PhD-FSTC-2014-21
The Faculty of Sciences, Technology and Communication

DISSERTATION

Defense held on 11/07/2014 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Frédéric PINEL
Born on 6/11/1968 in Maisons-Laffitte (France)

ENERGY-PERFORMANCE OPTIMIZATION FOR THE CLOUD

Designing massively parallel applications for Arrays of Wimpy Nodes inspired by the Savant syndrome

Dissertation defense committee

Dr Pascal Bouvry, dissertation supervisor
Professor, Université du Luxembourg

Dr Steffen Rothkugel, chairman
Associate Professor, Université du Luxembourg

Dr Jens Gustedt, vice chairman
Researcher, INRIA

Dr Laurent Lefèvre
Researcher, INRIA

Dr Bernabé Dorronsoro
Researcher, Université du Luxembourg

Abstract

The persistent trend in hosting Internet services towards the Cloud motivates the need for energy-efficiency. The analysis of the total cost of operation of a Cloud data center points to new directions for reducing the energy-related costs. The system architecture called Array of Wimpy Nodes (AWN) is such a direction, and sets the context for this work. The explosion of mobile computing renews the interest in AWN, by supplying low-power and low-cost hardware components that deliver not so low-performance. This thesis explores how to extract performance from the parallel architecture of AWN, by searching for massively parallel application designs. The exploration of parallel application design is conducted in the context of a complex combinatorial optimization problem, which does not involve "big data". The evaluation of current parallel designs suggests that new algorithms, rather than new algorithms' implementations, can deliver the necessary performance on AWN. Moreover, we show how statistical methods can guide the search for the new algorithms. Finally, we combine the previous findings to propose a method for the automatic generation of Map-Reduce programs, inspired by the Savant syndrome.

Acknowledgments

This thesis is the result of a multidisciplinary approach, which was made possible thanks to the contribution of many people. Fairly acknowledging the role of each is almost impossible, it must nevertheless be attempted.

The path followed in the course of this thesis was carefully reviewed, and required much guidance, from my supervisor Dr. Pascal Bouvry. Several key decisions were the result of his critical view. Pascal Bouvry also gave me the opportunity to undertake this work, for which I am very grateful. In general, the University of Luxembourg proved to be a very open and flexible organization, which made this work pleasant.

Dr. Dorronsoro was always available to share his expertise in metaheuristics and combinatorial optimization, which helped significantly. He also served as a mentor, in all aspects of the research activity. Critically, he was always willing to discuss ideas, which certainly took a great deal of his time, but helped solve difficult points.

Dr. Samee Khan, although remotely located from the University of Luxembourg, was able to share some of his broad knowledge, on topics such as distributed computing and statistical analysis. I also hope I was able to learn from his rigorous scientific approach.

Dr. Pascal Bouvry's team provided a great working environment. All team members were always available to share their knowledge and give advice, in a warm atmosphere. Grégoire Danoy, Johnatan Pecero and Sebastien Varrette were particularly helpful. The HPC cluster team was very helpful for using the University's clusters [1], but also provided general advice and support on distributed computing. Several students from the University of Luxembourg's Master in Computer Science also contributed to this work, as did V. Delplace during his masters's thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Green computing has become a broad term, that is employed in many different contexts, from electrical consumption costs to greenhouse gas emission. It was even considered an ethical issue [6]. In this work, we approach energy-efficiency from a cost perspective. The cost perspective is presented in Chapter 2. However, the cost benefits should not come with a reduced performance. The title of this thesis reflects this context.

Reducing costs of operation of Cloud data centers will become increasingly important. Energy-efficiency in the data center is a growing concern due to the increased capacity of data centers, a consequence of the widespread use of web applications, and the adoption of Cloud computing. The world wide web is still a defining factor in the IT industry, and more and more businesses adopt an on-line strategy. Cloud computing is largely targeted at web applications. Many successful web businesses are relying on Cloud computing, despite their capability, both financial and technical, to operate their own infrastructure, which indicates the business value of Cloud computing. Mobile computing is another trend that increases the reliance on web and Cloud computing. While increasingly powerful, the mobile devices play the role of internet client devices.

There are many past efforts to reduce the energy costs in the data centers, Chapter 2 reviews these efforts along three directions: hardware, power-aware resource management and energy-efficient applications. One frequent approach is server load consolidation, where careful resource management aims to power off unnecessary machines. An alternative proposal is the use of lower power components (CPU, memory, storage) to deliver an equivalent service, in terms of feature and performance, at a reduced energy cost [7]. Our work is set in the low-power alternative, it explores it's implications for application providers.

1.2 Approach

We have stated that our green objective for the data centers of Cloud providers is cost, and by cost we mean Total Cost of Operation (TCO). Chapter 2 will show that power consumption for the operation of a data center is not a major part of the TCO. Hardware purchase costs and critical power (the maximum power to provision to IT equipment [8]) are, combined they are estimated to represent 85% of the TCO. Many past efforts to reduce electrical costs do not address these cost components. However, low-power hardware can reduce the maximum power to be provisioned, and the recent explosion in mobile computing market is contributing increasingly powerful, but lower power hardware, at a low cost.

The question is therefore, can such evolutions in hardware play a role in the Cloud data centers, and can Cloud applications take advantage of this low-power hardware, which we will refer to as Array of Wimpy Nodes (AWN)? The question is analogous to the second choice in the following quote: "The microprocessor industry offers a choice of two strategies for efficiency: (1) start with a big, high-performance core and improve efficiency or (2) start with a small, low-power core and improve performance. We compare these two strategies and the data favors the latter for Microsoft Bing" [9].

The hypothesis followed in this thesis is that to successfully exploit the low-power hardware made affordable via the mobile evolution, new algorithms are necessary. New implementations of existing algorithms cannot deliver the required performance, as discussed in Chapter 3. The new algorithms are parallel algorithms, because of the intrinsic nature of the AWN.

Suggesting entirely new applications may seem unpractical. However, the changes that mobile, web, and Cloud computing are bringing in fundamental aspects of computer operation, set a precedent. We list a few signs of the current and expected changes. One example is the emergence of the distributed key/value "databases". In the traditional IT landscape, databases are monolithic applications running on large servers in privately owned data centers. Relational databases were seen as an unquestionable component in any distributed application. Many web applications use a seemingly weaker model, that exploits the largely distributed architecture of Clouds, in order to service multiple concurrent requests across large datasets [8]. A consequence is that traditional approaches to energy-efficiency, such as server consolidation, do not apply, because the load is distributed deliberately. Cloud providers offer distributed key/value stores as a service. Other examples can be seen in the operating system (OS) field. The OS field was also considered mature and stable until the emergence of "boot-to-browser", lightweight Gnu/Linux based OS, such as Firefox OS and Chrome OS. A similar change is predicted on the server-side. We have already witnessed the emergence of virtualization, an enabler of the initial Cloud offering. Re-

cent implications of virtualization point to lightweight, and application specific, VMs built on top of existing hypervisors (Xen), without a traditional OS. Within the OS, traditional features such as multi-user support, and file systems are considered immutable. However, web applications weaken the need for multi-user support, as all user interactions occur via a web server, running as a single user. Application security can be handled by other application-level services. Web applications expose storage services that are not tied to the file concept. The hypermedia, including hypertext, are not file based [10, 11]. Storage can now be implemented via Cloud based object storage services, which challenge the need for files. Questioning files for data storage has consequences, for example: Unix-flavored OS often use the file as the key abstraction for the services it provides. Beyond files and multi-user support, features considered to belong to the OS could migrate to the application layer, available as a Cloud software service.

The design of new algorithms suitable to the AWN is explored in the Chapters 4 to 6. The presented parallel designs are successive points on a specific scale. The scale indicates the reliance on human intervention. Chapter 4 presents manually designed parallel algorithms. Chapter 5 introduces the use of statistical tools to assist in the algorithm's design. Finally, Chapter 6 is a proposal for the automatic generation of a parallel program. The approach is applied to a use case, a specific application that we wish to parallelize, for execution on an AWN. The use case is an combinatorial optimization problem from the scheduling domain, it is described in Chapter 3. The application chosen fits our approach well, this may not be the case for other applications.

The problem of energy-efficiency in Cloud data centers is addressed through parallel program design, via the introduction of the AWN. Specifically, it is the process of parallel algorithm design that is addressed in this thesis. The design of parallel algorithm is not motivated by big data, or intensive computation, but should be applicable for smaller problems too, if the underlying computing platform evolves to the AWN. The parallelization of small problems was mentioned by D. Hillis, when discussing the Connection Machine [12]: "Now that doesn't mean that you couldn't invent a problem that can't be done by a lot of slow processors working together. It turns out that that such problems are surprisingly hard to invent. [...] The only problems that don't run well on a parallel machine are problems that just have a small fixed amount of data. For example, if you simulate the motion of nine planets of the solar system, you can represent that with 18 numbers. It's hard to see how you would use a whole lot of parallelism with only nine or even 100 planets. I'm not saying it's impossible. I'm just saying it's hard to see (which means it is a good problem for someone to work on)." The parallelization of small problems on slow processors is addressed in Chapter 6.

1.3 Summary of Contributions

The contributions proposed in this thesis are summarized below:

- I selected the AWN concept as a promising approach to energy-efficient Cloud computing, based on the analysis of the TCO of a Cloud data center, the design trends in web and Cloud applications, and the emergence of mobile computing.

- AWN was evaluated through the experimentation and simulation of multiple applications, on several AWN platforms. The applications experimented belong to different categories [13]: dense and sparse computations, structured grids, Map-Reduce, and bioinformatics. In addition to the evaluation of existing applications, I proposed and evaluated two new data-parallel designs of known algorithms, that solve a combinatorial optimization problem from the scheduling domain. Moreover, code-parallelism was simulated to study the effects of software pipelining, and contention to shared resource, in the context of AWN. The compared platforms were an ARM microserver, the GPU, and multicore CPUs.

- From the results of the previous evaluation, I inferred that changing the implementation of an algorithm could not deliver the necessary performance on an AWN: new parallel algorithms are necessary. In the context of combinatorial optimization, I modified two known algorithms (and not only their implementation) to improve the scalability, and yet provide the same capability. The proposed algorithms were validated on a GPU and a 40-core NUMA machine, and showed this approach to be promising.

- Achieving parallelism by modifying the algorithm makes the common trial-and-error approach to design unpractical, due to the large number of possible choices. I applied a technique from statistical model analysis, Sensitivity Analysis (SA), to guide the modification to an algorithm. The approach was experimented on the above mentioned combinatorial optimization problem from the scheduling domain. The results derived from the statistical analysis show that this approach can offer valuable insight for modifying an algorithm, without relying on human intuition.

- The insight provided by a statistical analysis can enable automatic program modification. I proposed and evaluated a method to automatically generate a Map-Reduce version of a given algorithm. The method is inspired by the Savant Syndrome, a medical condition that is found in people suffering from autistic spectrum disorder (but not only). Savants can perform seemingly sequential tasks very quickly in

an AWN-like environment: their mind. The method uses statistical analysis to implement the Savants' trait of pattern recognition, with statistical machine learning. The approach was experimentally validated on the same optimization problem from the scheduling domain. The parallel algorithms generated are well suited to the massively parallel AWN, even for small problems.

Chapter 2

Green Arrays of Wimpy Nodes

We open this chapter with a review on how power and energy impact the TCO of a cloud data center, and what are the current means to improve its energy-efficiency (Section 2.1). The findings from this review, and recent events in computing coincide to make the AWN concept a worthwhile approach to green computing. These events are the success of cloud computing to drive Internet business, and the explosion of mobile computing, commoditizing low-power components such as the ARM processors. We therefore present AWN next, in Section 2.2. Finally, we report our experimental results on the performance of a recent AWN implementation, in Section 2.3.

2.1 Green Computing

According to [14], energy-efficiency in a distributed system can be enhanced at three different levels: (a) energy-efficient applications, (b) power-aware resource management, and (c) efficiency of hardware. In this section, we argue that level (c), and indirectly (a), offer the best opportunities for energy-efficiency in cloud data centers. Energy-efficient applications are not normally perceived as a practical leverage for cloud providers, because they cannot usually control the applications submitted to their service. However, the shift in cloud offering (towards platform or software as-a-service) makes this possible: the application runtime is becoming under the cloud provider's control. Also, the cloud application designer has an indirect incentive to provide energy-efficient applications, through the price of the cloud service, which reflects the costs to the provider and is influenced by energy costs. Energy-efficient applications coupled with the emergence of energy-efficient hardware provide an incentive for cloud providers to offer services from an energy-efficient infrastructure.

	CPU	Chipset	Memory	I/O	Disk	Total
idle	38.4	19.9	28.1	32.9	21.6	141
SPEC CPU 2000 gcc	162	20.0	34.2	32.9	21.8	271
SPEC CPU 2000 mcf	167	20.0	39.6	32.9	21.9	281
SPEC CPU 2000 vortex	175	17.3	35.0	32.9	21.9	282
SPEC CPU 2000 art	159	18.7	35.8	33.5	21.9	269
SPEC CPU 2000 lucas	135	19.5	46.4	33.5	22.1	257
SPEC CPU 2000 mesa	165	16.8	33.9	33.0	21.8	271
SPEC CPU 2000 mgrid	146	19.0	45.1	32.9	22.1	265
SPEC CPU 2000 wupwise	167	18.8	45.2	33.5	22.1	287
TPC-C like dbt-2	48.3	19.8	29.0	33.2	21.6	152
SPECjbb	112	18.7	37.8	32.9	21.9	223
DiskLoad	123	19.9	42.5	35.2	22.2	243

Table 2.1: Subsystem Average Power (Watts) [4]

2.1.1 Hardware Efficiency

This section presents the hardware energy-efficiency issues in cloud data centers. First, we define the static, dynamic ranges, and the critical power. The power that a system consumes comprises, in general, two parts [15]:

- A static part that depends on system size and component type (computing, data storage and network elements); this consumption is incurred by leakage currents present in any powered system. The static part is quantified in a range, *the static range*, in %, which indicates how much of a component's power is static.
- A dynamic part that results from the usage of computing, storage, and network resources; caused by system activity and changes in clock rates. Similar to the static range, *the dynamic range*, in %, represents how much of a component's power is dynamic.

Critical power is the peak power level that can be provisioned to IT equipment (servers, networking) [8].

Direct server power

Historically, CPU were the most power consuming device in a server. CPU power is composed of static and dynamic power terms [14]. A modern CPU may consume up to $O(100)W$ [16]. Table 2.1, reproduced from [4], gives an example of power breakdown in a server, where the CPU subsystem is composed of four Pentium IV Xeon processors. The SPEC CPU 2000 are CPU-intensive, which explains the dominance of the CPU power in the results, up to 60% of the total power. Recent servers report a 42.3% energy

	Static Range	Dynamic Range	Average Load	Peak Power
CPU	30%	70%	30%	$O(100)W$

Table 2.2: CPU Static & Dynamic Power

share at 80% utilization [8]. The measurements highlight the influence of Dynamic Voltage and Frequency Scaling (DVFS), that enables the CPU's power to adjust to the workload. Also, current CPU are multi-core, where each core operates at a lower frequency than previous single core CPU. Finally, the CPU power consumption is found to correlate with the load, often linearly, although non-linear correlation was observed in database servers under slightly different metric [17]. Studies show average loads are low in current data centers, either 36% [14], between 10 to 50% [18] or slightly above 30% [19]. The cloud applications are more web oriented and rarely compute intensive or High Performance Computing (HPC) [20], therefore these observed loads apply well to the cloud. For example, a Google cluster of 20,000+ servers running typical data center loads including online services showed a 10-50% utilization from January until March 2013 [8]. These factors, DVFS, multi-core and low average loads, combine to reduce the static range to about 30% of the peak CPU [19, 14]. This means that although the CPU can consume a large share of the total power, in practice it is not the highest power consumer anymore [18]. The current average CPU power profile is summarized in Table 2.2.

The improved power consumption of CPU shift the focus to the power consumption of other server components, especially at lower utilization rates. This differs from the breakdown of [8], which show CPU energy share of 42.0% and DRAM of 12% using late 2012 generation servers, because this breakdown is presented at 80% average utilization.

Memory is now among the highest power consumers in a server [16]. Memory power usage depends on overall load and on the activity of the memory bank [4, 16]. But, this activity is not necessarily related to the CPU activity, but also to I/O under Direct Memory Access (DMA). The introduction of DDR3 with better energy management, the drop of DRAM from 1.8V to 1.5V has recently reduced the energy of memory [8]. Reducing the power consumption of memory systems is possible by observing the mismatch between the current DDR3 DRAM bandwidth and the memory usage patterns of web applications or distributed in-memory caching [21]. These applications typically do not require the high bandwidth offered by DDR3 because (a) network bandwidth is a limiting factor, and even then, (b) their needs are more oriented towards capacity and latency (2–6% bandwidth utilization reported by Microsoft Bing and Cosmos). Mobile Low Power DDR (LPDDR)2 memory offers a lower peak bandwidth than DDR3, with

the same capacity and similar latency: 4-5× energy reduction in exchange for 2× lower peak bandwidth. LPDDR2 are also more energy-proportional [22]. LPDDR2 achieve this by foregoing some circuits responsible for the static power of DRAM, but complicate the design for greater capacity memory systems. A proposed memory system design to meet capacity of servers provides a ×4–5 power reduction, often less than the DDR3 idle power. Besides the fact that this design proposal is not available now, reducing the memory power reveals the high power consumption (and large static range) of cache memory, and therefore a good cache hit ratio limits the effectiveness of LPDDR2. Previous work had recognized the high power of cache memory, about 25% of CPU power [23].

Other components, such PCI slots and disk, can also consume more power than the CPU [24]. Moreover, the majority of this power is static. The dynamic range is reported less then 50% for DRAM, 25% for disk drives (non Solid State Disk (SSD)) and 15% for networking switches [18, 25]. Therefore, overall, for non HPC or processor intensive loads, the server's dynamic power range is reported to be as low as 30% of the peak power, so when idle, a server consumes as much as 70% peak power [14, 18, 8].

Direct network power

The previous discussion on the contrast between dynamic ranges for CPU compared to that of a complete server also applies to data center networking. The communication infrastructure is reported to use as much as 30% of the total data center energy [26]. The network's dynamic range is much less than the servers (which we saw is much less than the CPU's). In [27], the authors notice that at full utilization (servers and network), the network consumes only 12% of the total power, whereas at 15% overall utilization, the network consumes 50% of the total power. They also claim that the network's dynamic range can be improved because modern plesiochronous links already increase the dynamic range in their performance and power. However, this requires reconfiguration of these links to adapt to the traffic intensity. Similarly, the dynamic range in data center networks is reported as low as 3 to 15% [26] of the total power. Also, an Ethernet switch's dynamic range was reported as low as 2% [25]. Moreover, only a part of the links are subject to DVS, as only the last hop (directly connected to the servers) switches are amenable to DVS. Consequently, the network dynamic range is estimated at 10%. A more ambitious, longer term, approach is to introduce passive optical network (PON) in the infrastructure [28]. Greentouch [29] is an on-going initiative by 30+ organizations to reduce the network energy consumption by 3 orders of magnitude. Their roadmap mentions the design hybrid low-power electronic/photonic devices, and addresses all network areas, from homes to data centers [30].

	Static Range	Dynamic Range
Server	70%	30%
Network	90%	10%
Infrastructure (distribution & cooling)	0%	100%

Table 2.3: Dynamic & Static Ranges of Data Center Components

Indirect power: cooling and power distribution

Extending our view from the servers and the network reveals another aspect to data center power: the necessary power distribution and cooling. Power distribution and cooling are reported to drive the majority of the costs in a cloud data center, but that includes capital expenses and operational expenses [8, 31]. Overall, in most data centers, 40-50% of consumed energy never reaches the computing resources: it is consumed by the cooling facilities or dissipated in conversions within the Uninterruptable Power Supply (UPS) and Power Distribution Unit (PDU) systems [14, 32, 31, 33]. This proportion matches the average Power Usage Effectiveness (PUE) of 1.7–1.9 found in a 2012 survey of 1,100 data centers [8].

The distribution of power to the computing components involves several steps, from the main supply to the rack, a process that is considered 90% efficient [8, 32, 31]. The efficiency depends on the load, typically performing worse under reduced load. For example, power supply units are 70% inefficient under 20% loads [19]. Moreover, the majority of the delivered power is dissipated as heat. The number of transistors integrated into todays Intel Itanium 2 processor reaches nearly 1 billion. If this rate continues, analysts predict that the heat (per cm2) produced by future processors would exceed that of the surface of the Sun, resulting in poor system performance [14]. For each computing watt of power (consumed by servers and networking), an additional 0.5–1W is required for the cooling system [14, 31]. In general, cooling is reported to represent 30-40% of the total data center power consumption [31, 8, 34].

The dynamic range for power distribution and cooling is assumed to be 100% [25, 8]. This means that when all critical equipment is off, the power distribution and cooling consume nothing.

The dynamic and static ranges for the different components of a data center, as presented in Section 2.1.1 to Section 2.1.1, are summarized in Table 2.3.

	% of TCO
Computing equipment	45%
Infrastructure	40%
Power consumption	15%

Table 2.4: TCO Decomposition

Indirect power: data center TCO

Extending our view again from the computing equipment, cooling and power distribution operating costs, reveals the capital expenses necessary to build the data center. A data center TCO is split between Capex and Opex [8]. Opex are the recurring costs related to operating the data center: power consumption, maintenance, personnel, etc. The previous sections discussed power consumption, a part of Opex. Capex are the costs required upfront, which are depreciated over time. Capex examples are the construction of a data center, power distribution, cooling equipment, servers and networking equipment, etc. Depreciation, or amortization, varies from 10 to 15 years for data center infrastructure (includes power and cooling), to 3 to 4 years for servers. Amortization is incurred monthly, translating upfront costs into recurring costs, which are added to the Opex to define the TCO.

The TCO distribution (including the amortized Capex) depends on many factors, such as computing equipment price, peak power, utilization rate, electricity prices, interest rates, etc. For a data center of 1.8 PUE, equipped with high-end rackable servers, 75% utilized on average, the cost of electricity represents 7% of the TCO. In contrast, for the same data center equipped with cheaper, more power consuming servers, the cost of electricity represents 26% [8]. However, in real-world data centers several other factors increase the share of infrastructure costs (Capex without computing equipment): the average utilization is much less than 75%, capacity is over-sized to allow the servers to be later upgraded (with additional RAM or disk) and in general to avoid the dramatic consequences of outages, and finally the data center is not fully occupied from the first day of operation. The adjusted TCO breakdown suggests that the cost of electricity represents about 15% of the TCO [8, 31, 34]. In general, the biggest share of a data center's TCO are the computing equipment and the infrastructure amortization. Power distribution and cooling is reported to represent about 70-80% of the construction costs [31, 8]. Infrastructure costs represent 20 to 50% of the TCO based on the real-world data center problems, approximated to 40%, while computing equipment represents 35 to 60% of the TCO, approximated to 45% [8, 34]. The Capex items: power distribution, cooling and space are considered linearly proportional to the critical power [8]. The overall TCO decomposition is summarized in Table 2.4.

	Static Range	Dynamic Range	Weight
Server power	70%	30%	70%
Network power	90%	10%	30%
Total computing power	76%	24%	100%
Power share in TCO	11%	4%	15%

Table 2.5: Total Power Breakdown

2.1.2 Power-aware resource management

This section reviews techniques to better manage existing resources in order to improve energy-efficiency. There is a wide body of work on this topic, however the previous section can help focus on the areas with most impact. From the previous Section 2.1.1- 2.1.1, we can summarize a data center's power consumption breakdown between static and dynamic ranges in Table 2.5. Infrastructure power (cooling and electrical distribution) approximately doubles the computing consumption 2.1.1, but by assuming it is 100% dynamic, and proportional to the computing power, does not affect the breakdown between static and dynamic power. Section 2.1.1 stated that electrical costs amount to 15% of the TCO. Therefore, from Table 2.5, the static computing range amounts to 11% of the TCO, while the dynamic range amounts to 4% only.

In contrast, infrastructure costs (amortized) which are linearly proportional to the critical power (independent of the actual power consumed) amount to 40% of the TCO. However, reducing the critical power is not achievable with power-aware resource management, by definition, because critical power is the maximum power that may be needed in the operation of the data center (with over-provisioning). Reducing the static power range is beneficial, because the average load in cloud data centers is typically low (30%). If computing power was fully dynamic (dynamic range of 100%), then at 30% load, the TCO would be reduced by 8%. Cooling represents 30-40% of the total power consumption, therefore if cooling power consumption was cut in half, the TCO reduction is 3%. Power-aware optimization of the dynamic range can reduce the TCO by a maximum of 4%, if all computing equipment are idle, while a 50% reduction in dynamic power would provide a 2% TCO reduction. Therefore, the power-aware management techniques surveyed are those aiming to reduce static power. Reducing static power is referred to as energy-proportionality [18].

Energy-proportionality

Energy-proportionality is a hardware design problem, as seen with the evolution of the CPU, which now has a large dynamic range [8, 18]. Until energy-

Power-aware technique	% of peak power
without DVFS	71%
with DVFS	56%
with DVFS and switching off under-loaded machines	15%

Table 2.6: Static Power of a Rack of Blades

proportional hardware is widely available, optimized management of existing hardware resources can in some cases improve energy-proportionality.

One logical proposition is to switch off lightly loaded machines, which suffer from high static power range. A way to accomplish this is to leverage Virtual Machine (VM) migration [35] to turn off under-utilized machines, after migrating their VM to another machine. In [36], the static power of a rack of server blades including networking switches and storage, is measured across several power-aware management schemes, and summarized in Table 2.6. Across an ensemble of machines, such as the rackable blade servers, energy-proportionality is possible.

Naturally, consolidation must not come at the expense of performance, or service availability. Unfortunately, the idle periods observed for various internet applications (web, mail, dns) are in the order of 100 ms, thus prevent switching off complete machines. In [19], the authors suggest that low-power sleep states can achieve energy-proportionality, if the transition delay from sleep state is about 10 ms. System-wide modifications are necessary to reach such a small transition delay. The authors note that such mechanisms exist in mobile computers (such as mobile phones, smartphones and tablets), a parallel that is further addressed in Section 2.2. CPU, DRAM, SSD, magnetic disk spin-down, provide or define appropriate sleep states. Power supplies however, do not offer the same efficiency across a wide power range, and is more efficient at higher loads. Finally, some integrated management tools are needed to transition the components to sleep state.

For online data-intensive applications, such as web search, switching off under-utilized machines, or transitioning to a low-power idle state is not advisable because of the latency constraint [37]. In contrast, Map-Reduce [38] is typically not so sensitive to latency. Low server utilization can be considered a consequence of the good design practices of high-performance distributed systems, such as those relying on distributed storage [8]. A distributed storage cluster size is based on data needs (and not based on load), and to keep latency low, requests are to be handled in parallel across all machines (regardless of their load). These under-loaded machines are therefore often active, preventing transition to idle state. A possible method to improve energy-proportionality is to reduce power at the server level (not at

the ensemble level) when in low activity, but not idle. The active low-power state is achievable provided (a) a low-power memory system state with fast transition delays (sub-μs) and (b) a coordination with the CPU low-power state transition. Currently, this option is not available.

Such a direction for energy-proportionality should not come as a surprise, for two reasons. First, the idea of consolidation implicitly assumes that the low-utilization results from poor planning, which lead to over-capacity in the data center. While this is probably the case in smaller private data centers (within companies), this seems unlikely in the case of public cloud data centers. Indeed, 85% of the TCO of a data center lies with equipment purchase (servers, network) and infrastructure. An observed utilization of 30% means that the decision to 85% of the TCO was mis-planned by a factor of 3. In addition, proponents of consolidation expect that management facing such a situation would seek to correct it by deploying consolidation that would reduce, at most, their TCO by 10%. The occurrence of such decisions in the operation of a multi-million USD data center seems unlikely, and if realized, would handicap the data center in the market. Second, both switching off servers and low-power idle states can only negatively impact latency, a critical performance objective. In two examples from [34], Google reported 20% revenue loss due to an additional delay of 500 ms to display search results, and Amazon reported a 1% sales decrease for an additional delay of as little as 100 ms. Generally speaking, the high fixed cost of servers and their limited lifetime (3 years), suggest it is better to leave servers on, in order to generate revenue, than to turn them off [34].

Memory systems represent a large share of the power consumption in a server, and is little energy-proportional. Several power management techniques were proposed to reduce the memory power in DRAM, and the static range [39]. Most techniques aim at transitioning DRAM chips to idle low-power states, analogous to server consolidation, however, these techniques are not suited to current cloud application requirements (high capacity but low bandwidth and latency) [40, 41, 42, 43, 44, 45, 46]. Other techniques aim at reducing the periodic refreshes of DRAM cells by observing that not all cells require refreshes [47, 48, 49]. Refreshing DRAM is power consuming and unconnected to the activity of the machine. Exploiting other internal mechanisms of DRAM, application specific buffering of memory access can reduce the DRAM's power consumption [50]. Such techniques degrade the latency, while preserving the bandwidth, which is not the requirement in online cloud applications. An interesting suggestion is to use the last level cache of an idle processor, interconnected to the others, to hold evicted data from another busy processor, thus limiting the access to memory [51]. A similar approach is the use of heterogeneous memory modules (heterogeneous in latency, bandwidth and power) and based on an application profile, to optimally allocate memory to a module [52]. These two approaches are akin to data placement strategies found in coarser grain, distributed storage.

Contrary to the consolidation of memory modules, memory pages could be spread to avoid hotspots, detrimental to static power [53, 54].

Networking is reported to represent a large share of the power consumption (30%), however, it is not currently energy-proportional (90% static power range): its power consumption is not related to its load, the network traffic. This low energy-proportionality is expected to become a major problem as other components of the data center improve in energy-proportionality, and because the performance improvements of disks (SSD) and CPU will demand faster networking. A similar discussion to the energy-proportionality in servers can be found in energy-proportional networking. Many works study how to consolidate traffic, and then switch off freed network links [55, 56, 57]. Related works better suited to cloud computing focus instead on enabling low-power active state. In [58], network architectures are said to be typically tree-like and over-subscribed, where server addresses are fragmented in VLAN, therefore preventing the dynamic allocation of services across all servers. This restriction limits power-aware resource management, such as consolidation, and also causes congestion in parts of the network and increases the risk of service unavailability. A proposed network architecture (topology, routing, network software) improves performance, however the server consolidation objective is not a suitable mechanism for reducing static power for online cloud applications. In [27], a new topology (flattened butterfly, which relies on adaptive routing) is proposed to reduce the total power consumption, and allows to adjust link speeds to the estimated traffic, similar to [59, 60] but with μs delays, and even switching off links (affecting the topology) in order to reduce the static power range. The flattened butterfly's adaptive routing also eases the dynamic adjustment of link speed.

2.1.3 Energy-efficient Applications

The last leverage to energy-efficient data center is the application. The software ultimately determines the behavior, hence the energy consumption of the data center, therefore it seems logical to devote some attention to this aspect. Applications could be re-designed to minimize power and energy.

One foundation for the design of energy-efficient application is a power model of a machine's instruction set [61]. Such modeling enables compiler optimizations for the reduction of energy in software [62, 63]. Initial findings point to memory operands for energy reduction. These operands trigger access to cache and main memories. Compiler level optimizations, such as standard loop optimizations (unrolling, tiling) improves performance, but can also reduce energy, by as much as 40%. However, the effect of these transformations is not straightforward to predict. On the other hand, interprocedural optimizations do not reduce energy [62]. The memory usage patterns are the main reason behind these observations. However, this ap-

proach only affects the dynamic power of machines, and even less: the dynamic power of the CPU and memory, which are not the biggest impact areas for energy-efficiency in cloud data centers.

Following the evolution in programming runtime, several works investigated the impact of garbage collection on energy [64, 65, 66, 67]. They are similar to [68], where the authors observe that the energy consumption in the memory system is a significant portion of overall energy expended in execution of a Java application, which is compatible to the power consumption analysis of Section 2.1.1. The authors suggest to control Java's garbage collector's (GC) behavior by accounting for memory banks, and switch off unused memory banks, from the application. This approach can reduce the static power in memory, a worthy objective. The actual switching off of inactive memory banks is performed outside the application, however the application can indirectly control this transition. This approach breaks the usual abstraction layers of a modern computing platforms, in a cross-layer fashion. The Garbage Collector (GC) can be modified to compact the heap, allocate objects to active banks first, and perform collection more frequently. The underlying system is modified to include an automated bank switch off mechanism and cache memories. The experimental results are obtained using a SPARC instruction set simulator [69], and a model of power estimations for all instructions obtained from [70]. Overall, the results reveal non-linear effects from the combination of several changes, which significantly improve upon the isolated changes (or first order effects). Only the memory bank switch off reduces static power, as we saw in Section 2.1.2. Although this lies outside of the GC optimization, its effectiveness is improved with the GC optimization.

The results from previous investigations can find their way into cloud data centers in several ways. However, optimizing code for energy in VMs hosted in the cloud does not guarantee much gain, because the execution environment of the VM is unknown to the cloud customer, by definition. Recent evolutions in cloud services provide some opportunity for greener applications. Indeed, cloud providers now offer abstracted software services, such as storage, database, messaging. These services are fully controlled by the cloud provider, unlike the same service packaged in a customer's VM. The cloud services could therefore be tuned by the cloud provider, to reduce energy. Moreover, cloud providers now offer platforms (such as Google's Appengine, various Map-Reduce frameworks), which do not execute a full VM, but operate from blocks of source code. The cloud provider then controls the compiler, the runtime, the software services, and the underlying infrastructure. In this context, the application energy transformations can be used in conjunction to other energy-efficiency measures, such as power-aware resource management. For example, energy-efficient Map-Reduce configurations [71, 72] could incorporate optimized compiler options to build the customer provided source code. Another direction to energy-efficient appli-

cations is the client-side javascript runtime, to move some computation out of the data center. Finally, cloud providers also offer access to complete applications (Google Docs), which provide even more control to the cloud operator.

2.2 Arrays of Wimpy Nodes

The biggest cost items in a cloud data center are the computing equipment purchase (45%) and the infrastructure (40%), as discussed in Section 2.1.1. Infrastructure costs are driven by the critical power requirements. An array of wimpy nodes (AWN) is a set of low-power, low-performance and low-cost machines. Successfully exploiting AWN appears promising to reduce significantly the TCO. The exact size of a wimpy node is an open question: "The more interesting discussions today are between low-end server nodes and extremely low-end (so-called wimpy) servers" [8]. AWN concurs with the opinion that energy-efficiency must be addressed at the hardware level [18], as discussed in Section 2.1. The recent explosion in mobile computing makes AWN a more realistic proposition. This section presents AWN, the main assumption of this work.

2.2.1 AWN as a distributed system

The first AWN implementation we present is a cluster of low-power System on Chip (SoC).

Early approaches

The first works on wimpy nodes were inspired by embedded computing. One of the first mention of the role of embedded computing for server design as the importance of power increases can be found in [13].

As computing evolved towards On-Line Transaction Processing (OLTP), the suitability of the processor design of the time was challenged. In [73], the authors reported that while processors were becoming increasingly complicated (increased instruction-level parallelism, speculative out-of-order execution, floating point operations and multimedia capability), they were poorly suited to the, then emerging, OLTP applications. From their characterization of the commercial loads, they designed a chip of eight simple Alpha processor cores with a two-level cache, packaged such as to allow their aggregation. Although each core is significantly slower than competing processor cores, the overall system was able to be 3–5 times faster, on the selected workloads. The workloads are inspired from TPC-B and TPC-D benchmarks, using an Oracle database server, and are parallel. The results were obtained from a simulator. Similar chip design experiments include Chip Multi-Processing (CMP), where even multiple smaller processor cores

Metric	Comparison with reference architecture
Performance	11% (webmail) – 86% (video streaming)
Performance/infrastructure-$	90%(webmail) – 650% (video streaming)
Performance/W	100%(webmail) – 720% (video streaming)
Performance/TCO-$	90%(webmail) – 600% (video streaming)

Table 2.7: Simulated AWN Performance on Web Applications [5]

compared favorably to higher end processors [74]. Although these works envision the OLTP loads, similar to cloud applications, they did not target energy-efficiency.

In [75], the authors propose an energy-efficiency benchmark, and present a system design that achieves ×3.5 better energy-efficiency than the previous sort benchmark winner. The key finding, which will be often reported, is that balance of design matters more than individual component performance (they also single out the CPU performance and power). Their energy-efficient system is based on a low-power CPU (Intel CoreDuo T7600, 34 W Thermal Design Power (TDP)), equipped with a fast I/O board, laptop-grade disks (5400 rpm), and DDR2 memory. Moreover, the authors stress the importance of software, especially OS components such file system (XFS). These findings mean that although balanced designs show superior performance, it remains specific to a workload.

One of the earliest studies [5] on the design of web application servers from power-efficient components (such as used for embedded computing) reported promising results in cost and energy, Table 2.7. This study included power consumption, cooling and local rack networking, albeit from simulations, specifications and industry reports, not measurements. The results compare the performance-per-infrastructure cost (everything but power and cooling), performance-per-Watt, performance-per-TCO for a selection of web applications (web search, web mail, video streaming, and Map-Reduce) between a reference architecture equivalent to a Xeon (2 processors × 4 cores @2.6 GHz) based machine (340W and $3.3k) and the equivalent of a P.A. Semi [1] PA6T (1 processor × 2 cores, @1.2 GHz) based machine (52W and $0.5k). The embedded component based rack uses memory sharing and remote low-power disks with flash-based disk caching, and relies on aggregated cooling housed in an enclosure with directed air-flow. An interesting finding is that a lower power and cost architecture (AMD Geode) performed worse on all metrics than the reference architecture, which indicate that there are other factors to consider.

At about the same time, a more detailed study measured the effectiveness of low-power components for Internet applications [7]. Their contri-

[1] acquired by Apple in 2008 to contribute to the iPod design

bution introduced the term Fast Array of Wimpy Nodes, which we adopt. The starting point is similar to previous work: new Internet applications could be hosted on more energy-efficient architectures. For example, they observe that CPU power increases faster than performance. In addition to low-power components, the authors expose the problem of balanced design, by assembling components that match the application's requirements and are right-sized in the overall design (computation, memory, I/O). The tested AWN nodes are an AMD Geode LX CPU @500 MHz, with 256 MB DRAM, which consumes 6 W. The benchmark application is a specifically developed distributed key/value store, based on Chord [76], and similar to `memcached`. The results show that with a $\times 3$ replication factor, a 720 nodes Fast Array of Wimpy Nodes (FAWN) could serve a 3.8 TB dataset at 506 K queries/second, or 101 queries/joule, consuming 5 KW (excluding power distribution and cooling), where an equivalent system using high-end servers would need 15 KW and cost 50% more. However, their FAWN design includes high-end servers, that act as front-ends to the wimpy nodes. The front-ends cache reads, to avoid a possible collapse in throughput when a single node becomes overloaded (when its keys are frequently accessed). However, for less than 1024 popular keys, an homogeneous FAWN without caching performs as well as with caching. Not all the results come from direct measurements, the results for Geode nodes are extrapolated from measurements on older embedded devices.

The initial findings were extended and refined in [77]. They identified the key benefit of AWN by observing that "dynamic power scaling techniques are less effective than reducing a clusters peak power consumption", because the peak power consumption determines the data center's infrastructure, and also influences the cost for servers. They considered several data-intensive workloads: I/O-bound workloads, memory/CPU-bound workloads, latency-sensitive but non-parallelizable workloads, large memory-hungry workloads. They also compare newer hardware, Atom @ 1.8 GHz based machine with 256 MB RAM, quad-core i7 860 @ 2.8 GHz with 2 GB RAM ("Desktop"), and a high-end $2\times$ quad-core i7 with 16 GB RAM ("Server"). The FAWN was more energy-efficient for I/O-bound (key/value store, `grep`) and memory/CPU-bound loads (matrix transpose multiplication, cryptography), but was less efficient on latency-sensitive and large memory-bound (machine learning) workloads, although with substantial program modifications the wimpy node performed $\times 2$ better (by using the memory more). The good performance of AWN on I/O-bound loads was also documented in [78], where, for the same power budget, an AWN based on Atom processors was able to deliver $\times 20$ more sequential I/O than a state-of-art high-end server based system, and for the same cost, delivered $\times 5$ more throughput. However, those results are derived from the benchmarking of a single node, extrapolated to a cluster.

In [33], the increasing imbalance in the design of data center servers (be-

tween CPU, memory and storage) was also raised. Moreover, this imbalance is reported to impact latency, which unlike bandwidth, cannot be addressed with parallelism [79]. This suggested the use of multiple machines with less powerful CPU, that match the other system components. The main objectives are to reduce the server costs and power. The superior reliability of servers is said to be superfluous, because servers are usually replaced after 3 years. The reference hardware, based on a 3.6 GHz CPU (with 15k RPM SCSI disks, and 2GB of main memory, which use 297 W at 60% utilization, and costs $2.3 K) is replaced with lower power machine, based on an AMD Athlon 64 4850e @ 2.5 GHz (with DDR2, which costs $500). The comparison is made on a real Microsoft Server 2003/II-S web application. The requests per second drops from 96 to 75, while the requests per joule increases from 0.33 to 1.25 (×3.9), and the requests per $ increases from 0.04 to 0.15 (×3.7).

In [9], the authors noted the evolution of workloads as a decisive event for the application of AWN. Internet applications shifted the focus away for single core performance towards I/O, memory, and power. As mentioned previously, the size of the data to process increasingly influences the design of systems (hardware and software) [8]. They chose for evaluation a web search service (Microsoft's Bing), which is more computationally intensive than similar applications (because of the use of statistical machine learning) which makes server performance more important. Each node keeps in memory a part of the overall web index, and ranks its indexes according to the query. The machines compared are a power-optimized Xeon L5420 quadcore @ 2.5 GHz and an Atom dualcore @ 1.6 GHz. The dynamic range of the Xeon is 38–75 W, while the Atom's is 1.4–3.8 W. However, the fewer Xeon nodes store more indexes, and thus operate at a high utilization, whereas each Atom operates at a lower utilization, which translates to a much reduced power consumption given its better energy-proportionality. On average, a Xeon core consumes 15.6 W, and an Atom core 1.6 W. A Xeon core achieves 2× the throughput of an Atom core, and is more robust when the query load varies. Therefore, an Atom core is 5× more energy-efficient then a Xeon core. The Xeon node achieves 98% latency objective, while the Atom 93%, and the variance of the Atom's latency objective is greater. Latency not only impacts the perceived user response time, but also the quality of the search results, as late results are not accounted for (cut-off), and latency slack is exploited to refine results. Reliability in case of node failure, is handled by spreading the query load across the other nodes. In this scenario, Atom nodes provided better latency as the increase in load is spread across more nodes, than for the Xeon nodes. Considering a 15 MW data center, the TCO of Atom nodes was greater than with Xeon nodes, mainly because the processor share in power is smaller than for Xeon nodes. The authors suggested machine modifications (increase the number of cores per node) to improve the share of processor in overall power. This study showed that

for data-parallel compute intensive applications, the Atom-based AWN does not reduce the TCO.

The potential benefits of AWN are not considered easily reachable. In [80], the author points several weaknesses in the AWN proposition. The software often needs to be parallelized, which is difficult and expensive task (when possible). This added development cost is rarely mentioned in cost comparison. Even if the application can be parallelized, the resulting code will often run at the speed of the slowest thread, and is impacted by the inter-thread communication parallel applications create. Each of the wimpy node requires some OS code and application data, replicating common code and data on all wimpy nodes' memory. The author states that for these reasons, practical parallel designs usually operate at the request level, leaving each request handling code intact, therefore, the wimpy nodes cannot afford to be too much slower than the current server nodes. The author's analysis is useful and correct (good software is expensive, parallelism is difficult, nodes currently require a full OS), however, the conclusion can nevertheless be incorrect. It can be incorrect because the analysis implicitly refers to costs, as of the time of writing. However, costs, as visible in markets, change. Changes in the cost assumptions can lead the same analysis to a different conclusion. The interest in AWN originates from an evolution in workloads, many of which are new Internet data-parallel applications (in 2009, 67% of the cloud market is Internet services [9]), without necessarily requiring high inter-thread communications. Moreover, many of the new applications are deliberately designed to take advantage of the evolving hardware. Although OS are complex softwares, they still evolve, such as minimal gnu/linux distributions, thanks in part to free software licenses [2]. The costs for software modifications are not useful in isolation, but are to be compared to the other costs in data center operation (such as the cost and capability of wimpy nodes), subject to change. Therefore, a general rule derived from facts valid only in a point in time are likely to be invalidated.

A similar analysis was presented in [81]. The authors evaluate AWN under complex database workloads (TPC-H based), using parallel databases (DB-X and Vertica). They compare Atom-based wimpy nodes to Xeon-based servers. The metric is price/performance, where price includes purchase cost and energy. A cluster of 5 Atoms results in an increase of 11–31% in price/performance compared to a single Xeon (depending on the exact benchmark). A cluster of 30 Atoms results in an increase of 23% in price/performance to a 6 Xeon cluster. These results confirm that not all applications are suited to the AWN. However, the costs considered do not cover the infrastructure costs (40% of the TCO), which depend on the critical power. It is difficult to observe the maximum power requirements in the two setups described. As mentioned previously, the purchase costs are a key

[2] `https://coreos.com/`, `https://www.docker.io/`

component of the analysis, and any change in the cost of wimpy node can impact the conclusions, as we discuss next.

Smartphones and the ARM processors

The previous works presented so far exploited embedded and laptop components (such as Atom processors) to bridge the gaps they noticed between current server architectures, emerging data center loads and increasing power-related costs. The success of smartphones, equipped with ARM processors, changes the situation. In [82], the author observes that mobile devices were becoming powerful (2008: 32-bit RISC with gnu/linux support, MB of RAM, Flash storage), while still dissipating little heat (less than 1W, no active cooling), and suggests that the mobile components could be applied to the data center, for power reduction. The anticipated objections to the adoption of mobile technology (unreliable "toys") are dismissed as the usual barriers to change, that were raised (and overcome) before, such as in the transition from mainframes to minicomputers, then onto RISC servers, then onto PC-type servers. The interesting perspective is that the potential for mobile technology does not only rely on the ARM CPU, but also on other devices, such as storage, and low-power memory. These non-CPU components now play a big role in low-power computing, such as SSD and LPDDR2. The low-power benefits of these technologies were visible from the battery operation, and heat dissipation. Software and power-aware resource management also played a role, but were not acknowledged. Although not a detailed study, this work detects that the emerging mobile market could provide the technological components to the data center.

Several works investigate the potential of ARM-based wimpy nodes for the data center. In [83], the authors compare a small cluster of 4 pandaboards (ARM Cortex A9) with a workstation (Intel Core2 Q9400), using several benchmarks: a web server, an in-memory database, and video transcoding. The results show that the ARM based cluster is 1.21× more energy-efficient for static web loads, 1.3× more for video transcoding, and 2.6–9× more for in-memory database. However, they do not provide much insight on how to overcome the performance penalty resulting from the use of ARM processors, or how to design data center scale systems for cloud loads.

The ARM processor characteristics are presented in [84]. Like the x86, the ARM A9 processor employs out-of-order execution, when previous ARM A8 or Atom processor do not. ARM processors are usually packaged with additional chips (such as GPU, memory controller, I/O ports), into a ARM-based SoC, whereas a typical x86 requires additional components (dissipating as much as 30 W). For example, the OMAP 4430 SoC found in the pandaboard, packages a dual-core A9, a GPU, a DSP, two M3 cores, a memory controller, an interface to external DDR and I/O ports. The ARM

designs have a larger dynamic range, in part due to more frequency-voltage points available to DVFS than x86. The authors evaluate the ARM A9 and A8 based SoC, a Xeon X3450-based server, and two laptops. Unfortunately, they set the maximum frequency of all the machines to that of the A9: 1 GHz, while the Xeon can run at up to 3 GHz. I consider, contrary to the author's position, that this is unfair, because the Xeon's higher clock frequency is a key feature of the server class processor.

ARM processors are also investigated for HPC [85, 86, 87]. In [85, 88], a Tegra 2 SoC (ARM A9-based) is evaluated with micro benchmarks (Stream, Dhrystone, SPEC CPU 2006) and Linpack. The same performance and energy-efficiency results were obtained, as in previously mentioned studies. One interesting observation is that only 6% of the total power consumed by an ARM node is used on the CPU cores, 30% on the Ethernet interface and memory module, and the rest (60+%) on PSU, USB, HDMI. The authors suggest two improvements to reduce this overhead: increase the core count (also suggested in [9]) and add additional logic, such as a Single Instruction Multiple Data (SIMD) unit (useful for scientific loads). These recommendations can be considered as the application of the balanced design idea to HPC.

Although the case for AWN was made before the ARM processors were widely used, there are surprisingly little academic publications on their impact for cloud data centers, which already exploit scale out designs. In contrast, the industry appears more interested in AWN, and most of the latest projects consistent with the review of past AWN designs.

In January 2014, ARM released a Server Base System Architecture (SBSA) specification [3], in collaboration with software (Canonical, Citrix -owns the Xen hypervisor-, Linaro, Microsoft, Red Hat, SUSE), and hardware companies (AMD, Dell, HP, Applied Micro, Texas Instruments). This specification is directly targeted at the data center. It defines the interface between the hardware and the system software (OS, hypervisor, firmware), such as to allow system software images to run on compliant hardware. It defines the CPU characteristics, such as endianess, required SIMD, interrupt controller (GICv2), hypervisor support features. This specification was greeted by the Open Compute Project [4], a project initiated by Facebook, to "enable the delivery of the most efficient server, storage and data center hardware designs for scalable computing". The Open Compute Foundation's chairman, and Facebook's vice president of infrastructure, recently joined Calxeda's board of directors. However, this did not prevent the pioneer in ARM-based server fabric (integrated cluster) from apparently closing business. AMD announced the launch of ARM based CPU for the server in

[3] `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0029/index.html`

[4] `http://www.opencompute.org`

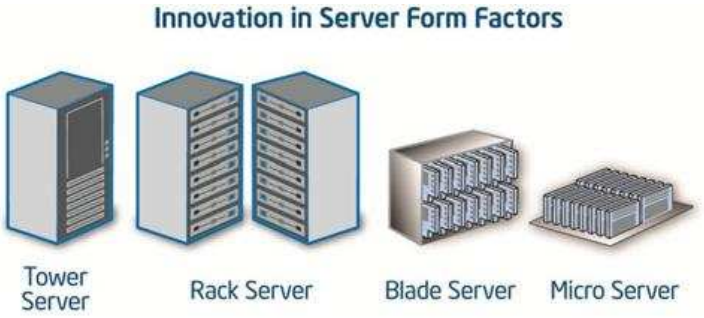


Figure 2.1: Microserver (source: Intel)

Q1 2014. Their A1100 is planned to include 4–8 A57 cores, a 64-bit ARM core @ 2 GHz, for an expected power of 25 W. Intel's recent Atom C2000 processor family is directed at AWN clusters, up to 8 cores @ 2.6 GHz with a minimum power of 6 W.

The industry's AWN implementations are often called microservers [5], Figure 2.1. Boston claim to be the first provider of ARM-based microservers [6]. Their microserver contains up to 192 cores in 48 independent quadcore ARM A9 @ 1.1 GHz and A15 @ 1.8 GHz (provided by Calxeda) consuming less than 300 W. Dell's Copper ARM microserver is an integrated cluster, of up to 48 nodes, each based on quadcore Marvell XP @ 1.6 GHz. HP offers a similar product, codenamed Project Moonshot, an integrated cluster of low-power nodes, up to 45 hot-pluggable servers. A possible node is the AMD X2150, low-power quadcore x86 @ 1.1–1.9 GHz, with an integrated GPU, consuming less than 22 W, for hosting remote desktops.

2.2.2 AWN in a System On Chip

As mentioned in Section 2.2.1, the ARM processor is actually a SoC, which packages more than just the CPU cores. For example, the ARM SoC designed by Calxeda [89] includes a networking component, to improve the interconnect across nodes. The idea of balanced design as the key to performance and energy-efficiency could motivate the implementation of a complete AWN in a single SoC, in contrast to assembling a cluster of SoC.

Graphical Processing Unit

The GPU implements the AWN concept as an accelerator chip, which delivers a part of the AWN in a low cost, low-power form. The GPU provides a dedicated chip for graphics related processing. The class of applications suited to GPU are those requiring computation intensive operations over a

[5] `http://www.intel.com/content/www/us/en/servers/microservers.html`
[6] `http://www.boston.co.uk/solutions/viridis/default.aspx`

large data set, which can be processed in parallel (as independently as possible), with throughput as the main performance objective [90]. While General Purpose GPU (GPGPU) was possible on the GPU (expressing non-graphics computation in graphics terms), interest lead the manufacturers to provide a more accessible API, such as Nvidia's CUDA SDK [91] or OpenCL [92]. There are several APIs now available which aim to ease the use of GPU, such as OpenACC [93]. A recent GPU board such as the Tesla K40 can deliver up to 1.43 Tflops with 2880 cores @ 775 – 875 MHz, and can support 12 GB of memory, requiring 235 W. Such a device is a special-purpose large cluster on a chip, for a modest power budget. The capability of GPU make it common component in the design of HPC systems, as evidenced by the presence of GPU in supercomputer rankings such as `www.top500.org`. Each core is less powerful than the ARM, laptop and embedded CPUs mentioned so far.

The programming difficulties raised in the context of wimpy nodes apply even more to the GPU. The design specificities of the GPU must be exploited in program designs to achieve the desired performance, such as the throughput oriented design objective. Also, the GPU being an accelerator, it is not meant as a general purpose computer, and does not provide networking, I/O nor standard programming interface (such as Unix) and libraries (although many are available). This leads the program design to split the processing between GPU and CPU.

In an effort to bridge the CPU/GPU differences and to benefit from the SoC integration, new hybrid platforms are packaging both technologies in a single SoC. The recent Nvidia Tegra K1 is such an example, it combines an ARM CPU and a low-power GPU [7]. The ARM multicore is itself heterogeneous, 4 A15 cores @ 2.3 GHZ, and one low-power (and lower performance) core. The GPU offers 192 cores. The main memory is DDR3 and LPDDR3 (as discussed in Section 2.1.1).

AWN in a System on a Chip

In [94], the authors study the performance of Atom-based nodes and Xeon-based node for the execution of `memcached` (a distributed, in-memory, key/value store). Their conclusion is that both platforms are inefficient, as the program executes far from the theoretical performance of both the network interface and the memory subsystem. They identify the cause for the bottlenecks (poor instruction caching, virtual memory bottleneck, branch prediction bottleneck) and propose a SoC of small CPU cores, a fast network card, and an FPGA to implement some `memcached` functions.

Intel Xeon Phi [8] [95] is a family of co-processors, that provides up to 61

[7] http://www.nvidia.com/object/tegra-k1-processor.html

[8] `http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html`

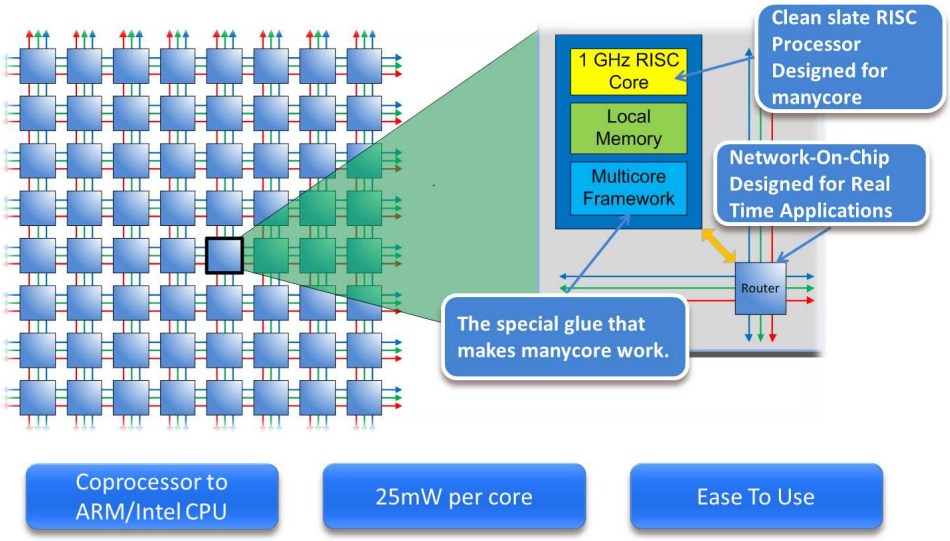


Figure 2.2: Epiphany SoC Cluster (source: Adapteva)

x86 cores (supporting 4 threads/core) @ 1 GHz, 6 – 16 GB memory, a PCIe interface (PCIe is the Phi's the form factor), a memory controller, power management functions, consuming 225 – 300 W (TDP) and is passively cooled. The programming environment is standard (C/C++, OpenMP, MPI, OpenCL), as is the ISA. Although targeted at HPC, the Phi can be considered an AWN on SoC, similar to the GPU. Prices are reported in the k$ range, similar again to the GPU.

Epiphany [9] is a 2 W (maximum) SoC cluster, composed of 16 nodes (32-bit RISC @ 1 GHz, dissipating 25 mW), arranged in a 2D array, connected by a low-latency mesh network-on-chip, Figure 2.2. Each node is an independent machine, with local memory, and ANSI-C programmable. The memory model exposed to the application is a unique shared memory address space. Memory locations outside a node is accessed through the mesh network. The network is based on atomic 32- bit memory transactions and is transparent to the program (except for latency). The key advantage of the Epiphany SoC is its programming model, which is ANSI-C. One Epiphany-based computer is Parallela, which includes a dual core ARM A9 CPU, an FPGA, the Epiphany 16 or 64 cores SoC, a 1 GB Ethernet, 1 GB RAM, consuming 5 W (for 66 cores), at 99$.

At the extreme end of the low-power AWN SoC range is the GA144 [10]. Although unsuitable for cloud data center applications, its extreme design highlights the strengths of AWN. This chip packages 144 independent nodes (F18A), which operate asynchronously (similar to event handlers). The

[9] `http://www.adapteva.com/`
[10] `http://www.greenarraychips.com/`

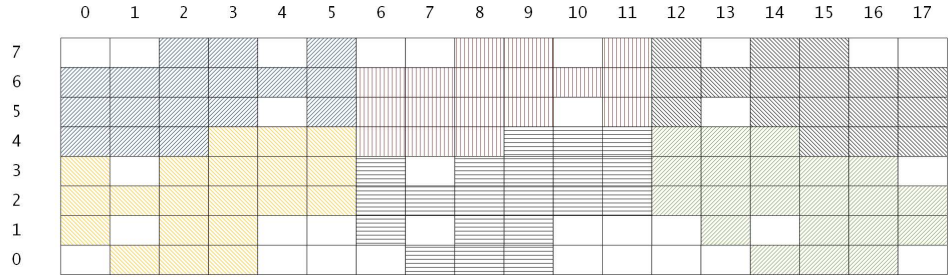


Figure 2.3: AES Node Placement on the GA144 [2]

maximum power requirement is 650 mW (each instruction consumes 7 picojoules). The SoC is almost completely energy-proportional, idle it consumes 14 μW, an idle node consumes 100 nW (each node can be suspended in mid-instruction). Each node has very limited memory (64 18-bit words), the full GA144 can hold 41472 bytes. Therefore, the GA144 needs an additional memory component to store larger programs. However, the instructions to be executed can be streamed to each node, freeing local memory for data. The available programming language is colorForth, a variant of Forth where color has syntactic meaning (the inventor of the Forth language founded GreenArrays). In [2], the authors implement two cryptographic algorithms: AES and RSA in Forth for the GA144. The AES code requires only 17 nodes of the 144, therefore multiple instances of AES can run in parallel, the placement of the running nodes per AES instance is shown in Figure 2.3. The placement depends on the need for I/O to and from the chip, and defines an interesting optimization problem. The RSA-1024 implementation uses 107 nodes. AES-128 encryption on the GA144 takes 37.9 μs (and 0.90 μJ), while an ASIC implementation takes 1.23 μs and a microcontroler 455 μs. RSA-1024 encryption on the GA144 takes 513.7 μs (and 24.6 mJ), while a reconfigurable cryptographic processor takes 3.84 μs, and a microcontroler 4,730 μs.

2.2.3 Abstract AWN Model

We summarize the above presentation of AWN, by defining an AWN model, to serve as a reference for the remainder of the document. An AWN is a distributed system, composed of low-power, low performance nodes, interconnected by a slower network than found within chips. The main benefit of the AWN is the energy-efficient scale-out, by combining independent low-power nodes, instead of relying on a single higher performance node (because its power increases faster than its performance). Scaling out cannot be reduced to adding cores, but also requires scaling out memory and networking, to accommodate for the Internet scale applications hosted in cloud data centers. Accelerators, such GPU, Xeon Phi, and AWN SoC, do

not meet this requirement. For reducing the data center TCO, the AWN should be competitively priced. The individual components from the mobile and embedded market (with orders of magnitude volume compared to traditional server technology), exploit the commodity-off-the-shelf (COTS) benefits.

2.3 Performance of AWN

Green computing is not only concerned with reducing power (either maximum or used) or energy, but must also address performance. Ideally, a green alternative should offer similar performance levels as current non-green solutions. In Section 2.2.1, we provided performance reports across several architectures and workloads. In summary, on Internet related workloads with data parallelism and I/O intensive (such as distributed in-memory storage, web search) the AWN provide better or equal performance. However, for more complex applications, either less parallelism or more CPU intensive (such database servers, machine learning enhanced web search), the improved energy-efficiency is usually obtained at the cost of a drop in performance. However, most of these results were not obtained with recent ARM-based AWN.

In this section, we report our own performance experiments of the first available AWN microserver, the Boston Viridis, within our HPC platform[11]. The Viridis microserver is a self contained, high-density 2U rack mount enclosure featuring 48 low-power SoC based on ARM A9 processors (quad-core), packaged in 12 Calxeda EnergyCard modules, and an integrated high-speed 10 GbE interconnect, Figure 2.4. The distinctive feature of the Viridis microserver are the Calxeda provided interconnect, and the power management. Each node runs Ubuntu 12.10 and a customized Linux kernel 3.5.0. The Viridis configuration is diskless, persistent storage is available via a network file system, external to the ARM microserver. The announced network uplink of 10 GbE was actually only 1 GbE.

2.3.1 Comparison of the Viridis ARM microserver and a GPU

In [96], we compared the performance of an ARM microserver and a GPU for Map-Reduce workloads, in terms of performance and power efficiency. The characteristics of the two platforms are summarized in Table 2.3.1. We have seen that both platforms can be considered AWN, although the GPU offers partial scale out capability, especially for lack of memory. For this comparison, the Map-Reduce applications' input size was limited by the

[11] `http://hpc.uni.lu`

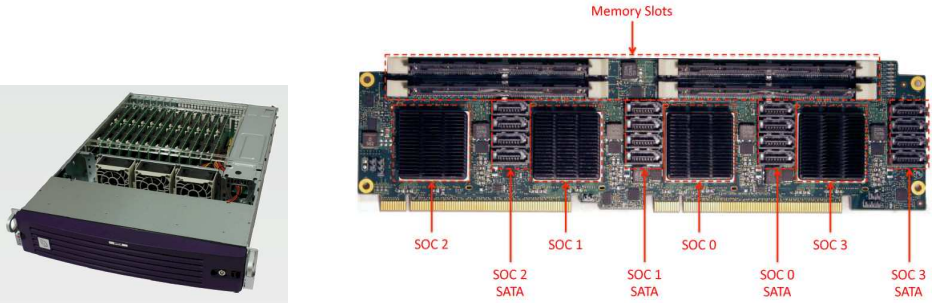


Figure 2.4: A Boston Viridis enclosure: general overview and Calxeda EnergyCard modules.

Table 2.8: GPU and ARM Microserver Overview

Hardware	Core count	CPU clock	Maximum power
Nvidia M2090 (GPU only)	512	1.3 GHz	225 W
Viridis ARM microserver	192	1.1 GHz	300 W

GPU's 6 GB of DDR5. The combination of these two platforms is also possible, as in the Mont-blanc project[12].

The assumption for the comparison is that both AWN could serve as the building block, to construct a much larger system (data center scale). The microserver and GPU can be considered comparable building blocks based on their total power (including the GPU host machine's power), core count and core clock frequency. Under this assumption, we wish to measure the performance and energy-efficiency of a single building block.

Map-Reduce workloads

The most popular Map-Reduce implementation today is Hadoop [97]. For this evaluation on the ARM microserver, we did not choose Hadoop, but another more recent Map-Reduce implementation from Nokia Research, Disco [98]. Disco core is implemented in Erlang, and runs user-supplied functions in Python or Ocaml. Moreover, like Hadoop, Disco users can use the language of their choice via an external interface. This choice is motivated by the following:

- Disco is simpler to use than Hadoop. This is a critical factor because the main feature of Map-Reduce is its simplicity for the user. Disco, by default, supports user-supplied functions in Python, a very productive language, popular in both the industry and academia. Disco's

[12] `http://www.montblanc-project.eu/`

simplicity is also reflected in its configuration, it is easier to configure Disco than Hadoop, although Hadoop provides more flexibility and features.

- Disco's more recent implementation is also simpler. It benefits from the Erlang language and runtime, which is designed for concurrent and distributed applications. This leads to a much smaller code size ($\times 6$) than Hadoop [99].

- Disco is often faster than Hadoop [99]. This is surprising given the slower runtimes of Erlang and Python compared to Java. Job overhead latency, distributed data storage throughput and latency are $\times 6 - 30$ better than Hadoop's. However, the total execution time for some large jobs is about the same.

The Disco services that control data access run on one of the nodes in the ARM microserver (all Map-Reduce services run in the ARM microserver). Also, the data items managed by Disco are, when possible, cached in the nodes' memory. This means that although the input data is small, data accesses may resolve to the network file system.

For the GPU, we selected Mars [100], a well-known Map-Reduce implementation for a single GPU, which executes most of the code on the GPU. The benchmark applications used in the comparison are taken as-is from the Mars code base. Mars compares favorably to Phoenix [101], a multicore implementation of Map-Reduce, which is an indication of its quality. Although the GPU is a good match for Map-Reduce applications, its characteristics bias the comparison in several aspects:

- The GPU global memory is very limited compared to the distributed storage of a cluster, making it unsuitable for larger jobs. The only alternative is to use additional resources, either CPU or other GPU, to provide this necessary capability, complicating its application.

- The data access is faster than in a distributed platform because the threads can directly access any data item in the shared memory space. This simplifies the design of the Map-Reduce framework.

- The scheduling of tasks is largely handled by the GPU microcode, and given the shared memory space, there is no need for data/process placement.

- The executable code is native on both the CPU and GPU, while Disco executes distributed Erlang and Python code.

We selected 5 benchmarks for this evaluation [98, 100]. They represent typical loads: web document searching (Word Count and String Match), float manipulation (Matrix Multiplication) and web log analysis (Page View

Count and Page View Rank). These applications are commonly used to compare Map-Reduce frameworks.

Performance results

Figure 2.5 shows measurements for the Word Count application. It is representative of all the Map-Reduce applications tested that include a reduce step: Word Count, Page View Count, Page View Rank. The input size is limited on the GPU by its 6 GB of global memory.

The energy-efficiency measures the amount of work done per energy unit. Energy is the average power over several instantaneous power measures. The power readings were obtained from the ARM microserver with Intelligent Platform Management Interface (IPMI), which captured the average power consumption over the time period (1 second) across all nodes. For the GPU, `nvidia-smi` was used. A baseline of 30 W is added to the measurements, to account for the power required by the host, a relatively low estimate.

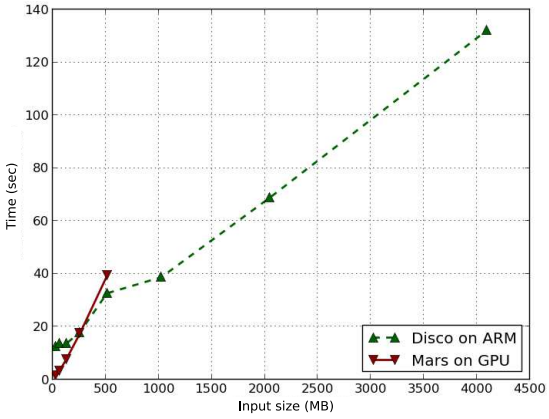
We notice that the results are similar, in terms of performance and power. The only difference relates to the small input sizes, where the Disco/Viridis suffers from more overhead such as initialization, related to the more complex software and hardware environment, than the Mars/GPU.

Figure 2.6 shows the performance measurements for String Match on both platforms. It is representative of all the Map-Reduce applications tested that do not include a reduce step: String Match and Matrix Multiplication. Mars on the GPU is less than one order of magnitude faster than Disco on ARM cluster. The GPU is much faster for small input sizes. The faster communication between threads (between map and reduce) is favoring the GPU where the data is stored in the GPU main memory. The Disco initialization is again penalizing the performance.

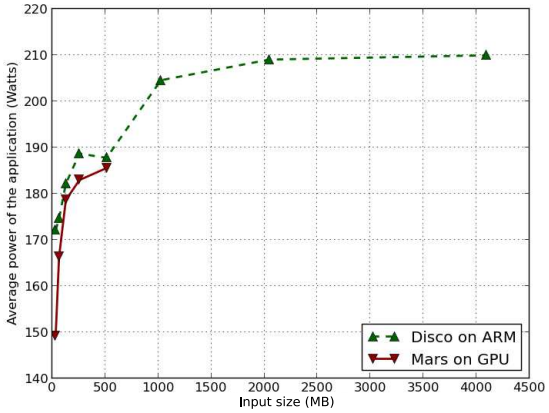
Two power plots, real and adjusted, are presented for the GPU, Figure 2.6b. This is a consequence of the short execution times on Mars. Indeed, the `nvidia-smi` power measurement command impacts the performance of the GPU if ran too often. Therefore the few real measures were modified based on measures from longer runs (processing bigger inputs).

Figure 2.7 shows the runtime of all applications, on the ARM microserver and the GPU. The largest input data is used in each of the benchmark. The breakdown of the runtime in the different steps is also shown. The left bar represents Mars on the GPU, while the right bar represents Disco on the ARM microserver. Mars includes the preprocessing and input/output (IO) times. The suffix '1' and '2' refer to a chained Map-Reduce execution, where an application is splitted in two Map-Reduce executions, the case only for the Page View Count (PVC) application.

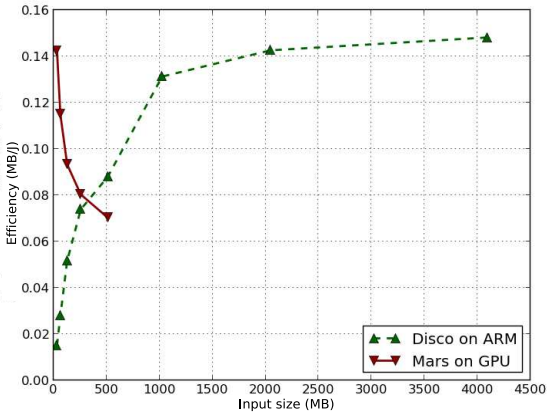
Disco is faster only for Word Count. The biggest difference in speed in favor of Mars is for Matrix Multiplication. Overall, the performance difference is less than one order of magnitude.



(a) Total time

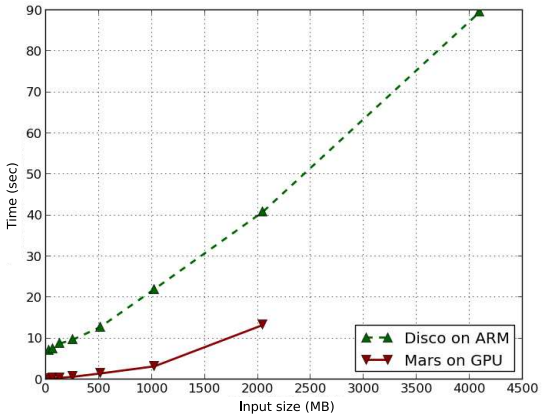


(b) Power average

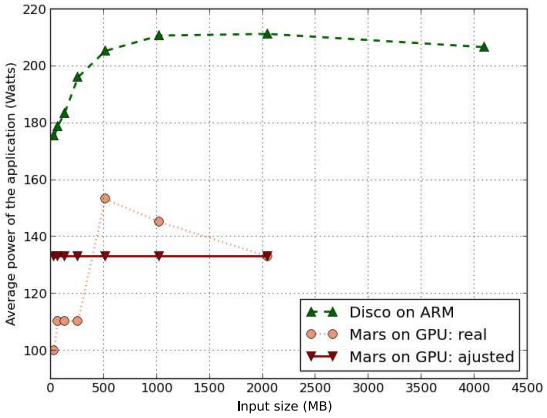


(c) Energy-efficiency

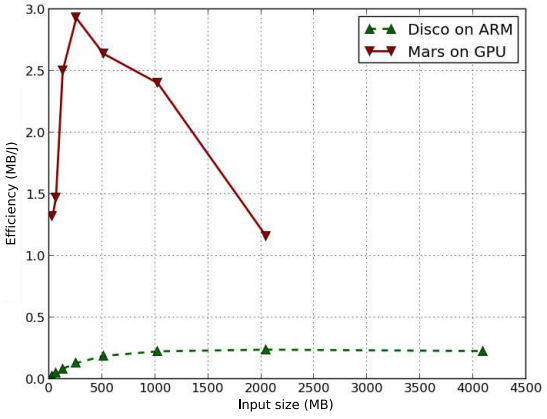
Figure 2.5: Results for the Word Count application



(a) Total time



(b) Power average



(c) Energy-efficiency

Figure 2.6: Results for the String Match application

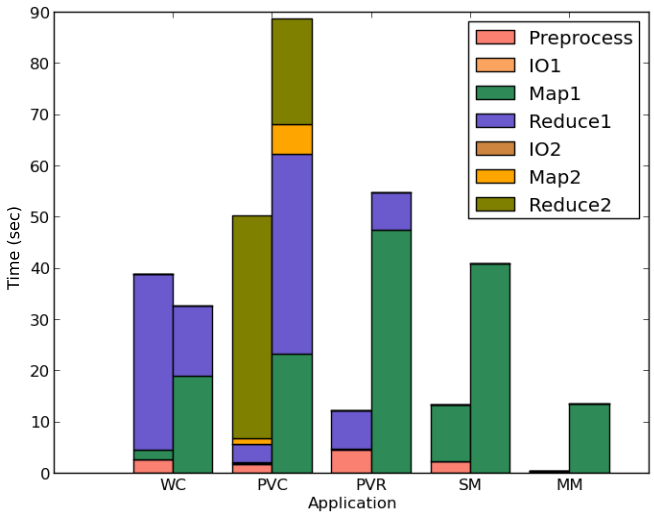


Figure 2.7: Performance per Application for the Largest Input. Mars/GPU: left, Disco/ARM microserver: right

Discussion

Both platforms were considered as equivalent building blocks for the assembly of a larger system. The fundamental differences in architecture leads to very different Map-Reduce frameworks: Mars/GPU runs native code on a tightly integrated chip, while Disco/Viridis runs bytecode on a distributed system. The same software runtime could further determine the architecture differences, but that would require the same codebase to be used on both platforms, which is hardly possible.

Even though, the performance is not so different between the two environments, except perhaps for Matrix Multiplication. One can only conjecture what the performance of a faster Map-Reduce application would be on a microserver with faster interconnect, but an order of magnitude improvement does not seem unreachable (given the performance difference between Python code and native code).

2.3.2 Comparison of the Viridis ARM microserver and a multi-core CPU

As an extension of the previous comparison, we compare the performance of the ARM microserver to a multi-core server. The assumed building blocks for the comparison is the Calxeda EnergyCard (4 integrated quadcore nodes) and multi-core server serve as the building block in a much larger system (data center scale). Under this assumption, we wish to measure the performance and energy-efficiency of a single building block. For the comparison, we selected to evaluate 4 nodes (16 cores) of the ARM microserver and 1

Table 2.9: CPU and ARM Microserver Overview

Hardware	Core count	CPU clock	Maximum power	Memory
Calxeda EnergyCard	16	1.1 GHz	25 W	4 GB
2× Intel Xeon E5-2660	16	2.26 GHz	2× 95 W (max. TDP)	32 GB

Table 2.10: Selected Workloads

Dwarf [13]	Selected workload
Dense and Sparse Computations	High Performance Conjugate Gradient, MrBayes
Structured grids (Represented by a regular grid; points on grid are conceptually updated together. It has high spatial locality.)	Cellular automata (Conway's game of life)
Map-Reduce	HiBench, PigMix, Starfish
Bioinformatics	AbySS, FASTA

Intel server (12–16 cores). The 4 nodes are packaged in a Calxeda EnergyCard, and should consume less than 30 W, where the Intel processors consume about 95 W, and 315 W at 75% utilization (estimated with the HP Power Advisor tool). The assumption is that both blocks hold about the same core count, and are building blocks for larger systems.

Selected workloads

The benchmark applications selected are inspired by [13], where the authors identify computation and communication patterns ("dwarfs"), as opposed to full applications or micro benchmarks. These patterns define classes of computation and communication pattens. Moreover, they list computation patterns that attempt to cover most of the applications today. To this list, we added benchmarks from the bioinformatics field.

We selected representative applications for several dwarfs.

The new High Performance Conjugate Gradient benchmark [102], developed as a modern alternative for the well-established Linpack benchmark, as implemented in the reference HPL. The design goal of HPCG has been to achieve a better correlation to real scientific application performance, by stressing a systems memory, network bandwidth and latency, balance and scatter/gather features which have a greater impact on general application performance than the compute rich, dense matrix computations exhibited in HPL. MrBayes [103] is a popular bioinformatics application for the Bayesian estimation of phylogeny, using Markov chain Monte Carlo methods.

Table 2.11: HPCG v1.1, MrBayes v3.2.2

Workload	Calxeda performance	Xeon performance
HPCG	0.75 GFlops	2.07 GFlops
MrBayes	4,200 s	400 s

Conway's game of life is a two-dimensional cellular automaton, where rules have a 9 cells neighborhood, and the grid is updated synchronously. The implementation is MPI-based. One node acts as the master, while the others perform the cell transitions. The decomposition is data-parallel, where the cells are equally split (approximatively) across the slave nodes. The chosen grid size is 200×200 cells. The simulation ends with 500 generations (all cells updated 500 times).

The first Map-Reduce applications are selected from HiBench [104]: Word Count (mostly CPU bound), TeraSort (CPU and I/O bound) and PageRank (balanced workload). The second Map-Reduce workload is PigMix [105]. PigMix is a set of Pig programs that are used as a benchmark to measure the comparative performance of the Pig programming language in a Hadoop environment. Apache Pig is a platform for analyzing large data sets that consists of a high-level language for expressing data analysis programs. Pigs infrastructure layer consists of a compiler that produces sequences of Map-Reduce programs. Pig is used to write complex Map-Reduce transformations using a simple scripting language, Pig Latin, such as aggregate, join and sort. The last Map-Reduce application is a TF-IDF benchmark from the StarFish project [106]. TF-IDF stands for term frequency-inverse document frequency. Both Map-Reduce benchmarks are implemented in Java.

AbySS, Assembly By Short Sequences [107], is a de novo parallel paired-end sequence assembler, developed to enable the efficient assembly of the vast amounts of data generated from large-scale sequencing projects, such as the sequencing of individual human genomes to catalog natural genetic variation. FASTA is a suite of bioinformatics applications [108] used to find regions of local or global similarity between Protein or DNA sequences. Since its introduction in 1985, it has become widespread in the bioinformatics community, seeing constant development and its FASTA file format becoming a de facto standard for describing bio sequences.

Performance results

Table 2.3.2 summarizes the results for the first two benchmarks. Surprisingly, the Calxeda SoC performs well compared to the single node 16-core Xeon. However, a single Calxeda node achieves only 0.19 GFlops, but with only 4 cores. For the MrBayes test, Calxeda performs an order a magnitude worse than the Xeon node. For most of these results, their statistical significance lies with the duration of the runs (HPCG requires about several hours

Table 2.12: Structured grid (cellular automata)

Machine	runtime
Calxeda	1.25 s
Xeon E5-2660	0.8 s

Table 2.13: Map-Reduce (Hadoop/HiBench)

Workload	Calxeda runtime (average)	Xeon runtime (average)
Word Count	395 s	301 s
TeraSort	88 s	186 s
PageRank 50 K	323 s	723 s
PageRank 1 M	877 s	1,067 s

to complete). The long running times should cancel the micro variations in the executions.

Table 2.3.2 presents the main result from the experiments, the Calxeda 4 node performance is surprisingly good given the frequent coordination (required by the synchronized updates) which should penalize the slower Calxeda interconnect.

The parameters for Word Count are: data_size = 500 MB, num_maps = 16, num_reds = 48. For TeraSort: data_size = 100 MB, num_maps = 48, num_reds = 24. For PageRank, data_size = 800 MB and 300 MB, num_maps = 96, num_reds = 48, num_iterations = 3, block = 0, block_width = 16.

The first results from the Map-Reduce HiBench applications show that, except for Word Count, the Calxeda ARM board performs better than the single node Xeon. This is driven by the CPU intensive nature of Word Count, where the faster and more advanced CPU helps.

Table 2.3.2 summarizes the performance results for the two additional Map-Reduce benchmarks. The first two columns report the runtime over 16 cores, while the two last columns report the runtime over 4 cores. The difference lies with the number of data nodes configured. For 16 cores, Calxeda is configured with one data node per node (4 data nodes), while for 4 cores, only one data node is used. For the Xeon node, only one data node was configured in both core counts. This configuration difference allows to

Table 2.14: Map-Reduce (Pig/Starfish)

Workload	Calxeda (16) runtime	Xeon (16) runtime	Calxeda (4) runtime	Xeon (4) runtime
PigMix	2,117 s	2,940 s	4,826 s	2,966 s
Starfish	4,383 s	7,555 s	17,060 s	7,038 s

Table 2.15: Bioinformatics

Workload	Calxeda runtime	Xeon runtime
AbySS v1.3.6	3,400 s	510 s
FASTA v36.3.6d fasta	240,000 s	50,000 s
FASTA v36.3.6d ssearch	50,000 s	7,500 s

show the performance difference per core, and also shows the good scalability of the Calxeda nodes.

The bioinformatics results of Table 2.3.2 show the poor performance of the Calxeda SoC compared to the Seon machine. Currently, the ARM SoC is not suitable for bioinformatics loads.

Discussion

We considered as building blocks for a larger system a Calxeda SoC, packaging 4 quadcore ARM A9 nodes, and a single node with two 8-core Xeon processors. The power consumption is an order of magnitude less for the Calxeda SoC, while the Xeon's clock is only twice the A9's.

The more balanced workload performs well on the ARM SoC (only three times slower than the Xeon). For example, HPCG aims to include computation patterns with lower computation-to-data-access ratios, irregular memory access, and fine-grain recursive computations. The same pattern is present in the cellular automata, where the computation-to-data-access ratio is also low. The Map-Reduce workloads are very favorable to the ARM SoC, as they perform even better than the single node Xeon. However, for the more CPU intensive workloads, such as the bioinformatics loads, the ARM SoC is too slow to represent a viable alternative, although about as energy-efficient.

2.4 Summary

The impact of energy on a cloud data center identified that critical power, the purchase price of equipment, and the poor energy-proportionality are the key factors to the TCO. AWN attempts to address precisely those factors. AWN is a lower-power alternative because power increases only linearly with capacity (as wimpy nodes are added to the cluster-like system). The huge market size of mobile computing have commoditized many computing components, lowering costs and attracting innovation. Finally, wimpy nodes are packaged in SoC, which package other components and improve the energy proportionality.

Even if green computing is possible with AWN, extracting performance from the AWN requires specific parallel algorithms. We have noticed that ex-

isting applications do not perform well on AWN, but that new applications, more parallel and distributed by design (such as Map-Reduce), perform well. This can be considered an instance of a more spontaneous co-design. Co-design [109] is the coordinated and deliberate design of both software and hardware in order to achieve an overall system quality. Here, wimpier nodes provide low-power alternatives for the cloud data center, prompting the re-design of Internet applications. An example are microservices [13], where Unix design patterns are provided with web protocols and leverage cloud computing, as pioneered by Netflix [14].

Finally, accelerators or special-purpose co-processors implement the AWN concept, but while they match the scale out of processing cores, they fail to do so for memory or I/O, and limit their benefits to some applications.

The next chapters present case studies of parallel designs to solve an optimization problem from the scheduling domain.

[13] `http://martinfowler.com/articles/microservices.html`
[14] `http://www.infoq.com/interviews/Adrian-Cockcroft-Netflix`

Chapter 3

Parallel Programming for Arrays of Wimpy Nodes

The usual meaning of parallel programming is to modify a program to benefit from parallel hardware while preserving the original algorithm (the underlying "recipe"). Except for lower-level changes (as performed by compilers), these changes are the result of a manual process. In this chapter, we experiment with this view of parallelism in the context of AWN. Section 3.1 addresses code-parallelism with software pipelining, and applies resource management optimization to extract performance from the AWN. Section 3.2 experiments with data-parallelism on two well-known algorithms from the optimization domain. We summarize our findings in Section 3.3.

3.1 Code Parallelism

In this section, we explore code parallelism in the context of AWN. Code parallelism extracts parallelism from instructions, as opposed to data. It is found in microprocessors' instruction-level parallelism. Processors provide instruction-level parallelism thanks to a number of architectural designs, such as: instruction pipelining, superscalar execution, speculative execution and out-of-order execution. Pipelining aims to improve the throughput, and not the latency, of a processor. The instruction cycle is decomposed into dependent steps, from 2 (controllers) to more than one thousand (Xelerated X10q network processor [110]), which can overlap in the processing of independent instruction cycles, as in a factory assembly line. A typical instruction cycle is: instruction fetch, instruction decode, execution, write-back. Compilers can help the processor by reorganizing the order of the generated machine code. Superscalar execution allows the execution of multiple instructions by multiple circuits. Speculative execution mitigates the delays of branches, by starting the execution of a code path before knowing if it will be taken.

Software pipelining [111] aims to offer the same benefits as processor pipelining, but at the software level. The chosen message-based decomposition is software pipelining (also called stream processing), can be traced back to D. Mc Ilroy in Unix shells. This technique is interesting because it can parallelize sequential codes, and is well suited to the Internet applications (as hosted in the cloud) where throughput is important. Once an application is decomposed into dependent steps, one question is where to locate each decomposed step in the AWN, so as to achieve a performance objectives, such as latency. This question can be stated as a scheduling problem. It relies on the a priori knowledge of the decomposed steps (or tasks, in the context of scheduling), and the AWN. We present methods for obtaining this a priori information [112] in Section 3.1.1.

The AWN, with it's performance limitations (Section 2), holds one potential benefit: they could reduce the contention present in modern architectures (because a large single machine is replaced with a cluster of smaller ones). This possible benefit is explored in Section 3.1.2.

With these elements, we proceed to analyze the opportunity of software pipelining on the AWN in Sections 3.1.3, 3.1.4.

3.1.1 Task Performance Prediction

Runtime task profiling approach

In order to make correct mapping decisions, the decision process needs to evaluate the different alternatives. This implies some prediction model of the performance of the tasks, on the distributed system available (for example, the AWN). This highlights a difference between the OS scheduler and the higher-level mapping considered here. The former cannot assess the overall service performance because their scope is the machine, and more generally because their objective is resource management (of hardware), rather than business performance. This limitation also occurs in multi-server (microkernel) OS [113], which can justify the use of such a predictor for a single machine.

Models to predict performance were usually derived from a detailed understanding of the inner workings of tasks and machines [114]. However, the growing complexity of these multi-core based computers do not lend themselves well to such an approach. These machines although sometimes considered parallel, actually share several components (memory, last level caches, I/O interfaces), which lead to contention [115, 116], as we will present in Section 3.1.2. Therefore, the actual performance of a task depends on the concurrent activity on other cores and processors. OS (virtual memory management, timesharing) and related tools (power management) further add to this hardware complexity. For example, paging faults considerably hurt an expected performance [117]. Modeling tasks is not simpler. They are

often considered to be defined by the source code in a high level language, however their behavior is set by compilers (optimization techniques) and runtime libraries. Moreover, some tasks can change behavior at execution time, in order to use a different amount of resource. For example: a task that relies on the slab allocator [118] can be requested to reduce its memory footprint. This prevents accurate prediction of a task's performance based on past performance in a different environment (different loads from different concurrent tasks). All these components prevent the accurate modeling of the computation, and the performance prediction of a given task on a given machine. Even if such an accurate model was developed, it would only be valid up to the next change in any of its components: hardware, OS, compiler, runtime library or task source program.

The question then is how to model task performance, when it depends on so many parts (each difficult to accurately model). Preferably, the prediction model should work for all possible machines and tasks. A possible approach is to avoid, as much as possible, a priori knowledge on the inner workings of the system, but should rather observe the execution of the tasks on the actual machines (hardware and related software such as OS). This is sometimes called runtime task profiling. The goal is not a descriptive, but a predictive model. This approach is based on a preliminary step to the mapping process, which measures the execution time of the tasks, on the targeted system, but during a limited time. In addition, the task profiling should explore the effects of contention, present in modern multi-core machines. This could be achieved by observing the task's performance when the machine is under various resource-specific loads, for a given profile run. Ideally, this extension should provide more accurate results but it ought not significantly increase the effort to build the model. The next section presents a review of previous efforts in this direction, grouped by field of application.

Distributed computing

As one of the first field to operate large computer infrastructures, HPC has identified the need for task performance prediction a long time ago. However, the original works focused around proprietary parallel machines, with dedicated system software. Since then, large infrastructure shifted to distributed systems composed of commodity machines and software.

The scheduling problem, presented in [119, 120], express the need for:

- profiling the tasks to execute,
- benchmarking the machines.

However it does not take into account the sources of contention in recent multi-processor, multi-core computers.

Several works [121, 122], and [123] investigate task runtime prediction from their past performance, on the same machines. This approach derives

from their context of application: a shared grid, where tasks controlled elsewhere are running on the same grid. This is a different problem from the one presented in this review. Here, the environment is completely controlled, but the problem is to map tasks in order to achieve specific results. Although this may seem an easier problem to solve, the problem of optimally mapping independent tasks on an heterogeneous system is NP-hard.

In [124], a Real-Time Advisor (RTA) predicts a task's performance. The RTA is used with a scheduler (real-time scheduler advisor), to place task on appropriate machines. The RTA predicts performance based on the observed runtime of the task when run on a vacant machine, and on the load of the machine where the task is planned to run. This formulation is due to the context for the prediction: how to map tasks on unreserved machines; which are under some load, outside of the mapper's control. Several aspects of the problem are similar to the problem defined in this thesis. Notably, the prediction of a task's runtime given a machine's load addresses the contention of shared resources. However, there are several differences. The tasks considered are computation-bound (busy loops), and therefore the shared resource is the processor. The machines are also single-core single-processor, so contention related to multi-core, multi-processor machines are not considered. Moreover, the predictor relies on the task's measured runtime, on the machine available, but only when run on a vacant machine (loadless). Finally, the scheduling problem is slightly different, the objective in [124] is to select the most suitable machine for a task, while the mapping problem is combinatorial: it tries to find the optimal mapping of all tasks, onto the vacant machines.

In [125], the authors design a prediction model for the grid, but opt to focus on a specific application in order to improve its accuracy. Applications run on a grid, where each computing node is a cluster. This is different from the context of this thesis, where a computing node is a machine. Contention at the machine level is therefore not included. Their approach is based on a limited run of the applications: executing the application against some input data, on one cluster. Predictions of runtime in other configurations (different data sets or clusters) are extrapolated from this initial measurement. A similar approach is reported in [126]. The host machine used for their study is a 32 processor Symmetric Multi-Processor (SMP) machine, but their work is applicable to other configurations. However, the tasks are quite specific. The execution model is that of work-stealing. A task is executed by the first available processor. It is not a long-running service, but rather a job, which is started and ends. A task can create other tasks. The objective of the study is to predict performance of the overall application. The tasks are profiled by monitoring their execution over a limited time. Memory contention is partially accounted for. The number of threads used for each task execution is varied, in order to assess the scalability of each task. A task which does not scale well reflects some contention (locking, cache conflicts,

etc) within the task. However, the objective is the task's scalability, and not the characterization of each task.

The problem of application performance prediction is studied in [127]. The authors propose an exploratory phase, called the pro-active training phase, which consists of running the tasks on the machines. They identify the cost of this training phase as a problem, and develop a method to minimize this effort. The method suggests to run the application on all nodes for a subset of the input values, and on the fastest machine for the full set of inputs. This formulation assumes a HPC application, which is run over a range of inputs. The data collected is then used for prediction of the real application on various machines. The problem formulation and the HPC setting differ from the problem defined here, but the approach to rely on actual executions and their observed runtime is very similar. The effort to minimize the preliminary phase is noteworthy, even if the solution is HPC specific. Moreover the question of contention in multi-core based machines is not considered.

In [128], the runtime of a task on a machine is predicted from the load of the task in isolation, the characteristic of the machine and the current load of the machine (due to other tasks). This prediction also relies on 5 rules that capture the interaction of tasks. This recent contribution is interesting because it bases its prediction on actual task execution, and limits their cost by only executing tasks in isolation. It also casts the problem in the context of quality of service, which is the end-user perspective included in the proposed direction. The literature review is also noteworthy. However, there are differences between approaches. The model used to predict task load does not address contention in multi-core, but mentions it as an OS concern. The tasks considered are essentially CPU-bound, because this allows the authors to link host load to task runtime. While this may be true for CPU-bound tasks, it is most unlikely in memory, network or any other contention prone situation. The domain of application for the profiling is HPC. Therefore the tasks are considered long-running (measurements are based on 5 second sampling), and directly related to a user id on the machine. The tasks in the present paper are different from these HPC tasks, because they are the instructions necessary to process a request, as part of a daemon or service. The 5 rules necessary to predict runtime from exploratory data are based on an understanding of the machines and the nature of task execution, which represents a big assumption.

A performance predictor, Dimemas, is presented in [129]. This simulator relies on execution traces of applications, on some characterized hardware. It can then predict the runtime of the same application on different hardware. The CPU burst, and network activity is considered in the model. The context is different from the one presented here, because the tasks are MPI-based HPC applications, and the varying hardware environment is the network performance. Therefore, a specific model for the task is used, which

is not the case in the proposed setting.

Runtime prediction is classified into three groups: code analysis, code profiling/analytic benchmarking, and statistical methods [130]. From such classification, the authors present a hybrid approach: statistical and analytic benchmarking. This classification clearly exposes different approaches to task performance prediction. However, two hypothesis in this work define a different problem to the one studied here. First, each task is assumed to have exclusive use of the machine on which it runs, such that a task's execution time does not depend on other tasks. This is clearly different from the problem defined here, where contention for shared resource, by concurrently running tasks, is one key hypothesis. Second, this previous work considers that a task execution time depends on its input data. This is perhaps specific to HPC environments. However, there is no such hypothesis in this thesis.

Thread scheduling

The method of profiling tasks based on their actual execution is proposed in [131]. This work looks at the execution of multiple threads of a process. It aims to identify data dependencies between threads. Data dependency occurs when multiple threads access the same data. Although the method is based on runtime analysis, compared to static source code analysis, it does not include machine characteristics, and does not consider contention beyond the data that threads share.

In [132], the relation workload and a machine's resource utilization (such as memory, CPU) is explored. The target application is capacity planning. The method relies on measured execution (called automated profiling). However, the relation sought does not involve performance estimation, because of the intended application in capacity planning. It does not consider contention in multi-core based machines.

Scheduling for simultaneous multi-threading architectures

The next papers consider how schedulers can improve the performance of threads when executed on Simultaneous Multi-Threading (SMT) architectures. SMT improves ILP by executing different threads at each cycle. The consequence is that some threads will achieve greater parallelism when co-scheduled together than other combinations.

This question shares some similarity with our question. Contention is present in SMT; however it is possible to minimize it in order to achieve greater performance. This depends on the nature of the threads (which are called tasks here). Co-scheduling threads on an SMT processor is analogous to mapping tasks on the same multi-core processor (or multi-processor machine). However, there are differences which prevent a direct application of the results from this field. The contention in SMT is limited in scope

(processor core), while this thesis places no restriction on the sources of contention within a machine (hardware and software stack). In addition, we are investigating a predictor for task runtime, while a SMT scheduler is concerned about processor utilization, a lower-level information. Nevertheless, some methods from the field of co-scheduling for SMT could be applied here.

A SMT scheduler [133, 134] minimizes contention on a superscalar processor, to improve utilization and performance of the threads. Their scheduler initially co-schedules threads according to fair policy, and then attempts to discover which threads run well together, by deliberately changing the co-schedule and observing the resulting performance. The adaptive nature of the approach is unsuitable to our question, because the unsuccessful co-schedules would impact the QoS and fail the SLA. Moreover, the combinatorial space of co-schedules is so large so as to make the above risk likely (because the scheduling is not limited to a SMT processor, but to an entire distributed system).

The target of [135] is the Simultaneous Multi-Threaded cores platform, such as Intel's HyperThreading. The objective is thread scheduling to reduce contention. This scheduling can either be performed at the CPU level, or at the OS level. The model is based on measurements of a real thread execution. However, there are notable differences. The approach is not exploration based (it does not require a preparation step which explores the behavior of tasks), but adaptive. The measurements deal with resource usage, such as caches and registers. Their model is based on the detail knowledge of the internals of the SMT processor, and therefore uses of a simulator to obtain results. The approach presented here tries to avoid both this knowledge and the use of a simulator.

Real-time and embedded system scheduling

In [136], the authors propose an energy-efficient soft real-time scheduler. The scheduler relies on runtime predictions, based on limited task execution, which is the approach considered in the present review. Soft real-time (meeting a fraction of all deadlines) expresses the problem of meeting SLA requirements, because SLA typically allows some deadlines to be missed, and sets penalties when more deadlines are missed. The deadlines capture the end-user view of performance. However, contention is not part of the model (which focuses on the CPU, for specific tasks). Also, energy efficiency is a consequence of DVFS control by the scheduler, which is not necessarily better than the race-to-idle policy. Finally, this scheduler is the finest grain OS scheduler, which operates at a lower-level than in our context.

A real-time scheduler where activities are subject to resource constraints is presented in [137]. The constraint is that shared resources are accessed sequentially. They mention both physical resources, such as disks, and logical

resources, such as critical sections guarded by mutexes. Only one task (according to our definition, not theirs) can access a shared resource at a time. Tasks are defined with statistical properties. However, how such properties are obtained is not described; it is likely that they are derived from actual executions. However, the task properties do not include resource contention. Constraints on shared resources are managed through scheduling access to resources, under a given model. Although their study does not match this review's scope of task profiling, it does address the higher level question of scheduling tasks under resource contention.

Internet Protocol routers

Some closely related works from a different field than data center computing are presented in [138, 139, 140]. These papers present and study programmable internet routers, based on network processors. One of the main issue with programmable routers is the mapping of tasks to processors.

In [138], the suitability to internet routing of different machine architectures is reported. They do mention contention as a critical bottleneck in network processors, but it is not part of the profiling or mapping study.

Dynamic profiling to support task mapping is proposed in [139, 140]. This profiling aims to characterize the tasks. The dynamic profiling is motivated by the variability of the input traffic, both in volumes and nature. Contention is not part of the study, although it is presented in their previous paper.

3.1.2 Task mapping with resource contention

In this section, we investigate the impact of resource contention when executing concurrent tasks, co-located on the same machine. This investigation is conducted by modeling contention to shared resources in modern machines, and DVFS, and simulating it with a modified scheduling heuristic [141].

Design innovations on modern processors allow multiple execution cores to be integrated into a single processor, with each core having its own resources. Yet, some interdependence between cores need be taken into account for optimal performance and energy efficiency. At different levels, cores in a processor share some resources leading to contention if applications running in parallel on the cores compete for the shared resources. Contention can induce not only to degrade the application performance, but also to inefficient use of energy, since cores waiting for a resource to become available dissipate energy without carrying out progress [142].

For example, we have carried out a simple experiment to demonstrate how contention for shared resources can slow down the application performance. In this experiment, we have executed one benchmark application (stream [143]) on a 2.4 GHz Intel Core 2 Duo. This application is main

memory bound. First, we have executed the application activating only one core and we measured the time that the application need to complete its execution. It took around 10 seconds. Then, we ran two stream applications in parallel on the two cores sharing a memory domain. The completion time of both applications was around 20 seconds because of the contention. This example clearly shows the negative effects on the performance and energy consumption without any consideration of contention on multi-core processors.

We are interested in the efficient executions of the applications on shared resources based computing machines with the aim of minimize both the energy consumption and the completion time. Since the resource manager is the component of a system responsible for deciding which application run on the processors simultaneously, resource allocation and scheduling are crucial for performance and energy efficiency.

We explore the impact of contention by simulating a resource allocation which models contention caused by concurrent tasks. The resource allocator is a heuristic based on the Min-Min resource allocation algorithm [144, 145]. The algorithm consider the multi-objective problem. This algorithms first compute the completion time for each job on each core. The core that has the minimum completion time for each application is selected (it corresponds to the first Min objective in the algorithm and is based on estimated completion times for the applications). Then the application with the overall minimum completion time is selected and affected to the machine or the core (it corresponds to the second Min objective of the algorithm). And this process is repeated. We have modified the first function of the Min-Min algorithm by adding a parameter that take into account the performance degradation on the completion time for the application when it is in memory contention conflict with another application that is in execution.

Problem statement

We consider computing architectures with a set P of M heterogeneous processor packages each package containing a set of m homogeneous/heterogeneous cores. The cores in general will be heterogeneous in time and energy requirements. These cores share last level cache and memory. One example of this kind of architecture could be machines with large shared memory having two 2.5 GHz Intel Xeon E5000 processor package series of two or four cores each. Each core may only execute one task at a time (i.e., no multi-tasking). Furthermore, we assume that processor packages are equipped with DVFS features.

We assume a set of independent tasks. They have to be individually processed by a single resource (*non-preemptive* mode). The tasks could specify hardware and/or software requirements over resources. These tasks could be memory bound and/or CPU intensive. For each task, we assume

that an estimated time to compute (Expected Time to Compute (ETC)) on each core has been provided. This ETC is estimated without consideration of contention. However, in our model we provide the resource allocation heuristic with a mechanism that penalize the objective function in the case of contention. An the final assignation of tasks to processing elements are done taking into account this situation. More precisely, assuming that the computing time needed to perform a task is known (assumption that is usually made in the literature [145, 146, 147]), we use the ETC model by Braun et al. [145] to formalize the instance definition of the problem as follows:

- A *number* of independent *tasks* to be scheduled.
- A *number* of heterogeneous *machine/cores* candidates to participate in the planning.
- The *workload* of each application (in millions of instructions).
- The *computing capacity* of each machine/core (in *mips*).
- Ready time $ready_m$ indicates when core m will have finished the previously assigned tasks.
- The Expected Time to Compute (ETC) matrix ($nb\_tasks \times nb\_machines$) in which $ETC[t][m]$ is the expected execution time of task t on core m.

In terms of resource allocation problem, the goal of this work is to assign the tasks on P, the set of mM processor packages so that the total completion time (i.e. makespan) of the last executed application over all the machines is minimum and the efficient use of energy is maximized. The completion time of a core m is defined as the time in which core m will finalize the processing of the previous assigned task as well as of those already planned tasks for the processing elements. This parameter measures the previous workload of a core. Notice that this definition requires knowing both the ready time for a core and the expected time to complete of the tasks assigned to the machine.

The proposed approach

The proposed solution is based on the Min-Min resource allocation algorithm [144, 145]. It is a two phase greedy heuristic. The algorithm starts with a set of all unmapped tasks and iteratively assigns tasks to processing elements by computing their expected minimum completion time. For each task this is done by first tentatively scheduling it to each core and estimating the task's completion time on each core. Also for each task, a metric function f_1 (i.e., the core that has the minimum completion time for

each application is selected) is computed over all expected completion times. Then the (task, core) pair with the best metric match is selected by using a selection function f_2 (i.e., the task with the overall minimum completion time is selected). After that, the task is mapped to the core. Again, this process is repeated with the remaining unmapped tasks.

We modified the Min-Min algorithm to adapt it to the contention problem. The modification lies essentially with the calculation of the quality of a mapping of a task to a core. The new scoring function takes into account potential memory contention and the voltage-frequency scaling of the individual cores and processors. In addition to all the task-to-core mappings considered, we consider the different voltage-frequency operating points.

The score of each mapping is calculated with

$$score = \alpha \times MCT + (1 - \alpha) \times Energy, \tag{3.1}$$

where MCT is the time the core takes to execute all its currently assigned tasks, and α sets the tradeoff between the objectives: energy and MCT. The energy spent executing the this task is calculated with the relation

$$Energy = V^2 \times CT, \tag{3.2}$$

where CT is the time needed to execute that task on this core. The total completion time is the sum of the completion time of each task, on this core is

$$MCT(m) = \sum_{i=1}^{n} CT_i(m). \tag{3.3}$$

As an hypothesis, the time needed to execute each task t on each core c, $CT_i(c)$, is considered known. These times are based on a dedicated machine, at maximum processor speed. They are presented in an estimated time to compute (ETC) matrix. The machines in our model are heterogeneous. The model is also inconsistent: one core may be faster than another to compute one task, yet slower than another core for another task.

The memory contention and voltage-frequency scaling are introduced as modifying factors to the ETC. Both of these can degrade the completion time of the task, as described in the following subsections.

Memory contention penalty The shared resource we model is the access to main memory. We define a memory contention when several memory-bound tasks are concurrently executed on the same machine. We suppose that the memory requests are not met with the various cache memories. Min-Min maps tasks to cores or processors, without specifying any execution order. This prevents the identification of a concurrent execution of memory-bound tasks. To reconcile these two views, we use a statistical approach. When considering the mapping of a memory-bound task to a core,

Parameter	Value
Tasks	36
Cores	6
Distinct machines	4
% of memory-bound tasks	50

Table 3.1: Parameters

we calculate the percentage of the time the other cores on the same machine spend executing memory-bound tasks. This reflects the probability for this task to execute concurrently with another.

$$Penalty_M = 1 + \sum_{c=1}^{n} pct_c. \tag{3.4}$$

The original ETC time is then multiplied by $Penalty_M$, to reflect the impact of memory contention. For example: on a 2 core single processor, if one core is executing only memory-bound tasks, then mapping another memory-bound task on the other core results in doubling the basic ETC for this task on this core. As mentioned earlier, this is in line with actual measurements made with a memory intensive benchmark application.

Voltage-frequency scaling penalty We assume that each core can operate at a distinct voltage-frequency. The frequency directly influences the performance of the core (lower frequency means greater CT), and the voltage its energy consumption. We suppose that each core of a processor can operate at different voltage-frequencies, and that memory-bound tasks are not affected by frequency changes; their CT do not change.

Simulation

In this section we present the setup for our simulations, and the results obtained.

Parameters The parameters are summarized in 3.1. There are four machines, two of them are composed of a single core processor, and the other machines of two cores or SMP processors. About half of the tasks are memory-bound.

The model for our voltage-frequency scaling is presented in table 3.2. These voltage-frequency operating points are arbitrarily chosen.

Results Table 3.3 summarizes the results obtained with our proposed algorithm across several weights (α), and a point of comparison with a model

Operating point	Performance penalty	Voltage (V)
0	×2.0	0.5
1	×1.5	1.0
2	×1.0	2.0

Table 3.2: Voltage-frequency operating points.

α	Makespan (time)	Energy (J)
0.5	310	380
0.8	240	880
1	205	2200
Min-Min	264	1055
(baseline)	195	

Table 3.3: Effect on contention and DVFS on task mapping

that ignores contention and DVFS (baseline model). We then correct the baseline Min-Min solution by taking into account the memory contention effect, as described in 3.1.2. The baseline makespan, when accounting for contention and DVFS, increases the makespan from 195 to 264. This adjustment also impacts the energy computed in the baseline model.

Higher values for α favor the makespan, while lower values favor energy. The α value of 0.8 provides the best trade-off between makespan and energy. We observe that our model for contention and DVFS influence the mapping founds (makespan ranges from 310 to 205), and impacts the original baseline results. Although this is based on a set of assumptions, these effects should not be ignored in mapping tasks to cores, and shows the potential benefit from using contention-free architectures, such as AWN.

3.1.3 Pipeline mapping on AWN with contention

The previous sections provided some background information which we can use to explore code-parallelism on AWN. This first study presents simulation results for the mapping of pipelined tasks onto a AWN, in order to assess its potential benefits [148].

The approach is to develop a model of the current multi-core processors, the tasks intended to run on these multi-core processors and their OS. The model is intended for middleware tools, such as schedulers. Multi-core processors suffer from contention for shared resources, such as main memory, which may significantly affect the expected performance and energy consumed. Moreover, current operating systems such as GNU/Linux timeshare

the processor cores, in order to overlap blocking I/O with computation. The kernel typically includes power management services which control the voltage/frequency scaling automatically, based on user specified policies (governor). The proposed model accounts for these facts, and therefore enables the scheduler of a data centers to make better decisions.

Contention is defined as the concurrent access to a shared resource within a machine [141]. This concurrency results when multiple cores execute different processes simultaneously, which may request access to the shared resource. Examples are: network interface, main memory, last-level cache, co-processors (such as a graphical processing unit, GPU).

In this section, the new model helps evaluate an alternative processing platform, AWN. It is important to note that the AWN processors can be multi-core themselves, but with a smaller number of cores than typically found in data center servers. The limited number of cores of AWN reduce contention, by reducing shared resources. A practical drawback to the millicluster comes from its distributed nature: it is a cluster. Each AWN has limited computing performance, therefore applications must be decomposed and distributed. This will break the binary compatibility for some programs. We address this requirement by suggesting to apply a well-known decomposition pattern; pipelining.

In this section:

- We propose a new model for multi-core processors that accounts for self-regulated DVFS by the operating system and time sharing of the processors by the operating system.
- We adapt a well-known off-line scheduling algorithm to this model, with the additional objective of energy efficiency. The main focus of this study is contention modeling, in order to improve scheduling accuracy. Therefore its benefit can only be assessed in combination with a scheduling algorithm. this thesis proposes such a scheduling algorithm, which aims at both maximum performance and minimum energy consumption. The algorithm retains the qualities of the original version, but extends it for energy minimization while applying the proposed model.
- Provides simulation results for different architectures, based on the new model, to put AWN proposal in perspective.
- Suggests and evaluates the decomposition pattern of pipelining, to port existing applications to AWN.

Problem definition

The context of the problem addressed in this study is the efficient mapping/scheduling of tasks to a set of processing elements. Mapping/scheduling

is defined as the procedure that assigns tasks (executable code) to some computational resources, the processing elements, and define a date at which the tasks should be executed, to optimize a performance objective. A processing element is a processor core, either a single-core processor, or one core of a multi-core processor. These cores can be considered components of a cluster of independent computers, or a computational grid. Efficiency refers to the fulfillment of some predefined objectives. This study considers dual objectives: (a) performance and (b) energy. The objective of a mapping algorithm is therefore to identify a mapping that delivers good performance at low energy consumption levels.

The field of scheduling has provided many algorithms to perform this mapping activity. However, all cluster computing and grid computing mapping algorithms rely on a simple model for the tasks and the processing elements.

Processing elements As already mentioned, the set of processing elements are processor cores. The cores are packaged in processors, which are themselves grouped in a computing machine. this thesis supposes that each core is capable of dynamic voltage frequency scaling (DVFS); that is, it can be operated on a set of supply voltages and different speed performance (associated to different clock frequencies) [149, 150]. DVFS seeks to exploit the convex relationship between the core supply voltage (that impacts the speed of execution) and the energy consumption. Moreover, different cores of a same processor are assumed able to independently operate at different voltage/frequency points. This assumption is used by the mapping algorithm.

Tasks A task is executable code, which is started to perform some amount of work and then stops after completion. For each task, an estimated time to complete (ETC) their work, on each of the different processors, is provided. The ETC matrix ($nb\_tasks \times nb\_cores$) in which $ETC[t][c]$ is the expected execution time of task t on core c. These times are provided for each task in isolation of the other tasks, without any consideration for contention for a shared resource. This assumption is often made in the literature [145, 151, 152].

In this work, the ETC of a task is further broken into three parts:

- I/O part, this represents the latency associated with an I/O operation;
- contention part (such as memory operations), this is the time spent in contention prone operations;
- a pure CPU part, the rest of the instructions of the task.

This classification is based on the instruction mix of the tasks. Memory intensive tasks perform significant load and store instructions to main memory. On the contrary, CPU-bound tasks perform little load and store instructions.

Energy-efficient mapping This study considers the energy-efficient execution of the tasks on shared resource based cores with the aim of minimizing both the energy consumption and the overall performance.

Performance is defined by two complementary indicators: the flowtime and the longest task time. The longest task is defined as the time needed for the longest task to complete. The inverse of Longest Task Time (LTT) is throughput. Indeed, if more than this throughput is queued into the application, then queues will build up in input of all the tasks which are too slow. By choosing a throughput based on the slowest task, queues can not build up. This is particularly applicable to pipeline systems.

Flowtime is the sum of the runtime for all tasks across the different cores.

$$FT = \sum_{t} ETC[t][c] \ , \tag{3.5}$$

where c is the core to which task t is mapped.

The energy E and power P relations used in this thesis are derived from the power consumption model in digital complementary metal-oxide semiconductor (CMOS) logic circuitry. Energy and power are defined as follows:

$$E = P \cdot T \ , \tag{3.6}$$

where T is the time needed to perform an application.

The mapping problem is therefore a multi-objective problem, where the objectives conflict with each other [153, 154]. That is, the aim is to maximize the performance of the system (i.e., to minimize the LTT and flowtime) while minimizing the energy consumption. This bi-objective problem can be approximated as a (scalar) single objective problem, defined as a weighted average of the two objectives. This is a traditional approach used to solve multi-objective problems. This way of tackling multi-objective optimization problems is widely accepted despite of its main drawbacks: only one solution from the Pareto front (a set containing the best non-dominated solutions to the problem) is found in each run, and only solutions located in the convex region of the Pareto front will be found. However, the use of the weighted function is justified in our case by the convex search space of the studied problem and also by the need of providing a unique solution, since there is not any decision maker to select the most suitable solution from a set of non-dominated ones. In the weighted approach, each objective is multiplied with a weight representing its importance. In this work, the weight for the LTT and flowtime is α, with real values in the interval [0...1]. The weight

for the energy consumption is $1-\alpha$. The α parameter is chosen to represent the relative importance of one objective or the other. This choice belongs to the user of the mapping algorithm. So, each the output to the mapping algorithm, with specific voltage/frequency points, can be evaluated with:

$$score = \alpha \cdot MS + (1 - \alpha) \cdot E \ . \tag{3.7}$$

Because the mapping algorithm aims to minimize both objectives, the optimal mapping is the one with the minimal *score*.

Model

This section presents the model of a processing element, a task, and the operating system services. This model is an extension of an existing model [141], which introduced the effect of memory contention.

The model takes as input:

- the proposed assignment of a task to a core,
- the current state of the mapping: the currently assigned tasks.

The model outputs three values:

- the flowtime for the resulting schedule;
- the LTT in the schedule;
- the associated energy consumption.

The model is not specific to any shared resource of a multi-core processor for which contention occurs. However, in order to clarify the presentation of the model, this thesis will choose a source of contention. Given the dominant impact of main memory access [155], for the rest of the paper, the shared resource modeled is that of main memory.

ETC modifiers Section 3.1.3 introduced the ETC. This time is provided for a task in isolation of any other (the task is running by itself on the machine), and at nominal voltage/frequency.

In the proposed model, the ETC of each task is decomposed into a latency part (for I/O), a contention-prone part, and the rest, a CPU part. This proportion can be derived from the instruction mix of the program, by examining the machine instructions to detect those which access the shared resource. In the case of contention for main memory, this would amount to the number of memory stores and loads. Alternatively, this part can be determined by analyzing the runtime behavior of a task. In an analogy with Amdahl's law for speedup under parallelization, the runtime T of a task can be rewritten as:

$$ETC[t][c] = T_{mem}[t][c] + T_{lat}[t][c] + T_{cpu}[t][c], \forall c \ , \tag{3.8}$$

where T_{lat}, T_{mem} and T_{cpu} are the runtime of the latency, contention-prone and CPU parts respectively. They are defined as follows:

$$T_{mem}[t][c] = \beta_t \cdot ETC[t][c] \forall c \ , \tag{3.9}$$

$$T_{lat}[t][c] = \gamma_t \cdot ETC[t][c], \forall c \ , \tag{3.10}$$

$$T_{cpu}[t][c] = (1 - \beta_t - \gamma_t) \cdot ETC[t][c], \forall c \ . \tag{3.11}$$

The contention factor is proportional to the memory parts of the other tasks running on the other cores of the same machine (which is the scope of our contention, section 3.1.3). Tasks running on the same core are not subject to contention because they do not run concurrently, but are preempted by the OS. This effect of contention on T_{mem} is approximated by the sum over all the other cores of the machine, of the proportion of runtime spent in memory related activity. Let N represent the other cores of the same machine, and β the ratio of memory runtime, then:

$$T'_{mem} = T_{mem} \cdot (1 + \sum_{c}^{N} \frac{\sum_{i}^{tasks} \beta_i \cdot ETC[i][c]}{\sum_{i}^{tasks} ETC[i][c]}) \ . \tag{3.12}$$

This statistical estimation is independent of the actual ordering of the execution of tasks on the different cores in the machine, because this is assumed to be the responsibility of the OS scheduler.

The time spent by the CPU waiting on the completion of an I/O operation is modified at this stage by the location of the endpoints to the communication. The target software architecture is a pipeline, therefore, the communication is uni-directional. If the location of the endpoints in this communication are on the same physical machine, then fast inter-process communication (IPC) are considered used transparently by the tasks. An example of such IPC is shared memory, and the appropriate synchronization primitives. In this case, the time allocated to latency is removed.

Finally, the modified ETC of a task is then:

$$ETC'(t, c) = T'_{mem}(t, c) + T'_{lat}(t, c) + T_{cpu}(t, c) \ . \tag{3.13}$$

Performance evaluation Performance was defined as LTT and flowtime in Section 3.1.3. These indicators are global, in the sense that they can only be computed on the entire schedule (unlike the ETC modifiers). When computing LTT, the latency is ignored when adding times, unless there is only one task allocated to a core. This is because if multiple tasks are assigned to a core, the OS scheduler can run them alternatively, to overlap I/O with computation (in the case of a single task, this is not possible). Both flowtime and LTT use the modified ETC values for each task, due to memory contention and local IPC.

Energy evaluation Energy-efficient scheduling using DVFS often consider that a scheduler should specify the voltage/frequency point of operation of each core. In this section, we depart from this approach. Indeed, inspection of the kernel power management tools of the GNU/Linux kernel version `2.6.35-24`, reveals that DVFS is very dynamic and self-regulated. Default values for the On-demand governor show a sampling rate of 10 ms (time period when a DVFS change is considered). Fundamentally, the complexity of a cluster, grid or cloud is such that whenever possible, local decision making should be preferred over a global one. In this case, the regulation is based on CPU utilization, which is also under the control of the kernel.

We suppose that the OS manages power using the `cpu-freq` tools under the on-demand governor. The on-demand governor implements the race-to-idle policy. Whenever there is a need for CPU, then the voltage/frequency is set to its maximum value. Later, when the utilization decreases, the voltage/frequency point of operation is chosen so as to match the needed load.

The makespan is used to compute the total energy spent executing the tasks. Makespan is defined as the maximum latest completion time CT over all the resources used in a mapping. Formally, for a processing element c, the completion time of c without consideration for contention, is defined as:

$$CT[c] = \sum_{t \in S(c)} ETC[t][c] \ , \tag{3.14}$$

where $S(c)$ is the set of tasks assigned to core c. The makespan MS is then defined as:

$$MS = \{\max\{CT[c] : c \in cores\} \ \}. \tag{3.15}$$

When busy, the cores are supposed to be operating a maximum voltage/frequency, under the on-demand governor. When a core has completed its tasks, then it switches to the lowest voltage/frequency. Formally:

$$E = BL \cdot N \cdot MS + \sum_{c} (P_{high} \cdot CT + P_{low} \cdot (MS - CT)) \ , \tag{3.16}$$

where BL is a constant power term, N is the number of machines powered on, P_{high} the CPU power consumption when operating at maximum voltage/frequency, P_{min} the CPU power consumption when operating at minimum voltage/frequency. A machine which is not used (no task assigned) is considered powered off.

Model parameters The previous subsection presented the model from which the required parameter list can be derived. It is presented in Table 3.4.

Table 3.4: Summary of parameters of the model

Parameter	Definition	Example value
α	weight for the performance objective	0.6
ETC	estimated time to complete a task on a core, in ms	1230.5
β_t	ratio, per task, of memory related operations	30%
γ_t	ratio, per task, of latency	20%
P_{low}	power at minimal DVFS operating point	10 W
P_{high}	power at minimal DVFS operating point	100 W
BL	constant power term	200 W

Applications of the model

The model presented is used to compare the relative performance of a two dual-core server and a AWN.

AWN Current many-core processors propose the view of a unified and shared memory. This leads to complicated memory path design which still requires great care from the programmer. Yet, this increased programming effort provides diminishing returns in terms of performance [13]. The well-known alternative to shared memory parallel programs is the message passing parallel model [156]. This model of parallel programming is widely used in the supercomputing area, where many distributed machines are typically needed. In addition to the large distributed system, this message passing model is effective when managing complexity, by forcing the decomposition of larger applications into many, smaller, independent lightweight processes [157]. Although decomposition is a well-known and frequently used technique to face complexity, it is also necessary when developing reliable programs, as highlighted by the practice of formal methods [158]. The decomposition pattern chosen is the pipeline. A software pipeline is simply a series of tasks, connected via uni-directional communication links. The advantage of a pipeline is to increase throughput, at the cost of increased delay for completing all the steps in the pipeline. Pipelining offers to increase the performance of sequential programs. So, this thesis supports the suggestion to combine the hardware devices from the mobile computing area with the middleware ideas and software from the distributed supercomputing area.

Suggesting alternatives to the current multi-core processor (with its ever increasing number of cores) design may appear a risky proposition. But, the current multi-core design does not appear to be the definitive answer to the problem of increased performance [13]. An implicit objective of the current design of multi-core processors is binary compatibility with existing pro-

Table 3.5: Processor specifications for platform comparison

Parameter	dual core	AWN
ETC		
perf. ratio	1	10
P_{low}	10 W	0.1 W
P_{high}	100 W	2 W
BL	200 W	20 W

grams. This objective conflicts with the other objective of designing parallel machines. The motivation for parallelism comes from, among other aspects, the combined limitations ("walls"), in memory access, power consumption and instruction-level parallelism. This leads to the formula [155, 13]: Power-wall + Memory-wall + ILP-wall = brick-wall.

It is interesting to point out that AWN can help address some of these limitations, and address the parallelism objective better then the current multi-core design. Indeed, many low-power components allows for a finer grain control of the power consumption in the cluster. Memory access and ILP can also operate in parallel in each AWN.

Comparison configuration The model proposed in this thesis will help to conduct a comparison of the different platforms. The first platform is two processor machine, each processor is dual-core. The second platform is a four node AWN, where each node is composed of a dual-core ARM processor.

The dual-core is based on the Intel 5400 series Xeon processor. The AWN is based on the ARM A9 Cortex processor. Table 3.5 summarizes the parameters for each platform. The ETC performance ratio indicates the relative speed of each processor. The ARM A9 is considered ten times less powerful than the Intel based processor. The power specifications are taken from the constructor's website.

To simulate the effect of pipelining, three large tasks are decomposed into four steps of a pipeline each. The cumulated ETC is about 10-20% greater for the pipeline tasks, to represent the increased workload to queue messages between tasks. The three different parts of the large tasks are overall preserved across the pipeline tasks, but not at each step.

Results The comparison is performed with a mapping algorithm based on the Min-Min resource allocation algorithm [144, 145], adapted to this model.

Figure 3.1 shows the effect of pipelining an application. The regular

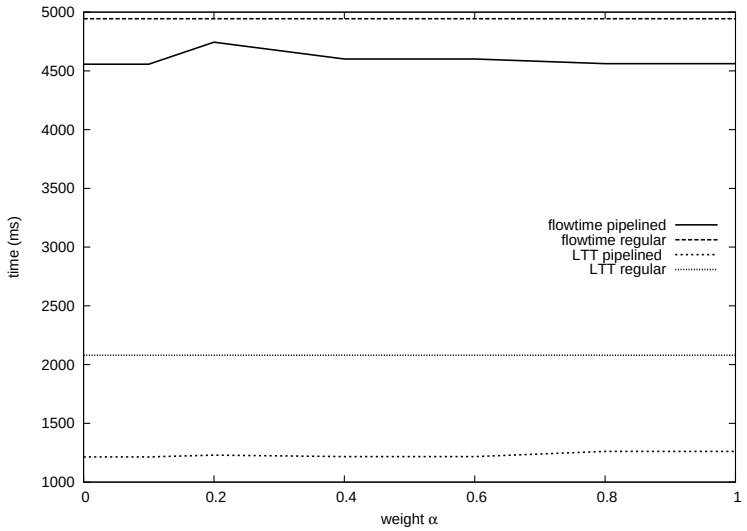


Figure 3.1: Performance for pipeline simulation on dual-core bi-processor

version of the application and the pipelined version are mapped on the same bi-processor machine. We can observe that for both performance objectives, the pipelined version is better. This is due to the better utilization of the CPU. The pipelined version offer much finer grain control over the coarse grain, regular version.

Figure 3.2 compares the regular version on the Intel bi-processor with the pipeline version on the AWN. We notice that the AWN performs worse, especially in term of flowtime. This is due to the performance penalty of 10 between the two processor speed, and the relatively small number of cores in the AWN, only a factor of 2.

Finally, Figure 3.3 show the energy results for all versions. Unsurprisingly, the AWN performs better than the others. We can notice that the pipeline version achieves better results than the regular version. This is an added benefit of the smaller tasks in a pipeline.

Conclusion

this thesis exposed the consequences of pipelining for energy-efficient mapping. A model that captures contention, operating system behavior for local scheduling and power management was presented. An existing mapping algorithm was adapted to rely on this model, and simulation experiments were conducted to evaluate the AWN microserver and compare it with multi-core processor based servers. Although the performance of AWN remains inferior to server class multi-core processors, it compensates in terms of total energy consumption and longest task time, which is directly related to throughput.

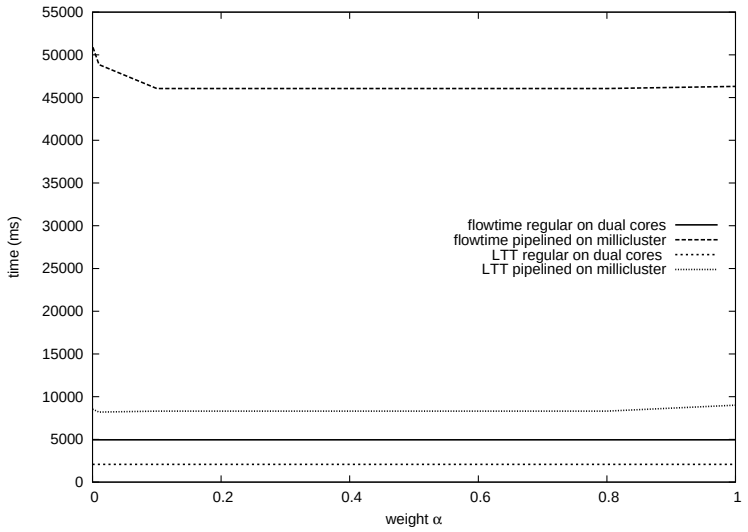


Figure 3.2: Performance comparison for pipeline simulation

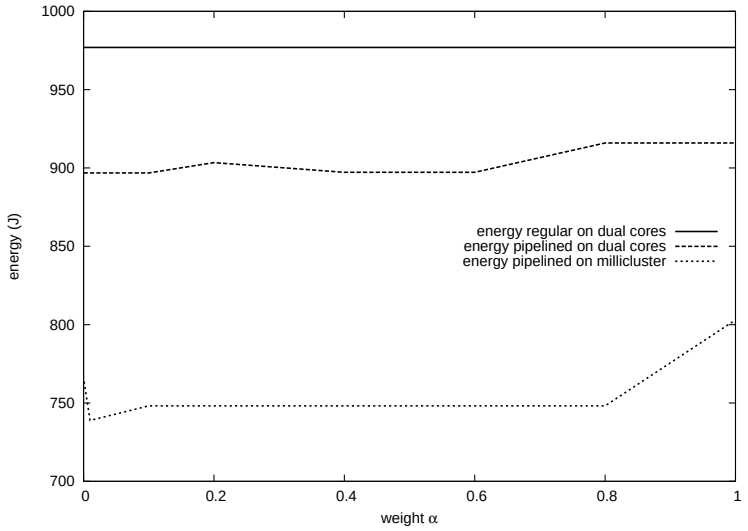


Figure 3.3: Energy comparison for pipeline simulation

3.1.4 Pipeline mapping on AWN with contention and soft deadlines

We further investigate pipelining as a candidate approach for parallelism for the AWN [159]. Soft real-time constraints are added to the tasks to account for the performance objectives of the overall application. The study compares three known heuristics, modified to suit our context. We also introduce SA as a tool to evaluate the proposed model.

We consider that an application is decomposed into a pipeline of tasks, connected by message queues. Several copies of the same tasks may be run concurrently, all serving the same queue. However, this possible set up is not investigated further here.

The objective is to match the performance of the application on data center class machines. However, the application's overall performance is now distributed across the tasks. Each task must respect some service level, in order for the application to do as well. Soft real-time deadlines provide a natural way to express this constraint. Each task is therefore given a deadline. The overall pipeline performance is equal to the slowest task's performance. The performance of a task is modeled as a time needed to process a unit of input, called the estimated time to complete (ETC) [160]. Therefore, the deadline for each task is the largest ETC. All tasks share the same deadline. In practice, that deadline is extended by a factor, *delta*, to account for variability in the ETC of the slowest task.

Energy model

We consider that each core is capable of dynamic voltage frequency scaling (DVFS); that is, it can be operated on a set of supply voltages and different speed performance (associated to different clock frequencies) [149, 150]. Energy-efficiency using DVFS often considers that application could specify the voltage/frequency point of operation of each core. This is not the assumption in this thesis. Indeed, inspection of the kernel power management tools of the GNU/Linux kernel version `2.6.35-24`, reveals that DVFS is very dynamic and self-regulated. Default values for the On-demand governor show a sampling rate of 10 ms (time period when a DVFS change is considered). Fundamentally, the complexity of a cluster is such that whenever possible, local decision making should be preferred over a global one. In this case, the regulation is based on CPU utilization, which is also under the control of the kernel. Here, the operating system manages power using the `cpu-freq` tools under the on-demand governor. The on-demand governor implements the *race-to-idle* policy. Whenever there is a need for CPU, then the voltage/frequency is set to its maximum value. Later, when the utilization decreases, the voltage/frequency point of operation is chosen so as to match the needed load. It should be mentioned that the automatic adjustment of the CPU frequency results in jitter, delayed comunications, in a large-scale system, which is the case of the AWN.

Processor is not the only component of the nodes of an AWN to consume energy. Because the intention is to capture the total energy consumption of the AWN, other components need to be included. The power model adopted is summarized by the relation:

$$P_m = P_{constant} + P_{high}, \tag{3.17}$$

where P_m is the total power of a machine (node in an AWN), $P_{constant}$ represents the constant power term, for components which do not scale according to voltage or frequency, this also include the network, and P_{high} represents the power increase, compared to the idle state, when components subject to DVFS are in high performance mode (in an active state, the machine dissipates $P_{constant} + P_{high}$). This is automatically adjusted by the OS and the hardware, and does not only include the processor. This model is preferable because it lends itself to experimental validation through power measurements.

Energy is the product of power and time. The energy should also reflect the race-to-idle policy. Total energy is defined below by:

$$E_m = P_{constant} \times CT_{max} + P_{high} \times \sum_{c}^{cores} CT_c, \tag{3.18}$$

where CT_c is the sum of all ETC of the tasks assigned to a core of a machine (its finishing time), and CT_{max} is the maximum CT_c over all cores of all machines. If no tasks are run on a machine, then that machine is considered switched off. When running a task, the core is at maximum voltage/frequency consuming P_{high}, when idle, only $P_{constant}$. The idle time lasts until the last core finishes its tasks. All machines in the AWN are considered identical in this study.

The nodes part of the AWN are multi-core, and are subject to contention. The contention considered in this study is related to components shared across a machine, such main memory. The contention factor is proportional to the memory parts of the other tasks running on the other cores of the same machine. Tasks running on the same core are not subject to contention because they do not run concurrently, but are preempted by the OS.

Contention effects impact the ETC of a task (access of a shared resource are serialized). Each task is defined by an ETC, which is split into:

- ETC under possible contention,
- the rest of the ETC, which is independent of sources of contention.

This effect of contention on a task's ETC is approximated by a delay added to its ETC. This penalty delay is the sum over all the other cores of the machine, of the time period spent concurrently in contention prone activity.

Scheduling pipelines in AWN

As mentioned previously, efficiently running software pipelines on a AWN of multi-core processors requires the precise scheduling of the tasks onto the cores. Before describing the algorithms proposed in this thesis, which is the topic of the next section, the nature of the scheduling in the AWN must

be made precise. Indeed, various components in a computing system use schedulers, making the term scheduling ambiguous.

The scheduler investigated here is a non-privileged instance of a program, operating at the cluster level. The main activity of the AWN scheduler is to periodically define the set of concurrent tasks in a processor, so as to minimize contention, meet task deadlines and save energy. It is similar to an OS long term scheduler, described in [161] by: "The long-term scheduler deals with the high-level or "big picture" issues; it is invoked infrequently and tasked with deciding which processes should inhabit the ready queue. The idea is that the long-term scheduler takes on the role of load balancing: it might try to maintain a mix of I/O-bound (i.e., those that perform mainly I/O) and computationally bound processes for example, in order to give the best overall system performance."

The AWN scheduler defines the current list of processes, per machine, that a usual OS scheduler (or short term) schedules for execution at a much higher frequency. The core on which to run the process is not important, because the machines are considered single processor (yet multi-core), and the exact core mapping is not important (as opposed to the concurrent set of processes).

Algorithms for simulation In this section, three scheduling algorithms are presented to meet the stated objectives. The particular context for the scheduling question lies at the intersection of two fields:

- distributed system, for the scheduling independent tasks (tasks in a pipeline have become independent tasks, which is another benefit of this program structure),
- soft real-time, because of the deadlines for each task.

The first algorithm evaluated is a variant of Min-Min [144, 145], a mapping algorithm for resource allocation. It originates from the field of distributed systems. It is a greedy algorithm, which considers all possible task-to-core mappings, and then performs the assignment of the best mapping, constructing the schedule incrementally. "Best" is defined here by the Energy Delay Product (EDP) $D \cdot E$, where:

- D is the total time by which all tasks exceeded their deadlines, $D = \sum_t^{tasks} max(0, deadline - finishing\_time)$,
- E is the total energy spent.

To influence the decisions of the algorithm, a paramater α is introduced, such that a mapping is considered better than the current best Max, if $\alpha \cdot D \cdot E < Max$.

The other two algorithms we consider are inspired from real-time scheduling algorithms: rate monotonic and earliest deadline first [162]. These algorithms have been proven optimal in a specific context, which is not the one set here, but similar. These variants are called Shortest Slack First (SSF) and Longest Slack First (LSF). SSF orders the tasks to schedule in increasing slack time, the difference between their deadline and ETC. It then assigns a core (which minimizes the EDP according the same rule as Min-Min) for each task in turn, incrementally constructing the schedule. LSF works identically, but orders the tasks by decreasing slack.

Before these algorithms can be experimentally compared, all the parameters of the different models (task, machine, energy, contention, algorithm) need to be set. It is best done once their respective influence is established. This is the topic of the next section.

Sensitivity analysis SA [163] of a model provides many benefits. First it determines the influence of each of the factors of the model. This allows future users of the model to focus on the most important parameters of the model, while ignoring the least influential. It also helps design models, because understanding the influence of each factor allows to verify of the model. For example, SA allowed the authors to uncover an error in the earlier version of a model. The SA reported that a presumed key factor had in fact almost no influence.

The objective for this SA is Factors Prioritization (FP), whose goal is [163] "to make a rational bet on what is the factor that one should fix to achieve the greatest reduction in the uncertainty of the output". A factor corresponds to our model parameters.

The different parameters for our model are presented in Table 3.6, with their range of possible values. Parameter maximum ETC is the maximum value used for the random generation of ETCs of all tasks. The maximum contention rate is the maximum value for the proportion of the ETC the task spends in contention prone instructions. This value is used for the generation of ETC. Deadline is the additional time added to the largest ETC to obtain the common deadline. It is an additional percentage to the largest ETC. Powers have been defined in Section 3.1.4. The values for P_{high} is taken from the specifications of the ARM A9 processor. The values for $P_{constant}$ are arbitrarily chosen, to reflect the non-processor related components. Parameter α is part of the objective function defined in Section 3.1.4. A small value for α indicates that sub-optimal scheduling decisions are allowed, whereas greater than 1 values indicate that scheduling decisions improve the objective by a greater margin.

Two SA are performed: a quantitative and a qualitative method. The quantitative method used is an extension to the Fourier Amplitude Sensitivity Test [164]. This method allows the computation of first order effects and

Table 3.6: Model parameter variation

Parameter	Probability	Range of values
maximum ETC	uniform	15 – 30 [time unit]
Contention rate max	uniform	30 – 60%
Deadline *delta*	uniform	0 – 30%
Power (constant)	uniform	15 – 25 W
Power high	uniform	0.1 – 2.0 W
α	uniform	0.7 – 1.2

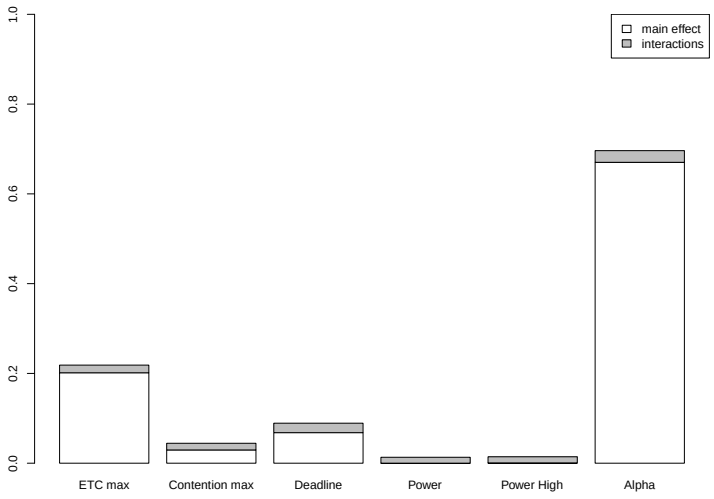


Figure 3.4: Fast99 of Performance

interactions for each parameter. Parameters interaction occurs when the effect of the parameters on the output is not a sum of their single (first order) effects. This variance decomposition method has the following desirable properties [163]. It is model independent (it does not place requirements on the type of model to work). It evaluates the effect of a parameter while all others are also varying. Finally, it copes with the influence of scale and shape (the probability density function and its parameters).

The results for this method are shown in Figures 3.4, 3.5. The output in the case of performance is the amount of time by which the deadlines were exceeded. Part of the hypothesis for the SA was a small number of machines, such that most tasks failed to meet their deadline. The amount of time by which they failed to meet their deadlines is the output.

Regarding the performance analysis, the factors have predominantly a

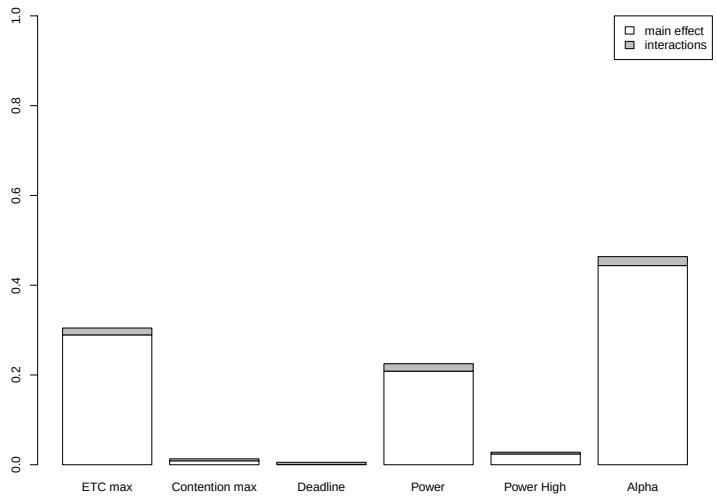


Figure 3.5: Fast99 of Energy

linear impact on the output. The most influent factor is α. Maximum ETC, the deadline factor and maximum contention playing minor roles. Indeed, the definition of α makes it play an influential role. Maximum ETC sets the maximum value for the task generation. The ETC for each task is randomly chosen in a given range. A higher value for maximum ETC leads to heterogeneous task ETC. Regarding the energy analysis, the three important factors are α, maximum ETC and power. This highlights the relation between performance and energy.

The qualitative method used is a One factor At a Time (OAT). It is commonly used for screening the least important factors from the rest. The method used here is the method of Morris. It also captures the linear and non-linear interaction of factors. Qualitative methods require less computations than quantitative ones.

The results for the method of Morris are shown in Figures 3.6, 3.7. The x-axis indicates the linear impact of the factor, while the vertical axis indicates the non linear impact.

Regarding the performance analysis, deadline and α show the strongest impact. Maximum ETC and maximum contention play a minor role. Both SA methods find the same four important factors, but in different order of importance. Regarding the energy analysis, maximum ETC, power and α have the most influence on the output. This is identical to the results of Fast 99.

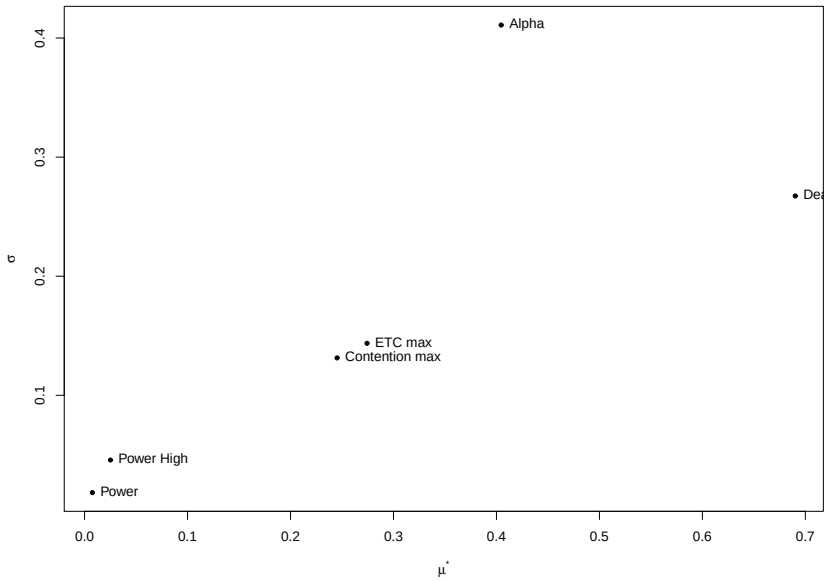


Figure 3.6: Morris of Performance

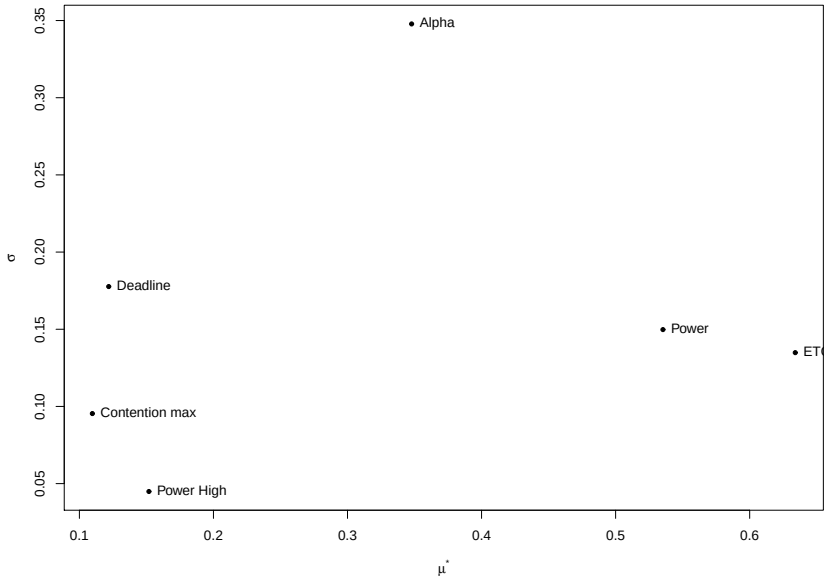


Figure 3.7: Morris of Energy

Comparison of the scheduling heuristics This section compares the three scheduling heuristics presented in Section 3.1.4. The SA results show

Table 3.7: Model parameters for heuristic comparison

Parameter	Value
# tasks	16
# machines	4
# processor/machine	1
# cores/processor	2
ETC range	5 – 25 [time unit]
Contention range	10 – 60%
Deadline *delta*	10%
Power (constant)	20 W
Power high	1.0 W

that special care should be taken when setting the parameters α, maximum ETC and to a lesser extent deadline and maximum contention. Therefore, experiments ran the different heuristics on the same ETC instances (30 instances for each run). Table 3.7 lists the parameter settings. The choice for the machines are based on ARM A9 processors. The power values come from the hardware specifications.

Figure 3.8 presents the performance results for the three heuristics. The x-axis lists different values for α, the heuristic score parameter. Performance is the amount of time by which the deadlines were exceeded. All heuristics reach their best score when $\alpha \simeq 1$. LSF is the best algorithm for performance, but only slightly better than Min-Min. However, it is faster than Min-Min.

Figure 3.9 presents the total energy results for the three heuristics. All heuristics reach their best score when $\alpha = 1$. SSF is the best algorithm for energy, and LSF is slightly better than Min-Min.

Conclusion

Software pipeline is a parallel application architecture to overcome the limited individual performance of each node of an AWN, and may allow to benefit from a low-power system. Three scheduling algorithms were introduced and evaluated to further reduce the energy consumption while meeting performance objectives, by adding soft real-time considerations to the software pipeline. The contention in multi-core processors is also part of the model.

3.2 Data Parallelism

The simulations of pipeline models for AWN and multi-core processors showed that pipelining for AWN does not match the performance objec-

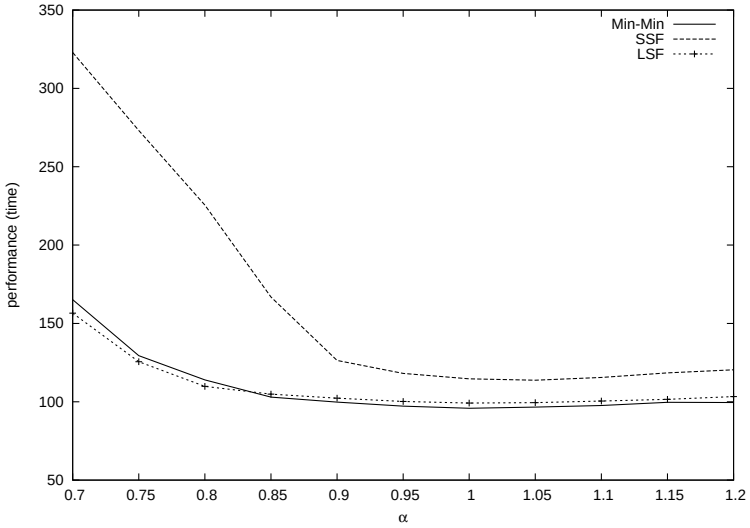


Figure 3.8: Comparison of 3 heuristics on performance

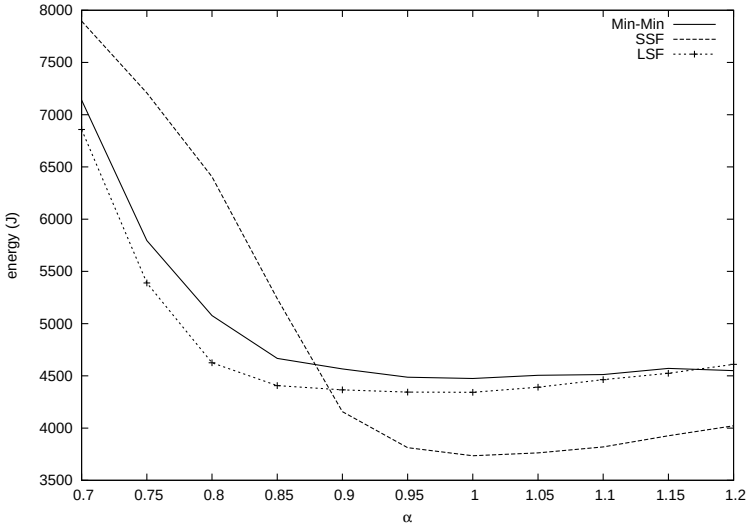


Figure 3.9: Comparison of 3 heuristics on energy

tives of brawny nodes, even for throughput. Another, more frequently used source of parallelism is data. Data parallelism keeps the programs as is, but splits the input data to be processed in independent pieces. When possible, this decomposition is more practical than code-parallelism. We explore this source of parallelism by designing data-parallel solvers for the optimization problem we selected, which is presented next, in Section 3.2.1. The chosen solvers are a heuristic, Section 3.2.2, and a metaheuristic, Section 3.2.3.

3.2.1 Use case: a scheduling optimization problem

For our investigation of data parallelism on AWN, we will often use the same problem: the independent task mapping optimization problem. The algorithms used to solve this problem (or find solutions of good quality) will be re-designed for increased parallelism, and evaluation on AWN. A possible confusion may arise because this problem was found in our previous investigations on code-parallelism (to support task placement of pipeline steps). In contrast, here the problem and its associated algorithms serve only as a test case for exploring the design of parallel application, the solvers' results are not used in any way.

The problem is how to assign independent tasks onto the different processors in an heterogeneous cluster, in order to minimize the makespan. Makespan is the completion time of the last machine (when the last machine finishes its tasks). Makespan is defined more formally later in this section. A machine is an independent computing unit, such as a single core in a multi-core processor.

This NP-complete problem [165] arises frequently in parameter sweep applications, such as Monte-Carlo simulations [166]. In these applications, many tasks with almost no interdependence are generated and submitted to a distributed system. In fact, more generally, the scenario in which the submission of independent tasks to a cluster is quite natural given that cluster users independently submit their tasks to the system and expect an efficient allocation of their tasks. We notice that efficiency means to allocate tasks as fast as possible and to optimize some criterion, such as makespan or flowtime. Makespan is among the most important optimization criteria of a distributed system; it is a measure of its productivity (throughput).

More precisely, assuming that the computing time needed to perform a task is known (assumption that is usually made in the literature [145, 146, 147]), we use the Expected Time to Compute (ETC) model by Braun et al. [145] to formalize an instance of the problem, as follows:

- A *number* of independent (user/application) *tasks* to be assigned.
- A *number* of heterogeneous *machine* candidates to participate in the planning. A machine is general term for a computing unit, such as a core.
- The *workload* of each task (in millions of instructions).
- The *computing capacity* of each machine (in *mips*).
- Ready time indicating when machine m will have finished the previously assigned tasks. In this work, we consider, without loss of generality, that all the machines are available to process the assigned tasks ($ready_m = 0$).

- The Expected Time to Compute (ETC) matrix (of size $nb\_tasks \times nb\_machines$) in which $ETC[t][m]$ is the expected execution time of task t on machine m.

We consider the task assignment as a single objective optimization problem, in which makespan is minimized. *Makespan*, the finishing time of latest task, is defined as:

$$\max\{completion[m] \mid m \in Machines\} \ , \tag{3.19}$$

where *completion* is the completion time of a machine. This time indicates when the machine will finalize the processing of the previous assigned tasks as well as of those already planned. Formally, for a machine m and a schedule S, the completion time of m is defined as follows:

$$completion[m] = ready_m + \sum_{t \in S^{-1}(m)} ETC[t][m] \ . \tag{3.20}$$

The complexity of the independent task mapping problem confines candidate algorithms to heuristic and metaheuristic approaches (except for small problem instances).

3.2.2 Data-parallel Min-Min for the GPU

This section presents the design of a parallel Min-Min for the GPU, that improves the performance for large problem instances [167], the runtime becomes $O(T^2 \times M/cores)$.

The most well-known heuristic algorithm for the independent tasks mapping problem is Min-Min [144]. As mentioned before, it is a deterministic, greedy algorithm that constructs the solution iteratively. It is one of several list algorithms, where the order of assignment can change (for example: Max-min). Due to its accuracy and simplicity, the algorithm has been used as a reference in many research papers since then [145, 168, 169] or as a component for the design of more efficient algorithms [170, 171, 172]. Faster sequential variants of the previous list algorithms were proposed in [173, 174], they sort the tasks to reach a runtime in $O(M \times T \times \log(T))$, instead of $(O(M \times T^2)$, at the expense of increased memory size. Nesmachnow et al. [175] proposed GPU implementations of two scheduling heuristics, including Min-Min. They report a maximum speedup of about 5 with respect to the sequential version of the heuristic. We obtain a greater speedup, as presented in Section 3.2.2, although their GPU hardware seems comparable to ours. Their paper does not detail the parallel algorithm used, therefore we cannot explain the difference.

Parallel Min-Min

The Min-Min algorithm iteratively proceeds in three steps. First, it finds the best machine assignment for each unassigned task (the first "min"). Here, best means minimal completion time. Second, it chooses among all the previous possible assignments, the one with the minimum completion time (the second "min"). Finally, it assigns that task to the corresponding machine. The process continues until all tasks have been assigned.

Algorithm 1 Pseudo-code for Min-Min heuristic on the GPU

```
for all Tasks of one solution do
    min_ct <<< Tasks >>> (results) // Step 1
    cudaMemcpy (results, temp, cudaMemcpyDeviceToDevice)
    // Step 2: parallel reduction
    n ← 2
    while Tasks/n ≥ 1 do
        min_task <<< Tasks/n >>> (temp)
        n ← n × 2
    end while
    assign <<< 1 >>> (temp, solution) // Step 3
end for
```

Our proposed GPU implementation for the Min-Min algorithm is presented in Algorithm 1. The `f <<< n >>> ()` notation reflects the CUDA macros: it indicates that kernel `f` is launched across `n` threads. The first step is the launch of the `min_ct` kernel. For each task, a thread finds the best machine for a given task, by selecting the machine with the minimum estimated completion time. This kernel is launched with `Tasks` threads, because the selection of the best machine can be conducted in parallel. Threads of previously assigned tasks are also run, but immediately return from the kernel. Then, the results are copied, from device memory to a temporary area on device memory, for the parallel reduction. The parallel reduction presented here (lines 5-9) is a simplified version of the code actually used to identify the best (minimal) task/machine assignment. Finally, one thread runs the `assign` kernel to update the solution with the best assignment. When all tasks have been assigned, the solution found can either be copied to the host memory or kept into the device memory, depending if the algorithm is run alone or not.

Experimentation

In this section, we study the performance of the parallel Min-Min. First, we describe in Section 3.2.2 the problem instances generated for the simulations. Then, we evaluate in Section 3.2.2 the GPU version of Min-Min heuristic proposed.

The computer used in the experiment is a Dell Precision T5400, which includes an Intel Xeon E5440 processors (dual processor, of 4 cores each), clocked at 2.83 GHz, with 16 GiB of main memory. The computer runs the GNU/Linux operating system Ubuntu Server (64-bit kernel, version 2.6.35-27). The GPU installed on this computer is a Nvidia Tesla C2050, with CUDA driver version 3.20 (capability 2.0). This GPU holds 14 multiprocessors (of 32 cores each), clocked at 1.15 GHz, and with a global memory of 2 GiB. All programs are written in C, except for the GPU kernels which are written in CUDA C. The operating system's Pthread library is used for the multi-threaded versions of the parallel CPU versions.

Problem instances We study six different instance sizes, specifically 512×16, 4096×128, $8,192 \times 256$, $16,384 \times 512$, $32,768 \times 1,024$, and $65,536 \times 2,048$, where the first figure is the number of tasks and the second the number of machines the tasks must be assigned to. A machine is independent computing unit, such as a core in multicore architectures.

The chosen problem instances are generated with high task and machine heterogeneity, which we consider realistic. This reflects different tasks and different processor types. Large problem instances should reflect different processor types, as a cluster is often the result of several machine acquisitions.

The instances were randomly generated (we created them as described in [160], using `R`), therefore 20 different instances were considered for every problem size in our experiments.

Performance Evaluation of Parallel Min-Min We evaluate in this section the performance of the parallel Min-Min design presented in Section 3.2.2. Because the Min-Min heuristic is a deterministic algorithm, and the parallel Min-Min performs exactly the same search as the sequential Min-Min, we only compare execution time. We implemented two different parallel versions of the algorithm: a multi-threaded CPU implementations (using 4 and 8 cores), and the GPU implementation.

We show in Figure 3.10 the speedup results of the different parallel Min-Min implementations. The speedup is measured as the time the sequential Min-Min requires over the CPU over the time of the corresponding algorithm. The x-axis is represented in logarithmic scale. As it can be seen, the two parallel Min-Min algorithms scale well with the problem size, providing similar speedup values for all of them. However, the algorithm does not scale so well with the number of cores. This is probably due to memory contention. The parallel Min-Min with 8 cores is always faster than the equivalent version with 4 cores, but the average speedup increases from 4.024 to 5.918 when doubling the number of cores from 4 to 8. It is worth emphasizing that the parallel Min-Min on 4 cores is achieving linear speedups in

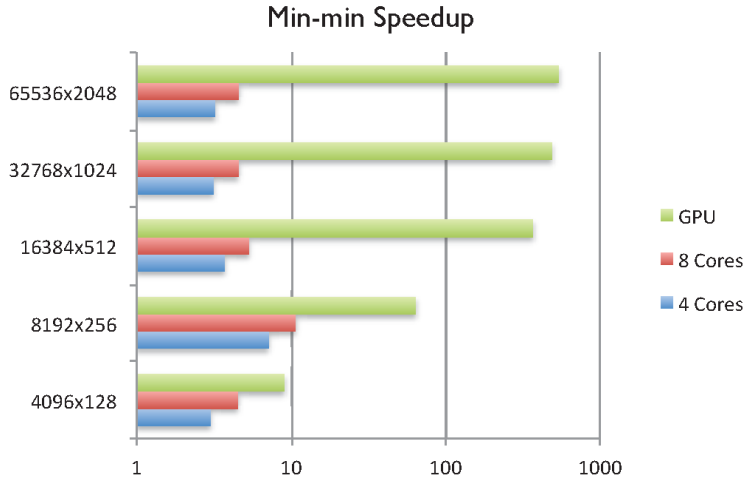


Figure 3.10: Speedup results of the GPU versus the equivalent sequential and parallel CPU Min-Min (logarithmic scale)

average for the considered problem sizes. Additionally, we notice that the algorithm performs super-linear speedups for the 8192 tasks instance: 7.11 for 4 cores and 10.67 for 8. We suspect this is due to higher cache hit ratios thanks to memory sharing.

We now turn to the results of the GPU version. We can see that its performance is clearly higher with respect to the CPU versions, with speedups ranging from 9 for the smallest instance to 538 for the biggest one (the sequential Min-Min algorithm takes more than 3 days to find a solution for this instance). Additionally, the algorithm scales well with the problem size, since the bigger the problem, the higher the speedup obtained. Therefore, the performance of the parallel GPU version with respect to the parallel CPU ones is better when the problem size increases.

We should mention that the GPU version is slightly different than the parallel CPU ones. They differ in the way the second step of the Min-Min algorithm is implemented, where it searches for the task that is earlier accomplished among those chosen in the first step. In the CPU version, tasks are iteratively traversed to choose the earliest accomplished one, while in the GPU algorithm this is done with a parallel reduction, benefiting from the massively parallelization provided by the GPU, which requires $\log_2(T)$ iterations (cf., Section 3.2.2).

The GPU program only uses the global device memory, therefore additional speedup may be achievable using the faster GPU memories, for example the read-only instance matrices. Also, the default parameter settings for the cache size were used.

Conclusion

In this section, we parallelized the well-known Min-Min heuristic. The observed speedups, although quite high, expose limits to the scalability of the data decomposition, although straitforward. The performance improvement between runs with 4 and 8 cores is not linear. With 4 and 8 cores, the speedup (compared to the sequential version) even decreases with problem size. With the GPU, the speedup increases with problem size, but less so with larger problems.

3.2.3 Parallel Cellular Genetic Algorithm

We mentioned that the selected optimization problem, because of its complexity, calls for inexact methods, heuristic or metaheuristic algorithms. Many evolutionary approaches have been applied to the selected problem [176]. Here, we evaluate a data-parallel version of a kind of genetic algorithm: a fine-grain multi-threaded Cellular Genetic Algorithm (CGA) for multi-core processors [177]. We present the CGA in the following section, and our design in Section 3.2.3.

Overview of parallel cellular genetic algorithms

CGA CGAs [178, 179, 180] are structured population algorithms with a high explorative capacity. The individuals composing their population are (usually) arranged in a two dimensional toroidal mesh. This mesh is also called grid. Only neighboring individuals (i.e., the closest ones measured in Manhattan distance) are allowed to interact during the breeding loop (see Figure 3.11). This way, we introduce a form of isolation in the population that depends on the distance between individuals. Hence, the genetic information of a given individual spreads slowly through the population (since neighborhoods overlap). The genetic information of an individual will need a high number of generations to reach distant individuals, thus avoiding premature convergence of the population. By structuring the population in this way, we achieve a good exploration/exploitation trade-off on the search space. This improves the capacity of the algorithm to solve complex problems [178, 181].

Individuals evolved in parallel across the population usually evolve together, which requires synchronization. Asynchronous evolution relaxes this global time constraint [182, 183]: individuals evolve independently and the population is not of the same age (underwent the same number of evolutions). Asynchronous models are also known to improve search capability [184]. In an asynchronous CGA, the population is updated with next generation individuals immediately after their creation. These new individuals can interact with those belonging to their parent's generation. Alternatively, we can place all the offspring individuals into an auxiliary population,

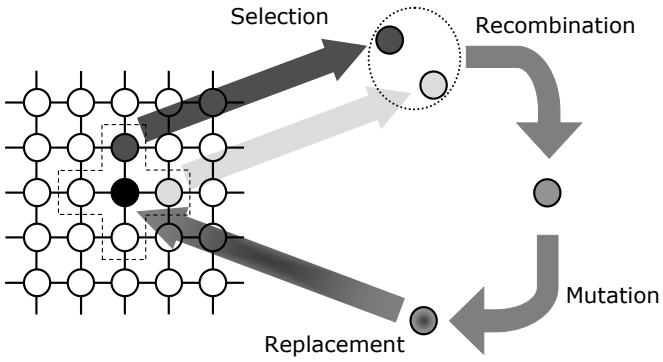


Figure 3.11: In cellular GAs, individuals are only allowed to interact with their neighbors.

Algorithm 2 Pseudo-code for a canonical CGA (asynchronous).

```
1: while ! StopCondition() do
2:     for all ind in population do
3:         neigh ← get_neighborhood(ind);
4:         parents ← select(neigh);
5:         offspring ← recombine(p_comb, parents);
6:         mutate(p_mut, offspring);
7:         evaluate(offspring);
8:         replace(ind, offspring);
9:     end for
10: end while
```

and then replace all the individuals of the population, with those from the auxiliary population, at once. This last version is referred to as the synchronous CGA model. As it was studied in [178, 185], the asynchronous CGAs converge the population faster than the synchronous CGAs.

A canonical CGA follows the pseudo-code of Algorithm 0. In this basic CGA, each individual in the grid is iteratively evolved (line 2). A generation is the evolution of all individuals of the population. Individuals may only interact with individuals belonging to their neighborhood (line 3), so parents are chosen among the neighbors (line 4) with a given criterion. Recombination and mutation operators are applied to the individuals in lines 5 and 6, with probabilities p_comb and p_mut, respectively. Afterwards, the algorithm computes the fitness value of the new offspring individual (or individuals) (line 7), and replaces the current individual (or individuals) in the population (line 8), according to a given replacement policy. This loop is repeated until a termination condition is met (line 1), for example: the total elapsed processing time or a number of generations.

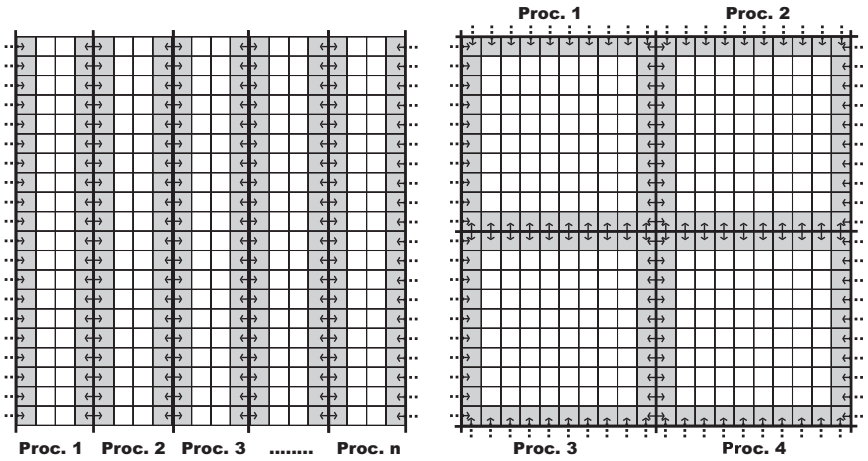


Figure 3.12: CAGE (left) and the combined parallel model of CGA (right)

Parallel CGA Some pioneer works in this line are those by Manderick and Spiessens [179, 186], Mühlenbein [187, 188], Gorges-Schleuter [189], and Collins [190]. With the popularity loss of massively parallel machines, some authors proposed different parallel implementations of cellular Genetic Algorithm (GA)s, more appropriate for the distributed architectures that started to be available from the 90's. In 1993, Maruyama et al. proposed in [191] a peculiar version of a parallel CGA for a cluster of machines in a LAN in which single solutions are located in the processors, and after every generation solutions exchange information with only one randomly selected neighbor, as an attempt to reduce communications overhead. This is similar to the distributed parallel Evolutionary Algorithm (EA) class presented in [181].

After this first work on parallel CGAs for LAN architectures, there are a number of more recent papers proposing other designs that better fit the dynamics of the canonical sequential model. Nakashima et al. proposed in [192] a combined CGA where the population is divided into smaller square sub-populations, interacting through their borders. An image of this model can be seen on the right-hand side of Fig. 3.12, where the gray cells represent the solutions exchanged in the inter-processors communications. Folino et al. contributed in [193] with CAGE, a parallel cellular GP in which the population is divided into groups of columns (or rows) which constitute sub-populations (see the graph on the left in Fig. 3.12) to be run on different processors. This way they can reduce the number of messages with respect to the previous model, but messages will be bigger.

Luque et al. compared the performance of several parallel GAs in LAN environments [194]. Among them, both distributed and cellular GAs were the best performing ones, being the cellular algorithm slightly slower (from

3% to 10%) than the distributed one, but providing better solutions. Later, the authors analyzed different parallel CGA designs in [195], and proposed the use of asynchronous communications among processors in [196]. Dorronsoro et al. proposed PEGA in [197], a new parallel GA distributed in islands, with a CGA in every island, which can be executed either in local area network environments or in *computational grids*. PEGA was successfully applied to the largest existing instances of the VRP problem, contributing to the state of the art with some new solutions. We proposed the first parallel implementations of CGA for multi-core architectures, with applications to real-world problems as DNA sequencing [198] or scheduling [177].

Finally, there are a number of implementations of parallel CGAs in GPU architectures. The first works had to deal with complex data structures to map the algorithm data to texture rendering based on GPU, i.e., the information contained in the solutions must be allocated in form of pixels in the GPU [199, 200, 201].

The appearance of tools like CUDA [202] or OpenCL [203] gave a major boost for the development of new parallel algorithms on GPU architectures. Among them, a few recent works are targeting Cellular EA. Soca et al. [204] proposed a framework for the implementation of cellular EAs on GPUs. Vidal and Alba also proposed a parallel version of CGA in a single GPU [205] and multiple ones [206]. Li et al. designed a fine-grained parallel immune algorithm [207].

A data-parallel CGA for the optimization problem

Parallel asynchronous CGA Our Parallel Asynchronous CGA (PA-CGA) is based on [198]. We partition the population into a number of contiguous blocks with a similar number of individuals (Figure 3.13). Each block contains $pop\_size/\#threads$ individuals, where $\#threads$ represents the number of concurrent threads executed. We partition the population by assigning successive individuals to the same block. The successor of an individual is its right neighbor. We move to the next row when we reach the end of a row (our grid is 2-dimensional). We assign each block to a different thread, which will evolve the individuals of its block.

In order to preserve the exploration characteristics of the CGA, communication between individuals of different blocks is made possible. As mentioned in the previous section, at each evolution step of an individual, we define its neighborhood. This neighborhood may include individuals from other population blocks. This allows an individual's genetic information to cross block boundaries.

Threads evolve their population block independently. They do not wait on the other threads to complete their generation (the evolution of all the individuals in their block) before pursuing their evolution. Hence, if a breeding loop takes longer for an individual of a given thread, the individuals evolved

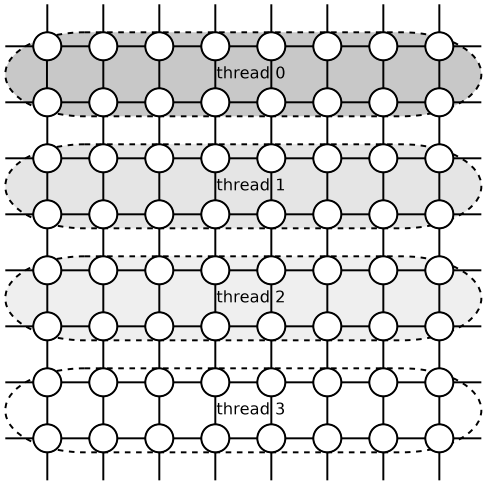


Figure 3.13: Partition of an 8×8 population over 4 threads.

by the other threads may go through more generations.

The combination of a concurrent execution model with the neighborhoods crossing block boundaries leads to concurrent access to shared memory. For example, the neighbor n of an individual i might belong to a different block. The thread evolving this other block could be updating individual n precisely when the thread evolving i is accessing it. Without care, this can result in incorrect results. This error can occur when selecting a parent from a neighborhood (non-atomic read operation), or recombining (a non-atomic read operation of the parent) that belongs to another block and is currently being replaced (a non-atomic write operation). To enable safe concurrent memory access, we synchronize access to individuals with a POSIX [208] read-write lock. This high-level mechanism allows concurrent reads from different threads, but not concurrent reads with writes, nor concurrent writes. In the two latter cases, the operations are serialized.

Asynchronous CGAs can visit individuals in different orders [185]. In this work, all threads will sweep through their population in the same fixed order. This means that we are using the line sweep policy in every block. Note that this is not exactly the same as the line sweep policy typically used in asynchronous CGAs. In our case, all blocks are updated concurrently. We experimented different sweep orders for different blocks, in hope of limiting memory contention, but we did not notice any significant improvement in the algorithm's execution speed. We attribute this to the unpredictable nature of the thread's execution, while the alternative sweep policies per thread assumed a predictable, fixed, thread execution by the operating system.

Algorithms 0 and 0 provide a more detailed description of the algorithm. Function $do\_parallel(f, parm)$ means that $f(parm)$ is executed by all threads in parallel, but on different data items. All threads join before

Algorithm 3 Pseudo-code for our proposed parallel asynchronous CGA (PA-CGA).

```
1: t_0 ← time();                                          ▷ record the start time
2: pop ← setup_pop();                                     ▷ initialize population
3: par ← setup_blocks(pop);                               ▷ set parameters for all threads
4: do_parallel(initial_evaluation, par);                  ▷ each thread evaluates its block
5: do_parallel(evolve, par, t_0);     ▷ each thread evolves its block, see Algorithm 0
```

the next instruction.

Algorithm 4 Pseudo-code for *evolve*().

```
1: while time() − t_0 ≤ time do
2:     for all ind in a thread's block do
3:         neigh ← get_neighborhood(ind);
4:         parents ← select(neigh);
5:         offspring ← recombine(p_comb, parents);
6:         mutate(p_mut, offspring);
7:         H2LL(p_ser, iter, offspring);
8:         evaluate(offspring);
9:         replace(ind, offspring);
10:    end for
11: end while
```

Function *initial_evaluation*() computes the fitness of all individuals in the initial population. The stop condition for this grid scheduling problem is a wall clock time. The asynchronous model moves the stop condition verification into *evolve*.

From Algorithm 0, we notice that the thread checks the current time after evolving all the individuals of its block. This could let the thread run for longer than the allowed time. We accept this approximation since one generation of the entire block takes less than 6 ms in our experiments, while the time is expressed in tens of seconds. The evolution step also performs a local search operation. This operation is presented in the next subsection. We parameterize Highest To Lower Loaded (H2LL), our problem-specific local search, with a number of iterations *iter*, which sets the number of passes. Finally, *evaluate*() computes the makespan of the schedule.

Local search and solution representation We also propose a new local search operator for the problem considered.

We refer to a machine's completion time as its *load*. The local search operator moves a task, randomly chosen, from the most loaded machine to a selected candidate machine (the most loaded machine's completion time defines makespan). The candidate machines are the N least loaded (N is a parameter). A candidate machine is selected if its new completion time, with

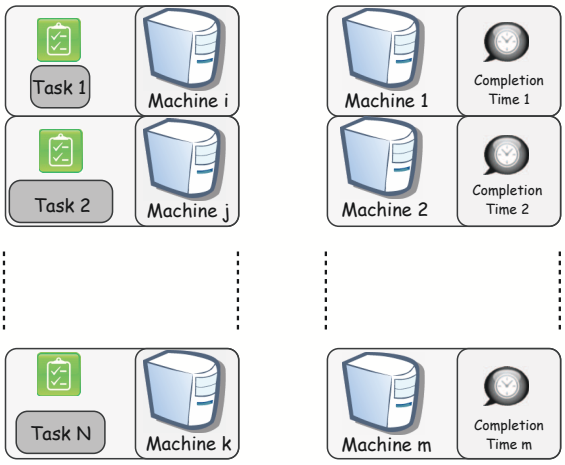


Figure 3.14: Representation of solutions. In addition to the task-machine assignments (left-hand side), we store the completion time for every machine too (right-hand side). Variation operators are only applied on the task-machine assignments.

the addition of the task moved, is the smallest of all the candidates. This new completion time must also remain inferior to the makespan. Algorithm 0 describes this operator.

The representation we use for independent task scheduling on grids is shown in Figure 3.14. It is composed of:

- an array S of integers, $S[t] = m$, representing the assignment of task t to machine m,
- an array CT of floating point values, $CT[m] = c$, representing the completion time of each machine m.

The completion times are often used, therefore maintaining up-to-date completion times for all machines speeds up computations. The *evaluate*() function of Algorithm 0 only finds the maximum completion time. The completion times are kept up-to-date by each operator (recombine, mutation, local search). Such updates are efficiently performed by adding or removing the ETC of a task on a machine to the appropriate completion time. As can be noticed from Algorithm 0, we use the transposed ETC matrix. This increases the cache hit rate, and thus the overall performance of the algorithm. Indeed, when accessing an ETC for a task on a machine, this ETC value is cached, but so are the neighboring values (caches operate on cachelines). If we store the transposed ETC matrix, then these neighboring values are the ETC values for the next few tasks on the same machine (exactly how many depends on the size of a cacheline). So, if the schedule assigns one of the

next tasks to the same machine, then this ETC value is present in cache. We measured an improvement in the algorithm's execution time of 5-10%. Indeed, this improvement is comparable to the uniform probability for such an event (next task assigned to same machine), $1/\#machines$, we use 16 machines in our experiments.

Algorithm 5 Pseudo-code for $H2LL$, our local search.

```
1: for all iter iterations do
2:     sort machines on ascending completion time
3:     task ← random task from last machines;
4:     best_score ← CT[last machines];                    ▷ makespan
5:     for all mac in pop_size/2 first machines do
6:         new_score ← CT[mac] + ETC[mac][task];
7:         if new_score < best_score then
8:             best_mac ← mac;
9:             best_score ← new_score;
10:        end if
11:    end for
12:    move task to best_mac if any
13: end for
```

Experimentation

This section presents the results of our experiments with PA-CGA. Section 3.2.3 describes both the parameterization of the algorithm and the instances of the problem we are solving. Section 3.2.3 reports and discusses the results.

Parameters and problem instances The algorithm parameters are summarized in Table 3.8. We are using a population of 256 individuals. The population is initialized randomly, except for one individual. The schedule for this individual results from the Min-Min heuristic [144]. The linear 5 (L5) neighborhood, also called Von Neumann neighborhood, is composed of the 4 nearest individuals, plus the individual evolved. This neighborhood is chosen to reduce concurrent memory access. The 2 best neighbors are selected as parents. The recombination operators used are the one-point (*opx*) and the two-point crossover (*tpx*). The mutation operator moves one randomly chosen task to a randomly chosen machine. The newly generated offspring replaces the current individual if it improves the fitness value. Finally, the termination condition is an execution time of 90 seconds. The number of threads used in all our experiments ranges from 1 to 4. All threads run on one processor. The processors used for the experiments are a 4-core Intel Xeon E5440 clocked at 2.83 GHz, with 6 MB L2 cache, a 6-core Intel Xeon L5640 @ 2.8 GHz with 12 MB L2 cache, and a quadcore ARM A9 @ 1.1 GHz with 4 MB of L2 cache (Calxeda ECX-1000).

Table 3.8: Parameterization of PA-CGA.

Population	16×16
Population initialization	Min-Min (1 ind)
Cell update policy	fixed line sweep per block
Neighborhood	linear 5
Selection	best 2
Recombination	one-point and two-point crossover, $p\_comb = 1.0$
Mutation	move, $p\_mut = 1.0$
Local search	H2LL, $p\_ser = 1.0$, $iter = 5, 10$
Replacement	replace if better
Stopping criterion	90 seconds, wall time
Number of Threads	1 to 4

The benchmark instances consist of 512 tasks and 16 machines. These instances represent different classes of ETC matrices. The classification is based on three parameters: task heterogeneity, machine heterogeneity and consistency [160]. Instances are labelled as `u_x_yyzz.k` where:

u stands for uniform distribution (used in generating the matrix).

x stands for the type of consistency (c for consistent, i for inconsistent, and s for semi-consistent). An ETC matrix is considered consistent when the following is true: if a machine m_i executes a task j faster than machine m_j, then m_i executes all tasks faster than m_j. Inconsistency means that a machine is faster for some tasks and slower for some others. An ETC matrix is considered semi-consistent if it contains a consistent sub-matrix.

yy indicates the heterogeneity of the tasks (hi means high, and lo means low).

zz indicates the heterogeneity of the resources (hi means high, and lo means low).

k numbers the instances of the same type.

We report computational results for the following 12 instances, for which we provide their Blazewicz [209] notation:

- $u\_c\_hihi.0 : Q16|26.48 \leq p_j \leq 2892648.25|C_{max}$;
- $u\_c\_hilo.0 : Q16|10.01 \leq p_j \leq 29316.04|C_{max}$;
- $u\_c\_lohi.0 : Q16|12.59 \leq p_j \leq 99633.62|C_{max}$;
- $u\_c\_lolo.0 : Q16|1.44 \leq p_j \leq 975.30|C_{max}$;

- $u\_i\_hihi.0 : R16|75.44 \leq p_j \leq 2968769.25|C_{max}$;
- $u\_i\_hilo.0 : R16|16.00 \leq p_j \leq 29914.19|C_{max}$;
- $u\_i\_lohi.0 : R16|13.21 \leq p_j \leq 98323.66|C_{max}$;
- $u\_i\_lolo.0 : R16|1.03 \leq p_j \leq 973.09|C_{max}$;
- $u\_s\_hihi.0 : R16|185.37 \leq p_j \leq 2980246.00|C_{max}$;
- $u\_s\_hilo.0 : R16|5.63 \leq p_j \leq 29346.51|C_{max}$;
- $u\_s\_lohi.0 : R16|4.02 \leq p_j \leq 98586.44|C_{max}$;
- $u\_s\_lolo.0 : R16|1.69 \leq p_j \leq 969.27|C_{max}$.

Results We now present and discuss the results of our computational experiments. The discussion includes a comparison with other algorithms in the literature.

We first study the speedup of the algorithm as an indication of its scalability: how performance improves with the number of threads. This study helps tune the optimal number of threads for the experiments. Speedup is usually defined as:

$$S(n) = time(1)/time(n) \ , \tag{3.21}$$

where n is the number of machines, or processor cores.

We exchange time for the total number of evaluations. With time, a performance improvement corresponds to a smaller execution time, but with evaluations, an improvement corresponds to more evaluations. This leads to the following definition of speedup used in this thesis:

$$S(n) = \#evaluations(n)/\#evaluations(1) \ , \tag{3.22}$$

where $\#evaluations(n)$ is the mean number of evaluations over 100 independent runs, and n is the number of threads.

Figures 3.15, 3.16 and 3.17 show how performance evolves with the:

- number of threads,
- number of local search iterations.

On the Xeon processors, we first observe that without local search (0 iteration) the performance decreases with the number of threads. This result is essentially due to thread synchronization. Without local search, the evolution of an individual requires less computation, but the same effort of synchronization (for recombination and replacement). So the proportion of synchronization over total computation is the highest. Given the partition of the population, a smaller block means that more individuals are on

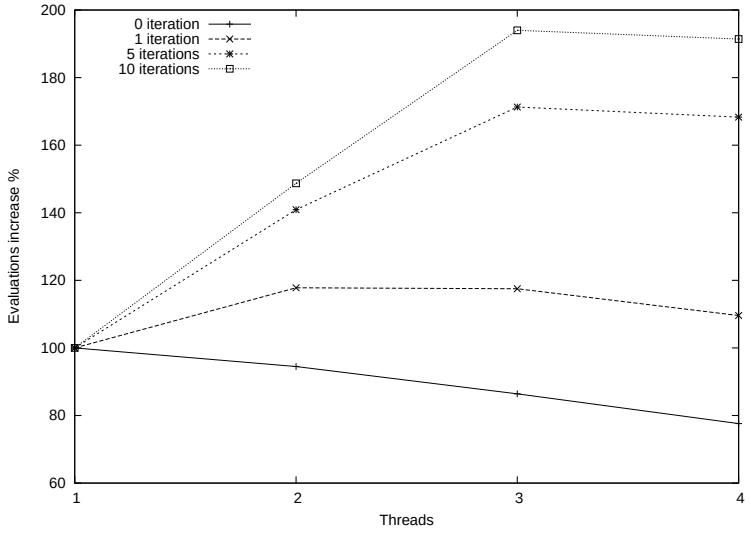


Figure 3.15: Speedup of the algorithm on Xeon E5440.

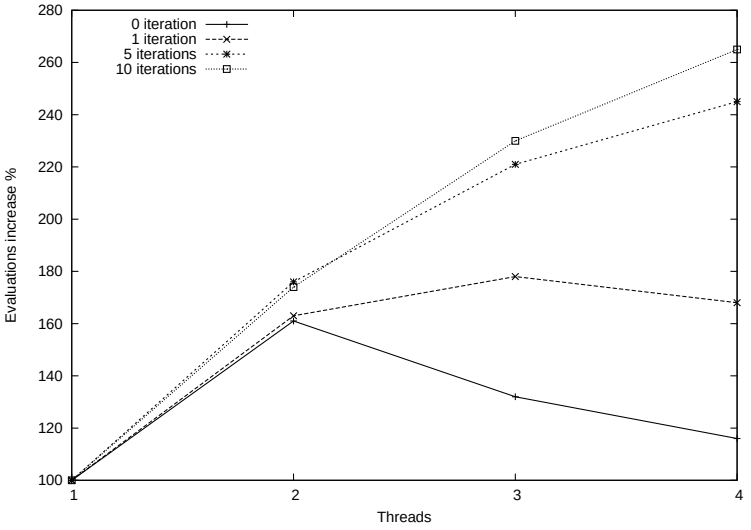


Figure 3.16: Speedup of the algorithm on Xeon L5640.

the boundary of the block, where the neighborhood therefore crosses block boundaries and may cause synchronization delays. The combination of these factors lead to more synchronization delays, which decreases performance when the number of threads increases.

As the number of local search iterations increases, more computation outside synchronization is performed (local search is performed on the offspring). This reduces synchronization delays, and we achieve positive speedups. Yet, we notice that with 5 or 10 local search iterations, there is no more per-

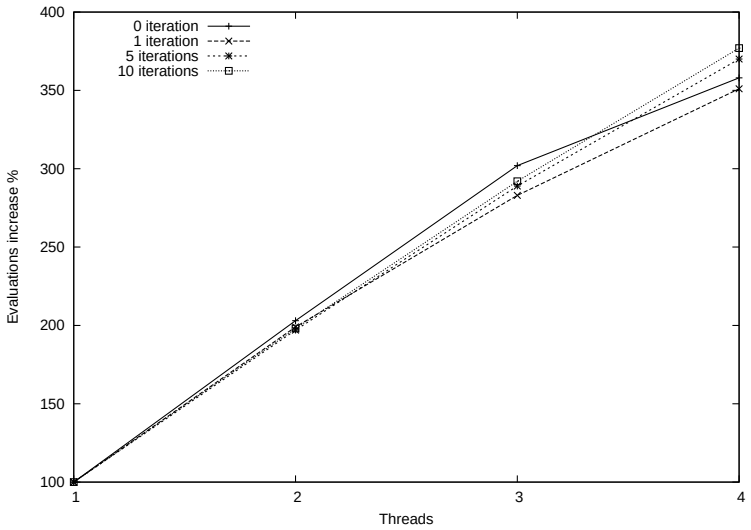


Figure 3.17: Speedup of the algorithm on ARM A9 ECX-1000.

formance gained when increasing the number of threads from 3 to 4. This is caused by the smaller block sizes. Although more time is spent in local search, the proportion of individuals on the boundary of their block increases, and with fewer individuals to process, synchronization is more frequent. Finally, the processor level 2 cache is shared across all running threads. Increasing the number of threads with little data locality negatively impacts performance. From the speedup results, we notice that 3 threads reach the maximum number of evaluations, so we adopt this model for the next studies in this thesis.

The Calexda ARM A9 results are similar to the Xeon with regards to local search iterations, but with a much improved scalability. The speedup is almost linear up to 3 threads. Without local search, the effect of synchronization appears with 4 threads, but less so than on the Xeon processors. The added computation of the local search iterations reduce the effect of synchronization. This is because the A9 is much slower than the Xeon for regular operations, while synchronization operations are similar in performance. The good synchronization performance of ARM A9 multi-core was also observed in other experiments (with a shared memory, multi-threaded Map-Reduce implementation).

Finally, the Xeon L5640 achieves $\times 4.6 - 7.5$ more evaluations than the ARM A9. However, the more the local search iterations, the faster the ARM ($\times 4.6$ slower for 10 local search iterations).

Next, we examine the influence of the recombination operators (*opx* and *tpx*), and the number of local search iterations (5 and 10). Figures 3.18–3.20 present these results. They are obtained over 100 independent runs. Three

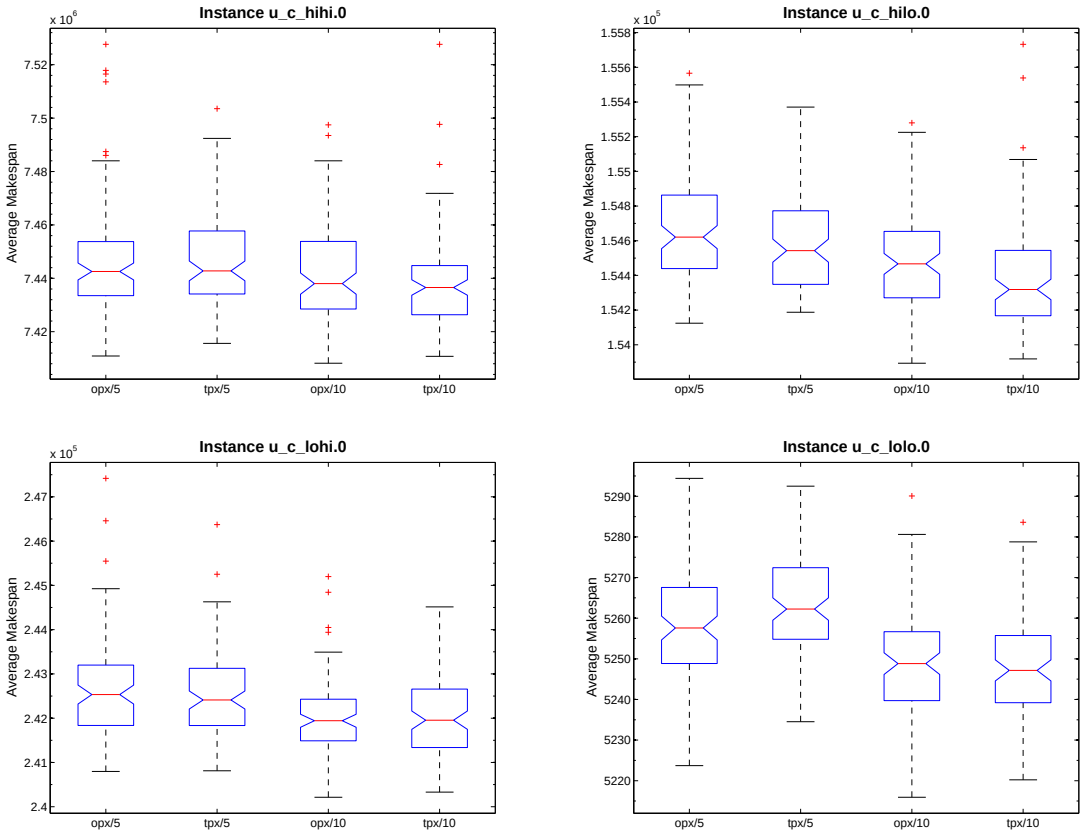


Figure 3.18: Comparison of recombination operators and local search iterations, consistant instances.

threads are used. A box plot is provided for each instance file. In these plots, when the notches in the boxes does not overlap, we can conclude, with 95% confidence, that the true medians differ. We notice that overall, the *tpx* recombination operator provides better mean makespan results than *opx*. Furthermore, 10 iterations of our local search $H2LL$ achieve a better mean makespan than 5. With statistical significance, we can state that $tpx/10$ performs better than $opx/5$ for all instances. It finds the best mean makespans in most instances, but not in all. For the consistent instances, *opx* and *tpx* find similar mean makespan values. For the next results of this section, we use the *tpx* recombination and 10 local search iterations.

Table 3.9 presents a comparison of our results with others found in the literature. Results for cMA + LTH (a CGA hybridized with Tabu search) [211] and struggle GA (a non-decentralized population GA) [210] are averages over 10 independent runs, and they were taken from the original papers. We propose 2 sets of results for our algorithm, PA-CGA. One for runs of 10 seconds,

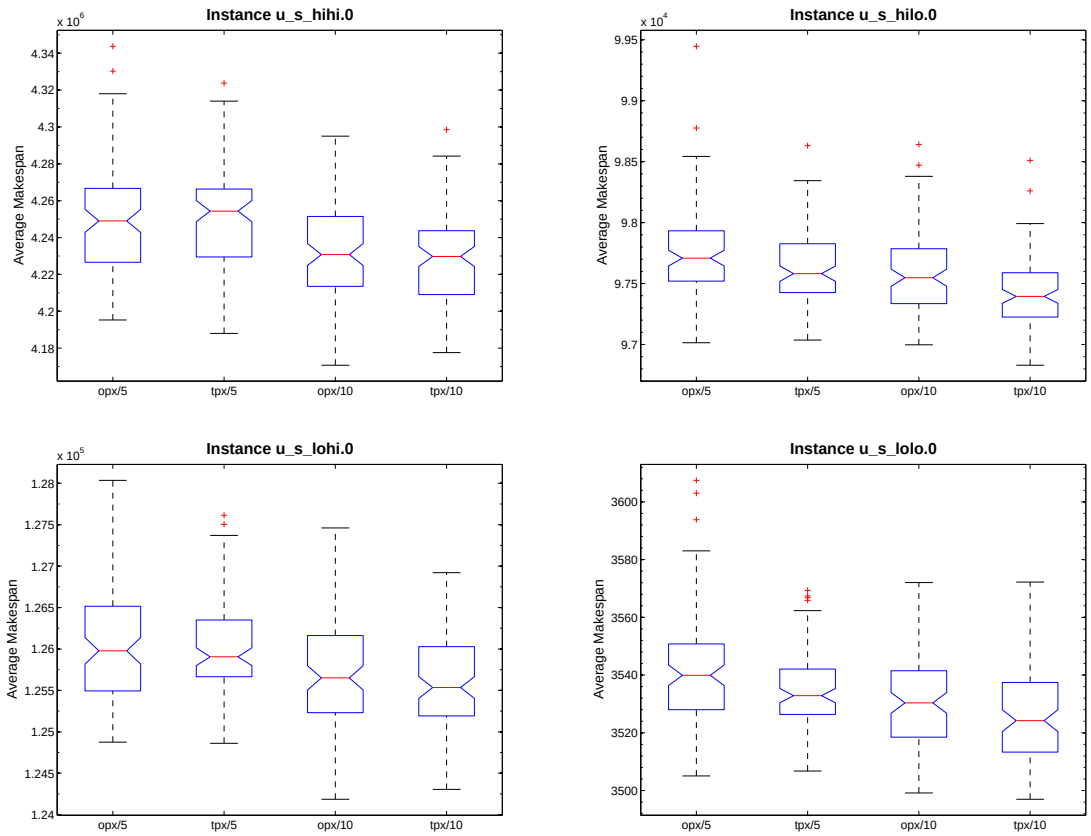


Figure 3.19: Comparison of recombination operators and local search iterations, semi-consistant instances.

Table 3.9: Comparison versus other algorithms in the literature. Mean makespan values.

instance	Struggle GA [210]	cMA + LTH [211]	PA-CGA 10 sec	PA-CGA
u_c_hihi.0	7752349.4	7554119.4	**7518600.7**	**7437591.3**
u_c_hilo.0	155571.48	**154057.6**	154963.6	154392.8
u_c_lohi.0	250550.9	247421.3	**245012.9**	**242061.8**
u_c_lolo.0	5240.1	**5184.8**	5261.4	5247.9
u_s_hihi.0	4371324.5	4337494.6	**4277497.3**	**4229018.4**
u_s_hilo.0	983334.6	97426.2	97841.6	**97424.8**
u_s_lohi.0	127762.5	128216.1	**126397.9**	**125579.3**
u_s_lolo.0	3539.4	**3488.3**	3535.0	3525.6
u_i_hihi.0	3080025.8	3054137.7	**3030250.8**	**3011581.3**
u_i_hilo.0	76307.9	75005.5	**74752.8**	**74476.8**
u_i_lohi.0	107294.2	106158.7	**104987.8**	**104490.1**
u_i_lolo.0	2610.2	**2597.0**	2605.5	2602.5

another for runs of 90 seconds. For both times, the results presented are averages of 100 independent runs. Makespan values in bold indicate the best results for an instance, or if 10 seconds of runtime of our algorithm

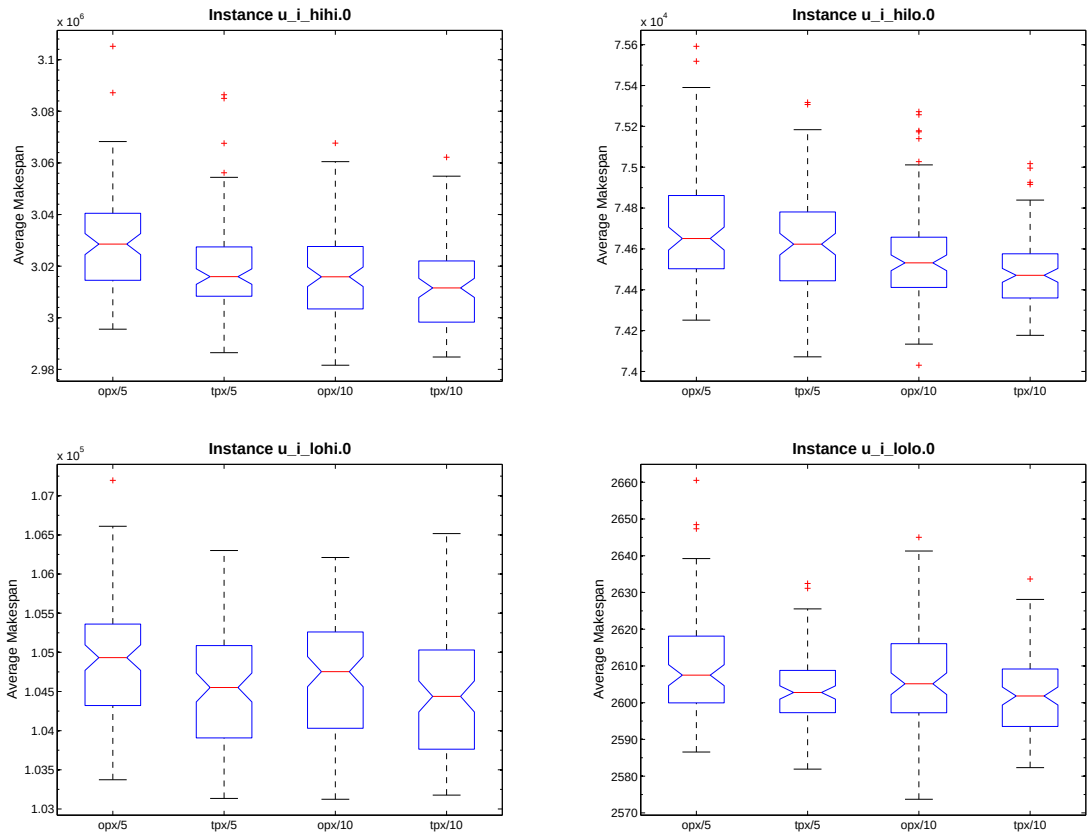


Figure 3.20: Comparison of recombination operators and local search iterations, inconsistant instances.

achieves better results than the literature. All experiments are performed on the Xeon E5440.

We present results for runs of 10 seconds because of the difference in computing platforms used in [211] and in our experiments. In [211], all experiments were conducted on a AMD K6 450 Mhz processor. This machine is slower than the one we use. To account for this difference, we wish to reduce the runtime in our experiments, in the same proportion. We therefore benchmark both machines, compute the performance ratio of the 2 machines, and apply it to our runtime. Unfortunately, we do not have access to a AMD K6 450 Mhz machine. However, there exists one benchmark whose results for this machine have been published, and is available for execution on our machine. It is the program TSCP 1.7.3 [212]. One advantage of this benchmark is that it implements a combinatorial algorithm, and does not test a specific processor feature. Executing the benchmark shows a performance ratio of 9 between the 2 machines. Therefore, we provide results

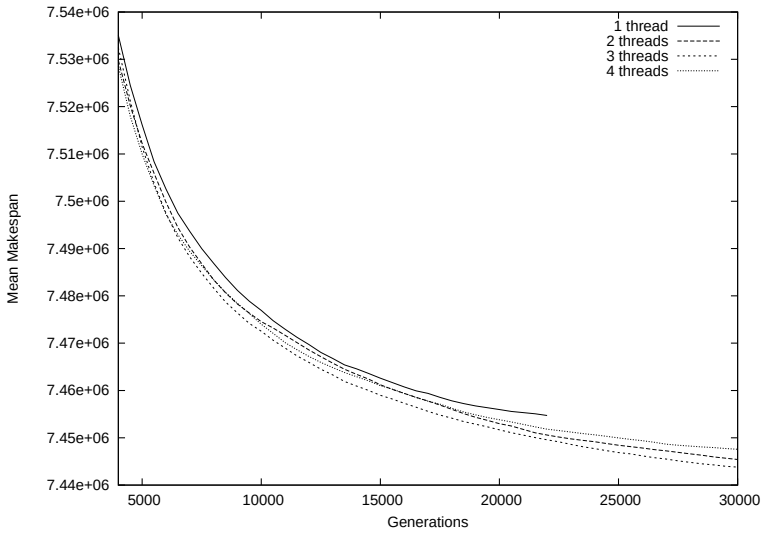


Figure 3.21: Evolution of the algorithm.

for runs of 90/9 seconds, as a comparison point.

Table 3.9 shows that PA-CGA improves most previous results. Particularly, it provides the best results for inconsistent instances (where the performance of a machine varies from one task to another) and for instances of high heterogeneity in tasks and resources. It improves half of the results for consistent, and semi-consistent instances. It does not improve results for instances where the tasks and resources have a low heterogeneity (homogeneous). These results are useful because inconsistent instances, and instances with high task and resource heterogeneity, represent the more complex problem formulation of independent task scheduling. Also, scheduling independent near-homogeneous tasks on near-homogeneous machines can be effectively addressed with alternative simpler and faster methods, such as heuristics [145]. We can also notice that our algorithm improves the results for instances with greater makespan values.

Figure 3.21 shows how makespan, averaged across the population (all threads) and over 100 independent runs, evolves with the number of generations. All runs process the `u_c_hihi.0` instance file. The stop condition is 90 seconds wall time. Each line corresponds to a different number of threads. In order to display differences, a subset of the domain (generations) is plotted. First, we notice that running the algorithm with one thread evolves for less generations than with more threads, in the allocated time. Also, one thread finds worse average makespan, at any generation. It is important to note that our algorithm configured for one thread represents the canonical asynchronous CGA of Section 3.2.3. With 4 threads, we observe that the algorithm converges faster initially, but fails to reach the best solutions. Running the algorithm with 3 threads finds the best solutions.

Conclusion

In this section, we manually designed a fine-grain multithreaded CGA for multi-core processors. The data parallelism was introduced at the population level. The work-to-synchronization ratio is tunable with the number of threads and the intensity of the local search operator. The resulting program improves the quality of the solution found (the performance of the search algorithm), and the speed of the program. However, as shown in Figure 3.15, increasing the number of threads is only beneficial up to 3 threads. With 4 threads, the program performs less iterations, and finds worse solutions than with 3 threads. This is a consequence of the increased inter-thread communication. The situation improves on a quadcore A9, due to its slower core but fast locking mechanism.

3.3 Summary

In this chapter, we explored code and data parallelism in the context of AWN. Simulations of software pipelines (code-parallelism) showed that the wimpy nodes, although more energy-efficient, cannot achieve the performance of brawny nodes, even when contention for shared resources is taken into account. Experiments with data-parallel versions of known algorithms showed clear benefits (such as super-linear speedup), but also exposed their limits, in scalability (with the size of input data, and the number of cores), but also in more basic configurations (Section 3.2.3).

An interesting finding is that although contention in access of shared components has a negative impact on performance and energy, it can also provide some benefits, as observed with the super-linear speedup of the parallel Min-Min (attributed to cache memory) and the improved solutions of a parallel genetic algorithm (attributed to the exploration/exploitation trade-off).

The usual approach to parallelism we experimented does not seem to allow us to meet our performance objectives for the AWN, another approach seems necessary. The unexpected positive influence of contention can serve as an inspiration for exploring alternative sources of parallelism, the topic of the next chapter.

Chapter 4

An Alternative Approach to Parallel Programming for AWN

4.1 Introduction

The previous chapter discussed the traditional approach to parallel design, where the semantics of the program were preserved by parallelization. Here, we propose a change in perspective: the semantics of the program is not preserved by parallelization. Although the program implements a different algorithm, it should nevertheless perform the same function or solve the same problem. This view represents an increase in abstraction: the program's function needs to be preserved, but not it's original program, and not even it's algorithm.

The chosen use case of the optimization problem suits this change of view, because the problem is well separated from the algorithm. Indeed, any solution found by an algorithm can be evaluated (its quality) regardless of how it was found. Therefore, an alternative approach is to change the original algorithm, to better match the AWN architecture, and yet find solutions of comparable quality.

The two sections of this chapter report on the experiments with two parallel algorithm designs, that modify a cellular genetic algorithm to better suit the AWN. The AWN platforms used are a 40-core shared memory processor (Section 4.2), and a GPU (Section 4.3).

4.2 The Parallel Cellular Genetic Algorithm Revisited

In this first section, we revisit the parallel asynchronous cellular genetic algorithm introduced in Section 3.2.3, by simplifying it at the expense of correctness [213]. The simplification consists in the removal of the thread locks when accessing shared memory. This deliberate error aims to improve the scalability of the parallel algorithm, while preserving its search capability. We experiment the change on three benchmark search problems (we do not use the scheduling problem of the previous chapter), and observe an improvement in runtime, and unexpectingly, a slight improvement in search capability, with statistical significance.

The source for parallelism in the proposed algorithm remains the population, it is a case of data-parallelism. The concurrency in a genetic algorithm's population is generally found in three ways: master-slave, island and cellular. The master-slave model dispatches the operators' work to a number of slaves. In the island model, the population is partitioned into isolated evolutionary processes, which periodically exchange individuals. The cellular model, introduced in Section 3.2.3, can be seen as fine-grained island model.

4.2.1 The PA-CGA Simplification

We present three parallel models for a PA-CGA: the *Island* [194], *Lock* [198, 177] and *Free*. The Free model, this section's contribution, removes the thread protection when accessing shared memory. It is an incorrect implementation of a PA-CGA: by removing the thread locks, data consistency is not ensured. This is meant to improve the runtime and scalability of the PA-CGA. However, this change is also expected to impact the search capability of the PA-CGA.

In the PA-CGA *Island* model, as presented in Algorithm 6, each thread operates on two populations: (a) its local partition, and (b) the global population. The local partition (a) is a thread-local partition, used for evolution over a number of generations, set to 100. Once 100 generations are completed, the thread-local partition (a) is copied to the global population (b). This copy is performed asynchronously (one individual at a time). The global population (b) is accessed by threads when they require individuals from another partition. This occurs at crossover, when a parent selected belongs to another partition. POSIX read-write locks [208] are used by the threads to read individuals from another partition, and to commit their partition to the global population. The Island model aims to reduce contention on the shared population by operating on a thread-local data as much as possible.

The PA-CGA *Lock* model, presented in Algorithm 7, is the closest to the

Algorithm 6 Island model

```
while < max_gens do
    while < max_gens & not every 100 gens do   ▷ evolve local partition 100×
        for all individual in local partition do
            individual ← evolved(individual)   ▷ "←" follows replacement policy
        end for
    end while
    for all individual in global partition do ▷ update the global partition
        rw_lock(global individual)
        global individual ← local individual
        rw_unlock(global individual)
    end for
end while
```

Algorithm 7 Lock model

```
for all gens do
    for all individual in global partition do
        child ← evolved(individual)
        rw_lock(global individual)
        global individual ← child          ▷ "←" follows replacement policy
        rw_unlock(global individual)
    end for
end for
```

classic asynchronous CGA. The only difference is that each thread evolves the individuals of its partition only. Each individual is protected with a POSIX read-write lock. This allows for concurrent read access. When an individual can be replaced with a better child, the change occurs immediately (provided a thread lock is acquired), and is then visible to all other threads. Individuals that are unreachable from other threads (placed in the "middle" of a partition) are still protected with a lock, although it is always granted. The Lock model requires more communication across threads than the Island model, however, changes (improvements) are communicated immediately.

Algorithm 8 Free model

```
for all gens do
    for all individual in global partition do
        child ← evolved(individual)
        global individual ← child        ▷ "←" follows replacement policy
    end for
end for
```

The PA-CGA *Free* model is the simplest of all models, as per Algorithm 8. A thread evolves its partition, and updates the global population immediately. However changes in the global population are made without any thread locking. This is apparently incorrect, because a thread may read an individual that is currently being updated (dirty read). Thus the individual read is a combination of the previous individual and its replacement. This is possible because of the representation of an individual; usually a large array of word size elements, not an atomic operation. This model is considered wait-free, because a thread's progress is bound by a number of steps it has to wait before progress resumes. The Free model is proposed to accelerate the execution by removing thread locks, yet retain the benefits of immediate updates to the population as in the Lock model. Increasing the number of threads makes dirty reads more frequent, because it reduces the size of each partition, each thread evolves its partition faster, and more individuals lie on the border of a partition.

4.2.2 Experimentation

To investigate the behavior of the Free model, we compare the models across a selection of benchmark problems, presented in Section 4.2.2. The behavior of the models is observed across several indicators, presented in Section 4.2.2. The parameters and environment is summarized in Section 4.2.2.

Table 4.1: Benchmark of combinatorial optimization problems

Problem	*Fitness function*	*n*	*Optimum*
MTTP	$f_{MTTP}(\vec{x}) = \sum_{i=1}^{n} x_i \cdot w_i$	200	-400.0
PPEAKS	$f_{PPEAKS}(\vec{x}) =$ $\frac{1}{N} \max_{1 \leq i \leq p}(N - HammingD(\vec{x}, Peak_i))$	100	100.0
MMDP	$f_{MMDP}(\vec{s}) = \sum_{i=1}^{k} fitness_{s_i}$ $fitness_{s_i} = 1.0$ if s_i has 0 or 6 ones $fitness_{s_i} = 0.0$ if s_i has 1 or 5 ones $fitness_{s_i} = 0.360384$ if s_i has 2 or 4 ones $fitness_{s_i} = 0.640576$ if s_i has 3 ones	240	40.0

Benchmarks

The benchmark problems selected for our comparison are well-known combinatorial optimization problems, displaying different features like multimodality, epistasis, large search space, etc. The reader is referred to [178] for details on these benchmarks. They are summarized in Table 4.1 (name, fitness value, number of variables –*n*–, and optimum). They are the Massively Multi-modal Deceptive Problem (MMDP) –instance of 40 subproblems of 6 variables each–, the MTTP –instances of 200 tasks–, and the PPEAKS problem, with 100 peaks.

Metrics

The metrics for the comparison aim to capture the behavior of the three algorithms as the number of threads increases. Our first metric is execution speed. It is the wall-clock runtime of the algorithms for the maximum number of generations. However, increased speed is useless if the search capability is degraded such that it requires more generations, therefore we add the following metrics:

- Success rate: the number of experiments when the optimum was found.
- Evaluation-efficiency: the number of evaluations required to find the optimum, when found. This is measured in evaluations (calculation of the fitness of an individual) instead of generations, because of the concurrent evolution in each partition.
- Time-efficiency: speed and evaluation-efficiency are combined by measuring the wall-clock time required to find the optimum (when found). This is useful from the perspective of a potential user of the algorithm.

Table 4.2: PA-CGA parameters

Parameter	*Value*
Population size	40×40
Asynchronous mode	fixed line sweep
Selection operator	L5, binary tournament
Crossover operator	two-point crossover
Crossover probability	1.0
Mutation operator	$\times 2$ flips
Mutation probability	1.0
Maximum generations	2500
Island synchronization period	100 generations
Runs	100

Experimental setup

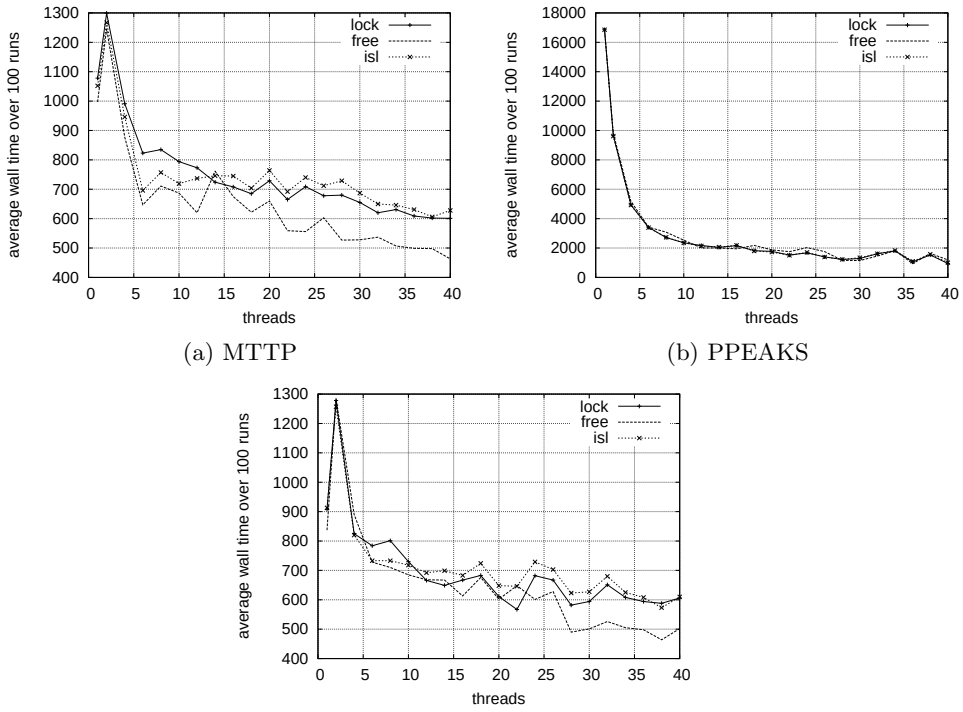
Table 4.2 summarizes the various parameters for the PA-CGA. The asynchronous mode sets the order in which the threads evolve the individuals in their partition [184]. Mutation consists in randomly flipping two items in the individual. The maximum number of generations is the stop condition per thread. The Island synchronization period specifies when the thread-local partition is committed to the global population (for other threads to access). For each benchmark, 100 searches or runs are performed. The individuals are randomly (uniform) generated for each run.

The computer used for the experiments is a Bullx S6030, where one board holds four Intel Xeon E7-4850 @ 2 GHz processors of 10 cores each. This computer can be considered an AWN, given the number of cores (40) and their relatively slow clock frequency. However, the total power requirement (in the order of the KW) prevents its use as a low-power infrastructure. The operating system is GNU/Linux 2.6.32-5-amd64 (Debian), GCC is version 4.4.5.

Results

In this section, we present the results from the benchmark problems, grouped by metric.

Runtime Figure 4.1 plots the average runtime (wall-clock) over the 100 runs (in msec), as defined in Section 4.2.2. We can observe that all models reduce their runtime as the number of threads increases. The Free model is the fastest and scales the best, which is expected given the wait-free design, although not significantly for PPEAKS, Figure 4.1b. The small difference between models for PPEAKS is due to the fitness function of PPEAKS, which is more time consuming than Minimum Tardiness Task



(a) MTTP

(b) PPEAKS

(c) MMDP

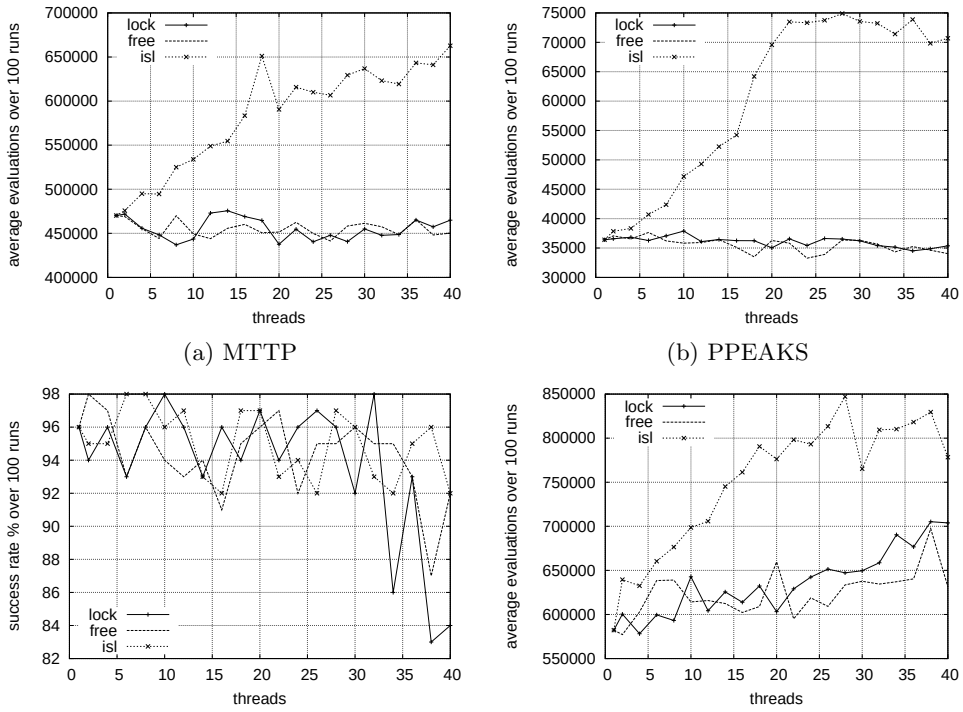
Figure 4.1: Runtime

Problem (MTTP) and MMDP and therefore dominates the synchronization delays. The speedup observed may seem low (especially for MTTP and MMDP), but the load is essentially due to synchronization.

Evaluation efficiency Figures 4.2 show the average number of evaluations needed to find the optimum (Section 4.2.2). Because this metric applies to runs where the optimum is found, we first discuss the success rate.

The success rate for the different PA-CGA models for MTTP and PPEAKS is 100% across the runs (and is not plotted). For MMDP, Figure 4.2c, the rate is below 100%. All models display about the same success rate, which also decreases from 35 threads and up. At this point, the partitions become too small, the generations too fast, thus reducing diversity in the partitions, which damage the search.

Regarding evaluation-efficiency, the Free model obtains similar or better results than Lock (Wilcoxon Signed-Rank test). On MTTP, PPEAKS and MMDP, Free is better in respectively 5, 10 and 20% of the cases. Also, the Lock and Free obtain constant results with the number of threads. The dirty reads in the Free model slightly help its evaluation-efficiency. The



(a) MTTP

(b) PPEAKS

(c) MMDP Success rate

(d) MMDP

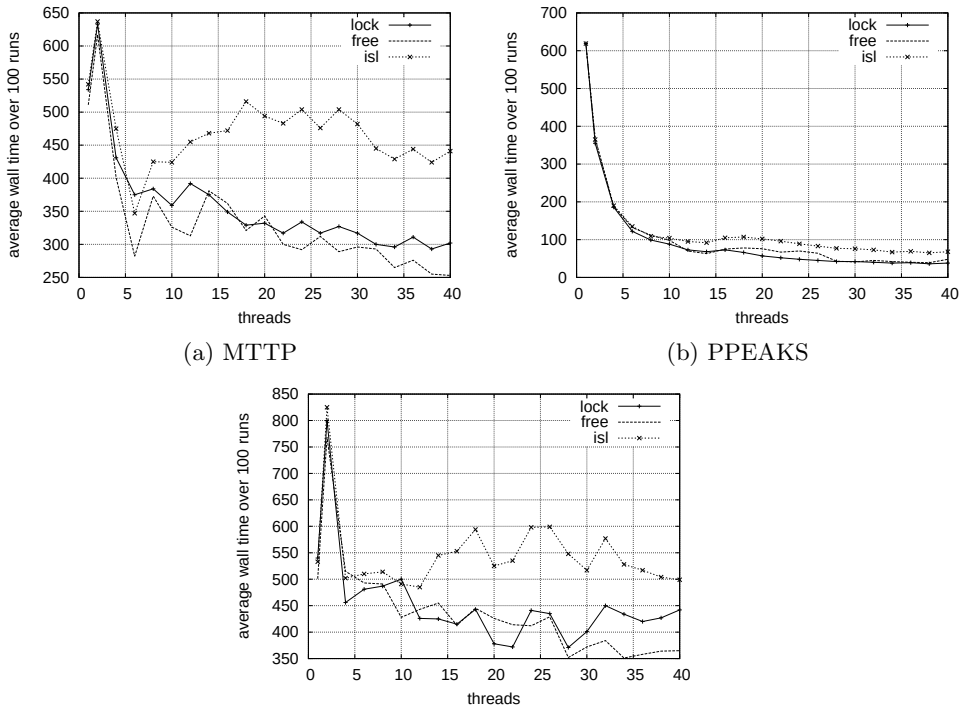
Figure 4.2: Evaluations to optimum (when found)

other observation is that the Island model does not scale well.

Time efficiency Figures 4.3 show the time elapsed to reach the optimum, when found (Section 4.2.2). For the Island model, Figures 4.3a, 4.3c show that the gain in runtime is offset by the loss in evaluation-efficiency. For PPEAKS, the gain in runtime is so high, that time-efficiency increases. The Lock and Free models improve their time-efficiency with a greater number of threads, mainly because of the gain in speed. The Free model obtains the best results. This is due to the surprisingly good evaluation-efficiency, which means that the dirty reads do not harm the search, and actually help.

4.2.3 Conclusion

We proposed a new parallel asynchronous CGA model, called Free. The Free model is based on a deliberate design error in the PA-CGA: all thread locks are removed, and access to the shared population leads to dirty reads. The absence of thread locks also makes it wait-free. It is the simplest PA-CGA design. This new model was compared to existing models: Island and Lock. The evaluation consisted in solving three benchmark problems (MTTP, PPEAKS, MMDP) using 1 to 40 threads, on a 40-core shared



(a) MTTP

(b) PPEAKS

(c) MMDP

Figure 4.3: Time to optimum (when found)

memory machine. These benchmarks are not computationally intensive, therefore the differences between models is more apparent. Experiments show that the Free model scales the best, and provides better or equal search capability, compared to the previously published Island and Lock models. The Free model, although at fault with respect to the original CGA design, preserves the principle behind evolutionary search. The dirty reads do not introduce noise, but provide another source of randomness in the sampling of previous individuals (parents). An another finding is that the well-known Island model provides worse search capability as the number of threads increases. Indeed, the improved runtime cannot compensate for the degraded efficiency of the model.

4.3 Parallel Cellular Genetic Algorithm for the GPU

The previous section presented a modification to the population-based decomposition in a CGA, for improved data-parallelism. While the change illustrated the benefits of the alternative approach to parallelism (Section 4.1), the performance gains were reported on experiments with a large pop-

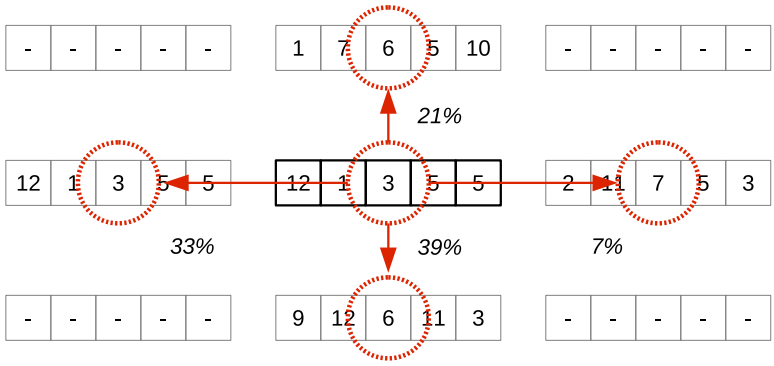


Figure 4.4: Common design of the two parallel recombination operators.

ulation (40×40).

This section presents another application of the proposed alternative approach, to a CGA but with smaller population sizes, the more usual case. In addition to the reduced population size, we target a hardware platform with more parallelism, a GPU. We design a CGA to solve the scheduling optimization problem for the GPU [167]. The optimization problem is described in Sections 3.2.1 and 3.2.3. The modified CGA applied to the scheduling problem is called GraphCell.

4.3.1 Parallel Synchronous CGA

GraphCell is a highly parallel synchronous cellular genetic algorithm for GPU architectures. We depart from the usual population-based decomposition, where one individual (or solution) is evolved by a single thread. Instead, new recombination operators run in a single thread per part of a solution (a task assignment in the context of the scheduling problem considered). In addition, each solution of the population is recombined in parallel. This leads to a high number of threads, which scales with the size of the problem, regardless of the chosen population size. The details are summarized in Algorithm 9, and a description is provided below.

GraphCell introduces two new recombination operators. In order to study the effect of these operators, in isolation and interaction, they are combined into a single operator. The combination is called Uniform Proportional Recombination (UPR). The two recombination operators of UPR are specifically designed for algorithms implementing cellular topologies in massively parallel architectures like GPUs.

Figure 4.4 shows how the recombination operators update each task of a solution. The arrays displayed represent the solutions, each cell of the array corresponds to a task assignment. The number in a cell of the array is the machine to which this task is assigned (the task is identified by it's index in the array). The circled task of the center solution shows the task

being updated. Nine solutions are shown. The solutions directly above, below, to right of, to the left of, the center solution define the neighbor solutions. The other solutions are ignored by both recombination operators. The new machine assignment for the circled task is computed by a dedicated thread (if $Tasks$ is the number of tasks of a solution, $Tasks$ threads are run to compute the new solution). The two proposed recombination operators follow the same design: the offspring solution is generated by assigning to each task, the machine of one of the neighboring solutions, according to a proportionate selection mechanism. The operators only differ in the criterion used for this selection:

- Fitness (UPR_f): we choose the assignment for each task of one neighboring solution with a proportionate selection mechanism based on the fitness of the solutions (i.e., the probability for choosing one neighboring solution is given by its fitness value over the sum of the fitness of all the neighbors). Therefore, parent i will be chosen with probability:

$$PF_{Solution_i} = \frac{Fitness(Solution_i)}{\sum_{j=0}^{No.\ Solution} Fitness(Solution_j)} \ .$$

- Completion time (UPR_{ct}): we choose the assignment for each task of one neighboring solution with a proportionate selection mechanism based on the estimated time to complete on the machine to which the considered task is assigned in the neighboring solution. In this case, the probability of choosing the assignment of neighboring solution i is:

$$PCT_{Solution_i} = \frac{ETC_{i,j}}{\sum_{k=0}^{No.\ Solution} ETC_{k,j}} \ ,$$

  where $ETC_{i,j}$ is the execution time of task j to the machine to which it is assigned in neighboring solution i. This value is given by the ETC matrix of the considered instance.

We study different combinations of these operators by defining the probability (P_{sel}) to apply the fitness based operator. The probability to apply the completion time operator is $1 - P_{sel}$. This decision is made for each task, and not for the entire solution.

The percentages shown in Figure 4.4 highlight this selection mechanism. From this description, we notice that the total number of threads used for the recombination UPR is: population size $\times$ solution size (which is the total number of tasks). This generates a high number of lightweight threads (more than 10^6), which is well suited to the GPU. Algorithm 9 presents the pseudo-code for GraphCell. GraphCell, the Min-min heuristic and the CGA, is executed only on the GPU.

GraphCell initializes the population of solutions randomly (with a uniform distribution), except for one solution, which is the result of the Min-min

Algorithm 9 Pseudo-code of GraphCell

```
1: // Population initialization:
2: // First, initialize one solution with Min-min (Section 3.2.2):
3: for all Tasks of one solution do
4:     min_ct <<< Tasks >>> (results)
5:     cudaMemcpy (results, temp, cudaMemcpyDeviceToDevice)
6:     n ← 2
7:     while Tasks/n ≥ 1 do
8:         min_task <<< Tasks/n >>> (temp)
9:         n ← n × 2
10:    end while
11:    assign <<< 1 >>> (temp, solution)
12: end for
13: // The rest of population is initialized randomly.
14: // The CGA:
15: while not stop_condition() do
16:    neighborhood_prob <<< Pop >>> ()
17:    upr <<< Pop × Tasks >>> ()
18:    mutate <<< Pop >>> ()
19:    fitness <<< Pop >>> ()
20:    replace <<< Pop >>> ()
21: end while
```

heuristic, also computed on the GPU as presented in Section 3.2.2. UPR is implemented in the two GPU kernels `neighborhood_prob`, and `upr`. Kernel `neighborhood_prob` computes the probability for each solution in the neighborhood to be chosen under fitness proportionality. This kernel is run by one thread per solution, because the fitness is defined per solution, but it could also be recomputed by each thread per task. The kernel `upr` randomly selects (with a uniform distribution) the recombination operator (UPR_f or UPR_{ct}) to apply for a task, according to probability P_{sel}. Then, the kernel randomly chooses a task assignment among the same tasks of neighborhood solutions, applying the proportionate selection mechanism. It computes the different probabilities for the estimated completion time proportionality, if needed, on demand because this is task dependent. Kernel `upr` is run by one thread per task per solution, because the assignment decision for a task is independent of all the other tasks. The other kernels, `mutate, fitness, replace` are launched with one thread per solution. Kernel `mutate` changes the assignment of a randomly chosen task, to a randomly chosen machine. Kernel `fitness` computes the makespan for the solution. Kernel `replace` replaces the old solution with the new computed solution if it is not worse (in terms of makespan). These last three kernels are standard operators in genetic algorithms. The stopping condition can either be a maximum number of evaluations, or time (wall-clock).

Table 4.3: Configuration for GraphCell

Population size	8×8 solutions
Population initialization	1 solution with Min-min and the rest random
Neighborhood	von Neumann
Recombination	Uniform Proportional Recombination (UPR)
Recombination probability	$p_r = 1.0$
Mutation	Random
Mutation probability	$p_m = 1.0/\text{numberOfTasks}$
Replacement	Replace if Better or Equal
Termination condition	100,000 generations

4.3.2 Experimentation

We present in Section 4.3.2 the GraphCell configuration, and its performance (in terms of solution quality) in Section 4.3.2.

We study six different instance sizes, $512 tasks \times 16 machines$, 4096×128, 8192×256, 16384×512, 32768×1024, and 65536×2048. The instances were generated according to Section 3.2.2.

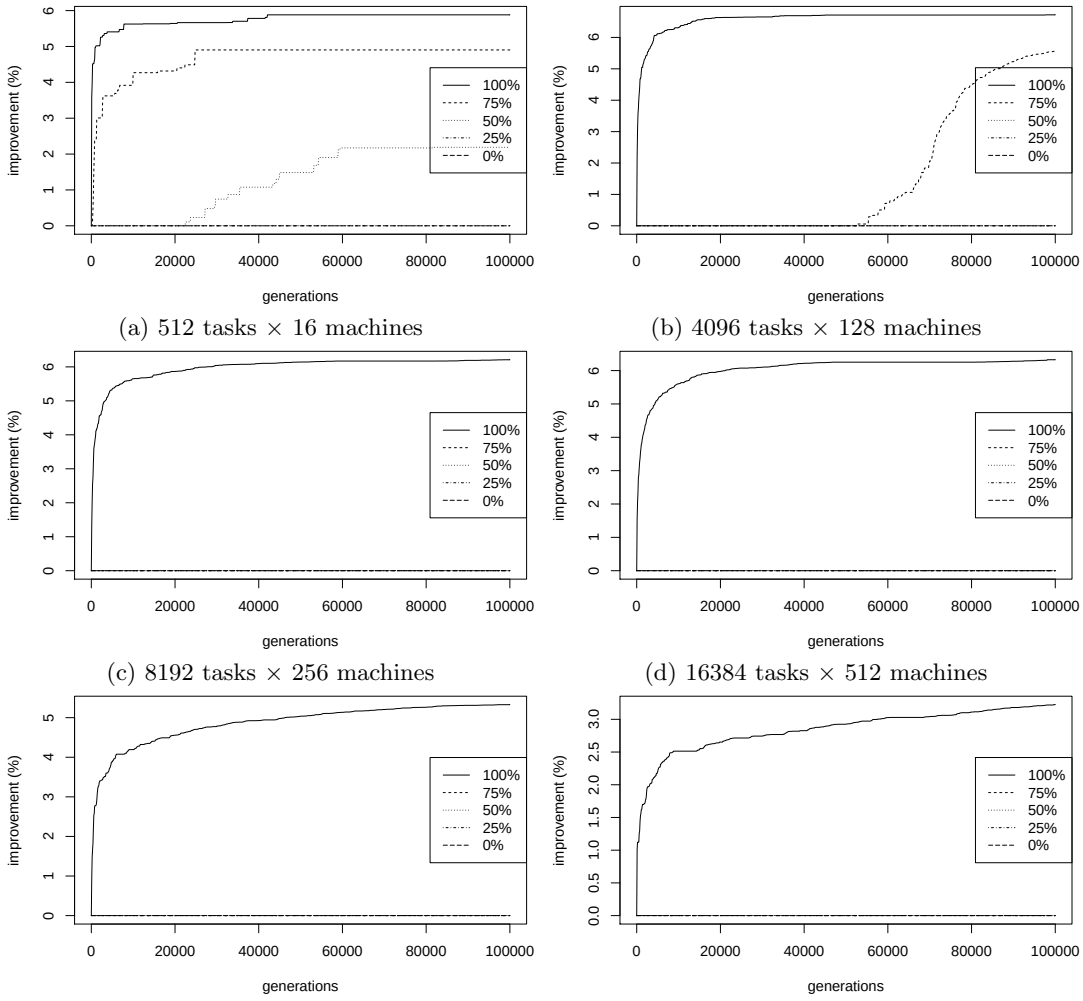
The computer used in the experiment is a Dell Precision T5400, which includes an Intel Xeon E5440 processors (dual processor, of 4 cores each), clocked at 2.83 GHz, with 16 GiB of main memory. The computer runs the GNU/Linux operating system Ubuntu Server (64-bit kernel, version 2.6.35-27). The GPU installed on this computer is a Nvidia Tesla C2050, with CUDA driver version 3.20 (capability 2.0). This GPU holds 14 multiprocessors (of 32 cores each), clocked at 1.15 GHz, and with a global memory of 2 GiB. The GPU kernels are written in CUDA C.

Algorithm configuration

Table 4.3 presents the configuration of the algorithms. The neighborhood used is von Neumann, also called L5. The recombination operators are combined into the new Uniform Proportional Recombination (UPR). The solutions are encoded in an integer array of length the number of tasks, where the content of a cell is the machine to which the task (denoted by the array index) is assigned. The mutation operator consists of assigning a random machine to a task with probability one over the number of tasks. Finally, new solutions replace previous solutions in-place, if their fitness is less or equal than the previous solutions. The execution stops after 100,000 generations (each solution is evolved 100,000 times).

Results

In this section, we analyze the quality of the solutions found by GraphCell, across different problem sizes. The effect of each of the two recombination



(a) 512 tasks × 16 machines

(b) 4096 tasks × 128 machines

(c) 8192 tasks × 256 machines

(d) 16384 tasks × 512 machines

(e) 32768 tasks × 1024 machines

(f) 65536 tasks × 2048 machines

Figure 4.5: Improvement of best solution, compared to Min-Min solution

operators is explored by varying the UPR parameter P_{sel}. When $P_{sel} = 1$, then the fitness proportionate selection is chosen (i.e., UPR_f). When $P_{sel} = 0$, then the estimated completion time proportionate selection is chosen (i.e., UPR_{ct}): other values reflect the interaction of the two operators. The results shown were obtained after 20 independent runs of the algorithm, solving a different problem instance every time.

Figure 4.5 shows the evolution of the improvement of the best solution in the population (averaged over the 20 instances), compared to the solution generated with the Min-min heuristic, for the five different configurations of UPR. These configurations are different values of the P_{sel} probability. These values are $P_{sel} = 1$, 0.75, 0.5, 0.25, and 0. Therefore, these plots show how the performance of the CGA part of GraphCell improves upon Min-min

with the five considered UPR configurations. We can observe that increasing the weight of the fitness proportionate operator improves the performance of the algorithm. Indeed, from the first generations the algorithms using $P_{sel} = 1$ (this is UPR_f) and 0.75 are able to improve Min-min solution by 3%, while the original algorithm with $P_{sel} = 0.5$ is not able to achieve this improvement in the 100,000 generations allowed. Therefore, GraphCell improves very quickly the initial Min-min solution, and the improvement continues until the end of the run, especially for the biggest instances.

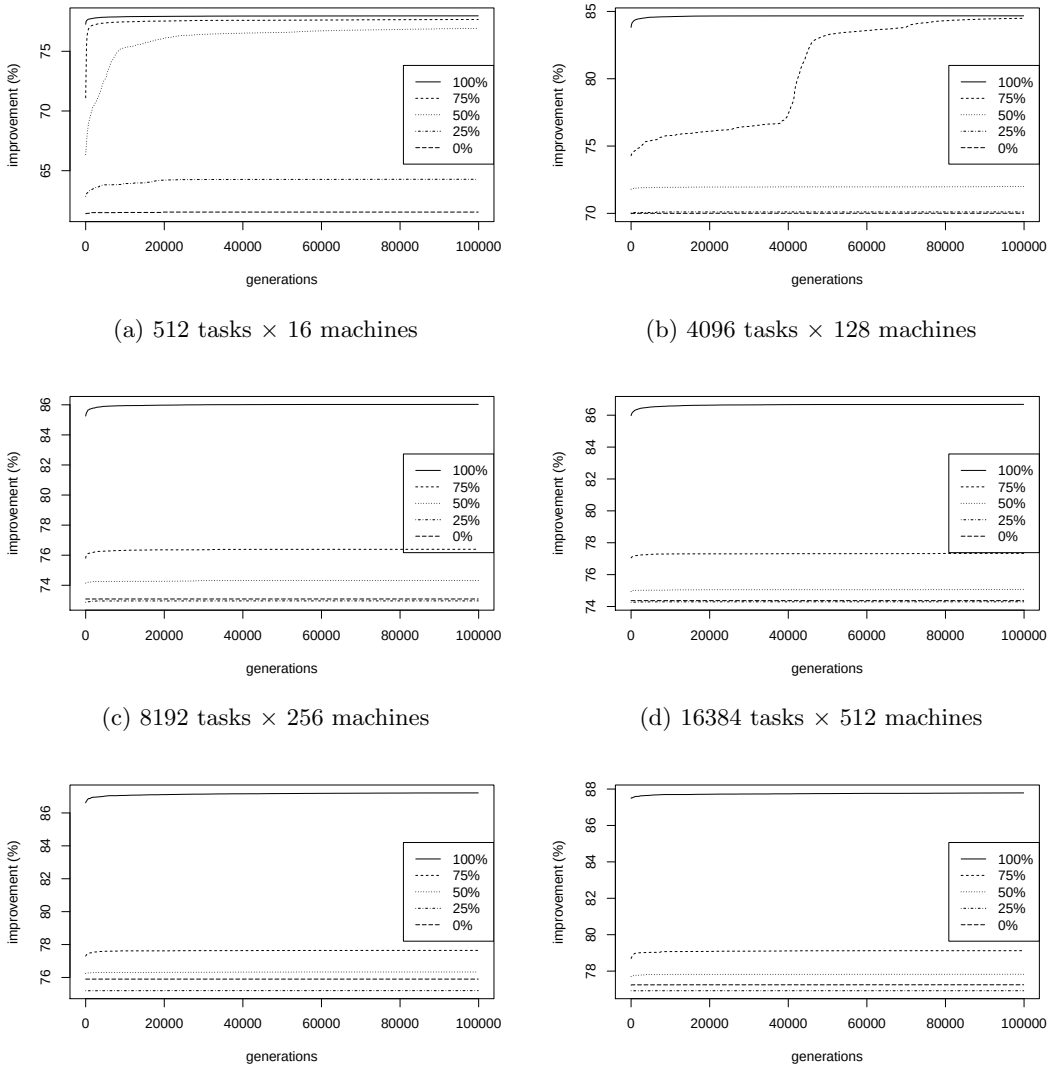
It is well known that increasing the neighborhood size leads to faster convergence speeds in cellular evolutionary algorithms [178]. Therefore we tried to accelerate the convergence speed of the algorithm by employing a larger neighborhood, namely C13, composed by the solution itself plus the 12 nearest ones in Manhattan distance. We ran the GraphCell algorithm for 50,000 generations with the two neighborhood structures on the 20 instances of each of the problem sizes, but did not notice any statistically significant differences (according to the unpaired Wilcoxon signed rank test) on the quality of solutions found (the p-values obtained were 0.4291, 0.718, 0.698, 0.57, and 0.5291 for the instances from smaller to larger, respectively). However, the runtime of the program increased with the neighborhood size.

Figure 4.6 reports the evolution of the average fitness over the population for the GraphCell algorithm with the five different UPR configurations. Since the population is initialized with random solutions, except one with the Min-min heuristic, the initial average fitness is low. We see that the results are similar to the ones discussed in the previous experiment. The value of P_{sel} has a significant impact on the improvement of the population during the run, and the larger the number of tasks to schedule, the more important the difference with $P_{sel} = 1$ is. With respect to the other configurations, only for the smallest instances two of them are able to perform significant improvements on the average quality of the solutions in the initial population: $P_{sel} = 0.75$ for 512 and 4096 tasks, and $P_{sel} = 0.5$ for 512 tasks.

4.3.3 Conclusion

In this section, we presented a new design for a parallel CGA, to solve a scheduling optimization problem, on the GPU. This new design goes beyond the usual population-based decomposition of parallel CGA, and extracts greater data-parallelism at the solution level, usually a large array of primitive types (such as integers or floats). This is often the case in solutions to combinatorial optimization problems. Even small problems (512 tasks assigned to 16 machines), arranged in a normal genetic population (64 solutions), can exploit thousands of weak cores.

The design illustrates the alternative approach of the introduction: the new parallel CGA does not preserve the original CGA. The modification is introduced in the parent selection and recombination operators, with two



(a) 512 tasks × 16 machines

(b) 4096 tasks × 128 machines

(c) 8192 tasks × 256 machines

(d) 16384 tasks × 512 machines

(e) 32768 tasks × 1024 machines

(f) 65536 tasks × 2048 machines

Figure 4.6: Improvement of fitness average across population, compared to fitness average across initial population

new operators that merge parent selection and crossover in a parallel algorithm. The new design is specifically designed for the GPU AWN, there is no equivalent sequential implementation (although it could be ported to a sequential machine). Other genetic operators could be modified in a similar fashion.

4.4 Summary

In this chapter we experimented with two parallel designs that deliberately change the original algorithm (and not just the algorithm's implementation), the alternative considered here. The first parallel design, Section 4.2, proposed a simplification in the population-based data-parallel CGA that, as a side-effect, introduces an error. This simplification improves the scalability of the program, by reducing the inter-thread communication. The error does not degrade the search capability of the algorithm, it improves it instead. This observation can be seen as a further encouragement to reconsider an algorithm in order to improve some property, in our case the scalability with the number of cores. The second parallel design, Section 4.3, replaces the standard genetic operators with more data-parallel alternatives, that implement a different algorithm. The result is several orders of magnitude more parallelism, available even to small problem instances.

These experiments illustrate that in order to extract the needed parallelism required by the AWN, the algorithm to parallelize can be modified without loss of capability. However, these experiments point to two weaknesses. First, only one of the algorithm extracts parallelism from small problems. Second and more significant, these designs are completely ad hoc, and manually designed, at the cost of time-consuming trial-and-error. Modifying an algorithm for increased parallelism is only practical if we have a more systematic method to find alternative designs.

Chapter 5

Algorithm Design with Sensitivity Analysis

5.1 Introduction

The previous chapter showed that changing the algorithm (and not the only the program) can improve scalability, a desired property for the concurrent algorithm, and still preserve its function or capability. The tested algorithm is a search algorithm; its capability can be measured as the quality of the solutions to an optimization problem. This raises the question: how to systematically identify what to change in the algorithm?

There are two directions to this. One direction is to first make changes that improve on the desired property, and then observe the resulting capability of the algorithm, hoping to preserve it, at least. This is the approach of the previous chapter. It is a natural approach because properties are easily measurable, for example: a profiler to measure the runtime property. The disadvantage of this approach is its uncertainty: the consequences of a change on the algorithm's capability are not predictable, therefore the process is essentially trial-and-error. Another direction is first to determine how each part of the algorithm contributes to its capability, then to preserve the key parts (semantically; the code can be different), but to change the rest to improve on a property. The changes made are better informed on the impact to the new algorithm's capability.

However, the second direction raises another question: how to determine the contribution of each part in an algorithm? Further decomposing this question, we may ask what method can help us. One answer is statistical analysis, and more specifically SA. The original purpose of SA is to analyze a model through the interaction of its parameters on the model's output, for example: to verify the occurrence of a known behavior that the model is intended to capture.

Section 5.2 introduces the SA method used in this chapter. We have

mentioned it earlier, in Section 3.1.4. SA is applied to the analysis of an EA: how its different components influence the capability of the algorithm, for the specific problem tackled. Then, in Section 5.3, the results of the SA are exploited to guide the modifications to the algorithm, to improve on a chosen property, while preserving the algorithm's capability.

5.2 Tuning Program Parameters with Sensitivity Analysis

We apply a generic SA method to measure the influence and interdependencies of the parameters on the capability of an EA [214]. The EA is the PA-CGA, presented in Sections 3.2.3 and 4.2.

The ultimate intention of the SA experiments is to guide algorithm modifications. However, the SA of an EA is immediately beneficial to set its various parameters for improved solutions. The nature-inspired EAs function by iteratively applying specific operators in order to modify potential solutions to a problem and converge to an optimal or near-optimal solution. Despite their application success, EAs remain highly dependent on their parameterization but also on the problem. It is common practice to augment a generic EA with problem specific components. As mentioned by De Jong in [215], the No Free Lunch theorems, stating that no single algorithm that will outperform all other algorithms on all classes of problems, induce several key questions, including: "what EA parameters are useful for improving performance?" Although a lot of work has been conducted in the field of parameter setting for EAs, most of these focused on searching for the best parameter values without considering if these parameters have a direct influence on the EA capability (its ability to find good solutions). Moreover, the effect of the different parameters and components of an EA can also be quantified with SA.

5.2.1 Sensitivity Analysis of a Program

SA originally targets the modeling activity. Here, we seek to apply this technique to a program.

The context of EA

Parameter setting can greatly influence the performance of EAs and therefore focused the interest of many researchers. Comprehensive surveys have been introduced by De Jong in [215], Eiben [216] and more recently by Kramer in [3].

As mentioned by Maturana et al. in [217], one of the main problems is to assess which parameters can lead to the algorithm transformation, i.e. improvement. They proposed a classification of parameters, distinguishing

behavioral parameters (operators probabilities, population size) and *structural parameters* (encoding, choice of operators). A similar classification was proposed by Smit and Eiben in [218] distinguishing between *numerical* and *symbolic* parameters. In this analysis we focus on behavioral, respectively numerical parameters setting.

The EA parameter space can be explored offline (before the search) or online (during the search). Eiben in [219] classified these parameter techniques as parameter *tuning*, and *parameter control*. In this work we are interested in parameter setting before the run (i.e. tuning), for which a taxonomy extension has been proposed by Kramer in [3] (see Fig. 5.1).

Tuning by hand induces user experience for setting the EA parameters beforehand. This solution is largely predominant in the literature in which parameters are usually set based on empirical evaluations as mentioned in [217].

The second tuning class, Design of Experiments (DoE), refers to Bartz-Beielstein work on Sequential Parameter Optimization (SPO) [220] which is a heuristic combining classical and modern statistical techniques. The objective is to design the experimental plan prior to the experiments.

Other works have attempted to analyze the sensitivity of parameters, but limited to the study of the independent influence of parameters values on the fitness. De Castro et al. in [221] studied the sensitivity of three parameters (number of antibodies, number of generated clones and amount antibodies to be replaced) of their Clonal Selection Algorithm (CLONEALG). Similarly, Ho et al. in [222] have analyzed the sensitivity of parameters of their Intelligent Genetic Algorithm (IGA), including mutation and crossover probabilities. In [223] Min et al. analyze the sensitivity of the population size and the termination condition (maximum number of generations) of a standard GA on a reverse logistics network problem. Finally, Geem et al. in [224] analyzed the sensitivity of Harmony Search (HS) parameters

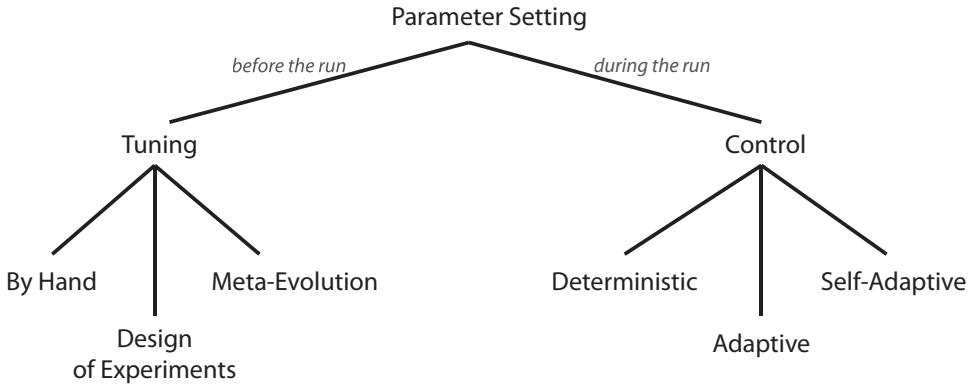


Figure 5.1: Parameter setting in EA's taxonomy [3].

(harmony size, memory considering rate and pitch adjustment).

The last parameter tuning class, meta-evolution, is also referred to as nested evolution. This is a two-level evolutionary process in which one algorithm optimizes the parameters of the second one. A recent approach has been proposed by Nannen and Eiben in [225], Relevance Estimation and Value Calibration of Evolutionary Algorithm (REVAC) parameters. It estimates the expected performance of the EA when parameter values are chosen from a Probability Density Function (PDF) and includes a measure of the parameter relevance (normalized Shannon entropy).

SA for EA

This section describes how SA can correct the drawbacks of the aforementioned approaches. SA tries to identify how uncertainty in each of the parameters influence the uncertainty in the output [163] of a model. This technique allows to answer the following question (among others): which factors cause the most and the least uncertainty in the output (screening). This measures the importance of factors in the model analyzed. It is useful in the DoE and parameter setting, because it allows to focus on the most influential factors, possibly setting arbitrary values to the least influential ones.

Moreover, this knowledge is also useful at design-time. The designer of a model intuitively develops an idea of its behavior. SA allows the designer to verify his hypothesis, and modify the model accordingly. This study therefore proposes to use SA to study the influential parameters of an EA on a specific problem class, i.e. scheduling problem of independent tasks in a grid. The objective is to reduce the chosen algorithm's parameter search space.

There are several ways to conduct a SA. Section 5.2.1 listed a few. Before presenting the suggested method, the desired characteristics of a method for SA are presented below. The method should:

- be model independent (it does not place requirements on the type of model to work on),
- evaluate the effect of a parameter while all others are also varying,
- cope with the influence of scale and shape (the probability density function and its parameters),
- quantify the influence of the uncertainty in factors,
- capture the interaction between factors.

These desired properties restrict the possible methods (such as using entropy as a measure of output uncertainty [225]). The chosen method is

based on decomposing the variance of the output, as introduced by Saltelli et al. in [163]. The exact implementation used is an extension to the Fourier Amplitude Sensitivity Test, called Fast99 [164]. This method allows the computation of first order effects and interactions for each parameter. Parameters interaction occurs when the effect of the parameters on the output is not a sum of their single (first order) effects.

5.2.2 Experimentation

The chosen method benefits from the properties presented in Section 5.2.1, therefore there are no model specific restrictions. First, the goal of the analysis must be stated and the output of the model defined accordingly. For an EA, this can be the quality of the solutions, the number of evaluations, the runtime of the implementation, etc. In our case, the output is the solution quality, which represents the fundamental function or capability of the algorithm. For each parameter of the model analyzed, the range of possible of values is required, along with their distribution in the range. These values come from experts in the application domain, or from the literature. Unless there are many parameters (greater than 30) or if the evaluation takes too much time (due to the number of parameters combinations), the Fast99 method mentioned in Section 5.2.1 is suitable. Otherwise, the qualitative method of Morris is better suited (it is a One-At-a-Time method, or OAT). The method then produces a list of parameter combinations, for which the model is evaluated. In the case of an algorithm, the implementation of the algorithm is run with the prepared parameter combinations. The number of combinations is $samples \times Nb\_parameters$. The method for the SA then collects the evaluation results and presents the linear and non-linear influence of each parameter. The next sections present its application.

Experiments configuration

The presented sensibility analysis is performed on an EA. This EA algorithm is the PA-CGA implemented for multi-core processors presented in Sections 3.2.3 and 4.2. The PA-CGA solves the scheduling of independent tasks problem, described in Section 3.2.1.

The PA-CGA is configured as follows. The population is initialized randomly, except for one individual. The schedule for this individual results from the Min-min heuristic [144]. The linear 5 neighborhood, also called Von Neumann neighborhood, is composed of the 4 nearest individuals, plus the individual evolved. The two best neighbors are selected as parents. The recombination operators used are the one-point (opx) crossover. The mutation operator moves one randomly chosen task to a randomly chosen machine. The newly generated offspring replaces the current individual if it improves the fitness value.

Table 5.1: Uncertainty in the model parameters

Factor	Distribution	Range of values
Population size	uniform	$8 \times 8 - 32 \times 32$
Mutation probability (P_mutation)	uniform	0.1 – 1.0
Mutation iterations (Iter_mutation)	uniform	1 – 5
Local search probability (P_search)	uniform	0.1 – 1.0
Local search iterations (Iter_search)	uniform	1 – 10
Load for local search (Load_search)	uniform	0.1 – 0.9
Threads	uniform	1 – 4

The parameters of the EA to analyze, called factors in the context of SA, are summarized in Table 5.1. For each factor considered in this study, a uniform distribution of the values is considered since we have no a priori indication of the correct values. Population size represents the dimension of the square shaped grid of the CGA, which can range between 8×8 to 32×32 individuals. Crossover rate is taken from a range between 0.1 to 1.0. Mutation is defined by its rate, ranging between 0.1 and 1.0, and the maximum number of mutations, ranging from 1 to 5. Local search is defined by the same properties (rate between 0.1 and 1.0 and maximum number of iterations between 1 and 10). The value range for the number of least loaded machines to consider is 0.1 to 0.9. This is an additional parameter of the problem-specific local search operator. Finally, as the algorithm can be parallelized, the number of threads varies between 1 and 4. The stop condition for each run of the PA-CGA is 100 generations. Each set of factors generated for the analysis is used for 4 runs. The result is the average of the makespan (solution quality for the scheduling problem considered) over those 4 runs. The SA therefore considers a total of 6400 parameters combinations. Sensitivity analysis is performed on the algorithm for two different instance files: *u_c_512x16_hihi_1* and *u_c_512x16_lolo_1*. The intention is to discover if different problem instances modify the factor prioritization results. The Fast99 implementation of the SA is provided by the R Sensitivity Analysis package [226].

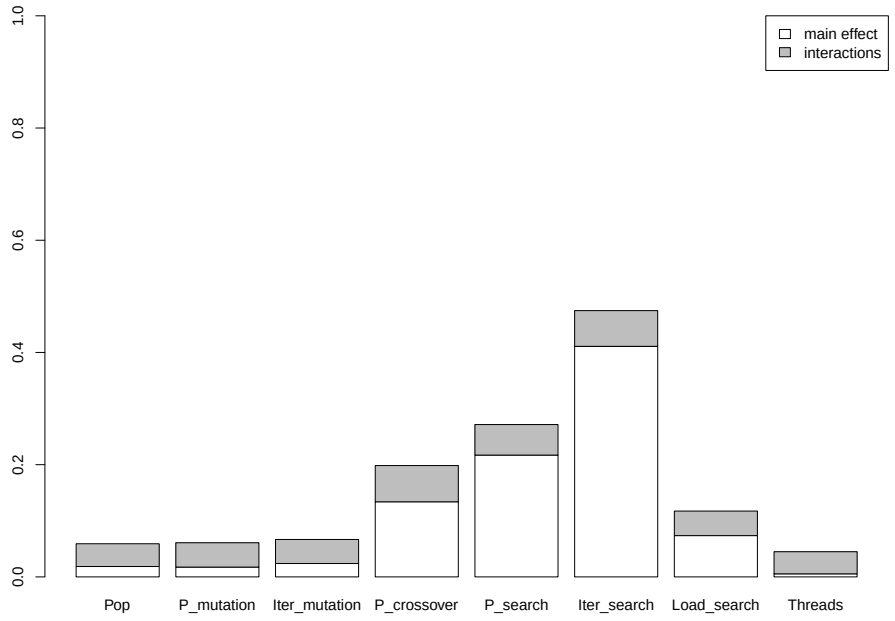


Figure 5.2: SA, *hihi* instance

Results

Figure 5.2 presents for each factor, their linear and non-linear (or interaction) effects on the output for the problem instance with high tasks and resources heterogeneity: the quality of the solution (the average makespan over 4 independent runs).

The benefits of the SA are immediately visible. Indeed, the local search parameters and notably the maximum number of iterations, influence the most the output. It is indeed twice more important than the second most influential parameter, the local search rate. This result is consistent with related works in the scheduling literature which enlightened the importance of the local search when dealing with hybrid metaheuristics [227]. This also justifies the hand tuning of the parameters performed for [177] The third most important parameter the crossover rate. This is highlighted in Figure 5.3 which analyzes the effects on the output of the GA parameters, thus using fixed values for the local search. It appears that the crossover rate is at least six times more important than all the other GA parameters.

These results also highlight that several parameters play a limited role, i.e. population size, mutation rate and iterations as well as the number of threads. This is also beneficial because values which have a positive impact on other aspects of the algorithm, such as runtime, can be selected without

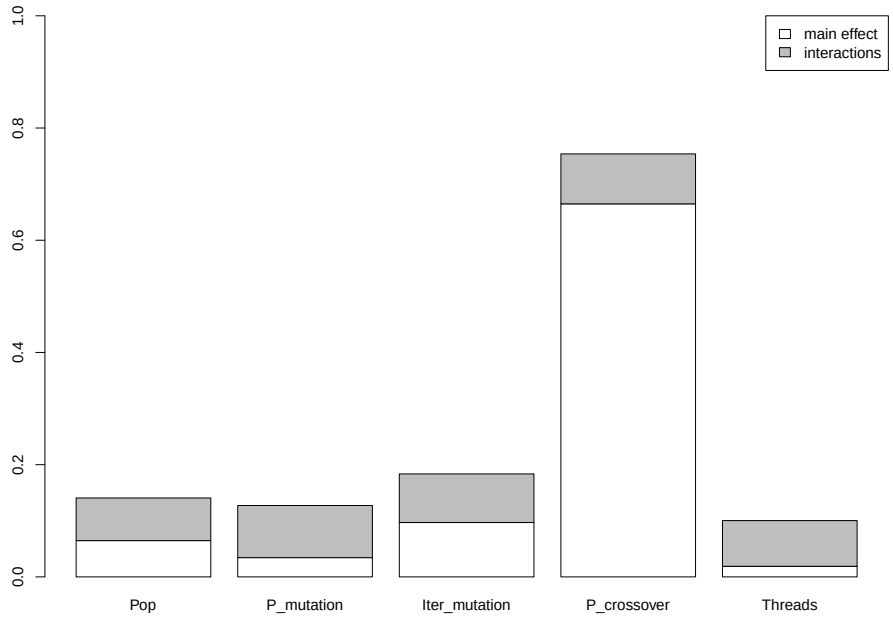


Figure 5.3: SA, *hihi* instance with fixed local search parameters

impacting the quality of the solutions. Indeed, this algorithm was designed to be run for a limited period of time (wall clock), therefore choosing a smaller population size and a higher number of threads will allow more generations.

Figure 5.4 shows the same analysis for the instance with low tasks and resources heterogeneity. The two most influential parameters are similar to the *hihi* instance, local search iterations followed by the of local search rate. One difference can be noticed at the level of the third parameter in terms of importance. This parameter now consists in the EA population size while the crossover rate was used for the *hihi* instance. As can be seen in Figure 5.5, crossover has indeed 40% less influence than population size. Finally the load for local search has almost no influence on the output in the *hihi* case. Figure 5.5 shows that there are significant interaction effects, which mean that the remaining parameters combined, influence the output more than individually. The interaction amount summarizes all the interactions (between two, three, etc parameters).

5.2.3 Conclusion

In this first section, a variance based SA has been proposed to study the influence and interdependencies of the parameters of a PA-CGA. Experi-

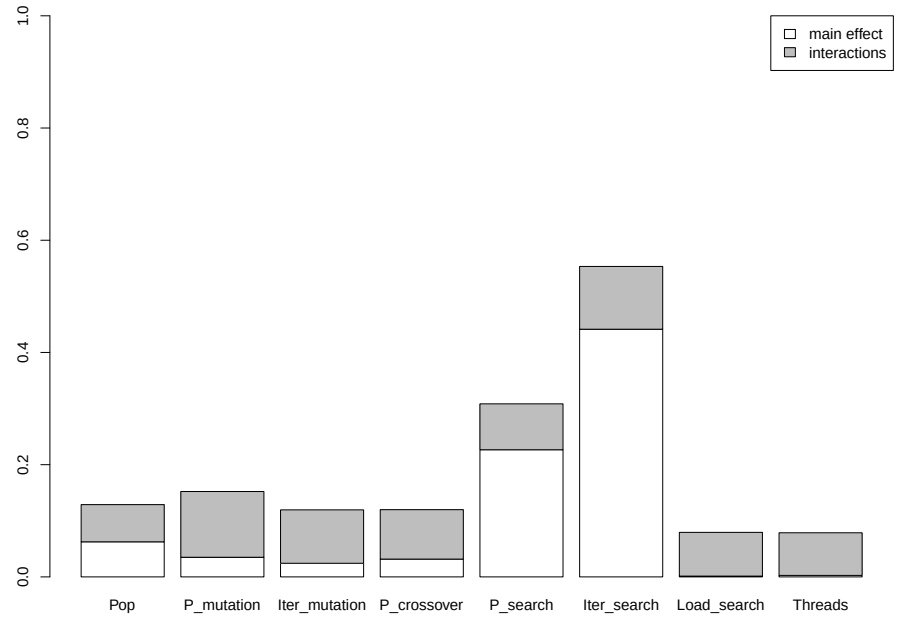


Figure 5.4: SA, *lolo* instance

mental results on two different instance classes of a scheduling problem have shown that, for both problem instances, the two most influential parameters are those related to local search. As expected, the genetic algorithm's parameters have a limited influence on the solution quality, except for the crossover rate and the population size, respectively for the *hihi* and *lolo* instances. Current implementations are available [226] to make this analysis a systematic step in any EA experiment. However, sensitivity analyses are expensive computationally, but an improved setup can result in faster execution (such as parallel execution).

5.3 SA-Guided Modifications to an Algorithm

In this section, the SA of a CGA with local search, presented in Section 5.2, is used to re-design a new and faster heuristic, called Two-Phase Heuristic (2PH), for the problem of mapping independent tasks to a distributed system [228, 229]. The proposed heuristic finds better solutions than Min-Min, and solutions of similar quality to the original CGA but in a significantly reduced runtime ($\times 1,000$ faster). The discovery process for these modifications is not trial-and-error. In this section, SA is not targeted at parallelism, nor energy-efficiency, but decreasing the runtime of an

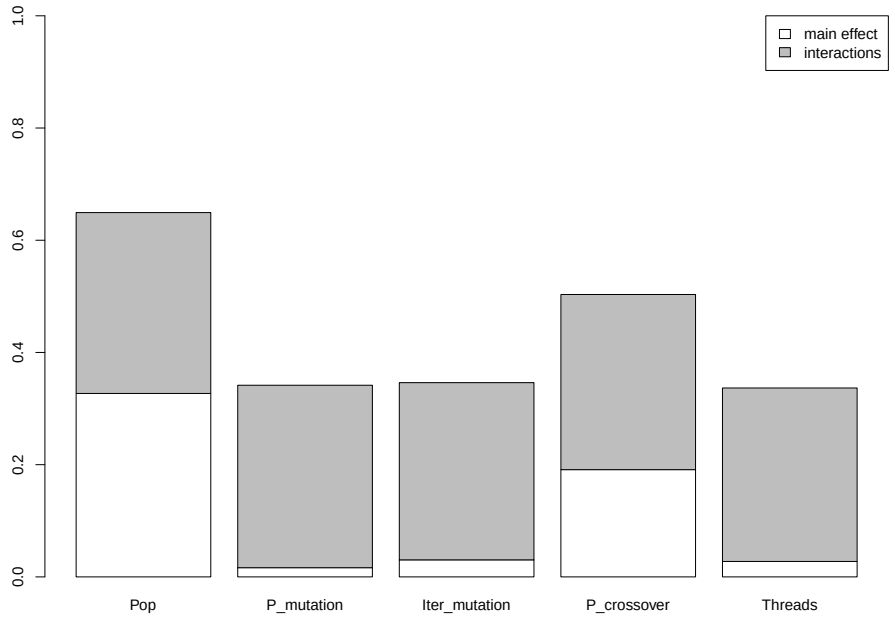


Figure 5.5: SA, *lolo* instance with fixed local search parameters

algorithm. It is exploited to illustrate an approach.

5.3.1 A Modified PA-CGA: the Two-Phase Heuristic

This section presents the proposed modifications to the original PA-CGA. First, the key findings of the SA, detailed in the previous section, are listed. Then, the implications are applied to the design of a new algorithm.

Summary of the SA

Our original algorithm for the SA guided modification is the PA-CGA, introduced in Section 3.2.3. Summarizing: the PA-CGA is a CGA, where one solution in the initial population is the output of the Min-Min algorithm, it also includes a problem-specific local search applied to every new solution generated. Finally, it is designed for concurrency, by splitting the population into several, partially independent blocks.

The SA conducted in Section 5.2.2 showed, Fig. 5.2, that the local search parameters and notably the maximum number of iterations influence the solution quality the most. Moreover, the number of local search iterations is twice as influent as the second most influential parameter: the local search rate. The chosen SA method is quantitative, therefore it justifies such com-

parisons, in contrast to qualitative methods that can only indicate the order in importance. The result of the analysis is consistent with related works in the literature that acknowledge the importance of local search in metaheuristics [227].

The results can also be interpreted to show that the other parameters play only a limited role for the considered problem instances, *i.e.*, population size, mutation rate and mutation iterations as well as the number of threads. The relative unimportance of the number of threads and population size on solution quality is significant. In Sections 3.2.3 and 4.2, concurrency was introduced in the PA-CGA by splitting the population across concurrent threads, to exploit multi-core parallelism. This source of parallelism has the disadvantages of requiring large population size, and exposed limitations in scalability with the number of cores. The SA measured that these disadvantages do not improve the capability of the algorithm much.

2PH description

As a consequence from the SA results, the modified algorithm, 2PH, is simply the sequential execution of Min-Min, followed by a single execution of the local search operator H2LL, both taken as is from the PA-CGA. The genetic evolution is removed completely. In addition, the number of iterations for the local search H2LL is increased from 5 to 30 or 100 (both values are experimented). The local search is performed only once for 2PH. In the original PA-CGA, the local search H2LL was applied on each new individual in the population, at each generation. Therefore, even a greater number of H2LL iterations, still performs far less local searches than in the PA-CGA.

Although the 2PH algorithm is simple, and considerably faster, the SA provides us with *a priori* evidence that it should also perform well (find good quality solutions). The next section reports exactly how well, in comparison with the original algorithm and Min-Min.

5.3.2 Experimentation

Experiments configuration

The benchmark application used to evaluate the 2PH algorithm is the independent task mapping problem, presented in Section 3.2.1. The quality of a solution is called makespan (the time when the last task mapped completes).

The 2PH algorithm is compared to Min-Min, and to the original PA-CGA. Wall-clock times for the 2PH and the PA-CGA implementations are useful to measure the speed of the algorithms, as mapping independent tasks can benefit from fast execution times. The experiments were performed on an Intel Core 2 Duo CPU P8800 @ 2.66 GHz running under Gnu/Linux

Table 5.2: Settings for the comparison with other algorithms in the literature.

Parameter	Value
Instance size	128 tasks $\times$ 16 machines
Instance classes	12
Instances per class	30
Runs per instance	10
PA-CGA runtime	3 seconds
PA-CGA population	8×8
PA-CGA threads	1
PA-CGA mutation probability	1.0
PA-CGA mutation iteration	1
PA-CGA crossover probability	1.0
PA-CGA search iterations	5
2PH search iterations	30, 100

operating system. Table 5.2 summarizes the different points of comparison for the evaluation of 2PH.

A total of 360 instances were used in the comparison (30 instances of 12 classes). The problem instances generated for this evaluation belong to different classes of ETC matrices. The classification is based on three parameters: (a) task heterogeneity, (b) machine heterogeneity, and (c) consistency [160]. In this work, instances are labeled as *g_x_yyzz* where:

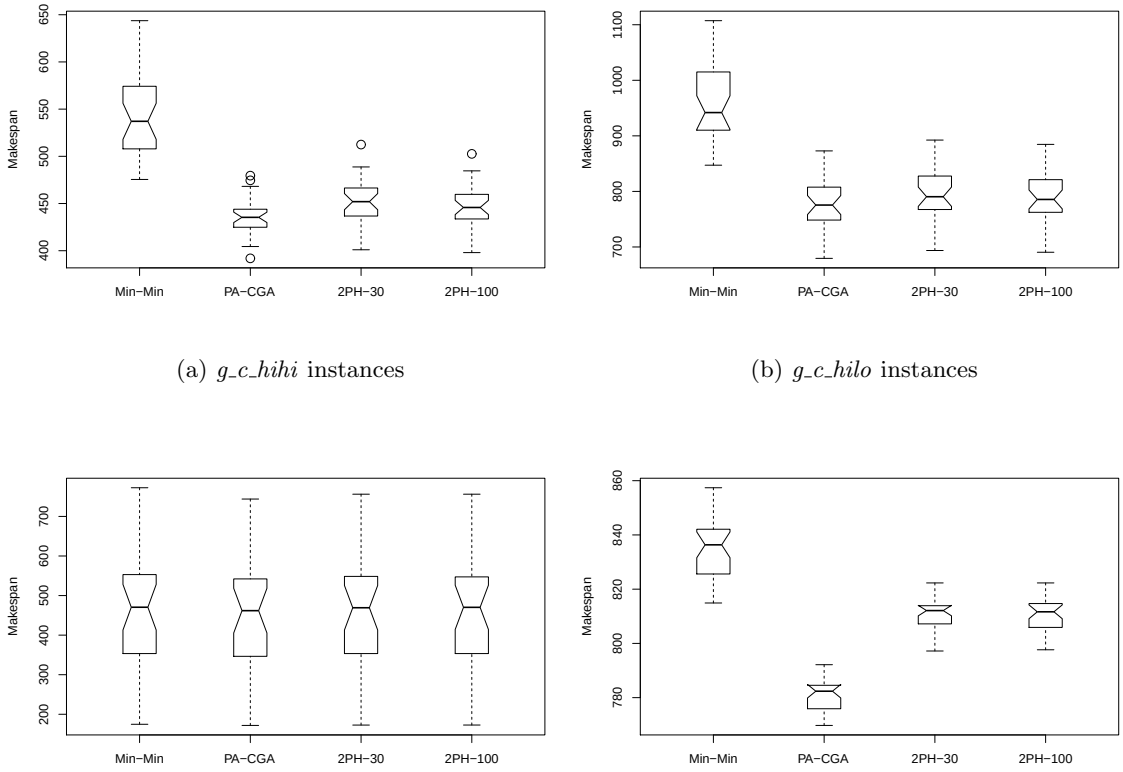
g stands for Gamma distribution (used for generating the matrix).

x stands for the type of consistency (*c* for consistent, *i* for inconsistent, and *s* for semi-consistent). An ETC matrix is considered consistent if a machine m_i executes a task t faster than machine m_j, then m_i executes all tasks faster than m_j. Inconsistency means that a machine is faster for some tasks and slower for some others. An ETC matrix is considered semi-consistent if it contains a consistent sub-matrix.

yy indicates the heterogeneity of the tasks (*hi* means high, and *lo* means low).

zz indicates the heterogeneity of the resources (*hi* means high, and *lo* means low).

PA-CGA is run for 3 seconds, wall-clock time, using 1 thread. The other parameters have identical values to those chosen for the SA. In our previous work [228], the algorithm showed similar performance for different run times ranging from 1 to 5 seconds. These times are far from those used in other previous works [177], where 90 seconds were used as time limit. However,



(a) *g_c_hihi* instances

(b) *g_c_hilo* instances

(c) *g_c_lohi* instances

(d) *g_c_lolo* instances

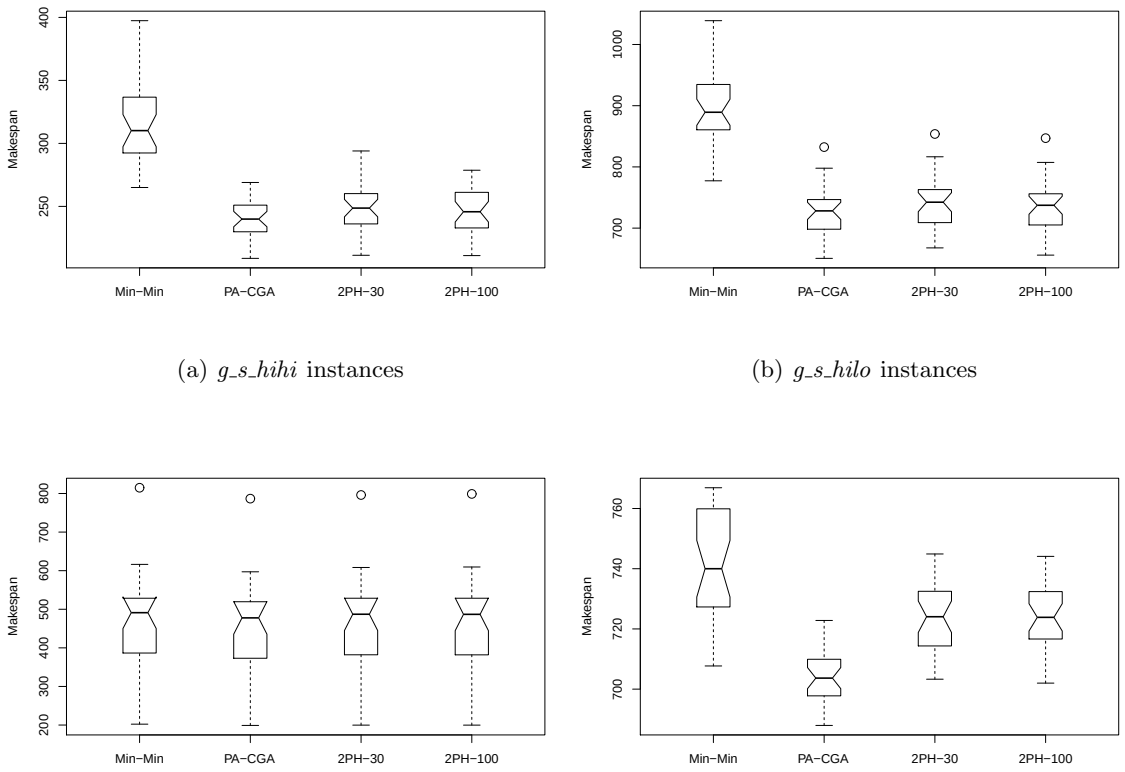
Figure 5.6: Makespan for the consistent instances.

PA-CGA with 1 thread completes over $100,000$ evaluations per second of runtime, which is sufficient for the algorithm to converge to good solutions. Moreover, the SA showed that the number of threads does not play a large role in the search for good solutions.

As mentioned earlier, two versions of 2PH with 30 and 100 iterations were chosen instead of 5 for the PA-CGA. 2PH with 30 iterations only completes in 3 milliseconds per problem instance.

Results

This section presents the experimental results of the different algorithms: (a) the Min-Min heuristic, (b) 2PH with 30 and 100 iterations, and (c) the PA-CGA. The results are shown as box-and-whisker plots. The boxplots are generated with the median of the makespan values obtained after the 10 independent runs for each of the 30 different instances of every problem class. The boxplots show the minimum and maximum values, as well as the first and third quartiles and the median value. The boxes with overlapping



(a) *g_s_hihi* instances

(b) *g_s_hilo* instances

(c) *g_s_lohi* instances

(d) *g_s_lolo* instances

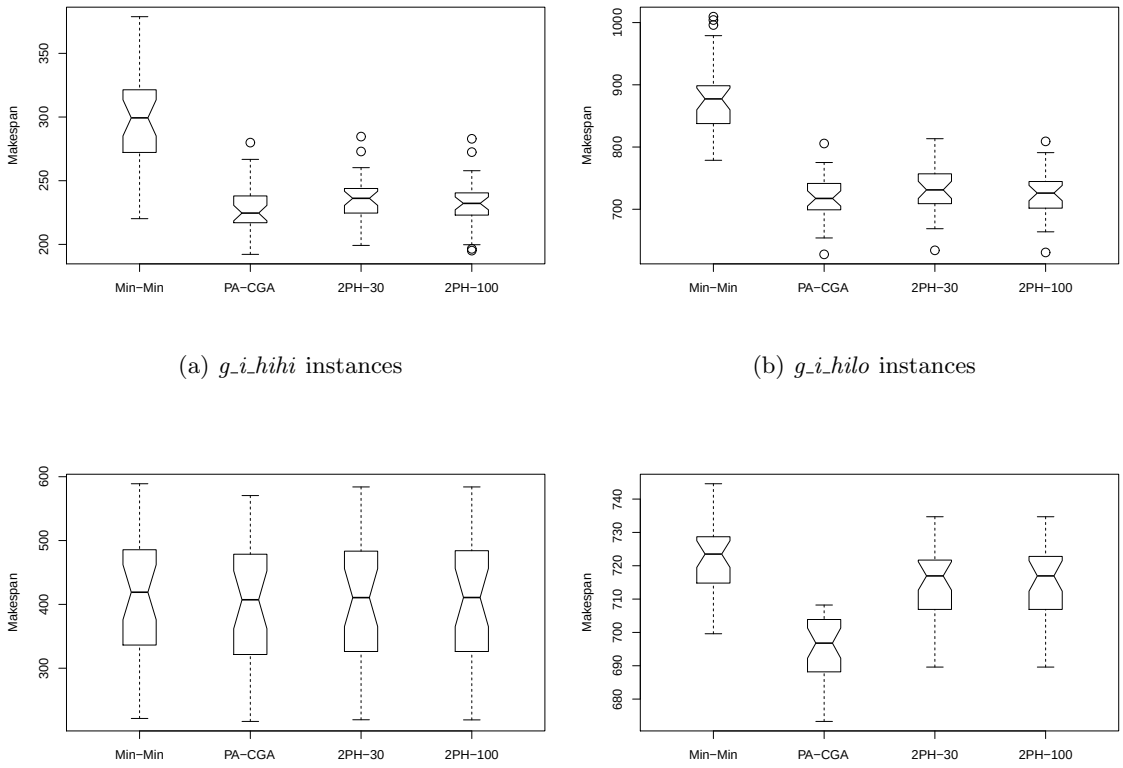
Figure 5.7: Makespan for the semi-consistent instances.

notches mean that there are not statistically significant differences (with 95% confidence level) between the algorithms they represent.

Overall, the 2PH improves the quality of the resource allocation significantly over Min-Min, and provides results of similar quality to PA-CGA, requiring only 3 milliseconds to achieve them.

The results for the consistent, semi-consistent, and inconsistent instances are shown in Fig. 5.6, Fig. 5.7, and Fig. 5.8, respectively. We see that there are not significant differences between 2PH with 30 and 100 iterations for any of the four problems considered with different resource and task heterogeneities. PA-CGA is clearly the best algorithm for problems with low task and resources heterogeneities, and Min-Min is always the worst one for every instance, with the exception of the instances with low task and high resources heterogeneities, for which all algorithms provide similar results. For the other problem classes, the 2PH algorithms show similar performance to the PA-CGA.

To evaluate the overall performance of the compared algorithms on all the problems, we used the Friedman statistic test to perform a ranking of the



(a) *g_i_hihi* instances

(b) *g_i_hilo* instances

(c) *g_i_lohi* instances

(d) *g_i_lolo* instances

Figure 5.8: Makespan for the inconsistent instances.

algorithms according to the solutions found. The Friedman test assigns small ranking values to those algorithms providing the highest solutions. Therefore, as the objective is minimization, those algorithms with highest rank value are the best performing ones. We computed a p-value of $1.955e-10$ with the Friedman test, so there are statistically significant differences with 95% confidence on the performance of the algorithms for all the problems considered in this work.

The rank is shown in Table 5.3. We can see that the ranking supports our conclusions on the results. PA-CGA is the best performing algorithm, followed by the two 2PH versions (very close from each other). However, the 2PH algorithms find the solution about $\times 1,000$ faster than the PA-CGA (2PH runs for a few milliseconds versus the 3 seconds of the PA-CGA). This makes the 2PH algorithm the best option for large scale systems. Finally, Min-Min is clearly the worst algorithm of the compared ones.

Table 5.3: Rank of the algorithms (higher rank is better).

Algorithm	Rank
PA-CGA	3.99
2PH-100	2.725
2PH-30	2.21
Min-Min	1.079

5.3.3 Conclusion

This paper exploits the results of the SA of a parallel asynchronous CGA, with local search. The analysis led to the design of a simple two-phase heuristic for the mapping of independent tasks. The new 2PH heuristic was compared against two algorithms from the literature, (a) the PA-CGA, and (b) the Min-Min heuristic. In most problem instances, 2PH found equivalent mappings in much less time (milliseconds versus seconds) than the CGA. The proposed heuristic also significantly improves the mappings found by the Min-Min heuristic, with little additional computation cost.

5.4 Summary

In this section, a SA method, a statistical procedure designed to evaluate models, was applied to a program: a problem-specific GA. The intent was to gain insight on how each component in the algorithm contributes to the quality of the solutions found, in order to guide modifications that improve a chosen property, with some confidence that its core function is preserved. The targeted property was the speed of execution of the algorithm, and the core function was the search for good solutions to a scheduling optimization problem. Experiments showed that SA provided correct insight: the straightforward modifications deducted from the analysis produced a new algorithm that runs $\times 1,000$ faster while finding acceptable solutions.

The results nevertheless suggest several improvements. First, the property improved was execution speed, and not parallelism. Second, the method is still largely manual, the SA results required interpretation, and the modifications were not automatic.

The finding from these experiments is that statistical methods are not only useful, but also practical because as we are given an original algorithm to improve, we can execute it at will to generate observations. The next section will further investigate statistical processing, as a path to automatic parallelization.

Chapter 6

Savant: Automatic Generation of Parallel Solvers

6.1 Introduction

The previous chapter showed how statistical analysis can successfully guide the search for algorithm modifications, to improve upon an algorithm's property, such as parallelism. One question is could this analysis and modification process be automated to generate new, parallel algorithms? In this chapter, we propose such an automatic parallel program generation, called the Savant approach [230]. The Savant approach relies on statistical analysis, but not on the sensitivity analysis of the previous chapter. Savant is both an approach to parallel program generation, and the name of the resulting parallel algorithm. Both uses of the term should be distinguishable in this chapter.

The Savant approach is inspired by the savant syndrome, as described in Section 6.2.2. The syndrome is a condition that affects up to 10% of people suffering from autistic spectrum disorder. Other occurrences of the Savant syndrome result from brain injury, either from accident or illness. The Savant syndrome is characterized by immediate display of spectacular abilities, such as performing seemingly sequential tasks in a very short time, using a massively parallel machine similar to the AWN: the brain.

In this chapter, we propose a design for a virtual savant, the Savant algorithm. This virtual savant is made parallel by design, it learns the behavior of a given algorithm with statistical machine learning. The given algorithm can therefore be considered parallelized, and the Savant algorithm represents its parallel version. The Savant approach should also be able to parallelize small problems (operating on small data, with limited computation), a valuable property in the context of AWN. Although parallelizing

small problems may not seem useful, it does force to challenge the current sources of parallelism, and may in turn be applied to larger problems.

Automatic parallelization is an ambitious task. The evaluation of the Savant approach to automatic parallelization is limited to one optimization problem, presented in the next section. Section 6.2 presents the Savant approach. Section 6.3 reports on the results of the approach to the considered use case.

6.1.1 Use Case for the Savant Approach

The Savant approach for automatic parallelization, presented in Section 6.2, is applied to the resolution of a combinatorial optimization problem. The problem is the independent tasks mapping problem, previously defined in Section 3.2.1. The key characteristics of the problem are summarized as follows. Solutions to this problem are the assignments of each task to a machine, such that a scheduling objective (makespan) is minimized. Makespan is the time when the last task finishes. Finding the task-to-machine mappings that minimizes makespan is an NP-complete problem. The problem is fully defined by (a) the tasks' estimated time-to-complete on each machine, and (b) the formula for makespan. The estimated execution times are presented in a task/machine matrix, called ETC.

The Savant approach generates a concurrent program from solved problem instances. The problem considered is NP-complete, therefore, the solved problem instances are obtained from an existing optimization algorithm. Several are used in the experiments:

- Min-Min, a well-known problem specific heuristic, previously presented in Section 3.2.2,
- a specialized CGA: the PA-CGA of Section 3.2.3, and configured in Section 6.3.1,
- a brute force algorithm that finds the optimal solutions for very small problems only.

The solved instances are the input to the Savant approach. However, because the solved instances come from an existing algorithm, the parallel solver generated by Savant can be considered a parallel version of the initial algorithm. Each of the algorithms listed above lead to the generation of a parallel program. The automatic parallelization perspective to the Savant approach suggest to briefly survey past automatic parallelization efforts, in the next section.

6.1.2 Automatic Parallel Program Generation

Parallelism was considered a part of the automatic build of executables from source code [231]. This optimization step is usually approached by

applying source-to-source transformations [232, 233]. Transformations include loop-unrolling, data access patterns, and rely on a careful inspection of data dependencies to extract concurrency from the source program. This approach to parallelization preserves the algorithm and most of the source code, by applying transformations that respect the semantics of the original program. The transformations are carefully defined, so as to guarantee identical behavior, and may even rely on formal reasoning [234]. Other authors apply AI techniques such as GA to select the source transformations and their order of application [235, 236, 237, 238]. In contrast, the Savant approach does not attempt to modify the implementation while preserving the algorithm, but produces a new algorithm.

In [239, 240], the authors review the applications of AI and machine learning to software engineering. Among the several applications mentioned, the transformation of programs for parallelism matches our objective. However, the approaches cited opt to restructure the source while preserving its semantics. Other related application domain include the generation of test cases for a given program, from inductive learning of programs from finite sets of input-output examples, and the synthesis of search programs for a Lisp code generator in the domain of combinatorial integer constraint satisfaction problems [241].

Genetic Programming (GP) [242] shares the same approach as the Savant approach. Indeed, GP aims to automatically evolve a program that performs a desired function. Moreover, the algorithm can be evolved to meet additional properties: a multi-objective search [237]. Parallelism can be introduced as an additional objective.

A combined GP and source-to-source transformation technique was introduced in [243]. Paragen disassembles a program into its constituent statements, and attempts to rebuild it in a parallel form using these statements and a combination of the parallel and serial functions available. The generated programs are tested for functional equivalence. Moreover, the functional equivalence are reported provable, however no detail is provided. The multi-objective search is performed using two populations, one per objective (correctness and parallelism). One unusual assumption is that the execution is considered on a synchronous virtual machine, where the cores operate at exactly the same clock, limiting concurrent access to shared memory.

In [244, 245], parallel programs are evolved using a linear GP for a multi-Arithmetic Logic Unit (ALU) processor register machine (MAP). An internal cross-bar network allows a number of registers to be shared. Using the linear GP, the programs are sequences of assembler instructions. The MAP is a virtual processor, and programs are simulated on this machine. The program generation is evaluated across 14 functions (including the Fibonacci series, numeric functions, boolean functions, artificial ant, and data classifications). Up to 8 ALU are considered, and each program length is 128 instructions at most. The termination condition is 10^8 tournaments or no

improvement over 10^7 tournaments. The focus of the study is the speed of the evolutionary search. An interesting finding is that evolving a parallel program is faster than evolving a sequential one.

In [246], the authors report on an unusual experiment that explores the generation of a parallel version of an initial program, by an evolutionary search similar to GP. The initial program performs self-replication. However, when replicated, the program is randomly altered. The alteration can introduce parallel instructions, that operate on the shared memory of the virtual machine used. Once the memory of the virtual machine is full, older programs are removed. The effect of these basic rules is that programs that replicate faster will survive, thus guiding the evolutionary search. The results show that initially the program undergo modifications on the sequential code that allow for faster replication. The programs are made shorter to accelerate the self-replication. Afterwards, the evolutionary process creates parallel versions, that perform the replication in parallel. The final solutions are able to use up to 32 threads for replication. The solutions are obtained after 10^9 clock cycles. Though limited to a simple function (self-replication), the unique evolutionary process is able to generate new programs that achieve the same self-replication in a parallel fashion.

The perspective on parallel program generation is common to the Savant approach, however we consider that the evolutionary approach is limited in its capability. We found that the programs evolved were relatively small ($O(100)$ assembly instructions), and required considerable computational effort to find; the stopping condition is the absence of progress in the last $10^6 - 10^8$ evolutions, or 10^9 clock cycles. The EA is a general technique that comes with drawbacks, such as the computational effort required by the large search space. Modeling parallelism as an objective function for the evaluation of a program is an elegant formulation of the problem, that fits well into the evolutionary search. However, introducing parallelism as an additional objective in a multi-objective search is an unreliable path to parallelism, because the achieved parallelism is unpredictable and often limited.

GA were considered capable of evolving rules of computation, instead of a solution to a problem [247]. The rules found by an evolutionary process need not be complete programs, but rather rules that react to state transition events, and in turn trigger additional state transitions. Such an application of the evolutionary process could in principle be used for automatic parallelization but we have not found previous work.

Cellular Automata (CA) rules were noticed to be amenable to evolutionary search [248]. The evolutionary process is used to find rules that allow the CA to achieve a desired function. Although the CA is often considered a parallel machine, the periodic computation performed in parallel at each cell (by the simple CA rule) is far less time consuming than the necessary communication, for reading the cell's neighborhood but mostly for

synchronization, therefore limiting the degree of parallelism.

6.2 The Savant Approach

6.2.1 A Parallel Algorithm Template

In Section 6.1.2, we mentioned that the previous GP approaches considered parallelism as an additional objective, which leads to an uncertain degree of parallelism. In contrast, Savant addresses the parallelism problem by specifying a target parallel template for the generated programs. The Savant approach only produces algorithms that conform to a design that guarantees a suitable form of parallelism. This chosen form of parallelism is meant to fit the AWN (Section 2.2), and should display:

- limited computation on each node, because the nodes are "wimpy",
- limited inter-node communication, because the nodes are considered connected over a slow interconnect, capable of connecting a great number of cores, unlike the shared memory machines mentioned in Section 6.1.2,
- scalability with the number of nodes, and exploit hundreds of cores, even on small problems.

The chosen algorithm template is a scatter-gather design, such as a Map-Reduce application. Open source and Free Software frameworks exist for this model, across different architectures (GPU, multi-core, clusters), which makes it a practical choice. Moreover, theoretical works have found it equivalent to Bulk Synchronous Parallel (BSP) and Parallel Random Access Machine (PRAM) [249], both well-studied parallel models.

In the chosen template, the input data is first processed independently by many mappers, whose results are then further processed, independently, by reducers. Independent processing means that the mappers and reducers do not communicate or otherwise synchronize. The mappers and reducers do not perform computationally intensive functions. In addition to this qualitative definition of the parallel model, we wish for a high number of mappers and reducers, even on small problems. Also, the new programs must scale with respect to problem size and hardware resources, meaning the mappers and reducers should decompose the problem into small parts.

Section 6.2.3 details the algorithm template, in the context of the selected use case. In summary, the template can be viewed as a universal algorithm, that is adapted to the algorithm to parallelize. However, before presenting the design, the following section introduces the source of inspiration, which influence the design of the template algorithm.

6.2.2 Analogy with the Savant Syndrome

We looked for previous occurrences where a massively parallel machine (similar to the AWN, composed of "wimpy" nodes to ensure the reliance on parallel processing), was able to solve small, sequential problems in a short time. This question lead to the Savant syndrome [250, 251, 252, 253, 254, 255].

People displaying symptoms of this syndrome can compute small sequential tasks, such as calendar computation (finding the day of the week of a given date), in a very short time (700 msec) [254], using largely unknown methods [252]. The Savants' methods for calendar computation are reported unknown because experiments showed that the distribution of the response times does not match those of known algorithms. Certain date calculations were expected to take longer to compute because of the calendar rules. Moreover, Savants can perform other date computations (such as finding the years that match a given weekday, date and month) with similar performance, which is considered more time-consuming for a computer algorithms [252].

Although not fully understood, neuroscientists believe that the Savants discover pattern recognition rules from data, which are later applied, in parallel, to new input [254, 255]. The rules extracted capture perceived regularities in the data, such as found in calendars and prime numbers. The Savant's perception of data appears different, more precise ("high resolution data" [250]) than our ordinary higher-level, conceptual memory [251, 253, 255]. Some descriptions of the Savant's internal perception of numbers [256] display synesthesia [257]. The learned pattern recognition is also consistent with studies of chess perception in players of different skill [258, 259].

These findings could explain their ability to perform calendar computation while ignoring the complicated details of calendars, or to enumerate prime numbers while ignoring what a prime number is, or even how to multiply and divide. The mental activities that some Savants describe incline us to believe that their pattern-recognition learning method is supervised [256].

6.2.3 Application to the Automatic Solver Generation

We now combine the different items presented above: we apply the Savant syndrome analogy (Section 6.2.2) to implement the desired parallel algorithm template (Section 6.2.1), for the generation of a parallel solver for the scheduling problem (Section 6.1.1).

Figure 6.1 provides an overview of the Savant algorithm. The "input" to the combinatorial optimization problem is an ETC matrix of N_{tasks} columns and $N_{machines}$ rows, and a fitness function which computes the makespan of a solution (Section 6.1.1). The "output" of a solver is a solution to a problem instance, the solution can be compactly represented by an array of integers or labels, where the index of the array denotes the task assigned,

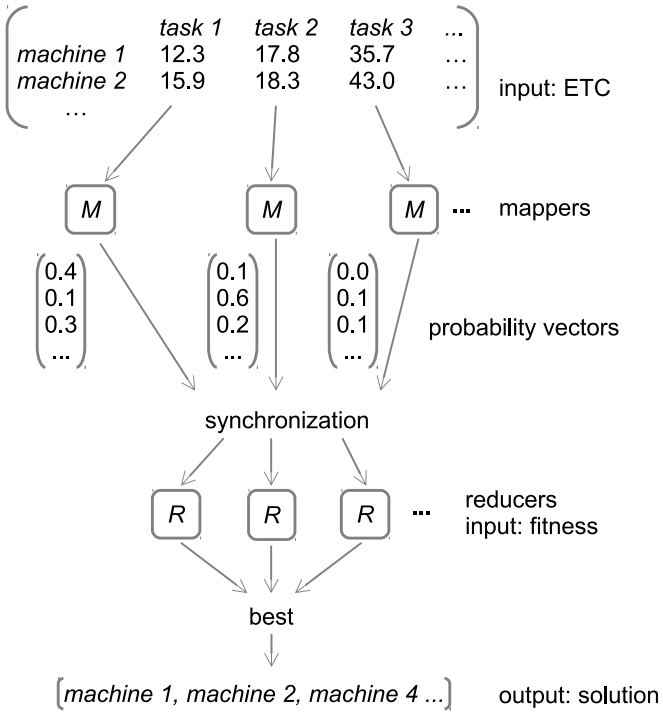


Figure 6.1: Overview of the Savant parallel algorithm

and each value in the array denotes the machine identifier to which the task is assigned. Typical problem instances involve many more tasks than machines, hundreds of tasks for even small problems.

The mapper

Concurrency in the Savant algorithm is first introduced by decomposing the tasks' assignment in a solution into multiple, independent task assignments. Each task assignment is performed by a mapper, in the Map-Reduce context, and represented by the *M* boxes in Figure 6.1. The approach is data-parallel, and each mapper operates independently of the others, avoiding communication. This output decomposition can exploit the parallelism of an AWN, even for small problem instances. By analogy with the Savant syndrome, each mapper is based on pattern recognition, and the tasks' assignments are parallel pattern recognitions. Each mapper is a multi-class classifier, implemented with Support Vector Machines (SVM), using `libSVM` [260]. The classes of the SVM are the machines the tasks are assigned to. The mappers' SVM implement supervised machine learning, which are trained with solved problem instances. The solved problem instances are obtained from an existing solver, which we wish to parallelize. Each mapper/SVM is trained independently of the others, and attempts to learn the assignment

logic from the solver. Each mapper is trained for a single task. Because the ETCs are randomly generated, according to procedure [145], the ETC data is sorted. The assumption behind sorting the ETC is that pattern recognition rules are considered dependent on the nature of the task and machine (smaller/larger task, slower/faster machine), but are common to similar tasks and machines across ETC instances. The tasks (ETC columns) are sorted in the order of increasing execution time, taken as the sum of ETC values across machines. The machines (ETC rows) are also sorted in the order of increasing execution time. The low-index tasks (the column number) have smaller execution times than the high-index tasks. The low-index machines (the row number) are faster than the high-index machines.

The reducer

We have not yet stated the output of a mapper because this output needs to address a concern in the above mentioned decomposition. The task decomposition of the solution is *a priori* a poor choice, because by definition the combinatorial optimization problem requires the full solution, while the proposed decomposition assigns each task independently. The reducers (the R boxes in Figure 6.1) are designed to bridge the gap between the independent mapper assignments and the full solution nature of the optimization problem. Moreover, the reducers use the other defining component of the optimization problem, the fitness function (which in the considered use case computes the makespan). The reducers assemble a full solution to the optimization problem, from the output of the mappers. The reducers are not limited to the assembly of the solution, but also search fit solutions (with low makespan). In order to keep the reducers generic, they perform a random search in the solution space, storing the best solution found. The random search is not very efficient, therefore, the search is guided by the output of the mappers. Therefore, a mapper's output is not a definite task assignment, but an assignment probability vector (Figure 6.1). The probability vector holds the probability to assign the task to the different machines, based on the learned behavior from the initial solver. SVM can be trained to produce such prediction probabilities. The probability vector does not only provide the most likely assignment (highest probability value), but also how likely the other assignments are. The vector allows for ambiguity in the task assignment (similar probabilities). Interestingly, the mental process of "analogy", when the brain associates events to categories, is known to be a non-deterministic function (that would otherwise map an event to a single concept), but defines weighted and temporary relations between an event and multiple concepts. The mappers' probabilities for task-to-machine assignment resembles these weighted links between events and concepts.

Each reducer generates solutions randomly, according to the assignment probability vector. Solutions generated are evaluated with the fitness func-

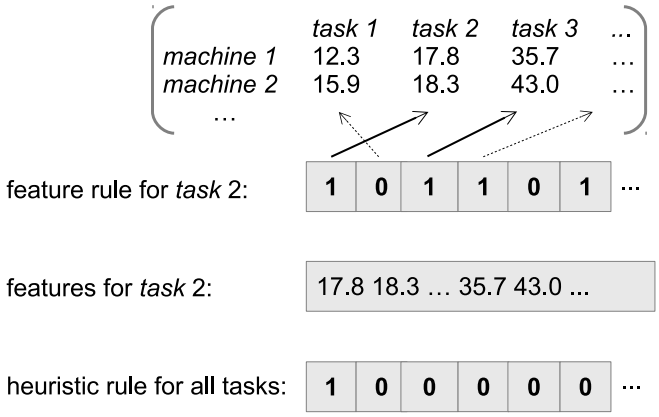


Figure 6.2: Feature selection rule

tion that computes the makespan, and the best solution is finally selected. The reducer's search for fit solutions is malleable [261], because the computation time required is only a function of the iterations (solution evaluations). Both the random generation and fitness evaluation are stateless and can be split across any number of nodes. For example, a single reducer thread performing 10K iterations can be split into two threads performing 5K iterations. Moreover, the generation of a single solution could also be parallelized, either data-wise (by partitioning the tasks or machines) or code-wise (by pipelining the task assignment and ETC lookup). The random number generation required for this search can be of low-complexity. The fitness evaluation can in general require significant computation, but not for makespan computation. Moreover, the makespan computation can also be parallelized.

The SVM feature selection

An important consideration for training the mappers in the Savant algorithm is the selection of features for the SVM. An obvious choice of features is the entire ETC matrix, because it includes all task and machine information. However, choosing the entire ETC matrix values for classification yields poor prediction results, and requires longer training time. We have chosen, after experimentation, to use a very simple feature selection rule: the features used for the mapper of a given task are the values of the ETC column of that task. In other words, a mapper/SVM will rely only on the estimated execution times of that task on every machine. The above mentioned feature selection rule is called the heuristic rule in the text below.

The good performance of the feature selection heuristic is surprising because the nature of both the combinatorial optimization problem (minimizing a complete solution fitness) and the solvers (which also operate on full

ETC) suggest that correctly assigning a task requires more information than just the profile of a single task. Therefore, we also investigated other feature selection rules. However, none improved the prediction scores of the above rule. First, we extended the features for a task's classifier to the ETC values of similar tasks. Because the ETC is sorted (Section 6.2.3), the similar tasks are the neighboring tasks in the ETC. Extending the neighborhood by 1–4 columns of the ETC lead to worse predictions on new, unseen instances.

In an effort to systematically explore feature selection rules, we used a genetic algorithm (CGA) that searched for the best features for each mapper/SVM. Searching for the features of each task's mapper can be cast as an optimization problem: what are the features that maximize the prediction accuracy of the mapper. The motivation was to depart from the heuristic feature sets, at the expense of a long computerized search. Another objective of the genetic search was to determine if different tasks required different feature selection rules (due to their different profiles), and if common patterns would emerge. In the CGA context, a feature selection rule for each task is represented as per Figure 6.2. The feature rule is a binary vector, of length the number of tasks. The binary representation allows to use the standard genetic operators (crossover, mutation). A '1' value means that all the ETC values for this task are part of the features. The selection rule representation must facilitate the detection of patterns. Selection rules for different tasks are thus comparable. Therefore, the vector index is relative to a task: the first binary value in the vector refers to the task for which the rule applies. The second and third vector values point to the left and right neighbors of the task, in the ETC matrix, and so on, as indicated by the arrows in the Figure. If there is no neighbor available (because we reached the edge of the ETC matrix), then that neighbor is taken in the other direction. As a consequence, the same rule vector will select different features for different tasks. The heuristic rule finally applied can be represented in the binary vector form, as indicated at the bottom of Figure 6.2 ("heuristic rule"). The heuristic rule is common to all tasks.

The CGA search found very irregular and complex feature sets. Incidentally, the best feature selections included the task's ETC column (a "1" in the first value of the rule vector). Although the search proposed features that improved the prediction (by 5–10%), the application on completely unseen instances proved slightly worse than the heuristic rule. Most likely, the CGA search suffered from overfitting, which we were not able to mitigate. Mitigation efforts included randomly changing the training problem instances.

Discussion

The Savant parallel algorithm presented provides much freedom in implementation. For example, the synchronization phase at the end of the SVM

mapping is not strictly required, the reducers could start generating partial solutions as soon as probability vectors become available. Moreover, there are numerous ways to communicate between threads, and several existing parallel frameworks are suited to the algorithm. We prefer to keep the algorithm's presentation more conceptual, rather than list the various implementation details which add little to the understanding.

In summary to this section, we would like to comment on the described parallel program generation. Our goal is to generate a parallel version of an existing algorithm. However, we did not only present a method, but a complete parallel algorithm, without saying anything of the original algorithm. The presented parallel algorithm appears generic and independent of the original algorithms, but this is not the case. The Savant parallel algorithm presented is indeed generic, but it is specialized to the original algorithm. The mappers are trained to learn the behavior of the original solver to parallelize, from the instances it solved. The reducers exploit the mapper's probabilistic view of the solution space, and apply the fitness function that completes the makespan minimization problem. Therefore, the Savant is a generic parallel solver that learns, through supervised learning, an original solver.

6.3 Experimentation

6.3.1 Configuration

The problem instances and evaluated algorithms

The scheduling problem instances for the combinatorial optimization problem are the ETC matrices. The ETC are generated following the procedure presented in [145]. As such, the ETC are randomly generated following probability distributions that attempt to represent real scenarios. We studied two types of ETC: high and low machine heterogeneity. The tasks are always chosen of high heterogeneity. The ETC instances of high task and machine heterogeneity are noted *hihi*, while instances of high task but low machine heterogeneity are noted *hilo*. Three problem sizes are simulated, instances of 12 tasks to map onto 4 machines (denoted 12×4), 128 tasks and 4 machines (denoted 128×4), and 512 tasks and 16 machines (denoted 512×16). The ETC data is sorted as indicated in Section 6.2.3.

The algorithms used to train the Savant mappers were listed in 6.1.1: an exhaustive search, the Min-Min heuristic, and the PA-CGA. For the smallest problem instances, the optimal solutions were found by an exhaustive search. The Min-Min heuristic was presented in Section 3.2.2. Table 6.1 summarizes the parameters used for the PA-CGA. The parameters are listed for completeness, the PA-CGA itself is described in Section 3.2.3.

Table 6.1: Parameters for the CGA

parameter	*value*
Stop condition	30s wall-clock time
Population size	8×8
Thread(s)	1
Neighborhood	Von Neumann
Crossover	Two-points, with probability 1.0
Mutation	Bit flip, with probability 1.0
Local search	10 iterations of H2LL, with probability 1.0
Replacement	Replace if better

Table 6.2: ETC instances for the SVM training and testing

ETC size $tasks \times machines$	*available training ETC*	*subsets of training ETC*	*test ETC*
12×4	1,000	200 – 1,000	200
128×4	1,000	200 – 1,000	200
512×16	4,000	1,000 – 4,000	200

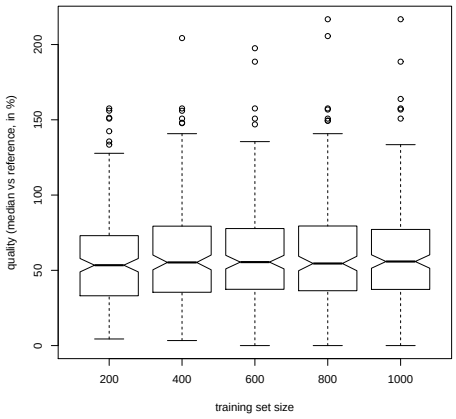
Training the mappers

Section 6.2.3 mentioned that the mappers are multi-class SVM classifiers, implemented in `libSVM`. The supervised training procedure follows the practical guide [1]: the chosen kernel is RBF, cross-validation is used to determine the SVM parameters, and the input data (the features extracted from the ETC with the heuristic rule) is scaled.

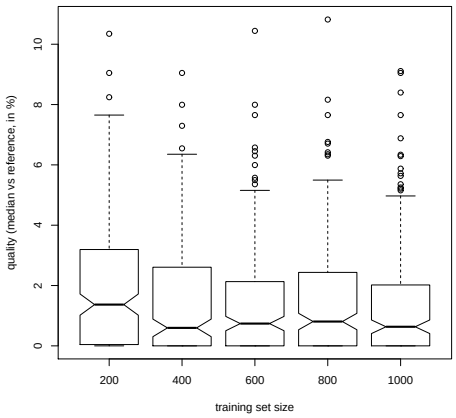
The amount of training data influences the quality of the SVM mappers, and the size of the SVM model (number of support vectors). Before we can evaluate the performance of the Savant algorithm, we must select the appropriate amount of training data for each problem size. We wish to select the fewest training observations that provide reasonably good results.

Table 6.2 is an overview of the observations used for the SVM. An observation is composed of an ETC matrix and the solution found by the algorithm we aim to parallelize. The table indicates the total number of available observations, and the different subsets that were evaluated for training the mappers. In all instances, 200 observations were used for testing. The test ETC are reserved for testing, and never used in training. We use a maximum training set of 1,000 observations for scheduling problems of 4 machines, and 4,000 observations for problems of 16 machines. The number of SVM classes are the number of machines. Therefore, we try to maintain a constant ratio between number of machines and observations, across the different problem sizes.

[1] `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`



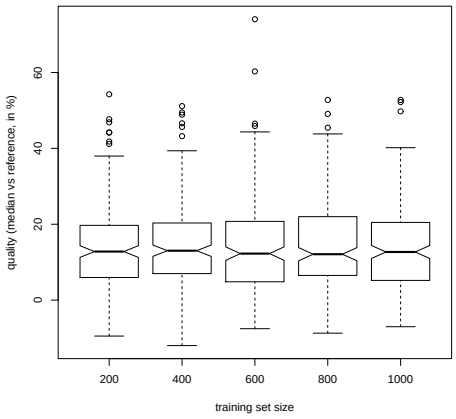
(a) After the Savant mapper



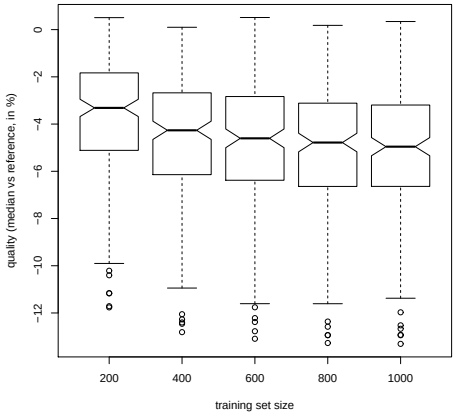
(b) After 10K iterations of the Savant reducer

Figure 6.3: Impact of training size on 12×4 *hihi* ETC with Savant/optimal.

Fig. 6.3–Fig.6.7 present a representative sample of the impact of training sizes on the Savant performance. The above mentioned figures present results as boxplots, where notches indicate statistical significance. The boxplots show the quality (makespan) of the solutions found by the Savant algorithm, using various training set sizes. The boxplots show Savant solution quality for the 200 unseen test ETC instances. Each figure depicts two boxplots: (1) results for the Savant from the mappers (the SVM classification), and (2) results for the Savant after 10K iterations of a reducer. In Section 6.2.3, we mentioned that a Savant mapper produces a probability vector. However, in order to evaluate how training set sizes influences the Savant's mappers, the first boxplots are computed with the most likely



(a) After the Savant mapper



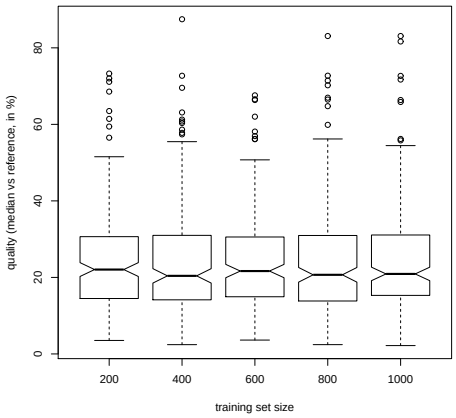
(b) After 10K iterations of the Savant reducer

Figure 6.4: Impact of training size on 128×4 *hihi* ETC with Savant/Min-Min.

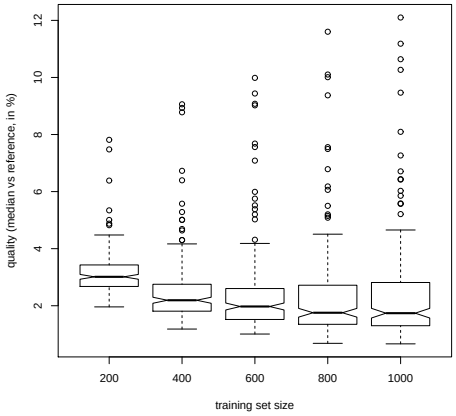
machine assignment.

In all figures, the quality of the solution is makespan. The quality is represented by the ratio of the makespan of the Savant solution relative to a reference solution:

- For the Savant mappers, the quality measure is the most likely SVM classification, relative to a reference solution.
- For Savant with a reducer, the quality measure is the Savant's median solution over 30 runs, relative to a reference solution, because the reducer is stochastic.



(a) After the Savant mapper

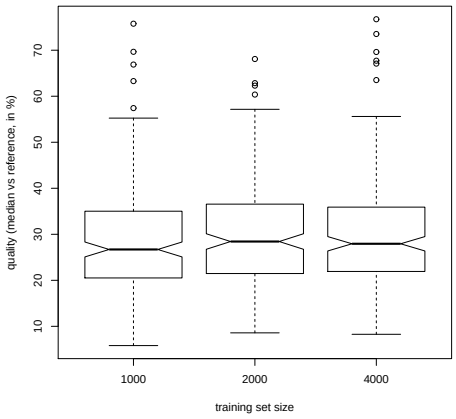


(b) After 10K iterations of the Savant reducer

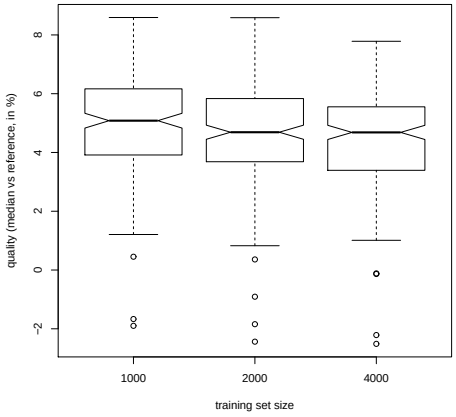
Figure 6.5: Training size impact on 128×4 *hihi* ETC instances with Savant/CGA.

Table 6.3: Reference solutions for training size comparisons

Figure	*ETC size*	*Observation source*	*Reference solution for each ETC*
Fig. 6.3	12×4	Optimal	optimal solution
Fig. 6.4	128×4	Min-Min	Min-Min solution
Fig. 6.5	128×4	PA-CGA	Best PA-CGA solution of 10 runs
Fig. 6.6	512×16	Min-Min	Min-Min solution
Fig. 6.7	512×16	PA-CGA	Best PA-CGA solution of 10 runs



(a) After the Savant mapper

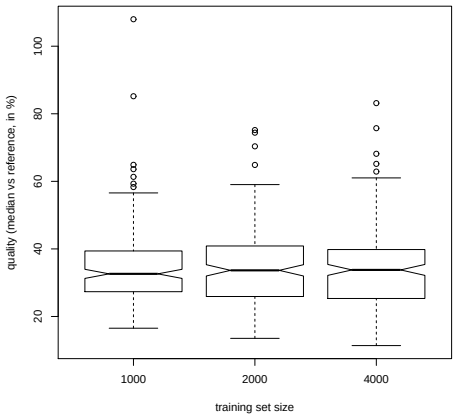


(b) After 10K iterations of the Savant reducer

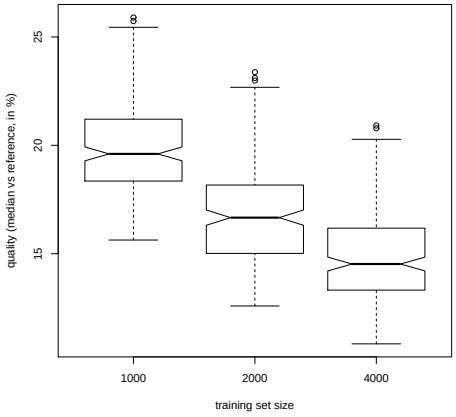
Figure 6.6: Impact of training size on 512×16 *hilo* ETC with Savant/Min-Min.

Table 6.3 summarizes the chosen solution references for each figure. The reference solution is simply a reference quality measure for a given ETC, that allows the comparison across training set sizes and ETC instances. The "Optimal" and "Min-Min" solutions are unique per problem instance (i.e. ETC), they are the reference solutions. However, for the CGA, the reference solution is the best of 10 runs (10 runs is enough given the variance of CGA results, as shown in the next section). This apparent mismatch between median and best solutions is acceptable because the reference only serves to produce a ratio that remains valid across training set sizes.

For problems of size 12×4, Fig. 6.3 shows the influence of the training



(a) After the Savant mapper



(b) After 10K iterations of step 2

Figure 6.7: Impact of training size on 512×16 *hihi* ETC with Savant/CGA.

set size on the Savant's performance, when trained with optimal solutions. We can observe that the training set size must be greater than 200 observations (solved ETC), although this is not apparent from the Savant mappers' results.

For problems of size 128×4, Fig. 6.4 and Fig.6.5 show that the Savant mappers' results do not reveal any difference in training set size. However, Savant after 10K iterations of the reducer shows that a training set size of 600 or more provides better quality solutions, for Savant trained with Min-Min or PA-CGA solutions.

For problems of size 512×16, Fig. 6.6, and Fig.6.7, also show that the Savant mappers do not benefit from additional training data. The results for

Table 6.4: Selected training set sizes

ETC size	*Number of observations*
12×4	600
128×4	600
512×16	4,000

the *hilo* instances (only Min-Min are presented here) show that after 10K iterations of the reducer, there is little benefit gained with more training data. However, *hihi* instances clearly show that more training data helps (only PA-CGA results are presented).

Therefore, for the further evaluation of the Savant algorithm of the next section, we select the smallest training set sizes that nevertheless provide good results (less training data yields less support vectors in the models). Table 6.4 lists the chosen training data size per ETC size.

6.3.2 Results

In this section, we report on the evaluation of the Savant algorithm.

The Savant mapper performance

This section compares the solutions to the scheduling problem found by the Savant mappers, to the ones found by the Min-Min heuristic, on unseen problem instances. The solutions are compared literally (genotype in the GA vocabulary), and not through their quality (phenotype). For the comparison, the reducer used only assembles the prediction results from the task classifiers, it does not apply the local search. Although, the goal of the Savant is the solutions' quality, the mappers are essential to the algorithm's behavior. The performance is reported in two ways: by similarity and probability to solution.

The similarity score is the count of correct task-to-machine classifications across 100 evaluation problem instances, for each task. The mapper's prediction follows the most probable machine assignment in the probability vector. Fig. 6.8 shows the similarity for 128×4 problems. It shows that the accuracy for the smaller and bigger tasks is lower than average. The average accuracy across tasks is approximately 82%, which is quite high, given that the Min-Min algorithm chooses machine assignments based on more information (the ETC values of all unassigned tasks, and the current machine completion times) than the 4 ETC values used by each Savant mapper. When below average, the accuracy is still greater than 60%. The assignment errors for larger tasks is understandable because the Min-Min assignment for larger tasks occurs later in the algorithm's execution, and depends on the previous task assignments, information that the Savant mapper does not

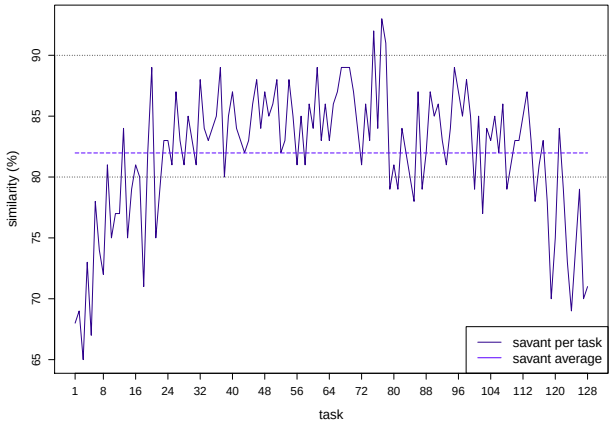


Figure 6.8: Savant mapper solution similarity for 128×4 problems.

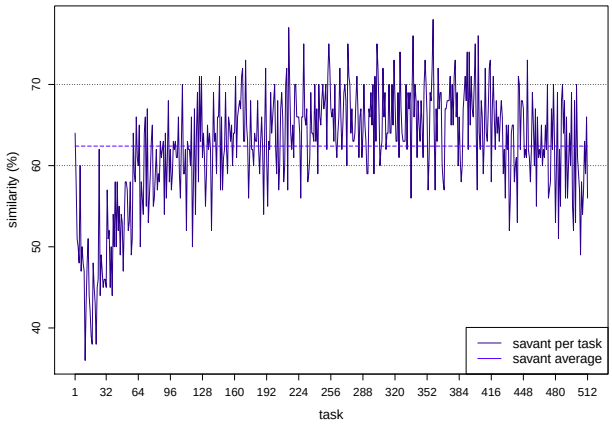


Figure 6.9: Savant mapper solution similarity for 512×16 problems.

have. The mismatch between the Savant's mapper and Min-Min could also be caused by the approximation in the task's sorting (Section 6.2.3), which determines the mapper model to apply for a task. However, errors for the smaller tasks is acceptable because they have little influence on the overall solution quality.

Fig. 6.9 reports on the similarity for 512×16 problems. The general shape of the plot is similar to that of the 128×4 problem size, where accuracy for small and large tasks are below average. The accuracy of the mappers is worse than for the smaller problems (63%). Overall, the accuracy of the classifiers, operating on only 16 factors (the ETC values for the task), is high. The prediction is more difficult for mappers because the assignment

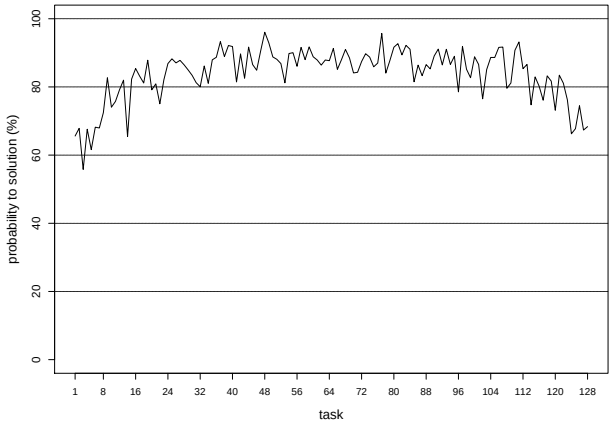


Figure 6.10: Savant mapper probability to solution for 128×4 problems.

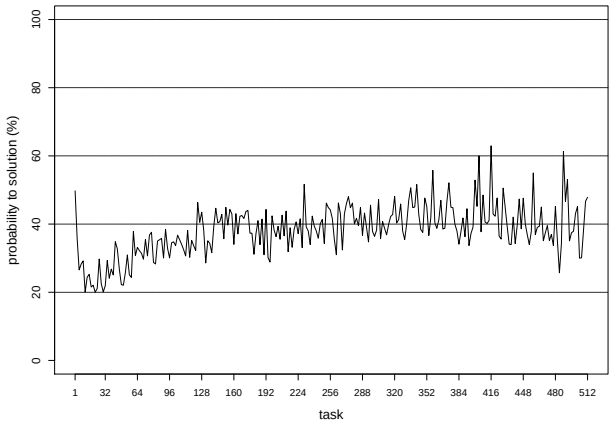


Figure 6.11: Savant mapper probability to solution for 512×16 problems.

is to be chosen among 16 classes.

The probability to solution is the probability from the mapper's probability vector, to choose the correct assignment. A correct assignment means matching the Min-Min assignment. The probability to solution is different to the similarity mentioned above (Fig. 6.8, and Fig.6.9). The similarity records the mapper's decision, which is the classifier's highest probability estimate. In contrast, the probability to solution is not necessarily the greatest probability. Such an indicator is very useful for the Savant evaluation, because even in case of mis-assignment (when the greatest probability is incorrect), the mapper's probability for the correct assignment influences the Savant's reducer. The probability to solution also represents a broader ac-

curacy measure of the mappers. Indeed, the probability estimates of a task for the various machine assignments vary greatly. Several assignments can have very close probabilities, reflecting an ambiguous choice, whereas some choices for machine assignments have very different probabilities, reflecting a strong preference.

For smaller problem instances, Fig. 6.10 shows that the prediction of the SVMs is highly accurate. Because of this high accuracy (greater than 50%), similarity and probability to solution are very similar. For the 512×16 problems, Fig. 6.11 shows much lower probabilities than the similarity of Fig. 6.9, because the probability of machine assignments is spread across more machines/classes. Also, the mappers are less accurate than for smaller problem instances. The distribution of the probabilities across tasks is different to Fig. 6.9, where the accuracy does not degrade for larger tasks.

The Savant capability

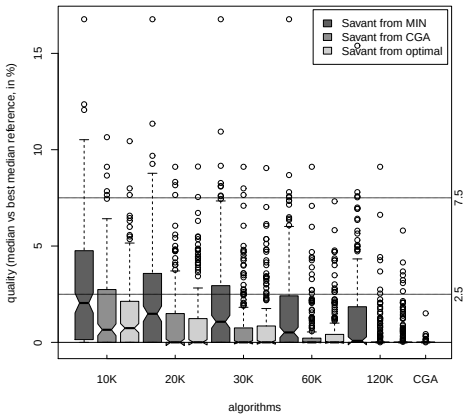
In this section, we investigate the Savant's capability: how fit are the solutions found by our automatically generated parallel Savant algorithm, for the considered combinatorial problem. The capability of the Savant algorithm can be observed from Fig. 6.12–Fig.6.17, covering the different problem sizes. For each problem size, we present the median and best solutions, over 30 runs, in separate figures. All results are obtained on the 200 unseen test instances.

All figures present the quality of the solutions found by the:

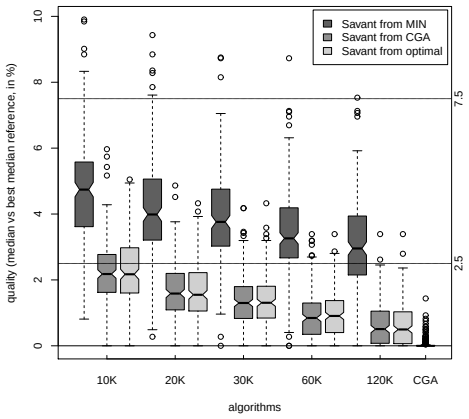
- Savant algorithm trained with optimal solutions, for 12×4 problems,
- Savant algorithm trained with Min-Min solutions,
- Savant algorithm trained with the best PA-CGA solutions.

Results for the three Savant versions are presented for several iteration count of the reducer: $10K$, $20K$, $10K \times tasks/4$, $10K \times tasks/2$, $10K \times tasks$. The size of the problem is then reflected in the number of reducer iterations. The results with the chosen number of iterations show convergence, and include the major solution quality improvements.

Savant results for 12×4 problems are presented in Fig. 6.12, and Fig. 6.13. The quality of the solutions are relative to a reference solution quality for each ETC. For the 12×4 problem size, the optimal solution is available through an exhaustive search. Therefore the reference is simply the optimal solution, which explains why no points are below the 0% mark. The boxplot to the far right presents the PA-CGA results. The Min-Min solutions are not shown, because they are far from the optimal (about 40% worse) and would further reduce the clarity of the figure, if plotted. We notice that the PA-CGA finds near optimal solutions. The PA-CGA is configured with



(a) *hihi*

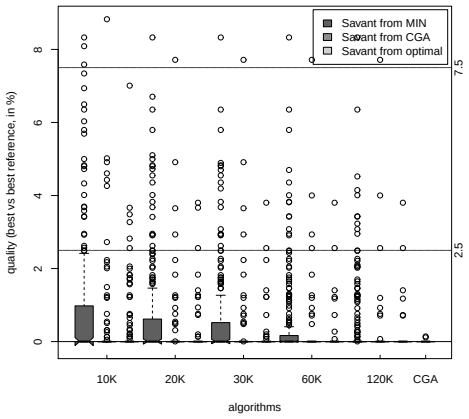


(b) *hilo*

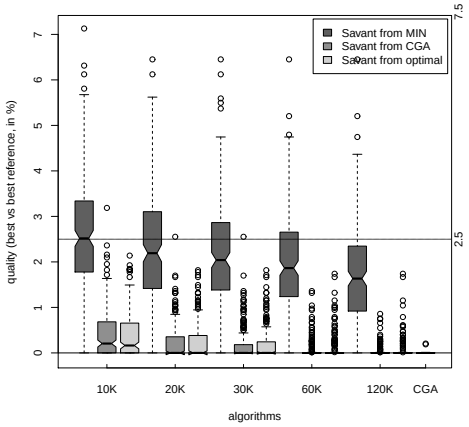
Figure 6.12: Savant median solutions for 12×4 ETC.

the Min-Min solution in the population, and also applies a problem-specific local search.

Overall, the Savant versions obtain good solutions quickly, less than 2.5% from the optimal in 10K iterations of the reducer. The solutions found by the Savant trained with Min-Min are much better than the ones found by Min-Min. The small solutions (12 tasks) allow the reducer to quickly improve the solutions, as shown in the makespan improvement between Fig. 6.3a, and Fig. 6.3b. The efficiency of the reducer is also due to the small ETC sizes, which allow the mapper to be more accurate, as the features of the SVM include 1/12 of the ETC for a 4-way classification. Results for the *hilo* instances are slightly worse (require $\times 10$ more reducer iterations), especially



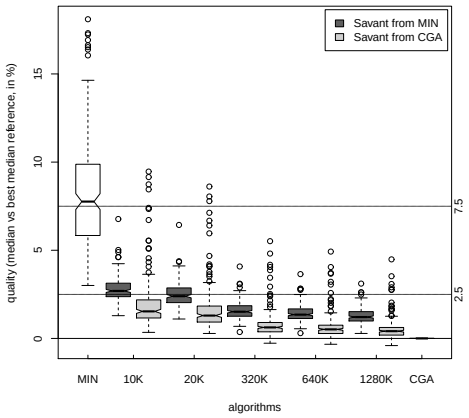
(a) *hihi*



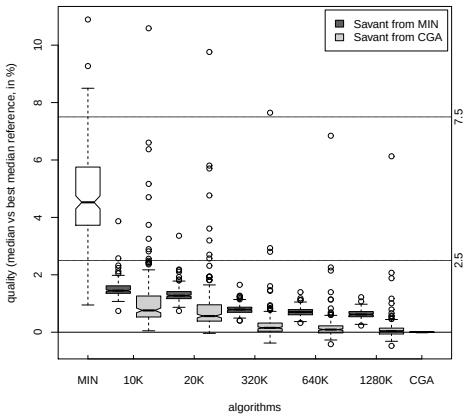
(b) *hilo*

Figure 6.13: Savant best solutions for 12×4 ETC.

for the Savant trained with Min-Min observations. The mapper for these instances seems to be less accurate on *hilo* instances. Min-Min assignment decisions are more difficult to learn for the mappers, because the assignment decisions depend more on information unavailable to the SVM, such as each machine's list of assigned task. However, these solutions are still considerably better than those found by Min-Min. Fig. 6.13 compares the best solutions of the Savant versions, with the best solution for each ETC. For this problem size, the best solution is the optimal. The Savant trained with Min-Min finds slightly worse solutions than when trained with the optimal or PA-CGA solutions, especially for *hilo* instances. This shows the influence of the mapper's accuracy on the overall Savant capability.



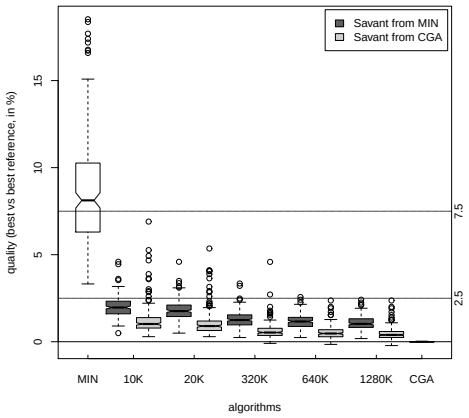
(a) *hihi*



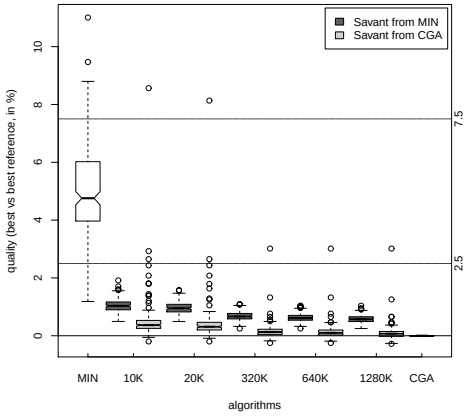
(b) *hilo*

Figure 6.14: Savant median solutions for 128×4 ETC.

Fig. 6.14, and Fig. 6.15 present the Savant results for 128×4 problems. For this problem size, the reference solution for normalizing the Savant results is the median PA-CGA solution over 10 runs, chosen because it is better than the Min-Min solution (the optimal solutions are not available for this problem size). The PA-CGA relative solution quality is thus near 0%. The first boxplot presents the Min-Min solutions. Results for *hihi* and *hilo* instances are similar. The difference in machine heterogeneity has less impact on makespan for 128 tasks than with 12 tasks. The Savant versions find very good solutions quickly. Trained with Min-Min, the Savant finds much better solutions than the Min-Min algorithm. The Savant even finds better solutions than the PA-CGA. For example, for *hilo* instances,



(a) *hihi*

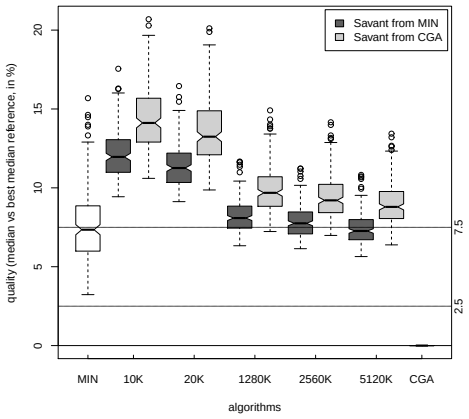


(b) *hilo*

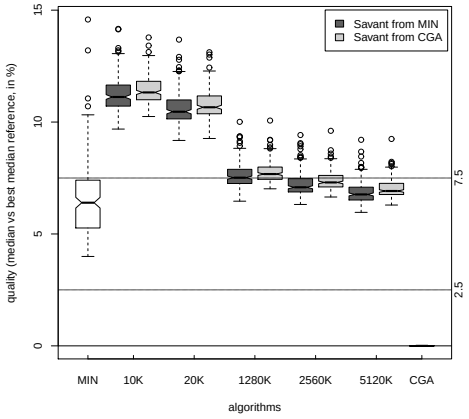
Figure 6.15: Savant best solutions for 128×4 ETC.

after 1,280K iterations, Savant finds better solutions than the PA-CGA in half of the problems (i.e. 100 instances). As a comparison, the PA-CGA performs about 6,000K fitness evaluations in the 30s configured to find the reported results. This is consequence of the reducer's random search, guided by the mappers' results, which if the mappers are accurate, can find new, fit solutions.

Fig. 6.16, and Fig. 6.17 present the Savant results for 512×16 problems. The reference solutions used for normalizing the Savant results are chosen in the same way as for 128×4 problems. Solutions found by the Savant trained with Min-Min are of similar quality to the Min-Min solutions. It does however, require more iterations than on smaller problems. Another



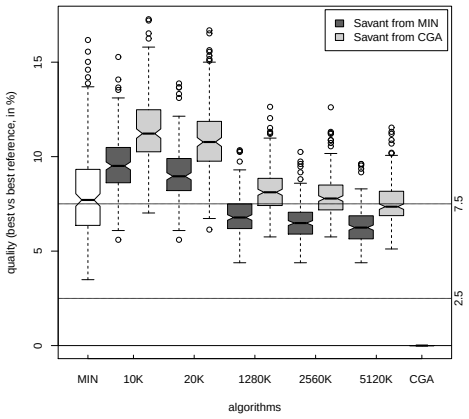
(a) *hihi*



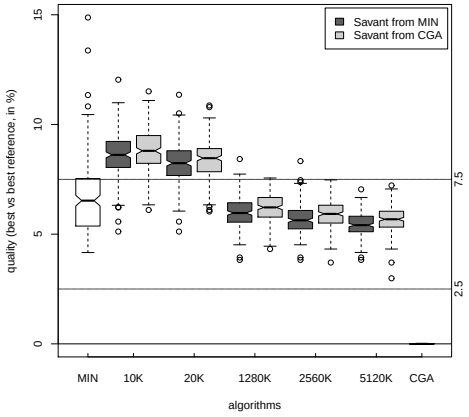
(b) *hilo*

Figure 6.16: Savant median solutions for 512×16 ETC.

difference with the results from the previous problem sizes is that Savant trained with Min-Min performs better than the Savant trained with PA-CGA observations. The Savant median solutions are less or equal to 7.5% of the PA-CGA median solutions. The Savant performance compared to the PA-CGA is not as good as in the previous, smaller, problem sizes. The increased problem size from 128 and 12 tasks demands more iterations from the reducer. Moreover, the increased problem size, especially in the number of machines, decreases the mapper's accuracy, which in turn requires more iterations from the reducer. The difference in the mapper's accuracy between Min-Min and PA-CGA is visible from Fig. 6.7a and Fig. 6.6a. The impact of the reduced accuracy is visible in Fig. 6.6b and Fig. 6.7b. However,



(a) *hihi*



(b) *hilo*

Figure 6.17: Savant best solutions for 512×16 ETC.

because the PA-CGA finds better solutions than Min-Min, Fig. 6.6, and Fig. 6.7 should not be compared too literally. Fig. 6.7b shows that increasing the training size from 2,000 to of 4,000 observations improves the solution quality, whereas increasing the training set size from 2,000 to 4,000 Min-Min observations does not. More observations could be necessary to train the Savant for the PA-CGA.

6.4 Summary

In this chapter, we presented a method to automatically generate a parallel search algorithm, from an existing sequential program. The approach was

applied to solve a combinatorial optimization problem from the scheduling domain. The Savant approach was able to automatically generate parallel solvers for this problem, in a Map-Reduce design. The generated Savant solver can exploit a massively parallel machine even for very small instances of this problem. Moreover, the resulting Savant algorithm is suited to the parallelism provided by AWN, by limiting the inter-thread communication and synchronization. The Savant solvers found solutions of comparable quality to the original sequential solvers, for different problem types and sizes.

However, there are several opportunities for improvements. The current approach requires the training of one model per small part of a solution. Although this is required only once (at design time), it remains a very time-consuming operation, that cannot scale to large problems. We plan to investigate how to project a Savant algorithm trained on a given problem size to another, larger problem.

Another improvement is to completely remove the fitness evaluations, currently part of the Savant reducer. In optimization problems, the computation cost of the fitness function is sometimes high. The fitness evaluations are meant to capture a key characteristic of the optimization problem. We plan to replace the fitness evaluations by a second pattern recognition process. The new pattern recognition could predict the full solution from the probability landscape of the different parts of the solution.

Chapter 7

Conclusion and Perspectives

In this thesis, we explored the parallel designs of solvers for an optimization problem. The parallel designs aimed to extract performance from the AWN, a possible evolution in computing that can reduce the energy-related costs of a Cloud data center.

The TCO perspective lead to identify critical power, the purchase price of equipment, and the poor energy-proportionality as the key factors to the costs in a cloud data center. The AWN proposal attempts to address those factors. The market size of mobile computing has commoditized the computing components in an AWN, lowering costs and attracting innovation. However, we noticed that existing applications do not perform well on AWN, but that new applications, more parallel and distributed by design (such as Map-Reduce), perform well.

Simulations of software pipelines (code-parallelism) showed that the "wimpy" nodes, although more energy-efficient, cannot achieve the performance of "brawny" nodes, even when contention for shared resources is accounted for. Experiments with data-parallel versions of known algorithms showed clear benefits (such as super-linear speedup), but also exposed their limits in scalability (with the size of input data, and the computing cores).

We proceeded to experiment two parallel designs that deliberately change the original algorithm (and not only the algorithm's implementation), the approach of the thesis. The experiments illustrated that to extract the needed parallelism required by the AWN, the algorithm to parallelize can be modified without loss of capability. However, these experiments pointed to two weaknesses. First, only one of the parallel design extracted parallelism from small problems. More significantly, the designs were *ad hoc*, and manually designed by a time-consuming trial-and-error process. Extending the search space of parallel designs to algorithmic changes is only practical if we can be guided.

Guidance in the search space was experimented with SA, a statistical procedure initially targeted for model evaluation. SA was instead applied

to a program (GA), to gain insight on how each part of the algorithm contributes to the quality of the solutions found. Experiments showed that SA provided correct insight: the straightforward modifications suggested by the analysis lead to a new algorithm that runs $\times 1,000$ faster while finding solutions of comparable quality to the original algorithm. However, the results suggested several improvements. First, the property improved was execution speed, and not parallelism. Second, the method was still largely manual, the SA results required interpretation, and the modifications were not automatic. The key finding from this experiment was that statistical methods are not only useful, but also practical, because we are given an original algorithm to improve upon, which we can execute on demand to produce the necessary observations.

The last experiments aimed to automate the statistical analysis $\rightarrow$ algorithm modification cycle, in order to automatically generate a parallel search program, from an existing sequential version. The automation method is called Savant, in reference to the Savant Syndrome. The Savant approach was able to automatically generate parallel solvers for the use case problem, in a Map-Reduce form. The generated Savant solver can run on a massively parallel machine, even for very small instances of this problem. Moreover, the resulting Savant algorithm is suited to the AWN, by limiting inter-thread communication and synchronization. The Savant solvers found solutions of comparable quality to the original sequential solvers, for different problem types and sizes.

Results for the Savant method are not completely satisfactory. The program generation is not fully automated, and the capability of the generated parallel algorithms needs to be improved. Moreover, the suitability of the Savant to other problems should be tested. Several paths to improvement can be explored. We have relied on decomposition for the problem in our use case. However, any decomposition calls for recombination. The field of collective intelligence, which includes Cellular Automata, illustrates how the combination of simple parts is "greater" than it's sum. The Savant's combination step could be improved, such as to avoid the repetitive fitness evaluations. Further automating the Savant method could be attempted with evolutionary computation. We have already applied evolutionary search for one aspect of the Savant's design. Other manual and problem specific design decisions could be systematically taken by an evolutionary search. Evolutionary computation is well suited to design-time activities, because the search is too time-consuming for runtime. The joint application of machine learning and evolutionary search has previously been identified as an appropriate technique for AI problems [262].

Appendices

Appendix A

Acronyms

2PH Two-Phase Heuristic . . . 127
ALU Arithmetic Logic Unit . . . 137
AWN Array of Wimpy Nodes . . . 10
BSP Bulk Synchronous Parallel . . . 139
CA Cellular Automata . . . 138
CGA Cellular Genetic Algorithm . . . 85
CMP Chip Multi-Processing . . . 25
CLONEALG Clonal Selection Algorithm . . . 121
DoE Design of Experiments . . . 121
DVFS Dynamic Voltage and Frequency Scaling . . . 16
DMA Direct Memory Access . . . 16
EA Evolutionary Algorithm . . . 87
EDP Energy Delay Product . . . 73
ETC Expected Time to Compute . . . 57
FAWN Fast Array of Wimpy Nodes . . . 27
FP Factors Prioritization . . . 74
GA Genetic Algorithm . . . 87
GC Garbage Collector . . . 24
GP Genetic Programming . . . 137
GPGPU General Purpose GPU . . . 33
H2LL Highest To Lower Loaded . . . 90
HPC High Performance Computing . . . 16
HS Harmony Search . . . 121

IGA Intelligent Genetic Algorithm . . . 121
LPDDR Low Power DDR . . . 16
LSF Longest Slack First . . . 74
LTT Longest Task Time . . . 63
MTTP Minimum Tardiness Task Problem . . . 107
MMDP Massively Multi-modal Deceptive Problem . . . 106
OAT One factor At a Time . . . 76
OLTP On-Line Transaction Processing . . . 25
PA-CGA Parallel Asynchronous CGA . . . 88
PDF Probability Density Function . . . 122
PDU Power Distribution Unit . . . 18
PRAM Parallel Random Access Machine . . . 139
PUE Power Usage Effectiveness . . . 18
REVAC Relevance Estimation and Value Calibration of Evolutionary Algorithm . . . 122
RTA Real-Time Advisor . . . 51
SA Sensitivity Analysis . . . 12
SIMD Single Instruction Multiple Data . . . 31
SBSA Server Base System Architecture . . . 31
SoC System on Chip . . . 25
SSD Solid State Disk . . . 17
SMP Symmetric Multi-Processor . . . 51
SMT Simultaneous Multi-Threading . . . 53
SPO Sequential Parameter Optimization . . . 121
SSF Shortest Slack First . . . 74
SVM Support Vector Machines . . . 141
TDP Thermal Design Power . . . 26
UPS Uninterruptable Power Supply . . . 18
VM Virtual Machine . . . 21
TCO Total Cost of Operation . . . 10

Appendix B

Thesis Output

The work presented in this thesis lead to the following output:

- 5 journal articles, and one in press (3 as first author) [229, 167, 263, 264, 265],
- 11 international conference articles [198, 177, 141, 148, 159, 112, 228, 266, 214, 230, 213],
- one book chapter [267],
- one poster presentation [96],
- two invited presentations [1],
- two panel participations [2] [268].

[1] http://www.ig.fpms.ac.be/content/fgps175, http://www.metz.supelec.fr/metz/personnel/vialle/seminars/RGE/ResumesRGE-090611Metz.pdf

[2] http://www.irit.fr/cost804/index.php/pubdocsmenuitem/doc_download/103-costic0804-istanbul-5-7nov2012pdf

Bibliography

[1] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos, "Management of an Academic HPC Cluster: The UL Experience," in *Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014)*. Bologna, Italy: IEEE, July 2014.

[2] T. Schneider, I. von Maurich, and T. Guneysu, "Efficient implementation of cryptographic primitives on the ga144 multi-core architecture," in *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*. IEEE, 2013, pp. 67–74.

[3] O. Kramer, "Evolutionary self-adaptation: a survey of operators and strategy parameters," *Evolutionary Intelligence*, vol. 3, pp. 51–65, 2010.

[4] W. L. Bircher and L. K. John, "Complete system power estimation: A trickle-down approach based on performance events," in *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*. IEEE, 2007, pp. 158–168.

[5] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt, "Understanding and designing new server architectures for emerging warehouse-computing environments," in *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*. IEEE, 2008, pp. 315–326.

[6] S. Murugesan, "Harnessing green it: Principles and practices," *IT professional*, vol. 10, no. 1, pp. 24–33, 2008.

[7] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "Fawn: A fast array of wimpy nodes," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 1–14.

[8] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines,"

Synthesis Lectures on Computer Architecture, vol. 8, no. 3, pp. 1–154, 2013.

[9] V. J. Reddi, B. C. Lee, T. Chilimbi, and K. Vaid, "Mobile processors for energy-efficient web search," *ACM Transactions on Computer Systems (TOCS)*, vol. 29, no. 3, p. 9, 2011.

[10] T. H. Nelson and T. H. Nelson, *Computer lib: Dream machines*. Tempus Books of Microsoft Press Redmond, 1987.

[11] D. C. Engelbart, "Reflections on our future," *Bulletin of the American Society for Information Science and Technology*, vol. 39, no. 6, pp. 44–46, 2013.

[12] D. Shasha and C. Lazere, *Out of Their Minds: The Lives and Discoveries of 15 Great Computer Scientists*, 1st ed. Copernicus Books, 1998.

[13] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from berkeley," Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Tech. Rep., 2006.

[14] A. Beloglazov, R. Buyya, Y. C. Lee, A. Zomaya *et al.*, "A taxonomy and survey of energy-efficient data centers and cloud computing systems," *Advances in Computers*, vol. 82, no. 2, pp. 47–111, 2011.

[15] A.-C. Orgerie, M. D. De Assuncao, and L. Lefevre, "A survey on techniques for improving the energy efficiency of large scale distributed systems," *ACM Computing Surveys*, vol. 46, no. 4, pp. 1–35, 2014.

[16] B. Diniz, D. Guedes, W. Meira Jr, and R. Bianchini, "Limiting the power consumption of main memory," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 290–301.

[17] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah, "Analyzing the energy efficiency of a database server," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 231–242.

[18] L. A. Barroso and U. Holzle, "The case for energy-proportional computing," *Computer*, vol. 40, no. 12, pp. 33–37, 2007.

[19] D. Meisner, B. T. Gold, and T. F. Wenisch, "Powernap: eliminating server idle power," in *ACM Sigplan Notices*, vol. 44, no. 3. ACM, 2009, pp. 205–216.

[20] Gartner, "Magic quadrant for cloud infrastructure as a service," https://www.gartner.com/technology/reprints.do?id=1-1IMDMZ5\&ct=130819&st=sb, 2013.

[21] K. T. Malladi, B. C. Lee, F. A. Nothaft, C. Kozyrakis, K. Periyathambi, and M. Horowitz, "Towards energy-proportional datacenter memory with mobile dram," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 37–48, 2012.

[22] D. H. Yoon, J. Chang, N. Muralimanohar, and P. Ranganathan, "Boom: enabling mobile memory based low-power server dimms," in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3. IEEE Computer Society, 2012, pp. 25–36.

[23] N. Bellas, I. N. Hajj, C. D. Polychronopoulos, and G. Stamoulis, "Architectural and compiler techniques for energy reduction in high-performance microprocessors," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 8, no. 3, pp. 317–326, 2000.

[24] L. Minas and B. Ellison, *Energy efficiency for information technology: How to reduce power consumption in servers and data centers*. Intel Press USA, 2009.

[25] X. Fan, W.-D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 13–23.

[26] D. Kliazovich, P. Bouvry, and S. U. Khan, "Greencloud: a packet-level simulator of energy-aware cloud computing data centers," *The Journal of Supercomputing*, vol. 62, no. 3, pp. 1263–1283, 2012.

[27] D. Abts, M. R. Marty, P. M. Wells, P. Klausler, and H. Liu, "Energy proportional datacenter networks," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 338–347.

[28] J. Baliga, R. Ayre, W. V. Sorin, K. Hinton, and R. S. Tucker, "Energy consumption in access networks," in *Optical Fiber Communication Conference*. Optical Society of America, 2008, p. OThT6.

[29] Greentouch, "Increase network energy efficiency by a factor of 1000 compared to 2010 levels," http://www.greentouch.org, 2014.

[30] P. Vetter, T. Ayhan, K. Kanonakis, B. Lannoo, K. L. Lee, L. Lefèvre, C. Monney, F. Saliou, and X. Yin, "Towards Energy Efficient Wireline Networks, An Update From GreenTouch," in *OptoElectronics and Communications Conference (OECC) 2013*, Kyoto, Japan, 2013. [Online]. Available: http://hal.inria.fr/hal-00925191

[31] J. Hamilton, "Internet-scale service efficiency," in *Large-Scale Distributed Systems and Middleware (LADIS) Workshop (September 2008)*, 2008.

[32] A. Pratt, P. Kumar, K. Bross, and T. Aldridge, "Powering compute platforms in high efficiency data centers."

[33] J. Hamilton, "Cooperative expendable micro-slice servers (cems): low cost, low power servers for internet-scale services," in *Conference on Innovative Data Systems Research (CIDR09)(January 2009)*, 2009.

[34] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, pp. 68–73, 2008.

[35] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 2005, pp. 273–286.

[36] N. Tolia, Z. Wang, M. Marwah, C. Bash, P. Ranganathan, and X. Zhu, "Delivering energy proportionality with non energy-proportional systems-optimizing the ensemble." *HotPower*, vol. 8, pp. 2–2, 2008.

[37] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power management of online data-intensive services," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE, 2011, pp. 319–330.

[38] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[39] S. Mittal, "A survey of architectural techniques for dram power management," *International Journal of High Performance Systems Architecture*, vol. 4, no. 2, pp. 110–119, 2012.

[40] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis, "Power aware page allocation," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 105–116, 2000.

[41] X. Fan, C. Ellis, and A. Lebeck, "Memory controller policies for dram power management," in *Proceedings of the 2001 international symposium on Low power electronics and design*. ACM, 2001, pp. 129–134.

[42] X. Li, Z. Li, F. David, P. Zhou, Y. Zhou, S. Adve, and S. Kumar, "Performance directed energy management for main memory and disks,"

ACM SIGARCH Computer Architecture News, vol. 32, no. 5, pp. 271–283, 2004.

[43] V. De La Luz, M. Kandemir, and I. Kolcu, "Automatic data migration for reducing energy consumption in multi-bank memory systems," in *Design Automation Conference, 2002. Proceedings. 39th.* IEEE, 2002, pp. 213–218.

[44] H. Huang, K. G. Shin, C. Lefurgy, and T. Keller, "Improving energy efficiency by making dram less randomly accessed," in *Proceedings of the 2005 international symposium on Low power electronics and design.* ACM, 2005, pp. 393–398.

[45] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic tracking of page miss ratio curve for memory management," in *ACM SIGOPS Operating Systems Review*, vol. 38, no. 5. ACM, 2004, pp. 177–188.

[46] V. Pandey, W. Jiang, Y. Zhou, and R. Bianchini, "Dma-aware memory energy management." in *HPCA*, vol. 6, 2006, pp. 133–144.

[47] M. Ghosh and H.-H. S. Lee, "Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3d die-stacked drams," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE Computer Society, 2007, pp. 134–145.

[48] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "Raidr: Retention-aware intelligent dram refresh," in *Computer Architecture (ISCA), 2012 39th Annual International Symposium on.* IEEE, 2012, pp. 1–12.

[49] C. Isen and L. John, "Eskimo-energy savings using semantic knowledge of inconsequential memory occupancy for dram subsystem," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on.* IEEE, 2009, pp. 337–346.

[50] J. Trajkovic, A. V. Veidenbaum, and A. Kejariwal, "Improving sdram access energy efficiency for low-power embedded systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 24, 2008.

[51] S. Mazumdar, D. M. Tullsen, and J. Song, "Inter-socket victim cacheing for platform power reduction," in *Computer Design (ICCD), 2010 IEEE International Conference on.* IEEE, 2010, pp. 509–514.

[52] S. Phadke and S. Narayanasamy, "Mlp aware heterogeneous memory system," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011.* IEEE, 2011, pp. 1–6.

[53] R. Ayoub, K. R. Indukuri, and T. S. Rosing, "Energy efficient proactive thermal management in memory subsystem," in *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*. IEEE, 2010, pp. 195–200.

[54] C.-H. Lin, C.-L. Yang, and K.-J. King, "Ppt: joint performance/power/thermal management of dram memory for multi-core systems," in *Proceedings of the 14th ACM/IEEE international symposium on Low power electronics and design*. ACM, 2009, pp. 93–98.

[55] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "Elastictree: Saving energy in data center networks." in *NSDI*, vol. 10, 2010, pp. 249–264.

[56] D. Kliazovich, P. Bouvry, and S. U. Khan, "Dens: data center energy-efficient network-aware scheduling," *Cluster computing*, vol. 16, no. 1, pp. 65–75, 2013.

[57] K. Bilal, S. U. R. Malik, O. Khalid, A. Hameed, E. Alvarez, V. Wijaysekara, R. Irfan, S. Shrestha, D. Dwivedy, M. Ali *et al.*, "A taxonomy and survey on green data center networks," *Future Generation Computer Systems*, 2013.

[58] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," in *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4. ACM, 2009, pp. 51–62.

[59] C. Gunaratne, K. Christensen, B. Nordman, and S. Suen, "Reducing the energy consumption of ethernet with adaptive link rate (alr)," *Computers, IEEE Transactions on*, vol. 57, no. 4, pp. 448–461, 2008.

[60] S. Nedevschi, L. Popa, G. Iannaccone, S. Ratnasamy, and D. Wetherall, "Reducing network energy consumption via sleeping and rate-adaptation." in *NSDI*, vol. 8, 2008, pp. 323–336.

[61] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: a first step towards software power minimization," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 2, no. 4, pp. 437–445, 1994.

[62] J. Zambreno, M. T. Kandemir, and A. Choudhary, "Enhancing compiler techniques for memory energy optimizations," in *Embedded Software*. Springer-Verlag, Heidelberg, 2002, pp. 364–381.

[63] C. Lee, J. K. Lee, T. Hwang, and S.-C. Tsai, "Compiler optimization on vliw instruction scheduling for low power," *ACM Transactions on*

Design Automation of Electronic Systems (TODAES), vol. 8, no. 2, pp. 252–268, 2003.

[64] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko, "Heap compression for memory-constrained java environments," *ACM SIGPLAN Notices*, vol. 38, no. 11, pp. 282–301, 2003.

[65] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Wolf, "Energy savings through compression in embedded java environments," in *Proceedings of the tenth international symposium on Hardware/software codesign.* ACM, 2002, pp. 163–168.

[66] P. Griffin, W. Srisa-An, and J. M. Chang, "An energy efficient garbage collector for java embedded devices," in *ACM SIGPLAN Notices*, vol. 40, no. 7. ACM, 2005, pp. 230–238.

[67] V. De La Luz, M. Kandemir, G. Chen, and I. Kolcu, "Energy-conscious memory allocation and deallocation for pointer-intensive applications," in *Embedded Software.* Springer-Verlag, Heidelberg, 2003, pp. 156–172.

[68] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko, "Tuning garbage collection in an embedded java environment," in *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on.* IEEE, 2002, pp. 92–103.

[69] B. Cmelik and D. Keppel, *Shade: A fast instruction-set simulator for execution profiling.* Springer-Verlag, Heidelberg, 1995.

[70] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye, *Energy-driven integrated hardware-software optimizations using SimplePower.* ACM, 2000, vol. 28, no. 2.

[71] Y. Chen, L. Keys, and R. H. Katz, "Towards energy efficient mapreduce," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-109*, 2009.

[72] N. Maheshwari, R. Nanduri, and V. Varma, "Dynamic energy efficient data placement and cluster reconfiguration algorithm for mapreduce framework," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 119–127, 2012.

[73] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, *Piranha: a scalable architecture based on single-chip multiprocessing.* ACM, 2000, vol. 28, no. 2.

[74] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The case for a single-chip multiprocessor," *ACM Sigplan Notices*, vol. 31, no. 9, pp. 2–11, 1996.

[75] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis, "Joulesort: a balanced energy-efficiency benchmark," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, 2007, pp. 365–376.

[76] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4. ACM, 2001, pp. 149–160.

[77] V. Vasudevan, D. Andersen, M. Kaminsky, L. Tan, J. Franklin, and I. Moraru, "Energy-efficient cluster computing with fawn: Workloads and implications," in *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*. ACM, 2010, pp. 195–204.

[78] A. S. Szalay, G. C. Bell, H. H. Huang, A. Terzis, and A. White, "Low-power amdahl-balanced blades for data intensive computing," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 1, pp. 71–75, 2010.

[79] D. A. Patterson, "Latency lags bandwith," *Communications of the ACM*, vol. 47, no. 10, pp. 71–75, 2004.

[80] U. Hölzle, "Brawny cores still beat wimpy cores, most of the time," *IEEE Micro*, vol. 30, no. 4, 2010.

[81] W. Lang, J. M. Patel, and S. Shankar, "Wimpy node clusters: What about non-wimpy workloads?" in *Proceedings of the Sixth International Workshop on Data Management on New Hardware*. ACM, 2010, pp. 47–55.

[82] A. Cockcroft, "Millicomputing: The future in your pocket and your datacenter," in *USENIX Conference, invited talk*, 2008.

[83] Z. Ou, B. Pang, Y. Deng, J. K. Nurminen, A. Yla-Jaaski, and P. Hui, "Energy-and cost-efficiency analysis of arm-based clusters," in *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*. IEEE, 2012, pp. 115–123.

[84] R. V. Aroca and L. M. G. Gonçalves, "Towards green data centers: A comparison of x86 and arm architectures power efficiency," *Journal of Parallel and Distributed Computing*, vol. 72, no. 12, pp. 1770–1780, 2012.

[85] N. Rajovic, L. Vilanova, C. Villavieja, N. Puzovic, and A. Ramirez, "The low power architecture approach towards exascale computing," *Journal of Computational Science*, vol. 4, no. 6, pp. 439–443, 2013.

[86] K. Fürlinger, C. Klausecker, and D. Kranzlmüller, "Towards energy efficient parallel computing on consumer electronic devices," in *Information and Communication on Technology for the Fight against Global Warming*. Springer-Verlag, Heidelberg, 2011, pp. 1–9.

[87] M. Jarus, S. Varrette, A. Oleksiak, and P. Bouvry, "Performance evaluation and energy efficiency of high-density hpc platforms based on intel, amd and arm processors," in *Energy Efficiency in Large Scale Distributed Systems*. Springer-Verlag, Heidelberg, 2013, pp. 182–200.

[88] N. Rajovic, A. Rico, N. Puzovic, C. Adeniyi-Jones, and A. Ramirez, "Tibidabo: Making the case for an arm-based hpc system," *Future Generation Computer Systems*, 2013.

[89] R. Courtland, "The high stakes of low power," *Spectrum, IEEE*, vol. 49, no. 5, pp. 11–12, 2012.

[90] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[91] C. Nvidia, "Compute unified device architecture programming guide," 2007.

[92] K. O. W. Group *et al.*, "The opencl specification," *A. Munshi, Ed*, 2008.

[93] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openaccfirst experiences with real-world applications," in *Euro-Par 2012 Parallel Processing*. Springer-Verlag, Heidelberg, 2012, pp. 859–870.

[94] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: designing soc accelerators for memcached," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ACM, 2013, pp. 36–47.

[95] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*. Newnes, 2013.

[96] V. Delplace, P. Manneback, F. Pinel, S. Varette, and P. Bouvry, "Comparing the performance and power usage of gpu and arm clusters for map-reduce," in *Cloud and Green Computing (CGC), 2013 Third International Conference on*. IEEE, 2013, pp. 199–200.

[97] D. Borthakur, "The hadoop distributed file system: Architecture and design," *Hadoop Project Website*, vol. 11, p. 21, 2007.

[98] P. Mundkur, V. Tuulos, and J. Flatow, "Disco: a computing platform for large-scale data analytics," in *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*. ACM, 2011, pp. 84–89.

[99] P. Mundkur, "Disco: Beyond mapreduce," 2013. [Online]. Available: \url{www.erlang-factory.com/upload/presentations/778/ef2013-disco.pdf}

[100] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 260–269.

[101] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 2009, pp. 198–207.

[102] J. Dongarra and M. A. Heroux, "Toward a new metric for ranking high performance computing systems," *Sandia Report, SAND2013-4744*, vol. 312, 2013.

[103] F. Ronquist, M. Teslenko, P. van der Mark, D. L. Ayres, A. Darling, S. Höhna, B. Larget, L. Liu, M. A. Suchard, and J. P. Huelsenbeck, "Mrbayes 3.2: efficient bayesian phylogenetic inference and model choice across a large model space," *Systematic biology*, vol. 61, no. 3, pp. 539–542, 2012.

[104] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*. IEEE, 2010, pp. 41–51.

[105] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a high-level dataflow system on top of map-reduce: the pig experience," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1414–1425, 2009.

[106] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A self-tuning system for big data analytics." in *CIDR*, vol. 11, 2011, pp. 261–272.

[107] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. Jones, and İ. Birol, "Abyss: a parallel assembler for short read sequence data," *Genome research*, vol. 19, no. 6, pp. 1117–1123, 2009.

[108] W. R. Pearson, "Flexible sequence similarity searching with the fasta3 program package," in *Bioinformatics methods and protocols*. Springer-Verlag, Heidelberg, 1999, pp. 185–219.

[109] W. H. Wolf, "Hardware-software co-design of embedded systems [and prolog]," *Proceedings of the IEEE*, vol. 82, no. 7, pp. 967–989, 1994.

[110] J. Carlstrom and T. Bodén, "Synchronous dataflow architecture for network processors," *Micro, IEEE*, vol. 24, no. 5, pp. 10–18, 2004.

[111] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan, "Software pipelining," *ACM Computing Surveys (CSUR)*, vol. 27, no. 3, pp. 367–432, 1995.

[112] F. Pinel, J. E. Pecero, P. Bouvry, and S. U. Khan, "A review on task performance prediction in multi-core based systems," in *Computer and Information Technology (CIT), 2011 IEEE 11th International Conference on*. IEEE, 2011, pp. 615–620.

[113] N. H. Walfield and M. Brinkmann, "A critique of the gnu hurd multi-server operating system," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 30–39, July 2007. [Online]. Available: \url{http://doi.acm.org/10.1145/1278901.1278907}

[114] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox, "Pace: A toolset for the performance prediction of parallel and distributed systems," *International Journal of High Performance Computing Applications*, vol. 14, no. 3, pp. 228–251, Fall 2000. [Online]. Available: \url{http://hpc.sagepub.com/content/14/3/228.abstract}

[115] D. Snowdon, S. Ruocco, and G. Heiser, "Power management and dynamic voltage scaling: Myths and facts," 2005.

[116] K. Klues, V. Handziski, C. Lu, A. Wolisz, D. Culler, D. Gay, and P. Levis, "Integrating concurrency control and energy management in device drivers," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 251–264. [Online]. Available: \url{http://doi.acm.org/10.1145/1294261.1294286}

[117] P.-H. Kamp, "You're doing it wrong," *Communications of the ACM*, vol. 53, no. 7, pp. 55 – 59, 2010. [Online]. Available: \url{http://cacm.acm.org/magazines/2010/7/95061-youre-doing-it-wrong/fulltext}

[118] J. Bonwick, "The slab allocator: an object-caching kernel memory allocator," in *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, ser. USTC'94. Berkeley, CA, USA: USENIX Association, 1994, pp. 6–6. [Online]. Available: \url{http://portal.acm.org/citation.cfm?id=1267257.1267263}

[119] T. Braun, H. Siegel, and A. Maciejewski, "Heterogeneous computing: Goals, methods, and open problems," in *High Performance Computing HiPC 2001*, ser. Lecture Notes in Computer Science, B. Monien, V. Prasanna, and S. Vajapeyam, Eds. Springer-Verlag, Heidelberg, 2001, vol. 2228, pp. 307–318, 10.1007/3-540-45307-5_27.

[120] S. Shivle, P. Sugavanam, H. J. Siegel, A. A. Maciejewski, T. Banka, K. Chindam, S. Dussinger, A. Kutruff, P. Penumarthy, P. Pichumani, P. Satyasekaran, D. Sendek, J. Smith, J. Sousa, J. Sridharan, and J. Velazco, "Mapping subtasks with multiple versions on an ad hoc grid," *Parallel Comput.*, vol. 31, pp. 671–690, July 2005.

[121] M. Dobber, R. D. van der Mei, and G. Koole, "Effective prediction of job processing times in a large-scale grid environment," in *HPDC*. IEEE, 2006, pp. 359–360.

[122] R. Wolski, "Experiences with predicting resource performance on-line in computational grid settings," *SIGMETRICS Performance Evaluation Review*, vol. 30, no. 4, pp. 41–49, 2003.

[123] P. A. Dinda and D. R. O'Hallaron, "Host load prediction using linear models," *Cluster Computing*, vol. 3, no. 4, pp. 265–280, 2000.

[124] P. A. Dinda, "Online prediction of the running time of tasks," *Cluster Computing*, vol. 5, pp. 225–236, 2002, 10.1023/A:1015634802585. [Online]. Available: \url{http://dx.doi.org/10.1023/A:1015634802585}

[125] L. Glimcher and G. Agrawal, "A performance prediction framework for grid-based data mining applications," in *IPDPS*. IEEE, 2007, pp. 1–10.

[126] R. Hoffmann and T. Rauber, "Profiling of task-based applications on shared memory machines: Scalability and bottlenecks," in *Euro-Par 2007 Parallel Processing*, ser. Lecture Notes in Computer Science, A.-M. Kermarrec, L. Bougé, and T. Priol, Eds. Springer-Verlag, Heidelberg, 2007, vol. 4641, pp. 118–128, 10.1007/978-3-540-74466-5_14.

[127] F. Nadeem, M. Yousaf, R. Prodan, and T. Fahringer, "Soft benchmarks-based application performance prediction using a minimum training set," in *e-Science and Grid Computing, 2006. e-Science '06. Second IEEE International Conference on*, dec. 2006, p. 71.

[128] S. Seneviratne and D. C. Levy, "Task profiling model for load profile prediction," *Future Generation Computer Systems*, vol. 27, no. 3, pp. 245 – 255, 2011. [Online]. Available: \url{http://www.sciencedirect.com/science/article/B6V06-511TN6Y-3/2/47447581cb2d7ac8595db2a4d5792f20}

[129] R. M. Badia, F. Escale, E. Gabriel, J. Gimenez, R. Keller, J. Labarta, and M. S. Muller, "Performance prediction in a grid environment," in *Grid Computing*, ser. Lecture Notes in Computer Science, F. Fernandez Rivera, M. Bubak, A. Gomez Tato, and R. Doallo, Eds. Springer-Verlag, Heidelberg, 2004, vol. 2970, pp. 257–264, 10.1007/978-3-540-24689-3_32. [Online]. Available: \url{http://dx.doi.org/10.1007/978-3-540-24689-3_32}

[130] M. Iverson, F. Ozguner, and L. Potter, "Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment," *Computers, IEEE Transactions on*, vol. 48, no. 12, pp. 1374 –1379, dec 1999.

[131] C. von Praun, R. Bordawekar, and C. Cascaval, "Modeling optimistic concurrency using quantitative dependence analysis," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 185–196.

[132] T. Abdelzaher, "An automated profiling subsystem for qos-aware services," in *In IEEE Real-Time Technology and Applications Symposium*, 2000, pp. 208–217.

[133] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreading processor," in *ASPLOS*, 2000, pp. 234–244.

[134] A. Snavely, D. M. Tullsen, and G. M. Voelker, "Symbiotic jobscheduling with priorities for a simultaneous multithreading processor," in *SIGMETRICS*. ACM, 2002, pp. 66–76.

[135] A. El-Moursy, R. Garg, D. H. Albonesi, and S. Dwarkadas, "Compatible phase co-scheduling on a cmp of multi-threaded processors," in *IPDPS*. IEEE, 2006.

[136] W. Yuan and K. Nahrstedt, "Energy-efficient soft real-time cpu scheduling for mobile multimedia systems," in *SOSP*, M. L. Scott and L. L. Peterson, Eds. ACM, 2003, pp. 149–163.

[137] H. Wu, B. Ravindran, E. D. Jensen, and P. Li, "Energy-efficient, utility accrual scheduling under resource constraints for mobile embedded systems," *ACM Trans. Embed. Comput.*

Syst., vol. 5, pp. 513–542, August 2006. [Online]. Available: \url{http://doi.acm.org/10.1145/1165780.1165781}

[138] T. Wolf, "Challenges and applications for network-processor-based programmable routers," in *Sarnoff Symposium, 2006 IEEE*, march 2006, pp. 1 –4.

[139] X. Huang and T. Wolf, "Evaluating dynamic task mapping in network processor runtime systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, pp. 1086–1098, 2008.

[140] Q. Wu and T. Wolf, "On runtime management in multi-core packet processing systems," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '08. New York, NY, USA: ACM, 2008, pp. 69–78.

[141] F. Pinel, J. E. Pecero, P. Bouvry, and S. U. Khan, "Memory-aware green scheduling on multi-core processors," in *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*. IEEE, 2010, pp. 485–488.

[142] A. Merkel, J. Stoess, and F. Bellosa, "Resource-conscious scheduling for energy efficiency on multicore processors," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 153–166.

[143] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," 1995.

[144] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on nonidentical processors," *Journal of the ACM*, vol. 24, no. 2, pp. 280–289, 1977.

[145] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen *et al.*, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed computing*, vol. 61, no. 6, pp. 810–837, 2001.

[146] A. Ghafoor and J. Yang, "Distributed heterogeneous supercomputing management system," *ECE Technical Reports*, p. 270, 1992.

[147] M. Kafil and I. Ahmad, "Optimal task assignment in heterogeneous distributed computing systems," *Concurrency, IEEE*, vol. 6, no. 3, pp. 42–50, 1998.

[148] F. Pinel and P. Bouvry, "A model for energy-efficient task mapping on milliclusters," *Proceedings of the Second International Conference*

on Parallel, Distributed, Grid and Cloud Computing for Engineering, 2011.

[149] J. R. Lorch and A. J. Smith, "Improving dynamic voltage scaling algorithms with pace," *SIGMETRICS Perform. Eval. Rev.*, vol. 29, pp. 50–61, June 2001.

[150] V. Venkatachalam and M. Franz, "Power reduction techniques for microprocessor systems," *ACM Computing Survey*, vol. 37, no. 3, pp. 195–237, 2005.

[151] A. Ghafoor and J. Yang, "A distributed heterogeneous supercomputing management system," *IEEE Computer*, vol. 26, no. 6, pp. 78–86, 1993.

[152] M. Kafil and I. Ahmad, "Optimal task assignment in heterogeneous computing systems," in *Heterogeneous Computing Workshop*. IEEE Computer Society, 1997, pp. 135–146.

[153] S. U. Khan and I. Ahmad, "A cooperative game theoretical technique for joint optimization of energy consumption and response time in computational grids," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 20, no. 3, pp. 346–360, 2009.

[154] S. U. Khan and C. Ardil, "A weighted sum technique for the joint optimization of performance and power consumption in data centers," *International Journal of Electrical, Computer, and Systems Engineering*, vol. 3, pp. 35–40, 2009.

[155] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, pp. 20–24, March 1995. [Online]. Available: http://doi.acm.org/10.1145/216585.216588

[156] A. S. William Gropp, Ewing Lusk, *Using MPI*. MIT Press, 1999.

[157] J. Armstrong, "The development of erlang," in *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ser. ICFP '97. New York, NY, USA: ACM, 1997, pp. 196–203. [Online]. Available: http://doi.acm.org/10.1145/258948.258967

[158] J.-R. Abrial, *Modeling in Event-B*. Cambridge University Press, 2010.

[159] F. Pinel, J. E. Pecero, S. U. Khan, and P. Bouvry, "Energy-efficient scheduling on milliclusters with performance constraints," in *Proceedings of the 2011 IEEE/ACM International Conference on Green Computing and Communications*. IEEE Computer Society, 2011, pp. 44–49.

[160] S. Ali, H. J. Siegel, M. Maheswaran, D. Hensgen, and S. Ali, "Representing task and machine heterogeneities for heterogeneous," *Journal of Science and Engineering, Special 50 th Anniversary Issue*, vol. 3, pp. 195–207, 2000.

[161] D. Page, *A Practical Introduction to Computer Architecture*, 1st ed. Springer-Verlag, Heidelberg, 2009.

[162] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. R. Kan, "Optimization and approximation in deterministic sequencing and scheduling: A survey," *Annals of Discrete Mathematics*, vol. 29, pp. 287–326, 1979.

[163] A. Saltelli, S. Tarantola, F. Campolongo, and M. Ratto, *Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models*. Wiley, 2004.

[164] A. Saltelli, S. Tarantola, and K. Chan, "A quantitative, model independent method for global sensitivity analysis of model output," *Technometrics*, vol. 41, pp. 39–56, 1999.

[165] E. Horowitz and S. Sahni, "Exact and approximate algorithms for scheduling nonidentical processors," *J. ACM*, vol. 23, pp. 317–327, April 1976.

[166] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman, "Heuristics for scheduling parameter sweep applications in grid environments," in *Heterogeneous Computing Workshop*, 2000, pp. 349–363.

[167] F. Pinel, B. Dorronsoro, and P. Bouvry, "Solving very large instances of the scheduling of independent tasks problem on the gpu," *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 101–110, 2013.

[168] G. Ritchie and J. Levine, "A hybrid ant algorithm for scheduling independent jobs in heterogeneous computing environments," in *23rd Workshop of the UK Planning and Scheduling Special Interest Group (PLANSIG 2004)*, 2004.

[169] P. Luo, K. Lu, and Z. Shi, "A revisit of fast greedy heuristics for mapping a class of independent tasks onto heterogeneous computing systems," *Journal of Parallel and Distributed Computing*, vol. 67, pp. 695–714, 2007.

[170] F. Xhafa, J. Carretero, E. Alba, and B. Dorronsoro, "Design and evaluation of tabu search method for job scheduling in distributed environments," in *Nature Inspired Distributed Computing (NIDISC)*

sessions of the International Parallel and Distributed Processing Symposium (IPDPS) 2008 Workshop. IEEE Press, 2008, pp. 2319–2326.

[171] F. Xhafa, E. Alba, B. Dorronsoro, and B. Duran, "Efficient batch job scheduling in grids using cellular memetic algorithms," *Journal of Mathematical Modelling and Algorithms*, vol. 7, no. 2, pp. 217–236, 2008.

[172] B. Dorronsoro, P. Bouvry, J. A. C. nero, A. A. Maciejewski, and H. J. Siegel, "Multi-objective robust static mapping of independent tasks on grids," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC), part of World Conference in Computational Intelligence (WCCI)*, 2010, pp. 3389–3396.

[173] C. O. Diaz, M. Guzek, J. E. Pecero, G. Danoy, P. Bouvry, and S. U. Khan, "Energy-aware fast scheduling heuristics in heterogeneous computing systems," in *High Performance Computing and Simulation (HPCS), 2011 International Conference on*. IEEE, 2011, pp. 478–484.

[174] E. Tabak, B. Cambazoglu, and C. Aykanat, "Improving the performance of independent task assignment heuristics minmin, maxmin and sufferage," *Parallel and Distributed Systems, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2013.

[175] C. M. Nesmachnow S., "Gpu implementations of scheduling heuristics for heterogeneous computing environments," in *Proceedings of the XVII Congreso Argentino de Ciencias de la Computación*, 2011, pp. 1563–1570. [Online]. Available: http://www.fing.edu.uy/inco/cursos/hpc/material/clases/gpu_hcsp_heuristics.pdf

[176] S. Nesmachnow, H. Cancela, and E. Alba, "Heterogeneous computing scheduling with evolutionary algorithms," *Soft Computing*, vol. 15, no. 4, pp. 685–701, 2010.

[177] F. Pinel, B. Dorronsoro, and P. Bouvry, "A new parallel asynchronous cellular genetic algorithm for scheduling in grids," in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–8.

[178] E. Alba and B. Dorronsoro, *Cellular Genetic Algorithms*, ser. Operations Research/Computer Science Interfaces. Springer-Verlag, Heidelberg, 2008.

[179] B. Manderick and P. Spiessens, "Fine-grained parallel genetic algorithm," in *Third International Conference on Genetic Algorithms (ICGA)*, J. Schaffer, Ed. Morgan Kaufmann, 1989, pp. 428–433.

[180] D. Whitley, "Cellular genetic algorithms," in *Fifth International Conference on Genetic Algorithms (ICGA)*, S. Forrest, Ed. California, CA, USA: Morgan Kaufmann, 1993, p. 658.

[181] E. Alba and M. Tomassini, "Parallelism and evolutionary algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 5, pp. 443–462, October 2002.

[182] T. Maruyama, A. Konagaya, and K. Konishi, "An asynchronous fine-grained parallel genetic algorithm," in *Proc. of the International Conference on Parallel Problem Solving from Nature II (PPSN-II)*, ser. Lecture Notes in Computer Science (LNCS). North-Holland, 1992, pp. 563–572.

[183] H. Muhlenbein, "Evolution in time and space - the parallel genetic algorithm," in *Foundations of Genetic Algorithms*. Morgan Kaufmann, 1991, pp. 316–337.

[184] E. Alba, M. Giacobini, M. Tomassini, and S. Romero, "Comparing synchronous and asynchronous cellular genetic algorithms," in *Proc. of the International Conference on Parallel Problem Solving from Nature VII (PPSN-VII)*, ser. Lecture Notes in Computer Science (LNCS), J. M. et al., Ed., vol. 2439. Granada, Spain: Springer-Verlag, Heidelberg, 2002, pp. 601–610.

[185] E. Alba, B. Dorronsoro, M. Giacobini, and M. Tomassini, *Handbook of Bioinspired Algorithms and Applications*. CRC Press, 2006, ch. Decentralized Cellular Evolutionary Algorithms, pp. 103–120.

[186] P. Spiessens and B. Manderick, "A massively parallel genetic algorithm: Implementation and first analysis," in *Proc. of the Fourth International Conference on Genetic Algorithms (ICGA)*, R. Belew and L. Booker, Eds. Morgan Kaufmann, 1991, pp. 279–286.

[187] H. Mühlenbein, "Parallel genetic algorithms, population genetic and combinatorial optimization," in *Proc. of the Third International Conference on Genetic Algorithms (ICGA)*. Morgan Kaufmann, 1989, pp. 416–421.

[188] H. Mühlenbein, M. Gorges-Schleuter, and O. Krämer, "Evolution algorithms in combinatorial optimization," *Parallel Computing*, vol. 7, pp. 65–88, 1988.

[189] M. Gorges-Schleuter, "ASPARAGOS - an asynchronous parallel genetic optimization strategy," in *Proc. of the Third International Conference on Genetic Algorithms (ICGA)*, J. Schaffer, Ed. Morgan Kaufmann, 1989, pp. 422–428.

[190] R. Collins and D. Jefferson, "Selection in massively parallel genetic algorithms," in *Proc. of the Fourth International Conference on Genetic Algorithms (ICGA)*, R. Belew and L. Booker, Eds. San Diego, CA, USA: Morgan Kaufmann, 1991, pp. 249–256.

[191] T. Maruyama, T. Hirose, and A. Konagaya, "A fine-grained parallel genetic algorithm for distributed parallel systems," in *Proc. of the Fifth International Conference on Genetic Algorithms (ICGA)*. San Francisco, CA, USA: Morgan Kaufmann, 1993, pp. 184–190.

[192] T. Nakashima, T. Ariyama, and H. Ishibuchi, "Combining multiple cellular genetic algorithms for efficient search," in *Proc. of the Asia-Pacific Conference on Simulated Evolution and Learning (SEAL)*, 2002, pp. 712–716.

[193] G. Folino, C. Pizzuti, and G. Spezzano, "A scalable cellular implementation of parallel genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 7, no. 1, pp. 37–53, February 2003.

[194] G. Luque, E. Alba, and B. Dorronsoro, *Parallel Metaheuristics: A New Class of Algorithms*. Wiley, 2005, ch. Parallel Genetic Algorithms, pp. 107–125.

[195] ——, *Optimization Techniques for Solving Complex Problems*. Wiley, 2009, ch. Analyzing Parallel Cellular Genetic Algorithms, pp. 49–62.

[196] ——, "An asynchronous parallel implementation of a cellular genetic algorithm for combinatorial optimization," in *Proceedings of the International Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 2009, pp. 1395–1402.

[197] B. Dorronsoro, D. Arias, F. Luna, A. Nebro, and E. Alba, "A grid-based hybrid cellular genetic algorithm for very large scale instances of the CVRP," in *High Performance Computing & Simulation Conference (HPCS)*, W. W. Smari, Ed., 2007, pp. 759–765.

[198] F. Pinel, B. Dorronsoro, and P. Bouvry, "A new parallel asynchronous cellular genetic algorithm for de novo genomic sequencing," in *Soft Computing and Pattern Recognition, 2009. SOCPAR'09. International Conference of*. IEEE, 2009, pp. 178–183.

[199] Q. Yu, C. Chen, and Z. Pan, *Advances in Natural Computation*, ser. Lecture Notes in Computer Science (LNCS). Springer-Verlag, Heidelberg, 2005, vol. 3612, ch. Parallel Genetic Algorithms on Programmable Graphics Hardware, pp. 1051–1059.

[200] Z. Luo and H. Liu, "Cellular genetic algorithms and local search for 3-SAT problem on graphic hardware," in *IEEE Congress on Evolutionary Computation*, 2006, pp. 10 345–10 349.

[201] J.-M. Li, X.-J. Wang, R.-S. He, and Z.-X. Chi, "An efficient fine-grained parallel genetic algorithm based on GPU-accelerated," in *IFIP International Conference on Network and Parallel Computing*. IEEE, 2007, pp. 855–862.

[202] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. NVIDIA Corporation, 2011.

[203] K. Group, "Open computing language," http://www.khronos.org/opencl/.

[204] N. Soca, J. L. Blengio, M. Pedemonte, and P. Ezzatti, "PUGACE, a cellular evolutionary algorithm framework on GPUs," in *IEEE Congress on Evolutionary Computation*, 2010, p. eProceedings.

[205] P. Vidal and E. Alba, *Nature Inspired Cooperative Strategies for Optimization (NICSO)*, ser. Studies in Computational Intelligence (SCI). Springer-Verlag, Heidelberg, 2010, vol. 284, ch. Cellular Genetic Algorithm on Graphic Processing Units, pp. 223–232.

[206] ——, "A multi-GPU implementation of a cellular genetic algorithm," in *IEEE Congress on Evolutionary Computation*, 2010, p. eProceedings.

[207] J. Li, L. Zhang, and L. Liu, "A parallel immune algorithm based on fine-grained model with GPU-acceleration," in *Proceedings of the 2009 Fourth International Conference on Innovative Computing, Information and Control*. IEEE Press, 2009, pp. 683–686.

[208] IEEE and The Open Group, "Posix (ieee std 1003.1-2008, open group base specifications issue 7)," http://www.unix.org, 2008.

[209] J. Blazewicz, J. K. Lenstra, and A. H. G. Rinnooy Kan, "Scheduling subject to resource constraints: classification and complexity," *Discrete Applied Mathematics*, vol. 5, pp. 11–24, 1983.

[210] F. Xhafa, "An experimental study on GA replacement operators for scheduling on grids," in *The 2nd International Conference on Bioinspired Optimization Methods and their Applications (BIOMA)*, Ljubljana, Slovenia, October 2006, pp. 212–130.

[211] F. Xhafa, E. Alba, B. Dorronsoro, B. Duran, and A. Abraham, "Efficient batch job scheduling in grids using cellular memetic algorithms," in *Metaheuristics for Scheduling in Distributed Computing Environments*. Springer-Verlag, Heidelberg, 2008, pp. 273–299.

[212] T. Kerrigan, "Tom kerrigan's simple chess program," http://www.tckerrigan.com/Chess/TSCP/.

[213] F. Pinel, D. Dorronsoro, P. Bouvry, and S. U. Khan, "It's not a bug, it'sa feature: Wait-free asynchronous cellular genetic algorithm," in *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, Heidelberg, 2013.

[214] F. Pinel, G. Danoy, and P. Bouvry, "Evolutionary algorithm parameter tuning with sensitivity analysis," in *Security and Intelligent Information Systems*. Springer-Verlag, Heidelberg, 2012, pp. 204–216.

[215] K. De Jong, "Parameter setting in eas: a 30 year perspective," in *Parameter Setting in Evolutionary Algorithms*. Springer, 2007, pp. 1–18.

[216] A. E. Eiben, Z. Michalewicz, M. Schoenauer, and J. Smith, "Parameter control in evolutionary algorithms," in *Parameter setting in evolutionary algorithms*. Springer, 2007, pp. 19–46.

[217] J. Maturana, F. Lardeux, and F. Saubion, "Autonomous operator management for evolutionary algorithms," *Journal of Heuristics*, vol. 16, pp. 881–909, 2010.

[218] S. K. Smit and A. E. Eiben, "Comparing parameter tuning methods for evolutionary algorithms," in *Proceedings of the Eleventh conference on Congress on Evolutionary Computation*, ser. CEC'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 399–406.

[219] A. E. Eiben, R. Hinterding, and Z. Michalewicz, "Parameter control in evolutionary algorithms," *IEEE Trans. Evolutionary Computation*, vol. 3, no. 2, pp. 124–141, 1999.

[220] T. Bartz-Beielstein, C. W. G. Lasarczyk, and M. Preuss, "Sequential Parameter Optimization," in *2005 IEEE Congress on Evolutionary Computation*, vol. 1. IEEE, 2005, pp. 773–780.

[221] L. de Castro and F. Von Zuben, "Learning and optimization using the clonal selection principle," *Evolutionary Computation, IEEE Transactions on*, vol. 6, no. 3, pp. 239 –251, Jun. 2002.

[222] S.-Y. Ho, H.-M. Chen, S.-J. Ho, and T.-K. Chen, "Design of accurate classifiers with a compact fuzzy-rule base using an evolutionary scatter partition of feature space," *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, vol. 34, no. 2, pp. 1031 – 1044, April 2004.

[223] H. Min, H. J. Ko, and C. S. Ko, "A genetic algorithm approach to developing the multi-echelon reverse logistics network for product returns," *Omega*, vol. 34, no. 1, pp. 56 – 69, 2006.

[224] Z. Geem, "Harmony search algorithm for solving sudoku," in *Knowledge-Based Intelligent Information and Engineering Systems*, ser. Lecture Notes in Computer Science, B. Apolloni, R. Howlett, and L. Jain, Eds. Springer Berlin, 2007, vol. 4692, pp. 371–378.

[225] V. Nannen and A. E. Eiben, "Relevance estimation and value calibration of evolutionary algorithm parameters," in *Proceedings of the 20th international joint conference on Artifical intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007, pp. 975–980.

[226] G. Pujol, *sensitivity: Sensitivity Analysis*, 2008, r package version 1.4-0.

[227] P. Moscato and C. Cotta, "A gentle introduction to memetic algorithms," in *Handbook of Metaheuristics*, ser. International Series in Operations Research and Management Science, F. Glover and G. Kochenberger, Eds. Springer New York, 2003, vol. 57, pp. 105–144.

[228] F. Pinel, J. Pecero, P. Bouvry, and S. U. Khan, "A two-phase heuristic for the scheduling of independent tasks on computational grids," in *High Performance Computing and Simulation (HPCS), 2011 International Conference on*. IEEE, 2011, pp. 471–477.

[229] F. Pinel, B. Dorronsoro, J. E. Pecero, P. Bouvry, and S. U. Khan, "A two-phase heuristic for the energy-efficient scheduling of independent tasks on computational grids," *Cluster Computing*, pp. 1–13, 2012.

[230] F. Pinel, B. Dorronsoro, P. Bouvry, and S. U. Khan, "Savant: Automatic parallelization of a scheduling heuristic with machine learning," in *Nature and Biologically Inspired Computing (NaBIC), 2013 World Congress on*. IEEE, 2013, pp. 52–57.

[231] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua, "Automatic program parallelization," *Proceedings of the IEEE*, vol. 81, no. 2, pp. 211–243, 1993.

[232] C. D. Callahan, K. D. Cooper, R. T. Hood, K. Kennedy, and L. Torczon, "ParaScope: A parallel programming environment," *International Journal of High Performance Computing Applications*, vol. 2, no. 4, pp. 84–99, 1988.

[233] F. Irigoin, P. Jouvelot, and R. Triolet, "Semantical interprocedural parallelization: An overview of the PIPS project," in *Proceedings of*

the 5th international conference on Supercomputing. ACM, 1991, pp. 244–251.

[234] X. Leroy, "Formal certification of a compiler back-end or: programming a compiler with a proof assistant," in *ACM SIGPLAN Notices*, vol. 41, no. 1. ACM, 2006, pp. 42–54.

[235] C. Ryan and P. Walsh, "Paragen ii: evolving parallel transformation rules," in *Computational Intelligence Theory and Applications*. Springer, 1997, pp. 573–573.

[236] K. P. Williams, "Evolutionary algorithms for automatic parallelization," Ph.D. dissertation, University of Reading, 1998.

[237] C. Ryan, A. H. van Roermund, and C. J. M. Verhoeven, *Automatic re-engineering of software using genetic programming*. Kluwer Academic, 2000.

[238] A. Nisbet, "Gaps: A compiler framework for genetic algorithm (ga) optimised parallelisation," in *High-Performance Computing and Networking*. Springer, 1998, pp. 987–989.

[239] D. Zhang and J. J. Tsai, "Machine learning and software engineering," *Software Quality Journal*, vol. 11, no. 2, pp. 87–119, 2003.

[240] M. Harman, "The current state and future of search based software engineering," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 342–357.

[241] S. Minton and S. R. Wolfe, "Using machine learning to synthesize search programs," in *Knowledge-Based Software Engineering Conference, 1994. Proceedings., Ninth*. IEEE, 1994, pp. 31–38.

[242] J. R. Koza, "Genetic programming as a means for programming computers by natural selection," *Statistics and Computing*, vol. 4, no. 2, pp. 87–112, 1994.

[243] P. Walsh and C. Ryan, "Paragen: a novel technique for the autoparallelisation of sequential programs using gp," in *Proceedings of the First Annual Conference on Genetic Programming*. MIT Press, 1996, pp. 406–409.

[244] S. M. Cheang, K. S. Leung, and K. H. Lee, "Genetic parallel programming: Design and implementation," *Evolutionary Computation*, vol. 14, no. 2, pp. 129–156, 2006.

[245] K. S. Leung, K. H. Lee, and S. M. Cheang, "Evolving parallel machine programs for a multi-alu processor," in *Evolutionary Computation,*

2002. CEC'02. Proceedings of the 2002 Congress on, vol. 2. IEEE, 2002, pp. 1703–1708.

[246] K. Thearling and T. S. Ray, "Evolving parallel computation," *Complex Systems*, vol. 10, no. 3, p. 229, 1996.

[247] D. E. Goldberg, "Genetic and evolutionary algorithms come of age," *Communications of the ACM*, vol. 37, no. 3, pp. 113–119, 1994.

[248] M. Mitchell, P. Hraber, and J. P. Crutchfield, "Revisiting the edge of chaos: Evolving cellular automata to perform computations," *arXiv preprint adap-org/9303003*, 1993.

[249] M. F. Pace, "Bsp vs mapreduce," *Procedia Computer Science*, vol. 9, pp. 246–255, 2012.

[250] H. Welling, "Prime number identification in idiots savants: Can they calculate them?" *Journal of autism and developmental disorders*, vol. 24, no. 2, pp. 199–207, 1994.

[251] L. Mottron, M. Dawson, I. Soulieres, B. Hubert, and J. Burack, "Enhanced perceptual functioning in autism: An update, and eight principles of autistic perception," *Journal of autism and developmental disorders*, vol. 36, no. 1, pp. 27–43, 2006.

[252] L. Mottron, K. Lemmens, L. Gagnon, and X. Seron, "Non-algorithmic access to calendar information in a calendar calculator with autism," *Journal of autism and developmental disorders*, vol. 36, no. 2, pp. 239–247, 2006.

[253] H. Darius, "Savant syndrome-theories and empirical findings," Ph.D. dissertation, University of Skövde, 2007.

[254] J. R. Hughes, "A review of savant syndrome and its possible relationship to epilepsy," 2010.

[255] L. Mottron, L. Bouvet, A. Bonnel, F. Samson, J. A. Burack, M. Dawson, and P. Heaton, "Veridical mapping in the development of exceptional autistic abilities," *Neuroscience & Biobehavioral Reviews*, 2012.

[256] D. Tammet, *Born on a blue day: Inside the extraordinary mind of an autistic savant*. Simon and Schuster, 2007.

[257] A. R. Luria, *The mind of a mnemonist: A little book about a vast memory*. Harvard University Press, 1968.

[258] W. G. Chase and H. A. Simon, "Perception in chess," *Cognitive Psychology*, vol. 4, no. 1, pp. 55 – 81, 1973. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0010028573900042

[259] N. Charness, E. M. Reingold, M. Pomplun, and D. M. Stampe, "The perceptual aspect of skilled performance in chess: Evidence from eye movements," *Memory & Cognition*, vol. 29, no. 8, pp. 1146–1152, 2001.

[260] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, pp. 27:1–27:27, May 2011. [Online]. Available: http://doi.acm.org/10.1145/1961189.1961199

[261] J. Blazewicz, M. Y. Kovalyov, M. Machowiak, D. Trystram, and J. Weglarz, "Preemptable malleable task scheduling problem," *Computers, IEEE Transactions on*, vol. 55, no. 4, pp. 486–490, 2006.

[262] R. Kurzweil, *The age of spiritual machines: When computers exceed human intelligence*. Penguin, 2000.

[263] P. Ruiz, B. Dorronsoro, G. Valentini, F. Pinel, and P. Bouvry, "Optimisation of the enhanced distance based broadcasting protocol for manets," *The Journal of Supercomputing*, vol. 62, no. 3, pp. 1213–1240, 2012.

[264] G. L. Valentini, W. Lassonde, S. U. Khan, N. Min-Allah, S. A. Madani, J. Li, L. Zhang, L. Wang, N. Ghani, J. Kolodziej *et al.*, "An overview of energy efficiency techniques in cluster computing systems," *Cluster Computing*, pp. 1–13, 2011.

[265] H. Hussain, S. U. R. Malik, A. Hameed, S. U. Khan, G. Bickler, N. Min-Allah, M. B. Qureshi, L. Zhang, W. Yongji, N. Ghani *et al.*, "A survey on resource allocation in high performance distributed computing systems," *Parallel Computing*, vol. 39, no. 11, pp. 709–736, 2013.

[266] J. Pecero, F. Pinel, P. Bouvry, and H. J. Fraire Huacuja, "A multi-objective grasp for energy-aware scheduling," *International Congress on Computer Science Research*, 2011.

[267] J. E. Pecero, F. Pinel, B. Dorronsoro, G. Danoy, P. Bouvry, and A. Y. Zomaya, "Efficient hierarchical task scheduling on grids accounting for computation and communications," in *Intelligent Decision Systems in Large-Scale Distributed Environments*. Springer-Verlag, Heidelberg, 2011, pp. 25–47.

[268] T. El-Ghazawi, F. Pinel, E.-G. Talbi, S. Vialle, and P. Bouvry, "Hpcs 2011 panel session: Graphical processing units (gpus): Opportunities and challenges," in *High Performance Computing and Simulation (HPCS), 2011 International Conference on*. IEEE, 2011, pp. 1–11.