dinda, P., Memik, G., and Dick, R. (2009). User- and process-driven dynamic voltage and frequency scaling. In *The 2019 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS'09, pages 11–22.

[97] Livermore Computing (2014). The Catalyst supercomputer. http://computation.llnl.gov/computers/catalyst.

[98] Lv, Q., Josephson, W., Wang, Z., Charikar, M., and Li, K. (2006). Ferret: A toolkit for content-based similarity search of feature-rich data. *SIGOPS Operating System Review*, 40(4):317–330.

[99] Lyberis, S., Pratikakis, P., Mavroidis, I., and Nikolopoulos, D. S. (2016). Myrmics: Scalable, dependency-aware task scheduling on heterogeneous manycores. *CoRR*, abs/1606.04282.

[100] Maiterth, M., Koenig, G., Pedretti, K., Jana, S., Bates, N., Borghesi, A., Montoya, D., Bartolini, A., and Puzovic, M. (2018). Energy and power aware job scheduling and resource management: Global survey — initial analysis. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 685–693.

[101] Manivannan, M., Papaefstathiou, V., Pericas, M., and Stenstrom, P. (2016). Radar: Runtime-assisted dead region management for last-level caches. In *International Symposium on High Performance Computer Architecture*, HPCA '16, pages 644–656.

[102] Marathe, A., Bailey, P., Lowenthal, D., Rountree, B., Schulz, M., and de Supinski, B. (2015). A run-time system for power-constrained HPC applications. In *High Performance Computing*, volume 9137 of *Lecture Notices in Computer Science*, pages 394–408.

[103] Marathe, A., Zhang, Y., Blanks, G., Kumbhare, N., Abdulla, G., and Rountree, B. (2017). An empirical survey of performance and energy efficiency variation on intel processors. In *Proceedings of the 5th International Workshop on Energy Efficient Supercomputing*, E2SC'17, pages 9:1–9:8.

[104] Marculescu, D. and Talpes, E. (2005). Variability and energy awareness: a microarchitecture-level perspective. In *Proceedings. 42nd Design Automation Conference, 2005.*, pages 11–16.

[105] Moore, G. E. (2000). Readings in computer architecture. chapter Cramming More Components Onto Integrated Circuits, pages 56–59.

[106] Mu'alem, A. W. and Feitelson, D. G. (2001). Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543.

[107] Mudge, T. (2001). Power: a first-class architectural design constraint. *Computer*, 34(4):52–58.

[108] Müller, M., Charypar, D., and Gross, M. (2003). Particle-based fluid simulation for interactive applications. In *Symposium on Computer Animation*, pages 154–159.

[109] Nagle, D. (2005). MPI – the complete reference, vol. 1, the MPI core, 2nd ed., scientific and engineering computation series, by marc snir, steve otto, steven huss-lederman, david walker and jack dongarra. *Scientific Programming*, 13(1):57–63.

[110] OpenMP Architecture Review Board (2013). OpenMP application program interface version 4.0.

[111] Pan, A. and Pai, V. S. (2015). Runtime-driven shared last-level cache management for task-parallel programs. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 11:1–11:12.

[112] Papaefstathiou, V., Katevenis, M. G., Nikolopoulos, D. S., and Pnevmatikatos, D. (2013). Prefetching and cache management using task lifetimes. In *International Conference on Supercomputing*, ICS'13, pages 325–334.

[113] Patki, T., Lowenthal, D. K., Rountree, B., Schulz, M., and de Supinski, B. R. (2013). Exploring hardware overprovisioning in power-constrained, high performance computing. In *Internations Confernece on Supercomputing*, ICS'13, pages 173–182.

[114] Patki, T., Lowenthal, D. K., Sasidharan, A., Maiterth, M., Rountree, B. L., Schulz, M., and de Supinski, B. R. (2015). Practical resource management in power-constrained, high performance computing. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 121–132.

[115] Planas, J., Badia, R. M., Ayguadé, E., and Labarta, J. (2009). Hierarchical task-based programming with starss. *International Journal on High Performance Computing Applications*, 23(3):284–299.

[116] Podobas, A. and Brorsson, M. (2010). A comparison of some recent task-based parallel programming models. In *Multiprog*.

[117] Quinlan, S. and Dorward, S. (2002). Awarded best paper! - venti: A new approach to archival data storage. In *FAST*.

[118] Rajovic, N., Carpenter, P., Gelado, I., Puzovic, N., Ramirez, A., and Valero, M. (2013). Supercomputing with commodity CPUs: Are mobile SoCs ready for HPC? In *Proceedings of the 2013 ACM/IEEE Conference on Supercomputing*, SC'13, pages 1–12.

[119] Ravichandran, K., Lee, S., and Pande, S. (2011). Work stealing for multi-core hpc clusters. In *Euro-Par*, pages 205–217.

[120] Reinders, J. (2007). *Intel Threading Building Blocks*. First edition.

[121] Rimal, B. P., Choi, E., and Lumb, I. (2009). A taxonomy and survey of cloud computing systems. In *Proceedings of the 2009 Fifth International Joint Conference on INC, IMS and IDC*, NCM '09, pages 44–51.

[122] Rountree, B., Ahn, D., de Supinski, B., Lowenthal, D., and Schulz, M. (2012a). Beyond DVFS: A first look at performance under a hardware-enforced power bound. In *IPDPS Workshops PhD Forum*, pages 947–953.

[123] Rountree, B., Ahn, D. H., de Supinski, B. R., Lowenthal, D. K., and Schulz, M. (2012b). Beyond dvfs: A first look at performance under a hardware-enforced power bound. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12, pages 947–953.

[124] Rountree, B., Lownenthal, D. K., de Supinski, B. R., Schulz, M., Freeh, V. W., and Bletsch, T. (2009). Adagio: Making dvs practical for complex hpc applications. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 460–469.

[125] S. Ashby and, P. B., Chen, J., Colella, P., Collins, B., Crawford, D., Dongarra, J., Kothe, D., Lusk, R., Messina, P., Mezzacappa, T., Moin, P., Norman, M., Rosner, R., Sarkar, V., Siegel, A., Streitz, F., White, A., and Wright, M. (2010). The opportunities and challenges of exascale computing. DOE Technical Report.

[126] Samaan, S. (2004). The impact of device parameter variations on the frequency and performance of VLSI chips. In *International Conference On Computer Aided Design*, ICCAD'04, pages 343–346.

[127] Sánchez Barrera, I., Moretó, M., Ayguadé, E., Labarta, J., Valero, M., and Casas, M. (2018). Reducing data movement on large shared memory systems by exploiting computation dependencies. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, pages 207–217.

[128] Sarangi, S. R., Greskamp, B., Teodorescu, R., Nakano, J., Tiwari, A., and Torrellas, J. (2008). Varius: A model of process variation and resulting timing errors for microarchitects. *IEEE Transactions on Semiconductor Manufacturing*, 21(1):3–13.

[129] Sarood, O., Langer, A., Gupta, A., and Kale, L. (2014). Maximizing throughput of overprovisioned hpc data centers under a strict power budget. In *SC*, pages 807–818.

[130] Schling, B. (2011). *The Boost C++ Libraries*.

[131] Sherwood, T., Perelman, E., and Calder, B. (2001). Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, PACT '01, pages 3–14, Washington, DC, USA. IEEE Computer Society.

[132] Shoga, K., Rountree, B., and Schulz, M. (2014). Whitelisting MSRs with msr-safe.

[133] Shoukourian, H. (2015). *Adviser for Energy Consumption Management: Green Energy Conservation*. Dissertation, Technische Universität München.

[134] Sifakis, E., Neverov, I., and Fedkiw, R. (2005). Automatic determination of facial muscle activations from sparse motion capture marker data. *ACM Transactions on Graphics*, 24(3):417–425.

[135] Singh, K., Bhadauria, M., and McKee, S. A. (2009). Real time power estimation and thread scheduling via performance counters. *SIGARCH Computer Architectures News*, 37(2):46–55.

[136] slurmsim (2018). Slurm simulator. https://github.com/BSC-RM/slurm_simulator.

[137] Song, S., Ge, R., Feng, X., and Cameron, K. W. (2009). Energy profiling and analysis of the hpc challenge benchmarks. *Internations Journal on High Performance Computing Applications*, 23(3):265–276.

[138] Subcomittee, A. (2014). Top ten exascale research challenges. Technical report, US Department of Energy.

[139] Symeonidou, C., Pratikakis, P., Bilas, A., and Nikolopoulos, D. S. (2013). DRASync: Distributed region-based memory allocation and synchronization. In *EuroMPI*, pages 49–54.

[140] Tan, X., Bosch, J., Vidal, M., Álvarez, C., Jiménez-González, D., Ayguadé, E., and Valero, M. (2017). General purpose task-dependence management hardware for task-based dataflow programming models. In *2017 IEEE International Parallel and Distributed Processing Symposium*, IPDPS'17, pages 244–253.

[141] Teodorescu, R. and Torrellas, J. (2008). Variation-aware application scheduling and power management for chip multiprocessors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 363–374.

[142] TOP500 (2018). TOP500 Supercomputer Site. http://www.top500.org.

[143] Totoni, E., Langer, A., Torrellas, J., and V.Kale, L. (2014a). Scheduling for hpc systems with process variation heterogeneity. Technical report, University of Illinois at Urbana-Champaign.

[144] Totoni, E., Torrellas, J., and Kale, L. V. (2014b). Using an adaptive hpc runtime system to reconfigure the cache hierarchy. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'14, pages 1047–1058.

[145] Tschanz, J., Kao, J., Narendra, S., Nair, R., Antoniadis, D., Chandrakasan, A., and De, V. (2002). Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage. *Solid-State Circuits, IEEE Journal of*, 37(11):1396–1402.

[146] Tzenakis, G., Papatriantafyllou, A., Kesapides, J., Pratikakis, P., Vandierendonck, H., and Nikolopoulos, D. S. (2012). BDDT: Block-level dynamic dependence analysis for deterministic task-based parallelism. *SIGPLAN Notices*, 47(8):301–302.

[147] Valero, M., Moreto, M., Casas, M., Ayguade, E., and Labarta, J. (2014). Runtime-aware architectures: A first approach. *Supercomputing frontiers and innovations*, 1(1).

[148] Vandierendonck, H., Chronaki, K., and Nikolopoulos, D. S. (2013). Deterministic scale-free pipeline parallelism with hyperqueues. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 32:1–32:12.

[149] Vandierendonck, H., Pratikakis, P., and Nikolopoulos, D. S. (2011). Parallel programming of general-purpose programs using task-based programming models. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar'11, pages 13–13.

[150] Vassiliadis, V., Parasyris, K., Chalios, C., Antonopoulos, C. D., Lalis, S., Bellas, N., Vandierendonck, H., and Nikolopoulos, D. S. (2015). A programming model and runtime system for significance-aware energy-efficient computing. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 275–276.

[151] Vidal, R., Casas, M., Moretó, M., Chasapis, D., Ferrer, R., Martorell, X., Ayguadé, E., Labarta, J., and Valero, M. (2015). Evaluating the impact of openmp 4.0 extensions on relevant parallel workloads. In *OpenMP: Heterogenous Execution and Data Movements - 11th International Workshop on OpenMP, IWOMP 2015, Aachen, Germany, October 1-2, 2015, Proceedings*, pages 60–72.

[152] Wall, D. W. (1991). Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 176–188.

[153] Zheng, G., Bhatelé, A., Meneses, E., and Kalé, L. V. (2011). Periodic hierarchical load balancing for large supercomputers. *International Journal on High Performance Computing Applications*, 25(4):371–385.

[154] Zuckerman, S., Suetterlein, J., Knauerhase, R., and Gao, G. R. (2011). Using a "codelet" program execution model for exascale machines: Position paper. In *International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 64–69.

List of Figures

2.1 Ferret implementation in OpenMP 4.0/OmpSs 15

4.1 Task-graph of blackscholes application. No dependencies exit between tasks, only barrier synchronization between iterations. 39
4.2 Parallel execution of Pthreads and task-based versions of bodytrack on an 8-core machine and native input size. Different parallel regions correspond to different colors. White gaps in the figure, represent idle time. 40
4.3 Task-graph of bodytrack application. Edges show task data dependencies. 41
4.4 Task-graph of canneal application. Only barrier synchronization among tasks. 43
4.5 Task-graph of dedup application. Data dependency edges force the correct order of writting the data chunks to the output. 44
4.6 Parallel execution of the task-based version of dedup on an 8-core machine and native input size. Different task types correspond to different colors. 45
4.7 Task-graph of ferret showing the pipelined execution model. Edges show data dependencies among different tasks. 47
4.8 Task-graph of fluidanimate application. Edges represent data dependencies among tasks 49
4.9 Task-graph of freqmine application. Edges represent data dependencies among tasks. 50
4.10 Task-graph of streamcluster application. Edges represent data dependencies among tasks. 51
4.11 Task-graph of swaptions application. 52
4.12 Comparison of lines of code between our task-based implementations and the original Pthreads or OpenMP versions. 53
4.13 Comparison of scalability between the task-based implementations and the original (Pthreads/OpenMP) versions. 55
4.14 Comparison of scalability between the task-based implementations and the original (Pthreads/OpenMP) versions. 56
4.15 Comparison of scalability between the task-based implementations and the original (Pthreads/OpenMP) versions. 57
4.16 Comparison of scalability between the task-based implementations and the original (Pthreads/OpenMP) versions. 58

4.17 Speedup comparison for Pthreads and OmpSs with the same granularity, as well as our optimized OmpSs version, when run on 16 cores. Results are normalized to the sequential version of the original code. 59
4.18 Runtime breakdowns when running on an 8-core configuration. 62

5.1 Performance obtained when the `freqmine` application is run on 64 different 12-core Intel Xeon E5-2695v2 sockets under different power budgets. 66
5.2 Executions of `swaptions` under 40 W power capping. 68
5.3 Executions of `blackscholes` near a synchronization point under 40 W. 69
5.4 Comparison between the even and the best configuration observed by application profiling. The speedup is computed over the execution of the even resource distribution for 80 W. 71
5.5 Comparison of best configuration found by exhaustive and scoped online analyses for different *monitoring window* size, when running under a 80W power constraint. The size of the monitoring window can influence the precision of the analysis. 76
5.6 Performance benefits using monitoring windows of 1 second under 80 W power limit. 78
5.7 Energy consumption reduction using 1 second monitoring windows under a 80 W power bound. 80

6.1 Total power consumption trace for Conservative, Variability agnostic and Variability aware scheduling policies, when running the same workload. Considering socket variability maximizes performance and meets the power budget. 86
6.2 Power, active cores and component activity ratio traces when running on 12 cores of a single socket. Architectural components shown are the fetch unit (FE) and L1 cache. The activity ratios are the number of retired micro operations per unhalted cycle, relevant to each architectural component. In the case of cores, activity ratio is the number of active cores. For memory the activity ratio is measured as the number of references (for caches) or LLC misses (for main memory) per cycle. 89
6.3 Power, active cores and component activity ratio traces of MPI, multinode applications when running on 12 cores per socket. Results show only activity on one socket, but the other sockets demonstrate similar behavior. Architectural components shown are the fetch unit (FE) and L1 cache. The activity ratios are the number of retired micro operations per unhalted cycle, relevant to each architectural component. In the case of cores, activity ratio is the number of active cores. For memory the activity ratio is measured as the number of references (for caches) or LLC misses (for main memory) per cycle. For multi-node applications PMC data is collected for all the processes and individual predictions are made for each socket. 90

6.4 Comparison between the variability ratios over all 256 sockets as observed for *cholesky* and *sparseLU* benchmarks. Variability ratio here is the fraction the Power consumption of the benchmark on a given socket, divided by the power consumption of the same benchmark on a reference socket. The cpu bound *cholesky* detects variability more precisely than *sparseLU*. 93
6.5 Actual and predicted power consumption, using the PMC-based (Optimized PMC) model, for *blackscholes* under three distinct sockets. Same CPU chip model is mounted on all three sockets, utilizing all 12 available cores. 94
6.6 Framework for the *SLURM+PRVP* and *SLURM+PMCVP*. The upper right box shows the steps for training the PMC model, while Power Ratio training stage is shown in the box below. 97
6.7 Comparison of average power prediction error for all models over all sockets. The error bars show the standard error deviation across all sockets for the corresponding application. 98
6.8 Power and Average turnaround time reduction over SLURM extended. Results SLURM+PE and SLURM+PEVA values range as shown by the corresponding lines, depending on the estimation provided. 100
6.9 Energy consumption Reduction over *SLURM extended*, under different budgets and traffic scenarios. SLURM+PE, which is variability agnostic fail to save any energy, while SLURM+VTVP performs best. 100

List of Tables

3.1 Benchmark training set for PMC-based power prediction model. 30

3.2 NAS Multi-Zone benchmarks are multinode applications that use both MPI and OpenMP to express parallelism. MPI is used for inter-node communication and OpenMP is used for intra-node parallelizations of loops. 31

4.1 PARSEC Benchmark Suite 38

4.2 PARSEC parallelization model and properties characterization. 63

5.1 Optimal configurations per application and power bound in terms or Watts and active cores per socket 72

6.1 Architectural component activity ratios formulas for Intel Broadwell Architecture, inferred from Intel's 64 and IA-32 Architectures Software Developer's Manual [1] . . 88

6.2 Benchmark training set for PMC-based power prediction model. 92

List of Abbreviations

ALU Arithmetic Logic Unit.
AMC Assymetric Multi-Core.

BPU Branch Prediction Unit.

CPU Central Processing Unit.
CUDA Compute Unified Device Architecture.

DDR3 Double Data Rate.
DIMM Dual Inline Memory Module.
DOE Department of Energy.
DVFS Dynamic Voltage and Frequency Scaling.

FE Fetch Unit.
FPU Floating Point Unit.

GPU Graphics Processing Unit.

HPC High Performance Computing.

ILP Instruction Level Parallelism.
IVP Ideal Variability Prediction.

L_{eff} Effective Gate Length.
LLC Last Level Cache.
LLNL Lawrence Livermore National Laboratory.

MPI Message Passing Interface.
MSR Model-Specific Register.

NAS-MZ NASA Multi-Zone.
NASA National Aeronautics and Space Agency.
NUMA Non-Uniform Memory Access.

PE Power Estimation.
PEVA Power Estimation Variability Aware.
PMC Performance Monitoring Counters.
PMCVP Program Monitoring Counter-based Variability Prediction.
POXIS Portable Operating System Interface, Unix.
PR Power Ratio.
PR-CB Power Ratio - Computational Bound.
PR-MB Power Ratio - Memory Bound.
PRVP Power Ratio Variability Prediction.
Pthreads Posix Threads.

RAPL Running Average Power Limit.

SPMD Single Program Multiple Data.

TDG Task Dependency Graph.
TDP Thermal Design Power.
TLP Thread Level Parallelism.

UMA Uniform Memory Access.

V_{th} Voltage Threshold.
VT Variability Trained.
VTVP Variability Trained Variability Prediction.