Task scheduling on heterogeneous multi-core

Mohammed Khatiri

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Préparé dans le cadre d'une cotutelle entre la Communauté Université Grenoble Alpes en France et l'Université Mohammed Premier d'Oujda au Maroc

Spécialité : **Informatique**

Arrêté ministériel : le 6 janvier 2005 – 25 mai 2016

Présentée par

Khatiri Mohammed

Thèse dirigée par **Denis TRYSTRAM** et **El Mostafa DAOUDI**

préparée au sein du Laboratoire d'Informatique de Grenoble **(LIG)** dans l'École Doctorale **MSTII** et le laboratoire de Recherches en Informatique **(LaRI)** dans Centre d'Etudes Doctorales Sciences et Techniques

Ordonnancement de tâches sur multi-coeur hétérogènes

Task scheduling on heterogeneous multi-core

Thèse soutenue publiquement le **26-09-2020**,
devant le jury composé de :

El Miloud JAARA
Professeur à l'université Mohammed Premier d'Oujda, Maroc, Présidente

Pierre MANNEBACK
Professeur à l'université de Mons, Belgique, Rapporteur

Swann PERARNAU
Docteur en sciences, laboratoire national d'Argonne, USA, Rapporteur

Mostapha ZBAKH
Professeur à l'ENSIAS, université Mohammed V, Rabat, Maroc, Examinateur

Nadia Brauner VETTIER
Professeur des universités, université Grenoble Alpes, France , Examinateur

Denis TRYSTRAM
Professeur des universités, LIG, Grenoble INP, France, Directeur de thèse

El Mostafa DAOUDI
Professeur à l'université Mohammed Premier d'Oujda, Maroc, Directeur de thèse

Remerciements (Acknowledgement)

I would like to thank all the members of the jury for their helpful feedback and comments, and especially the three reviewers, *Pierre MANNEBACK*, *Swann PERARNAU* and *EL MILOUD JAARA* for reviewing the whole dissertation.

Mes travaux de thèse ne seraient pas ce qu'ils sont sans **Denis TRYSTRAM** et **El Mostafa DAOUDI**. Je souhaite chaudement remercier *Denis* pour ses conseils avisés, sa patience, et surtout pour les discusions et la motivation qui m'ont offert. Je remercie *El Daoudi* qui m'a fait découvrir le monde de la recherche, pour son encadrement et ses conseils précieux. Je souhaite aussi vivement remercier Frederic WAGNER et Nicolas Gast qui m'ont beaucoup appris et ont vivement amélioré la qualité de mon travail. Je tiens à remercier tous les membres des équipes DATAMOVE/POLARIS à Grenoble et LARI à Oujda pour ce formidable cadre de recherche et d'ambiance de travail, un grand merci, mille mercis à Annie simon Je tiens à remercier Alfredo Goldman au Brésil et Erik Saul aux États-Unis qui m'ont accordé des stages de recherche au niveau international. Un grand merci à l'équipe pédagogique de l'UFR et de l'ENSIMAG et à tout le personnel de l'UGA.

Ma mère **Rahma BOUKABOUS** et mon père **Mohammed KHATIRI**, aucun mot ne saurait exprimer tout mon amour et toute ma gratitude. Merci pour vos sacrifices le long de ces années. Merci pour votre présence rassurante. Merci pour tout l'amour que vous pourvoyez à notre petite famille. Un grand merci spécial à ma magnifique soeur **Imane** et à sa nouvelle petite fille **Shahd**, pour l'amour, la tendresse et surtout pour votre présence dans ma vie depuis notre enfance. Grand merci ma petit soeur **Insaf** et mon frère **Youssef**, je vous aime beacoup et je vous souhaite la réussite dans votre vie, avec tout le bonheur qu'il faut pour vous combler. Merci aussi mon beau frère *Rabie*.

Ma merveilleuse femme **Manal SAADI**, Je ne trouverai jamais de phrase pour t'exprimer mes sentiments et mon amour, merci d'avoir donné un sens à ma vie. Merci pour ton amour, ton soutien, ton sacrifice et tes encouragements qui ont toujours été pour moi d'un grand réconfort. *Je t'aime* ...

Je tiens à remercier aussi mon beau-père **Mohammed SAADI** et ma belle-mére **Karima BELKASMI**, merci pour vos prières, vos encouragements et votre soutien m'ont toujours été d'un grand secours. Merci aussi a mes frères **Adil** et **Alae-Edine**, Je vous souhaite la réussite dans votre vie.

De grands mercis à mon cousin **Mounir GRARI**, merci pour ton soutien et ton encadrement. Merci à ma grand-mère, mes oncles, mes tantes et tous mes cousins.

Merci à mes meilleurs amis, Khalil, Mehdi, Yassine et Mohammed, merci pour tous mes amis au Maroc, Merci à tous mes amis en France, merci pour votre soutien et votre présence dans ma vie.

Abstract / Résumé

Abstract

Today, high performance computing platforms (HPC) are experiencing rapid and significant development, they are bigger, faster, more powerful, but also more complex. These platforms are more and more heterogeneous, dynamic and distributed. These characteristics create new challenges for the scheduling problem which corresponds to the allocation of tasks to the different and remote processors. The first challenge is how to effectively manage the heterogeneity of resources which can appear at the computation level or at the communication level. The second challenge is the dynamic nature of tasks and data, To face this challenge, the development must be supported by effective software tools to manage the complexity. In this dissertation, we are interested in both on-line and off-line scheduling problems in heterogeneous resources on a dynamic environment. The crucial performance feature is the communication, which is ignored in most related approaches.

Firstly, we analyze the **Work Stealing** on-line algorithm on parallel and distributed platforms with different contexts of heterogeneity. We start with a mathematical analysis of a new model of **Work Stealing** algorithm in a distributed memory platform where communications between processors are modeled by a large latency. Then, we extend the previous problem to two separate clusters, where the communication between two processors inside the same cluster is much less than an external communication. We study this problem using simulations. Thus, we develop a lightweight *PYTHON* simulator, the simulator is used to simulate different **Work Stealing** algorithms in different contexts (different topologies, different tasks type and different configurations).

In a second part of this work, we focus on two **offline** scheduling problems . Firstly, we consider the scheduling problem of a set of periodic implicit-deadline and synchronous tasks, on a real-time multiprocessor composed of m identical processors including communication. We propose a new tasks allocation algorithm that aims to reduce the number of tasks migrations, and limits migration (of migrant tasks) on two processors. Secondly, we model a recent scheduling problem, which concerns the **micro-services** architectures which aim to divide large applications (*Monolithic applications*) into several micro connected applications (**micro-services**), which makes the scheduling problem of micro-services special. Our model allows us to access several research directions able to identify effective solutions with mathematical approximations.

Résumé

Aujourd'hui, le développement des plates-formes de calcul haute performance (HPC) est considérable, elles sont toujours plus grandes, plus rapides, plus puissantes, mais aussi plus complexes. Ces plates-formes sont plus en plus hétérogènes, dynamiques, mais surtout, distribuées. Ces caractéristiques créent de nouveaux défis pour le problème de ordonnancement qui correspond à l'allocation des tâches aux différentes machines distantes. Le première défi est de savoir comment gérer efficacement l'hétérogénéité des ressources qui peut apparaitre au niveau du calcul ou au niveau des communications. Le deuxième défi est le caractère dynamique des tâches et des données, Pour relever ces défis, il faut accompagner ce développement par des outils logiciels efficaces pour gérer la complexité. Dans cette thèse, nous sommes intéressés aux problèmes d'ordonnancement **en ligne** et **hors-ligne** dans des ressources hétérogènes avec un environnement dynamique. La caractéristique de performance cruciale est la communication, qui est ignorée dans la plupart des approches existantes.

Dans une première partie, nous analysons l'algorithme d'ordonnancement du *vol du travail* **en ligne** sur des plateformes parallèles et distribuées sous plusieurs contextes d'hétérogénéité. Nous commençons par une analyse mathématique d'un nouveau modèle de l'algorithme *vol du travail* sur plate-forme à mémoire distribuée où les communications entre les processeurs sont modélisés par une grande latence. Ensuite, nous étendons le problème précédent à deux clusters où la communication entre deux processeurs à l'intérieur d'un cluster est beaucoup plus petit qu'une communication externe. Nous étudions ce problème à l'aide de simulations. Ainsi, nous développons un propre simulateur, qui sera utilisé pour simuler différents algorithmes de *vol de travail* dans différents contextes (différentes topologies, différents types de tâches et différentes configurations).

Dans une deuxième partie, nous nous concentrons sur deux problèmes d'ordonnancement **hors ligne**. Tout d'abord, l'ordonnancement d'un ensemble de tâches périodiques à échéance implicite et synchrones, sur une plate-forme *temps réel* composée de m processeurs identiques où la communication entre eux est importante. Pour ce problème, nous proposons un nouvel algorithme d'allocation de tâches qui vise à réduire le nombre de migrations de tâches, et limiter la migration (des tâche migrante) à deux processeurs. Ensuite, nous modélisons un problème d'ordonnancement récent, qui concerne les architectures *micro-services* qui visent à diviser les grandes applications (*Applications monolithiques*) en plusieurs petites applications (*micro-services*) connectées. Les *micro-services* ont des caractéristiques très spécifiques, ce qui rend spécial le problème d'ordonnancement (des micro-services). Sans apporter de solution complète, cette modélisation nous permet d'accéder à plusieurs directions de recherche capables de déterminer des solutions efficaces avec des approximations mathématiques.

Contents

1 Introduction

1.1 Context

The computer has made human life easier, by performing complex jobs in a short time, storing and managing multiple information (data) easily. The use of a typical desktop computer has become necessary in several fields, to perform basic computation and data management, even if they are far from computer science. For instance, the use of a simple computer in a pharmacy facilitates the drug store and prescription management.

Afterward, the use of computer science has expanded and the use of simple computers becomes insufficient. The scientific and business fields have started to use intensively computing resources to deal with a complex problem that needs massive computational resources. For instance, climate modeling, finding increasingly bigger prime numbers, analyzing data from genetic information contained in DNA sequences, buying Pattern Analysis and so on. Such fields need sophisticated machines that can work with very high efficiency.

For this purpose, High-performance computing (HPC) community brings an effective solution by combining the computing power of several machines in a way that offers higher performance than one could get out of a typical desktop computer. A first example are supercomputers which are composed of many nodes or CPUs (Central Processing Units) and GPUs (Graphics Processing Units). Another example are large parallel systems with multiple parallel machines connected by high-performance networks. Sometimes such machines are located in different locations (different countries and continents). In the last ranking of TOP500[1] in November 2019, the most powerful supercomputer reached the speed of 1.65 exaflops[2] with more than $2,000,000$ nodes. This means that it can perform more than one billion of billion operations per second. Such platforms require specific skills to install and use them and are way too expensive, they also consume a lot of energy.

With the rise of the internet, the need for computing power and data storage capacity grows more and more, and it is no longer confined in solving complex problems,

[1] www.top500.org
[2] One exaflops corresponds to 10^{18} floating operations per second

but it becomes essential to run millions of web applications used by companies for their employees and customers, and the millions of smartphone applications used everywhere in the world. These applications provoke an explosion of data produced that need massive computational resources to store and manage them. Some of these applications are critical and require a real-time processing, they must be processed in the time and not exceeding the deadline. The main challenge of these applications is processing speed and execution [80].

Cloud computing and data centers are an effective solution for these applications to benefit from High Performance Computing and storage. Cloud computing consists of using remote computer servers via the internet, to run applications, store and manage data. The data centers are buildings used to house many computer and data storage systems in good conditions like security, air conditioner, etc. These systems are used by cloud computing users on-demand, cloud computing users can rent and manage servers by them-self or use just the remote applications hosted in the cloud.

Today, the High Performance Computing community provides many supercomputers for scientific and business fields to perform complex operations and analyze massive data, and offer the cloud computing which facilities for millions of companies and smartphone applications to cover the huge demand for power computing and data storage.

The mechanism for managing these systems is transparent for the users and it does not require specific skills except the few command lines to perform the reservation and launch the execution. But, in the background, these systems are composed of multiple parallel and heterogeneous system interconnect in a very complicated network. The complexity of these systems requires a pearl of wisdom, rational thought, and knowledge to manage different this huge amount of resources, and take advantage of its power in an efficient way.

The biggest challenge of the computational systems is the scheduling tasks problem which represents the heart of effective resource management. Scheduling consists of determining where and when to perform each task in the system, in order to optimize one or more objectives with various constraints [80, 27]. For example, the most common objectives are the Makespan (the total execution time) [19, 80] and the energy consumed during the execution[4]. The scheduling can be off-line (static) if we know in advance information about different tasks (like the arrival time, their execution times on each processor, number and type of resources etc.), in this case, the scheduling is determined before execution. The scheduling can also be on-line (dynamic) if the characteristics of a task are known only when it

arrives in the system. In the on-line method, the scheduling is determined during the execution.

As we mentioned before, the evolution of the digital world opens a huge potential of using highly dynamic and versatile parallel and distributed systems. A direct consequence of such an increasing complexity creates new challenges for the scheduling problem that can not be addressed by classical methods.

1.2 Challenges

A first difficulty is how to effectively manage the heterogeneity of resources. The heterogeneity of such a system can appear either at the level of computation when the different processors of the platform do not have the same computation speed or at the level of communication when the communication between two processors depends on their location or both. In the literature, there are several algorithms that deal with this problem (heterogeneity), but all the efficient algorithms are off-line and task information is considered known before execution[77].

A second difficulty is the dynamic character of the tasks and data. Indeed, new tasks are created during the execution, some stopped unexpectedly, etc. This dynamicity makes the scheduling problem more complex which increased challenges. Again, there exist several solutions to deal with, and the field of scheduling is very active. However, this constraint added to the previous one (dealing with heterogeneity) is highly competitive since both are somehow contradictory[16, 25]. Heterogeneity is well treated with off-line algorithms, but the tasks dynamicity deprives and does not provide any information about tasks before starting execution, which makes the problem even more complicated.

Determining an effective solution for such problems is not straightforward, and the classical methods do not work well since the problem is totally dynamic. Mathematical modeling of the problem is a good way to understand the problem in detail and provides the opportunity to do some mathematical analysis to assess the different solutions. The mathematics analysis help to get an idea of the effectiveness of solutions, for example, bounds (upper or lower) on the Makespan, on the number of used processors, etc.

Usually, theoretical results assume some constraints that lead to idealized models, for instance, where communications are neglected, or in the best case, they are taken into account only partially. This fact creates a distance between theoretical models and practical tools. This distance is affected by three aspects: the level of

heterogeneity of resources, the dynamic character of the tasks and the multiple scale of platforms. This distance could be a barrier between theoretical analysis (giving worst-case or average-case bounds) and the development of practical tools that can be used for solving actual problems.

Simulations are another way to analyze such problems, it helps to get an idea about the distance between theoretical and practical sides. Simulation results can help to get an idea about the efficiency of theoretical bounds. We can also do parametric tests to assess the impact of different parameters. The simulation is also helpful to test different solutions to get an intuition about the behavior of the execution, specially when the theoretical results can not provide an effective evaluation of the solution.

We are interested in this thesis in both on-line and off-line scheduling problems in heterogeneous resources with a dynamic environment. The crucial performance feature is the communication, which is ignored in most related approaches.

We proposed to focus on work stealing models since it combines naturally on-line facilities and it allows implicitly to deal with heterogeneity (this means that this mechanism is transparent for the users). We performed a new mathematical analysis while taking into account communications between processors and we developed an extensive experimental campaign (based on simulations) of the variants of the models.

We studied also several scheduling problems within the off-line context. More precisely, we proposed an algorithm for real-time scheduling that avoids tasks migration. Then, we presented a model for a recent problem of container scheduling posed by micro-services architecture and containers. Through this model, we opened several research directions that are able to determine efficient solutions with mathematics approximations.

1.3 Contributions

The nature of the work conducted in this thesis concerns both theoretical and simulation results. Thus, the contributions are divided into the two following parts:

Contributions of Part 1 : we analyze the **Work Stealing** on-line algorithm on parallel and distributed platforms with different contexts of heterogeneity. The classical distributed-memory model with only one level of memory hierarchy where

the communications are modeled by a large latency, and an extension of this model to heterogeneous architecture composed of two separate computing clusters. Here, the communications are heterogeneous (communication between two processors inside a cluster is much less than an external communication). In both cases, the target objective is the expected makespan.

Our main result is to be able to prove the following bounds :

1. bound on the expected makespan (denoted by $C_{\max}$) of W unit independent tasks scheduled by **Work Stealing** on p processors including latency λ.

$$\mathbb{E}\left[C_{max}\right] \leq \frac{\mathcal{W}}{p} + 4\lambda\gamma \log_2 \frac{\mathcal{W}}{\lambda}$$

2. bound on the expected makespan of W unit tasks with a DAG of precedence that has a critical path of D scheduled by **Work Stealing** on p processors including latency λ.

$$\mathbb{E}\left[C_{max}\right] \leq \frac{\mathcal{W}}{p} + 6\lambda\gamma D$$

Where γ is a positive constant < 4.03.

The analysis was done using adequate potential functions.

For the two-clusters case, the worst case analysis was too hard, and we believe that even if we do an analysis as in the first problem (one cluster), the bounds will be very far from reality. The worst-case analysis aims to suppose the worst case during all the execution time, which is strange if we consider that communications are outside clusters (we pay the worst) all time. For this reason, we analyzed this problem using simulations. Thus, we developed a lightweight **PYTHON** simulator, the simulator is used to simulate different **Work Stealing** algorithms in different context (different topologies, different tasks type and different configurations), the simulator is simple to use and does not need big configuration. In this thesis, the simulator is used for two main reasons: - to evaluate and discuss the quality of the first theoretical analysis, - to perform an experimental study for Work stealing on two-clusters. The simulator is open source and the code is available on github[3].

Contributions of Part 2 : The second series of contributions concern off-line scheduling policies on several heterogeneous contexts.

[3]https://github.com/mkhatiri/ws-simulator

First, we studied the ***real-time scheduling*** in a platform where communications matter, More precisely, we consider the scheduling problem of a set of periodic implicit-deadline and synchronous tasks, on a real-time multiprocessor composed of m identical processors including communication, we propose a new tasks allocation algorithm that aims to reduce the number of tasks migrations, and each migrant task migrates between two processors only.

Second, we investigated a recent scheduling model for **micro-services** architectures. The micro-services architectures aims to divide large applications (*Monolithic application*) into several connected mini-applications (which is called micro-services). These micro-services work together independently to form the whole application. Micro-services can also be executed on different machines. In case of overload, a micro-service can be duplicated and the load is distributed between the duplicated instances, these instances can be executed on different machines. Micro-services can shared the same machine, but each micro-service requires a percentage of CPU and memory. The sum of CPU required for micro-services in the same machine must be less than 1, the same for the Memory requirements. The CPU and Memory requested ratio depends on the micro-services, some micro-services require more CPU than memory, some others require more memory than CPU. Most of these applications are build on cloud computing systems.

The allocation of the different micro-services and their duplicated instances on cloud machines form a new scheduling problem. Due to the cost of a cloud, the target objective is the number of machines used to build the application (set of microservices). Our contribution here is to propose a model for microservices applications, that is used to perform some theoretical analyses in order to give directions to find approximations solutions.

1.4 Content

The chapters of this manuscript are organized as follows, Chapters 2 presents a general related work that summarizes the most relevant references that are linked with our work. Chapters 3 presents the first model of *Work Stealing* on distributed platform including communication delay, presents the two models of tasks used, and describes in detail the analyze using potential functions to bound the total completion time. Chapters 4 presents the different variants of *Work Stealing* algorithm, and presents the architecture of our simulator, and exhibits a first use of the simulator to study experimentally the first model in order to discuss and assess the theoretical bound. Chapters 5 presents the second model of *Work Stealing* on the hierarchical platform of two-clusters, and present different strategies and variants to minimize the overhead, the chapter shows many simulations to compare and discuss these strategies. Chapters 6 focus on a problem of scheduling on real-time systems, and describes the different steps of a new semi partitioned algorithm that reduces the number of migrations. Chapters 7 describes the new architecture of microservice-based applications and proposes two models of the scheduling problem of microservices on clouds. Finally, Chapters 8 presents a general conclusion of this thesis.

2 Related work

2.1 General resource management problem

Scheduling is one of the most important problem while managing distributed resources (allocating a set of jobs/tasks to available resources). This problem consists in determining where to put the tasks and when to start them. There is a huge literature on this problem with thousands papers published each year. This is an old problem and the foundations of the field is on the late sixties [42, 43]

The static version of this problem (off-line) is known to be difficult (the basic version of scheduling a set of independent tasks on two identical machines targeting the minimization of the maximum completion time (makespan denoted C_{max}) is NP-complete, and strongly NP-hard for an arbitrary number of processors [39].

Today, the evolution of parallel and distributed platforms leads to a large number of variants of this problem including various objectives, various types of hardware components, memory constraints, various types of resources, on-line versions, decentralized decision making, etc. Of course, most of these variants are still NP-hard.

There exist several efficient mechanisms for implementing scheduling policies, including the classical list scheduling of Graham, priority queues, packing algorithms, etc. *Work stealing* is one of these mechanisms, efficient in distributed frameworks.

2.2 Work Stealing algorithms

Work Stealing is a decentralized list scheduling algorithm where each processor maintains its own local queue of tasks to execute. When a processor has nothing to execute, it selects another processor uniformly at random to steal one or more tasks if possible. Several works are available on this algorithm,

2.2.1 Theoretically-oriented works

WS has been studied originally by Blumofe and Leiserson in [20]. They showed that the expected Makespan of a series-parallel precedence graph with $\mathcal{W}$ unit tasks on p processors is bounded by $E(C_{\max}) \leq \frac{\mathcal{W}}{p} + \mathcal{O}(D)$ where D is the length of the critical path of the graph (its depth). This analysis has been improved in Arora *et al.* [11] using potential functions. The case of varying processor speeds has been studied by Bender and Rabin in [16] where the authors introduced a new policy called *high utilization scheduler* that extends the homogeneous case. The specific case of tree-shaped computations with a more accurate model has been studied in [76]. However, in all these previous analyses, the precedence graph is constrained to have only one source and an out-degree of at most 2 which does not easily model the basic case of independent tasks.

Simulating independent tasks with a binary precedences tree gives a bound of $\frac{\mathcal{W}}{p} + \mathcal{O}(\log_2(\mathcal{W}))$ since a complete binary tree of $\mathcal{W}$ vertices has a depth $D \leq \log_2(\mathcal{W})$. However, with this approach, the structure of the binary tree dictates which tasks are stolen. In complement, [40] provided a theoretical analysis based on a Markovian model using mean field theory. They targeted the expectation of the average response time and showed that the system converges to a deterministic Ordinary Differential Equation. Note that there exist other results that study the steady state performance of WS when the work generation is random including Berenbrink *et al.* [17], Mitzenmacher [65], Lueling and Monien [60] and Rudolph *et al.* [74]. More recently, in [79], Tchiboukjian *et al.* provided the best bound known at this time: $\frac{\mathcal{W}}{p} + c.(\log_2 \mathcal{W}) + \Theta(1)$ where $c \approx 3.24$.

Acar *et al.* [1, 2] studied data locality of WS on shared-memory and focus on cache misses. The underlying model assumes that the execution time takes m time units if the instruction incurs a cache miss and 1 unit otherwise. A steal attempt takes at least s and at most ks steps to complete ($k \geq 1$ multiple steal attempts before succeeding). When a steal succeeds, the thief starts working on the stolen task at the next step. This model is similar to the classical WS model without communication since the thief does not wait several time steps to receive the stolen task.

2.2.2 Work Stealing and communication issues

In all these previous theoretical results, communications are not directly addressed (or at least they are taken implicitly into account by the underlying model). WS largely focused on shared memory systems and its performance on modern platforms (distributed-memory systems, hierarchical plateform, clusters with explicit commu-

nication cost) is not really well understood. The difficulty lies in the problems of communication which become more crucial in modern platforms [10].

Dinan *et al.* [31] implemented WS on large-scale clusters, and proposed to split each tasks queues into a part accessed asynchronously by local processes and a shared portion synchronized by a lock which can be access by any remote processes in order to reduce contention. Multi-core processor based on NUMA (non-uniform memory access) architecture is the mainstream today and such new platforms include accelerators as GPGPU [81]. The authors propose an efficient task management mechanism which is to divide the application into a large number of fine grained tasks which generate large amount of small communications between the CPU and the GPGPUs. However, these data transmissions are very slow. They consider that the transmission time of small data and big data is the same.

Besides the large literature on theoretical works, there exist more practical studies implementing WS libraries where some attempts were provided for taking into account communications.

2.2.3 Work Stealing Applications

SLAW is a task-based library introduced in [44], combining work-first and help-first scheduling policies focused on locality awareness in PGAS (Partitioned Global Address Space) languages like UPC (Unified Parallel C). It has been extended in HotSLAW, which provides a high level API that abstracts concurrent task management [64]. [57] proposes an asynchronous WS (AsynchWS) strategy which exploits opportunities to overlap communication with local tasks allowing to hide high communication overheads in distributed memory systems. The principle is based on a hierarchical victim selection, also based on PGAS. Perarnau and Sato presented in [70] an experimental evaluation of WS on the scale of ten thousands compute nodes where the communication depends on the distance between the nodes. They investigated in detail the impact of the communication on the performance. In particular, the physical distance between remote nodes is taken into account. Mullet *et al.* studied in [67] Latency-Hiding, a new WS algorithm that hides the overhead caused by some operations, such as waiting for a request from a client or waiting for a response from a remote machine. The authors refer to this delay as *latency* which is slightly different that the more general concept we consider in our paper. Agrawal *et al.* proposed an analysis [3] showing the optimality for task graphs with bounded degrees and developed a library in Cilk++ called *Nabbit* for executing tasks with arbitrary dependencies, with reasonable block sizes.

2.3 Real Time Scheduling

The problem treated in this thesis (Chapter 6) consist of real-time scheduling, which is a particular type of scheduling dealing with tasks with a deadline. Real-time scheduling has to guarantee that all tasks are executed before their deadlines The literature on real time scheduling is very wide, we just specify the ref which have a direct link with our work.

To overcome the problem of the partitioned approach and increase the utilization rate of the system, recent works [7, 50, 61, 62, 23] have introduced the semi-partitioning scheduling in which most of tasks are assigned to particular processors as the partitioned scheduling, but the remaining tasks (unasigned tasks) are allowed to migrate between processors. In other words, each remaining task is splitted into a set of sub-tasks and each one of them is affected to a processor. This approach allows migration but reduces the number of migrant tasks compared to the global approach.

The Semi-partitioning algorithm EKG [8] cuts the set of processors into groups each one is composed of k processors and limits migration within the same group. In addition, a task can migrate between two processors only. Note that EKG allows to schedule optimally a set of periodic implicit tasks on m processors when k=m (EKG with one group). Since EKG allocates migrant and non-migrant tasks simultaneously, this can generate a great number of migrant tasks.

Kato et al. [51] have proposed the EDHS algorithm which improves the EKG algorithm. It proceeds into two separate steps to allocate the tasks: during the first one, the tasks are assigned according to a given partitioning algorithm in order to minimize the number of migrant tasks generated by the EKG algorithm. The second one consists in allocating migrant tasks on multiple processors according to a second algorithm.

2.4 Containers scheduling

The problem treated in this thesis (Chapter 7) consist of containers scheduling on cloud infrastructures in microservice-based application context [35]. Many studies investigated the placement of containers on machine for specific applications in order to meet different objectives. The authors in [71] propose a new decision making method for database container placement, they use Markov Decision Processes to

provide probability assurances for high QoS. The authors in [75] propose a dynamic container resource allocation mechanism (CRAM) which find the optimal allocation that meets varying QoS demands on heterogeneous cluster based on game theory.

More than that, there are several recent approaches that use different techniques for multi-objective optimization. As in [12] where they propose a cloud-based Container Management Service (CMS) which uses the constraint programming to schedule containers on the cloud. The CMS captures the heterogeneous requirements resource of containerised applications and examines the cluster state, then it uses the constraint programming to deploy these containerised applications. The work aim to increase the deployment density, the scalability, and resource efficiency. The work in [48] propose ECSched, an efficient container graph-based scheduler. ECSched schedule container on heterogeneous clusters and make high-quality and fast placement decisions, the authors map the problem to a graphic data structure and model it as minimum cost flow problem. The authors in [47] address the container deployment algorithm to satisfy multiple objectives on heterogeneous clusters. They formulate the problem as a vector bin backing problem with heterogeneous bins, and they propose a new container deployment algorithm to improve the tradeoff between load balancing and dependency awareness with multi-resource guarantees.

3 Analysis of work stealing with latency

Contents

We study in this chapter the impact of communication latency on the classical *Work Stealing* load balancing algorithm. Our work extends the reference model in which we introduce a latency parameter. By using a theoretical analysis and simulation, we study the overall impact of this latency on the Makespan (maximum completion time). We derive a new expression of the expected running time of a *bag of independent tasks* and *task graph with precedences* scheduled by Work Stealing. This expression enables us to predict under which conditions a given run will yield acceptable performance. For instance, we can easily calibrate the maximal number of processors to use for a given work/platform combination. All our results are validated through simulation on a wide range of parameters.

3.1 Introduction

The motivation of this work is to study how to extend the analysis of the Work Stealing (WS) algorithm in a distributed-memory context, where communications matter. WS is a classical on-line scheduling algorithm proposed for shared-memory multi-cores [11] whose principle is recalled in the next section. As it is common, we

target the minimization of the *Makespan*, defined as the maximum completion time of the parallel application. We present a theoretical analysis for an upper bound of the expected makespan and we run a complementary series of simulations in order to assess how this new bound behaves in practice depending on the value of the latency.

3.1.1 Motivation for studying WS with latency

Distributed-memory clusters consist in independent processing elements with private local memories linked by an interconnection network. In such architectures, communication issues are crucial, they highly influence the performances of the applications [46]. However, there are only few works dealing with optimized allocation strategies and the relationships with the allocation and scheduling process is most often ignored. In practice, the impact of scheduling may be huge since the whole execution can be highly affected by a large communication latency of interconnection networks [45]. Scheduling is the process which aims at determining where and when to execute the tasks of a target parallel application. The applications are represented as directed acyclic graphs where the vertices are the basic operations and the arcs are the dependencies between the tasks [27]. Scheduling is a crucial problem which has been extensively studied under many variants for the successive generations of parallel and distributed systems. The most commonly studied objective is to minimize the makespan (denoted by $C_{\max}$) and the underlying context is usually to consider centralized algorithms. This assumption is not always realistic, especially if we consider distributed memory allocations and an on-line setting.

WS is an efficient scheduling mechanism targeting medium range parallelism of multi-cores for fine-grain tasks. Its principle is briefly recalled as follows: each processor manages its own (local) list of tasks. When a processor becomes idle it randomly chooses another processor and steals some work (if possible). Its analysis is probabilistic since the algorithm itself is randomized. Today, the research on WS is driven by the question on how to extend the analysis for the characteristics of new computing platforms (distributed memory, large scale, heterogeneity). Notice that beside its theoretical interest, WS has been implemented successfully in several languages and parallel libraries including Cilk [37, 55], TBB (Threading Building Blocks) [72], the PGAS language [31, 64] and the KAAPI run-time system [41].

3.1.2 contributions

In the first work, we study how communication latency impacts work stealing. This chapter has three main contributions. First, we create a new realistic scheduling model for distributed-memory clusters of p identical processors including latency denoted by λ. Second, we provide an upper bound of the expected makespan. We consider the case of a bag of independent tasks and the case of a bag of tasks with precedence constraints. Our bounds are the sum of two terms. The first is the usual lower bound on the best possible load-balancing $\frac{\mathcal{W}}{p}$, where $\mathcal{W}$ and p are the total amount of work and the total number of processors respectively. The additional term depends on the dependence model. In the case of independent tasks, this additional term is $16\lambda \log_2(\frac{\mathcal{W}}{\lambda})$. In the case of DAG, this term is proportional to the critical path. The analyses are based on adequate potential functions. There are two reasons that distinguish this analysis in regard to the existing ones: finding the right function (the natural extension does not work since we now need to consider in transit work). Its property is that it should diminish after any steal related operation. We also consider large timesteps of duration equal to the communication latency.

3.2 Task Models and Work-Stealing Algorithm

We consider a discrete time model of a parallel platform with p identical processors. In this section, we introduce the two models of task dependence that we will study and how they affect the work stealing algorithm.

3.2.1 Dependence between tasks

The tasks can be independent or constrained by a directed acyclic graph (DAG) of precedence. The total amount of processing work to complete all tasks is denoted by $\mathcal{W}$. When the tasks have some dependencies, we denote by D the critical path of the DAG (that corresponds to its depth).

3.2.1.1 Independent Tasks

Our first case of interest is to consider unitary independent tasks. For this case, we denote by $w_i(t) \in \mathbb{N}$ the number of unit of work that processor i has at time t (for $i \in \{1 \dots p\}$). At unit of work corresponds to one unit of execution time. The total amount of work on all processors by $\mathcal{W}(t) = \sum_{i=1}^{p} w_i(t)$. At $t = 0$ all work is on one processor. The total amount of work at time 0 is $\mathcal{W} = w_1(0)$. In this model, all tasks are independent which means that when a processor steals from another processor,

it could steal as many tasks as there are available. We will assume for this model that that each successful steal divides the work in two equal parts.

3.2.1.2 DAG of precedence

We consider a model similar to [11, 79] where the workload is composed of $\mathcal{W}$ unitary tasks that have precedence constraints represented by a DAG. This DAG has a single source that represents the first task and that is originally located on a given processor. We consider that the scheduling is done as in [11]: each processor maintains a double-ended queue (called deque) of activated tasks. If a processor has one or more task in its deque, it executes the tasks at the head of its deque. This takes one unit of time. After completion, a task might activates 0, 1 or 2 tasks that are pushed at the end of the deque. The activation tree is a binary tree whose shape depends on the execution of the algorithm. It is a subset of the original DAG and has the same critical path. We define the height of node of this tree as follows. The height of the source as D (*i.e.*, the length of the critical path). The height of another task is equal to the length of its father minus one. We assume when a processor steals work from another processor, it steals the activated tasks with the largest height.

3.2.2 Work Stealing Algorithm

Work Stealing is a decentralized list scheduling algorithm where each processor maintains its own local queue or deque of tasks to execute. When a processor i has nothing to execute, it selects another processor j uniformly at random and sends a work request to it. When processor j receives this request, it answers by either sending some of its work or by a fail response.

We analyze one of the variants of the WS algorithm that has the following features:

- **Latency:** All communication takes a time $\lambda \in \mathbb{N}^+$ that we call the latency. Figure 3.1 presents an example of Work Stealing execution with latency λ, a work request that is sent at time $t - \lambda$ by a thief will be received at time t by the victim. The thief will then receive an answer at time $t + \lambda$. As we consider a discrete-time model, we say that a work request arrives at time t if it arrives between $t-1$ (not-included) and t. This means that at time t, this work request is treated. The number of incoming work requests at time t is denoted by $R(t) \in \{0, 1, \ldots, p-1\}$. It is equal to the number of processors sending a work request at time $t - \lambda$. When a processor i receives a work request from a thief j, it sends a part of its work to j. This communication takes again λ units

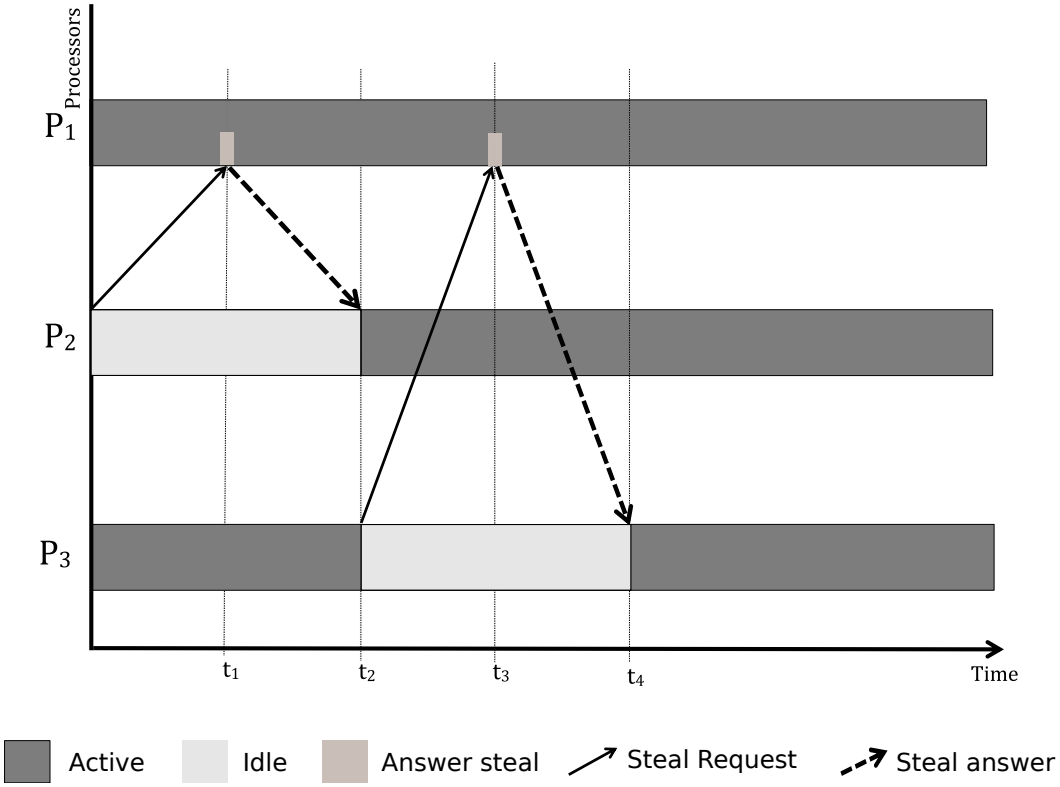


Fig. 3.1: Example of a work stealing execution

of time. The processor j receives the work at time $t + \lambda$. We denote by $s_i(t)$ the amount of work in transit from P_i at time t. At end of the communication s_i becomes 0 until a new work request arrives.

- **Single work transfer:** We assume that a processor can send some work to at most one processor at a time. While the processor sends work to a thief, it replies by a fail response to any other work request. Using this variant, the work request may fail in the following cases: when the victim does not have enough work or when it is already sending some work to another thief. Another case might happen when the victim receives more than one work request at the same time. It deals a random thief and send a negative response to the remaining thieves.

- **Steal Threshold:** The main goal of WS is to share work between processors in order to balance the load and the speed-up execution. In some cases however it might be beneficial to keep work local and answer negatively to some work requests. We assume that, in the case of independent asks, if the victim has less than λ units of work to execute, the work request fails (answering such a work request could increase the makespan as the time to answer a request is λ units of time). We do not make such an assumption for the case of DAG

because we assume that a processor does not know if the computation time of all the tasks that will be activated by its local tasks will be less than λ.

- **Work division:** This division of work depends on the model of dependence:

  - If all tasks are independent, we suppose that the victim sends to the thief half of its work:

    $$w_i(t) = w_i(t-1)/2$$

    and

    $$s_i(t) = w_i(t-1)/2$$

  - In the case of the DAG of precedence, a processor can only answer positively if it has two or more activated tasks in its deque. In this case, it sends its task that has the largest height.

3.3 Analysis of the Completion Time

This section contains the main result of the work which is a bound on the expected makespan. Before presenting the detailed analysis, we first describe its main steps before jumping into the technical analysis of the two models (independent tasks and DAG of precedence).

3.3.1 General Principle

We denote by $C_{\max}$ the makespan (*i.e.*, total execution time). In a WS algorithm, each processor either executes work or tries to steal work. As the round-trip-time of a communication is 2λ and the total amount of work is equal to $\mathcal{W}$ and the number of processors is p, we have:

$$pC_{\max} \leq \mathcal{W} + 2\lambda \#\textit{Work Requests}$$

where p is the number of processors. This leads to a straightforward bound of the Makespan:

$$\mathcal{C}_{max} \leq \frac{\mathcal{W}}{p} + 2\lambda \frac{\#WorkRequests}{p} \tag{3.1}$$

Note that the above inequality is not an equality because the execution might end while some processors are still waiting for work.

The key element of our analysis is to obtain a bound on the number of work requests. For that, we use what we call a potential function that represents how the tasks are (un)balanced. We bound the number of work requests by showing that each event involving a steal operation contributes to the decrease of the potential. Our approach is similar to the one of [79] but with one additional key difficulty: communications take λ time units. At first, it seems that longer communications should translate linearly into the time taken by work requests but this would neglect the fact that longer communication also reduce the number of work requests.

In order to analyze the impact of λ, we reconsider the time division as periods of duration λ. We analyze the system at each time step $k\lambda$ for $k \in \mathbb{N}$. By abuse of notation, we denote by $w_i(k)$ and $s_i(k)$ the quantities $w_i(k\lambda)$ and $s_i(k\lambda)$. We also define the total number of incoming work requests in the interval $(\lambda(k-1), \lambda k]$ by $r(k) = \sum_{j=1}^{\lambda} R((k-1)\lambda + j)$ and we denote by $q(r(k))$ the probability that a processor receives one or more requests in the interval $(\lambda(k-1), \lambda k]$ (this function will be computed in the next section). Note that since a steal requests takes at least λ units of time to be answered, we have $0 \leq r(k) \leq p$.

For both our models, the key steps of the analysis are as follows:

1. First, we will define a potential function $\phi(k)$ that is such that we can bound the expected decrease of the potential as a function of $r(k)$, the number of work requests in the time interval $(\lambda(k-1), \lambda k]$:

   $$\mathbb{E}\left[\phi(k+1) \mid r(k)\right] \leq h(r_k)\phi(k)$$

   This is done in Lemma 3.3.1 for the case of independent tasks and 3.3.2 for the case of DAGs.

2. Second, we will show that this bound implies that the number of work requests is upper bounded by :

   $$\mathbb{E}\left[R\right] \leq p\gamma \log_2 \phi(0)$$

   where $\gamma = \max r/(-p \log_2 h(r))$.
   This will be done in Lemma 3.3.3.

3. We will then obtain a bound on the Makespan by using Equation (3.1).

3.3.2 Expected Decrease of the Potential

Our results are based on the analysis of the decrease of the potential. The potential at time-step $k\lambda$ is denoted by $\phi(k)$. Its definition depends on the task dependence model. In the following, we analyze its decrease in two separate cases.

3.3.2.1 Independent Tasks

In the case of independent tasks, we define the potential as

$$\phi(k) = 1 + \frac{1}{\lambda^2} \sum_{i=1}^{p} \phi_i(k), \tag{3.2}$$

where

$$\phi_i(k) = w_i(k)^2 + 2s_i(k)^2 \quad \text{for i} \in \{1 \dots, p\}$$

This potential function always decreases. It is maximal when all the work is contained in one processor which is the potential function at time 0 and is equal to $\phi(0) = \mathcal{W}^2$. The schedule completes when the potential becomes 0.

The rational behind the definition of the potential function is as follows. First, the term 1 is only here to ensure that the potential is never smaller than one (which ensures that $\log \phi(k) \geq 0$). Up to the multiplicative factor $1/\lambda$, the rest of Equation (3.2) is composed in two terms: $\sum_{i=1}^{p} w_i(k)^2$ and $2\sum_{i=1}^{p} s_i(k)^2$. These terms serve to measure how unbalanced is the work: it is maximal when all jobs is located on a processor. As stated in Lemma 3.3.1, each event related to a steal request decrease the potential:

- When a steal arrives at a processor with w_i jobs, approximately $w_i/2$ jobs remain on this processor while $w_i/2$ jobs go into a term s_i. In the potential, this transforms w_i^2 into $(w_i/2)^2 + 2(w_i/2)^2 = 3w_i^2/4$;

- When some work arrive at a processor, a term s_i is transformed into a term w_j. Because of the factor 2, in the potential, this transforms $2s_i^2$ into s_i^2.

We denote by $\mathcal{F}_k$ all events up to the interval $((k-1)\lambda, k\lambda]$.

Lemma 3.3.1. *For the case of independent tasks, the expected ratio between $\phi(k+1)$ and $\phi(k)$ knowing $\mathcal{F}_k$ is bounded by:*

$$\mathbb{E}[\phi(k+1) \mid \mathcal{F}_k] \leq h(r(k))\phi(k),$$

where the potential is defined as in Equation (3.2) *and*

$$h(r) = \frac{3}{4} + \frac{1}{4}\left(\frac{p-2}{p-1}\right)^r \tag{3.3}$$

Proof. To analyze the decrease of the potential function, we distinguish different cases that corresponds to processors that are executing work, sending or answering work requests. We show that each case contributes to a diminution of the potential.

Between time $k\lambda$ and $(k+1)\lambda$, a processor does (at least) one the following four things. Note that these cases covers all possible behaviors of the processor.

Case 1 ($s_i > 0$) The processor started to send work to another (idle) processor j before time $k\lambda$. This means that processor j will receive $s_i(k)$ tasks at a time $t < (k+1)\lambda$. Note that by assumption, $s_i(k) \geq w_i(k)$ because processor i has executed some if its own work since it decided to send half of its work to j. There are now two cases:

- **Case 1a**. If no additional work requests has been received between t and $(k+1)\lambda$, it holds that

$$\begin{aligned} w_i(k+1) &\leq w_i(k) \\ w_j(k+1) &\leq s_i(k) \\ s_i(k+1) = s_j(k+1) &= 0. \end{aligned}$$

  This implies that the potential of i and j at time step $k+1$ satisfies:

$$\begin{aligned} \phi_i(k+1) + \phi_j(k+1) &= w_i(k+1)^2 + w_j(k+1)^2 \\ &\quad + 2(s_i(k+1)^2 + s_j(k+1)^2) \\ &\leq w_i(k)^2 + s_i(k)^2 \\ &= w_i(k)^2 + 2s_i(k)^2 - s_i(k)^2 \\ &\leq \frac{2}{3}\phi_i(k) \\ &= \frac{2}{3}(\phi_i(k) + \phi_j(k)). \end{aligned}$$

  The last inequality holds because $w_i(k)^2 + 2s_i^2(k) = \phi_i(k)$ and $s_i^2(k) = (s_i^2(k) + 2s_i^2(k))/3 \geq (w_i^2(k) + 2s_i^2(k))/3 = \phi_i(t+1)/3$, and the last equality holds because $\phi_j(k) = 0$.

- **Case 1b**. If one or more work request has been received between t and $(k+1)\lambda$ (by either processor i or j), then this processor will send some of its work of this processor. It should be clear that this will further decrease the

potential (see Case 2b below). This shows the inequality $\phi_i(k+1)+\phi_j(k+1) \leq \frac{2}{3}(\phi_i(k)+\phi_j(k))$ also holds in this case.

Note that if $w_i < \lambda$, processor i might become idle before $(k+1)\lambda$. In this case, it will send a work request. This will not modify the potential as the work request will be received after time $(k+1)\lambda$.

Case 2 ($s_i = 0$ and $w_i \geq 2\lambda$) The processor has work and it is available to respond to work requests. We distinguish two cases: (case 2a) if this processor receives one or more requests or (case 2b) if it does not receive any request.

- **Case 2a** – If processor i receives one or more work requests between $k\lambda$ and $(k+1)\lambda$, it will respond positively to one processor (say processor j) by sending it half of its work. All other work requests will fail. This implies that $w_i(k+1) \leq w_i(k)/2$ and $s_i(k+1) \leq w_i(k)/2$ and $w_j(k+1) = s_j(k+1) = 0$, which implies that

$$\mathbb{E}\left[\phi_i(k+1)\right] = w_i^2(k+1) + 2s_i^2(k+1) \leq \frac{3}{4}w_i^2(k) = \frac{3}{4}\phi_i(k) \tag{3.4}$$

- **Case 2b** – If processor i does not receive any work requests, it will only execute work, in which case $\phi_i(k+1) \leq \phi_i(k)$

Finally, the probability that processor i receives no work request between $k\lambda$ and $(k+1)\lambda$ given that r processors sent a work request is equal to $((p-2)/(p-1))^r$. This shows that Case 2a occurs with probability $1-((p-2)/(p-1))^r$ while Case 2b occurs with probability $((p-2)/(p-1))^r$. Hence

$$\mathbb{E}\left[\phi_i(k+1)\right] \leq \phi_i(k)\left[\left(\frac{p-2}{p-1}\right)^r + \frac{3}{4}\left(1-\left(\frac{p-2}{p-1}\right)^r\right)\right] = h(r)\phi_i(k).$$

Case 3 ($s_i = 0$ and $\lambda \leq w_i < 2\lambda$) The processor has less than 2λ units of work and therefore may or may not be able to answer work requests depending if they arrive before its remaining work is less than λ units of work. If a work request is received then we fall back to Case 2a. Otherwise, the processor only executes work and :

$$\begin{aligned}\phi_i(k+1) &= (\max(0, w_i(k) - \lambda))^2 \\ &\leq \frac{1}{2}w_i(k)^2 \\ &= \frac{1}{2}\phi_i(k).\end{aligned}$$

Case 4 ($s_i = 0$ and $w_i < \lambda$) If processor i is idle or became idle between $k\lambda$ and $(k+1)\lambda$, there are two sub-cases. The first one is if this processor receives some work between $k\lambda$ and $(k+1)\lambda$. In this case this processor is the processor j of the **Case 1** above and its contribution to the potential has already been taken into account. The second one is if this processor does not receive work during $k\lambda$ and $(k+1)\lambda$ in which case its potential is $\phi_i(k+1) = 0$.

Note that in addition to all this decrease, at least one processor executed λ units of work during $[k\lambda, (k+1)\lambda)$ (otherwise there would be nothing to compute and the schedule would be finished). This contributes to the decrease of (at least) $\lambda^2/3$ to one of the $\phi_i(k)$.

Using the variation of each of these scenarios we find that the expected potential time $k+1$ is bounded by:

$$\begin{aligned}\mathbb{E}[\phi(k+1) \mid \mathcal{F}_k] &\leq 1 + \frac{1}{\lambda^2}\left(-\frac{\lambda^2}{3} + \sum_{i \in \text{ Case 1}} \frac{2}{3}\phi_i(k) + \sum_{i \in \text{ Case 2}} h(r(k))\phi_i(k)\right) \\ &\leq \max\left(\frac{2}{3}, h(r(k)), \frac{1}{2}, 0\right)\phi(k) \\ &= h(r(k))\phi(k)\end{aligned}$$

where the last equality holds because $h(r(k)) \geq 3/4 \geq 2/3$. □

3.3.2.2 DAG of precedence

In the case of DAG, the potential will depend on the maximal height of the tasks that a processor has in its lists. Denoting by $h_i(k)$ the maximal height of the tasks that processor i has in its deque, we define a quantity $w_i(k)$, that we call the potential work of processor i, as:

$$w_i(k) = \begin{cases} (2\sqrt{2})^{h_i(k)} & \text{if the processor } i \text{ has two or more tasks in its deque} \\ \frac{1}{2}(2\sqrt{2})^{h_i(k)} & \text{if the processor } i \text{ has only one tasks in its deque} \\ 0 & \text{if the processor } i \text{ does not have any tasks} \end{cases}$$

For the tasks in transit, we define the potential work in transit $s_i(k) = \frac{1}{2}(2\sqrt{2})^h$, where h is the height of the task in transit.

The potential is then defined similarly as in Equation (3.2):

$$\phi(k) = 1 + \frac{1}{\lambda^2}\sum_i \phi_i(k) \qquad \text{where } \phi_i(k) = w_i^2(k) + 2s_i^2(k). \tag{3.5}$$

We are now ready to prove the following lemma, that is an analogue of Lemma 3.3.1 for the case of dependent tasks.

Lemma 3.3.2. *For the case of DAG, the expected ratio between $\phi(k+1)$ and $\phi(k)$ knowing $\mathcal{F}_k$ is bounded by:*

$$\mathbb{E}[\phi(k+1) \mid \mathcal{F}_k] \leq h(r(k))\phi(k),$$

where the potential is defined as in Equation (3.5) *and $h(r)$ is as in Lemma 3.3.1,* i.e.*, $h(r) = \frac{3}{4} + \frac{1}{4}\left(\frac{p-2}{p-1}\right)^r$.*

Proof. Similarly to the proof of Lemma 3.3.1, we study the expected decrease of the potential by distinguishing three cases: the case $s_i > 0$ (work arriving at a thief) and the case where a processor is or becomes idle correspond to cases 1 and 4 of Lemma 3.3.1 and can be analysis exactly as what was done since Cases 1 and 4 of the proof of Lemma 3.3.1 do not depend on the task dependence model. Moreover, the distinction between Case 2 and Case 3 that was done for independent tasks is not necessary for DAGs as there is no steal threshold for DAG. Last, the analysis of Case 2b of Lemma 3.3.1 is similar: if a processor does not receive any work request, the potential does not grow.

Hence, in the reminder of the proof, we focus on the only interesting case which what happens when a processor receives one or more work request (Case 2b of the proof of Lemma 3.3.1. We distinguish two cases depending on how many tasks are in the deque of processor i when it receives a work request.

1. If processor i has only one task (say of height h), then it cannot send work. In this case, it will complete its tasks that will activate at most 2 tasks of height $h-1$. In such a case, the potential work of processor i will go from $w_i(k) = \frac{1}{2}(2\sqrt{2})^h$ to at most $w_i(k+1) = (2\sqrt{2})^{h-1}$. This shows that

$$\begin{aligned}\phi_i(k+1) = w_i^2(k+1) &\leq (2\sqrt{2})^{2h-2} \\ &= (2\sqrt{2})^{2h}/8 \\ &= w_i(k)^2/2 \leq \frac{1}{2}\phi(k).\end{aligned}$$

2. If processor i has two or more tasks, then by construction, it can have at most two tasks of maximal height (say h). In this case, the processor will send one task (of height h) to the thief, and will at the mean time execute the other task. In this case, the maximal height of the task of its deque will be $h-1$ and the task

sent will be of height h. This implies that $w_i(k+1) \leq (2\sqrt{2})^{2h-1} \leq w_i(k)/(2\sqrt{2})$ and $s_i(k+1) \leq \frac{1}{2}(2\sqrt{2})^h \leq w_i(k)/2$. This shows that:

$$\begin{aligned}\phi_i(k+1) &= w_i^2(k+1) + 2s_i^2(k+1) \\ &\leq w_i^2(k)/8 + 2w_i^2(k)/4 \\ &= \frac{5}{8}\phi_i(k).\end{aligned}$$

As both $1/2$ and $5/8$ are strictly smaller than the $3/4$ of Equation (3.4), the rest of the proof of Lemma 3.3.1 can be applied *mutatis mutandis* to prove Lemma 3.3.2. □

3.3.3 Bound on the Number of Work Requests

We are now ready to use Lemma 3.3.1 and 3.3.2 above to obtain a bound on the number of work requests. Let us define the constant γ as follows:

$$\gamma \stackrel{\text{def}}{=} \max_{1 \leq r \leq p} \frac{r}{-p \log_2(h(r))}$$

where h is defined in Equation (3.3).

Lemma 3.3.3. *Let $\phi(0)$ denote the potential at time 0 and let τ be the first time step at which the potential reaches 1. Then, for both the independent tasks and the DAG models, the number of incoming work requests until τ, $R = \sum_{k=0}^{\tau-1} r(k)$, satisfies:*

$$\begin{aligned}&(i) \qquad \mathbb{E}[R] \leq p\gamma \log_2 \phi(0) \\ &(ii) \qquad \mathbb{P}[R \geq p\gamma(\log_2 \phi(0) + x)] \leq 2^{-x}.\end{aligned}$$

Proof. By definition of γ, for a number of work requests $r \in \{0, \ldots, p-1\}$, we have $\log_2(h(r)) \leq \frac{r}{-p\gamma}$ which implies that $h(r) \leq 2^{-r/(p\gamma)}$.

Let $X_k = \phi(k) \prod_{i=0}^{k-1} 2^{r(i)/(p\gamma)}$. By Lemma 3.3.1 and 3.3.2, this shows that

$$\begin{aligned}\mathbb{E}[X_{k+1} \mid \mathcal{F}_k] &= \mathbb{E}[\phi(k+1) \mid \mathcal{F}_k] \prod_{i=0}^{k} 2^{r(i)/(p\gamma)} \\ &\leq \phi(k) 2^{-r(k)/(p\gamma)} \prod_{i=0}^{k} 2^{r(i)/(p\gamma)} = X_k\end{aligned}$$

This shows that $(X_k)_k$ is a supermartingale for the filtration $\mathcal{F}$. As τ is a stopping time for the filtration $\mathcal{F}$, Doob's optional stopping theorem (see *e.g.*, [34, Theorem 4.1]) implies that

$$\mathbb{E}[X_\tau] \leq \mathbb{E}[X_0]. \tag{3.6}$$

By definition of X, we have $X_0 = \phi(0)$ and $X_\tau = \phi(\tau)2^{R/(p\gamma)}$. As $\phi(\tau) = 1$, this implies that

$$\mathbb{E}\left[2^{R/(p\gamma)}\right] = \mathbb{E}\left[\phi(\tau)2^{R/(p\gamma)}\right] \leq \phi(0), \tag{3.7}$$

By Jensen's inequality (see *e.g.*, [34, Equation (3.2)]), we have $\mathbb{E}[R/(p\gamma)] \leq \log_2\left(\mathbb{E}[2^{R/(p\gamma)}]\right)$. This shows that

$$\mathbb{E}\left[R\right] \leq p\gamma \log_2 \phi(0)$$

Moreover, by Markov's inequality, Equation (3.7) implies that for all $a > 0$:

$$\mathbb{P}\left[2^{R/(p\gamma)} \geq a\right] \leq \frac{\phi(0)}{a}.$$

By using $a = \phi(0)2^x$, this implies that $\mathbb{P}[R \geq p\gamma(\log_2 \phi(0) + x)] \leq 2^{-x}$ □

In the next lemma, we show that the constant γ can be bounded by a constant that is independent of the number of processor p.

Lemma 3.3.4. *The constant γ of Lemma 3.3.3 is such that $\gamma < 4.03$.*

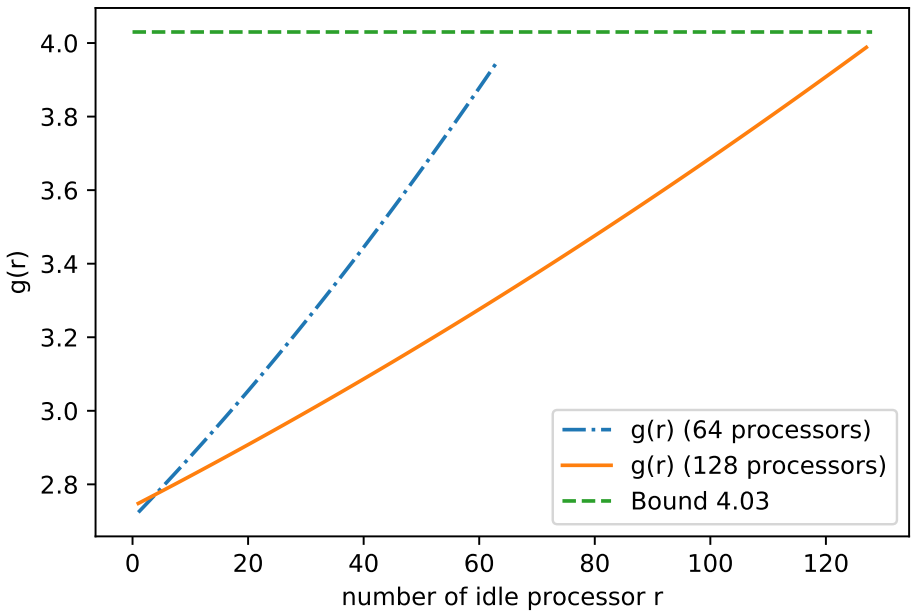


Fig. 3.2: Function $g(r)$ as a function of the number of idle processor r for $p = \{64, 128\}$ processors. We observe that g is increasing and upper bounded by 4.03.

Proof. Let define the function g as

$$g(r) = \frac{r}{-p\log_2\left(\frac{3}{4} + \frac{1}{4}\left(\frac{p-2}{p-1}\right)^r\right)} \tag{3.8}$$

By definition, the constant γ is the maximum of g:

$$\gamma = \max_{1 \leq r \leq p-1} g(r)$$

The function g is displayed in Figure 3.2 for the case of 64 processors and 128 processors. As we observe on this curve, g is increasing and is therefore bounded by $g(p-1) < 4.03$. This is what we prove in the remainder of this proof. Let $x = (p-2)/(p-1)$ and $y = x^r$, we have $y \in (0,1)$. We have

$$\frac{1}{g(r)} = -p\frac{\log_2(3/4 + y/4)}{\log_x(y)} = -\frac{p \log x}{\log 2} f(y),$$

where $f(y) = \log(3/4 + y/4)/\log(y)$. The first derivative of f is

$$f'(y) = \frac{y \log y - (3+y)\log(3/4 + y/4)}{y(3+y)(\log y)^2}$$

The first derivative of the numerator of $f'(y)$ is $\log y - \log(3/4 + y/4)$ which is negative for $y < 1$. Thus it implies that $f'(y)$ is decreasing. As $f'(1) = 1$, this shows that $f'(y) \geq 0$ and therefore that f is increasing. This implies that $g(r)$ is increasing (because y is decreasing in r and $g(r) = \alpha/f(y)$ for $\alpha = -\log 2/(p \log x) > 0$).

As g is increasing, $\gamma = g(p-1)$. Now, for all $p \geq 2$:

$$\begin{aligned}
\left(\frac{p-2}{p-1}\right)^{p-1} &= \left(1 - \frac{1}{p-1}\right)^{p-1} \\
&= \exp\left((p-1)\ln(1 - \frac{1}{p-1})\right) \\
&\leq \exp\left(-(p-1)\frac{1}{p-1}\right) = \frac{1}{e}
\end{aligned}$$

This shows that

$$\begin{aligned}
\gamma &= g(p-1) \\
&\leq \frac{1}{2 - \log_2\left(3 + \frac{1}{e}\right)} < 4.03
\end{aligned}$$

□

3.3.4 Analysis of the Makespan

We are now ready to prove the bound on the total completion time $C_{\max}$ that is summarized by the following theorem:

Theorem 3.3.5. *Let C_{max} be the Makespan of $\mathcal{W}$ unit independent tasks scheduled by WS with latency λ. Then,*

$$(i) \qquad \mathbb{E}\left[C_{max}\right] \leq \frac{\mathcal{W}}{p} + 4\lambda\gamma \log_2 \frac{\mathcal{W}}{\lambda}$$
$$(ii) \qquad \mathbb{P}\left[C_{max} \geq \frac{\mathcal{W}}{p} + 4\lambda\gamma \log_2 \frac{\mathcal{W}}{\lambda} + x\right] \leq 2^{-x/(2\lambda\gamma)}.$$

Let C_{max} be the Makespan of $\mathcal{W}$ unit tasks with a DAG of precedence that has a critical path of D scheduled by WS with latency λ. Then

$$(i) \qquad \mathbb{E}\left[C_{max}\right] \leq \frac{\mathcal{W}}{p} + 6\lambda\gamma D$$
$$(ii) \qquad \mathbb{P}\left[C_{max} \geq \frac{\mathcal{W}}{p} + 6\lambda\gamma D + x\right] \leq 2^{-x/(2\lambda\gamma)},$$

The constant γ is the same as in Lemma 3.3.3. In particular $\gamma < 4.03$.

Proof. Both cases will be proved separately.

Independent tasks – By Lemma 3.3.3, the number of incoming work requests until τ is bounded by $p\gamma \log_2 \phi(0)$, with $\phi(0) = 1 + \mathcal{W}^2/\lambda^2$. Moreover by definition, the schedule is finished at time τ. Thus by Equation (3.1) we have

$$\begin{aligned}\mathbb{E}\left[\mathcal{C}_{max}\right] &\leq \frac{\mathcal{W}}{p} + 2\lambda\gamma \log_2(1 + \frac{\mathcal{W}^2}{\lambda^2}) \\ &\leq \frac{\mathcal{W}}{p} + 4\lambda\gamma \log_2\left(\frac{\mathcal{W}}{\lambda}\right) + 4\lambda\gamma,\end{aligned}$$

where we used that $\log_2(1+x) \leq 1 + \log_2(x)$ for $x \geq 1$.

By the same way, we use Lemma 3.3.3 (ii) and equation 3.1 we obtain:

$$\begin{aligned}\mathbb{P}\left[C_{max} \geq \frac{\mathcal{W}}{p} + 4\lambda\gamma \log_2\left(\frac{\mathcal{W}}{\lambda}\right) + 4\lambda\gamma + x\right] &\leq \mathbb{P}\left[2\lambda\frac{R}{p} \geq 2\lambda\gamma \log_2 \phi(0) + x\right] \\ &= \mathbb{P}\left[R \geq p\gamma \log_2\left(\frac{\mathcal{W}}{\lambda}\right) + p\gamma\frac{x}{2\lambda\gamma}\right] \\ &\leq 2^{-x/(2\lambda\gamma)}\end{aligned}$$

DAG of precedence. The case of DAG is similar, the only difference being the expression of the potential at time 0. In the case of a DAG, the potential at time 0 is equal to $1 + \frac{1}{4}(2\sqrt{2})^{2D} = 1 + \frac{1}{4}8^D \leq 8^D$ where D is the critical path. This shows that $\log_2 \phi(0) \leq D \log_2 8 = 3D$ which implies that

$$\mathbb{E}\left[\mathcal{C}_{max}\right] \leq \frac{\mathcal{W}}{p} + 6\lambda\gamma D.$$

□

The bounds that we obtained in Theorem 3.3.5 are composed of a term due to the computation of tasks ($\mathcal{W}/p$) and an overhead term related to the depth. In case of independent tasks the depth is $\log(W)$ but we obtain a slightly better bound due to tasks not being divided below λ. We actually think bounds on the DAG could be refined when considering non-unit tasks.

3.4 Conclusion

We presented in this paper a new analysis of Work Stealing algorithm where each communication has a latency of λ. Our main result was to show that the expected Makespan of a load of W on a cluster of p processors is bounded by $W/p + 16.12\lambda \log_2(W/(\lambda))$ for the case of independent tasks and by $W/p + 24.18\lambda D$ in the case of a DAG of depth D.

Our analysis makes use of a potential functions whose expected decrease per unit time can be bounded as a function of the number of work requests. We then use this to derive a theoretical upper bound on the expected makespan. We also extend this analysis one step further, by providing a bound on the probability to exceed the bound of the makespan.

This work will certainly be the basis of incoming studies on more complex hierarchical topologies where communications matter. As such, it is important as it allows a full understanding of the behavior of various Work Stealing implementations in a base setting.

Work Stealing Simulator

4

Contents

4.1 Introduction

4.1.1 Context

The analysis of the classical *Work Stealing* algorithm is a difficult combinatorial problem [20]. It becomes even more difficult on more complex environments. For example, the analysis is much more difficult in the case on distributed memory than on shared memory since communication matter.

Our first analysis in Chapter 3 with potential function was not intuitive. The main idea (Section 3.3.1) is to see what happens at each step and its impact on the potential function (how much it decreases). One of the most difficult challenge was to find the adequate potential function which decreases when a steal operation occurs. Then, the proof used the worst case scenario at each step to get the worst decrease of the potential function. In case of one cluster with m processors, the worst decrease happens when all the processors act as thieves except one ($m-1$ steal requests).

We are interested in the analysis of *Work Stealing* algorithm on more complex environments including non homogeneous ones. In particular, we are interested in platforms with multiple clusters where each cluster contains a set of shared memory processors. The clusters are linked via a not uniform interconnection network. As processors in the same cluster communicate through a shared memory, communications cost are almost negligible. The processors in different clusters communicate through the interconnection network and thus, communications are explicit (latency or bandwidth) and costly.

4.1.2 Why using simulator?

The heterogeneity of communications and the mechanism of *Work Stealing* generate an interesting combinatorial problem, which is more difficult than the initial case in which we use on one cluster with homogeneous communications. Moreover, a mathematical analysis using the potential functions is not effective, because it is very difficult to find an adequate potential function. Moreover, the worst case scenario is too far from the reality compared to the model of *Work Stealing* on one cluster. The worst case scenario in multiple clusters is not just when all the processors act as thieves except one, but also when all the processors steal outside their own clusters. This worst case scenario is one of the most difficult barrier to analyze the model of the *Work Stealing* algorithm on multiple clusters.

Therefore, we have to rely on simulations to observe what happens when we use the *Work Stealing* algorithm on multiple clusters platforms. We performed simulation to understand how the algorithm behaves when the communication time increases, and to get an idea about the average completion time according to different parameters (communication time, number of processors, etc...). And to compare different strategies that take communication time and cluster's topology into account. At the same time, we used the simulator to validate the theoretical analysis in the basic case of one cluster, and show how much the Makespan is far from the experienced Makespan.

There exist several simulators on parallel and distributed computing. Many of them are developed for a specific research projects by researchers and are undocumented, and/or no longer maintained. However, there exist several High quality simulators like SimGrid [22] that include many features and allow to consider complex situations like congestion, cache effects for particular architectures. However, such simulators are usually very computationally expensive, and they require a long execution time. Our purpose is less ambitious since we target simple processing units to observe a single aspect of execution process, which is the work steeling algorithm on platforms with different topologies. For this work, we developed a specific lightweight **PYTHON** simulator. Our simulator is quite flexible and easy to use and update. Moreover, it allows getting more insight on the result. Thus, we are interested in using our own representations for interpreting the simulation results.

4.1.3 Objective

The objective of our simulator consists in running different models of the *Work Stealing* algorithm. It executes an application on a platform, an application consists of a list of tasks with or without dependencies, and the platform consists of multiple processors linked by a specific topology. The simulator allows to execute a scenario with a specific task on a specific platform. It is designed to be sufficiently flexible to meet the different needs to analyze the *Work Stealing* algorithm and to compare different victim selection strategies. It offers various types of applications and various topologies. Moreover, its architecture facilitates the development of other types of applications and other topologies for interconnecting the processors. Even more than that, the simulator is fast. It also shows in details the results of each simulation. These results could be numerical (execution time, number of steal requests, etc...) or graphical (Gantt chart, real time execution etc...).

In this chapter, we give in Section 4.2 an overview of the different variants of the *Work Stealing* algorithm. Then, we present in Section 4.3 the architecture of our light *Work Stealing* simulator. In Section 4.4 we use our simulator to assess the validity of our analysis presented in Section 3. Then, we show the latency intervals exhibiting an acceptable Makespan on a single cluster, and we conclude the section by studying the impact of simultaneous responses.

4.2 Variants of the Work Stealing algorithm

The *Work Stealing* algorithm schedules an application (set of tasks) in a distributed platform composed of p processors linked by a specific topology. Many algorithms and implementation variants of the Work Stealing algorithm exist in the literature. In

particular, we present the different task models of the scheduled application. Then, we describe different types of platform topologies possible and how they impact the victim selection. We conclude by describing different policies for steal answers.

4.2.1 Application Task model

The type of scheduled application is an important issue. As we see on Chapter 3.3.4, the bound of Makespan depends on the type of the scheduled application. The application defines the characteristics of the tasks, the dependencies between them and how the work could be divided during a steal operation.

In the literature, many researchers are interested in analyzing *Work Stealing* algorithms using different task models. The most used task models can be classified as follows:

4.2.1.1 Divisible load

The divisible load represents applications with independent unit tasks. It has been introduced in [18] and experimented by [33]. It considers the work as a divisible load where the initial amount is represented by a single big task. Then, during execution, each task can be divided on request into two subtasks containing each a part of its work. For instance when a steal request occurs in a busy processor it sends a positive response in a form of a new task containing a part of the local work and updates accordingly its current content. Many theoretical studies on *Work Stealing* use this divisible load model since it simplifies the theoretical analysis [79].

4.2.1.2 DAG of tasks

This type represents an application as a set of tasks constrained by a directed acyclic graph (DAG) of precedence [27]. This DAG has a single source that represents the first active task. The processing time of a task can be unitary as in [11] or depend on the size of the task [79]. The scheduling of such type is done in [11], each processor maintains a double-ended queue (called deque) of activated tasks. If a processor has one or more task in its deque, it executes the tasks at the head of its deque. After completion, a task might activate other tasks that are pushed to the end of the deque. A task is active in the DAG only when all its precedents have been executed. The activation tree could be a binary tree or a fork-join whose shape depends on the execution of the algorithm. It is a subset of the original DAG and has the same critical path. We define the height of nodes of this tree as follows. The height of the source as D (i.e., the length of the critical path). The height of another task is equal

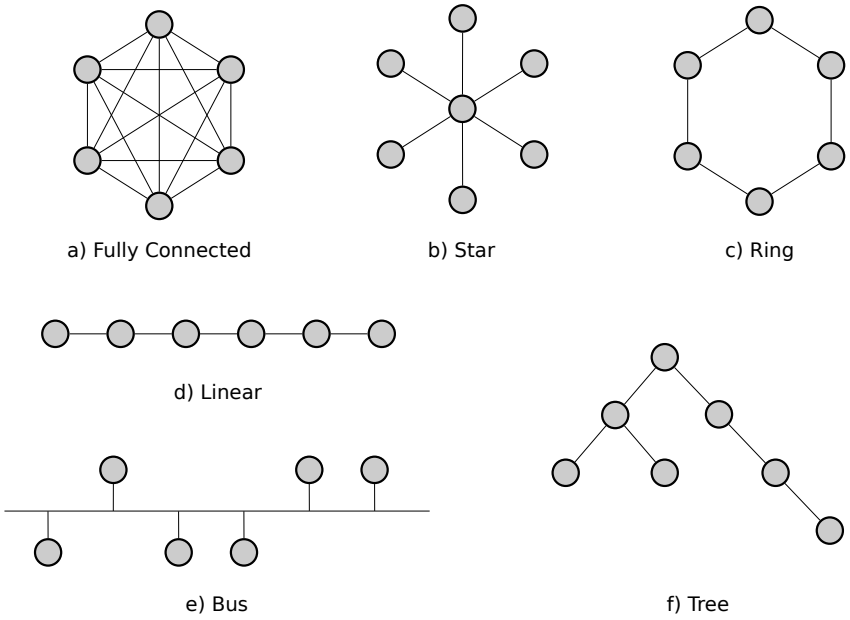


Fig. 4.1: Multi-clusters topologies

to the length of its father minus one. We assume when a processor steals work from another processor, it steals the activated tasks with the largest height.

4.2.1.3 Adaptive tasks

The adaptive tasks represent a dynamic application that reacts specifically to the steal requests. At the beginning, all the workload is stored as one big task which is located on a given processor. Then during a steal operation, the processor shares a part of its task and creates the merge task that brings together the result of the two parts at the end. In general, the processing time of an adaptive task depends on its size and the algorithm used. The processing time of the merge task depends on the size of the tasks that proceeded it and the algorithm used to merge the results. The adaptive tasks have been studied in [66, 28] and introduced in [73] to solve the prefix problem.

4.2.2 Platform topologies

The platform topology defines the location of the processors in the platform and characterizes the communication times between them (latency or bandwidth). There exist many topologies in the literature that can be classified as follows:

- **One cluster :** The same topology used in Chapter 3. The processors are fully connected in a cluster. The communications between them are homogeneous and can take place simultaneously with no extra overhead, and the communi-

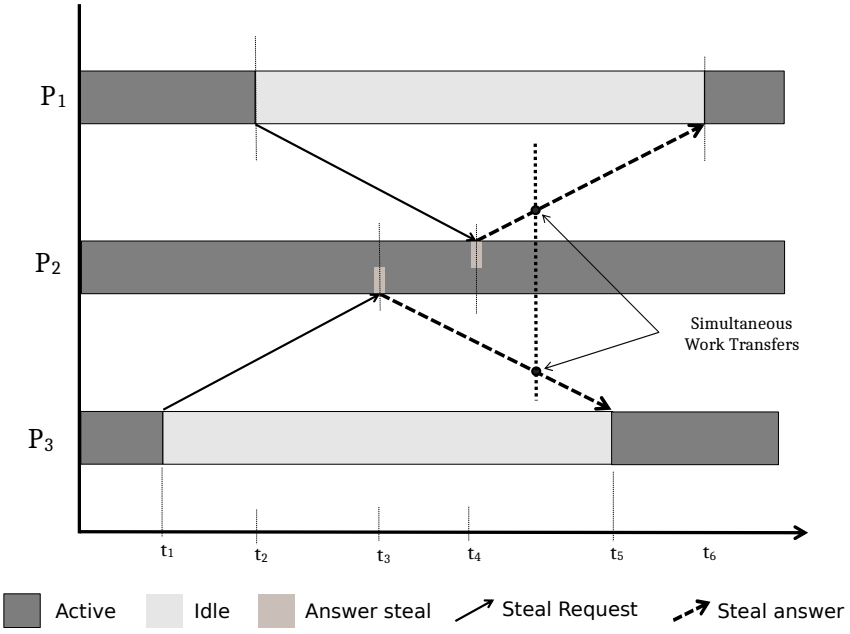


Fig. 4.2: Distribution of work in the case of simultaneous responses

cation costs are dominated by the latency. Thus, this communication can be modeled by a constant delay (denoted by λ in Chapter 3). We can model the shared memory processors by a single cluster topology if we consider that the communication time takes 1 time step.

- **Two clusters :** the processors are divided into two clusters. The processors in the same cluster communicate via shared memory. We consider that this communication takes 1 time step. The clusters are connected via an interconnect network that performs the communications between processors in different clusters. Since the communication cost between cluster is much larger than the communication inside the clusters, the communication between the processors is heterogeneous and creates victim selection issue (explained in Section 4.2.3).
- **Multiclusters :** the processors are divided into several clusters that are linked via a network in different topologies as shown in Fig 4.1. In these topologies, the communication between processors depends on their location and also on the location of their clusters on the topology.

4.2.3 Victim selection

The *Work Stealing* algorithm on complex topology with the heterogeneity of communication creates new questions about the victim selection strategy. Sometimes, the victim selection should take into account the characteristic of the topology (distance between processors, the communication time, etc...). Thus, the victim selection

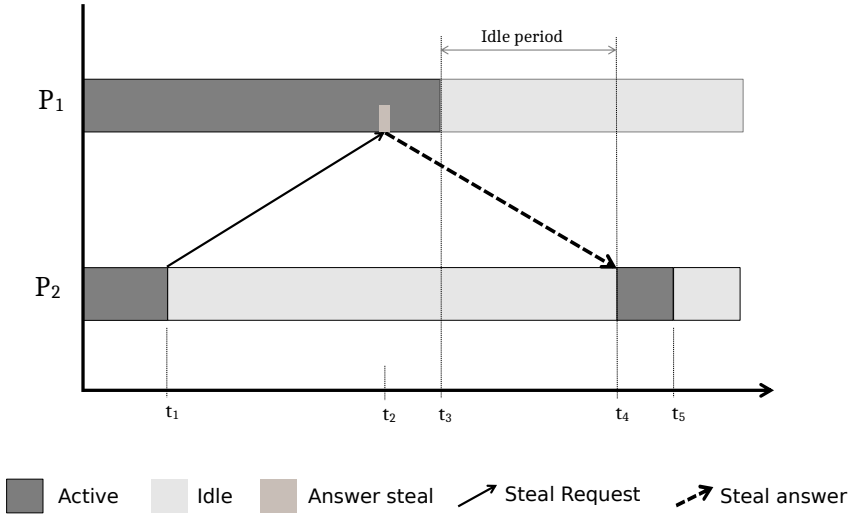


Fig. 4.3: Example of creating artificial idle times

strategies is an important question especially on structured topology. In the next chapter, we study in detail the victim selection questions.

4.2.4 Steal answer policies

4.2.4.1 Simultaneous responses

There exist in the literature two main variants for handling steal responses, namely, the single and simultaneous responses. We consider here both techniques as follows:

- **Single work transfer (SWT)** is a variant where the processor can send some work to at most one processor at a time. The processor sends work to a thief and it replies by a fail response to any other steal requests. Using this variant the *steal request* may fail in the two following cases: when the victim does not have enough work or when it is already sending some work to another thief.

- **Multiple work transfers (MWT)** Each processor can respond and send work to several processors simultaneously. The received requests are handled sequentially. In the classical model, the processor always answers by sending half of its work. In case of simultaneous requests it arranges them in a series and answers in the same way. Fig 4.2 gives an example of such simultaneous work transfers. In this figure $W_i(t)$ denotes the work on P_i at time t.

4.2.4.2 Steal Threshold

The main goal of *Work Stealing* is to share work between processors to balance the load and speed-up the execution. In some cases however it might be beneficial to keep work local and answer negatively to some steal requests.

Fig 4.3 shows an example of this case on two processors. At time t_1 processor P_2 sends a steal request to P_1. At t_2 P_1 receives this request and answers by sending half of its local work, which is less than the communication duration. At t_3 P_1 finishes its remaining work and becomes idle. Then, both processors are idle in the time period between t_3 and t_4. This clearly is a waste of resources since the whole platform is idle while there is yet some work to execute. Moreover, such a behavior can be chained several times. This effect is not purely theoretical as it has been observed during our initial experiments.

It is possible to prevent this from happening by adding a threshold on steal operations. We introduce a *steal threshold* which prohibits steals if the remaining local work becomes too small.

4.3 Simulator Architecture

We present in this section the global architecture of our simulator. First, we describe the basic mechanism of our simulator. Then, we explain how the simulator manages different variants of *Work Stealing* (described in Section 4.2) using different independent engines.

Basic Mechanism. During an execution of *Work Stealing*, the processors switch between different states over time. For example, a processor is active when it executes work. Once it finishes its work, it becomes idle. Then, if its tasks queue is not empty, it pops a task and it becomes active again, otherwise, it becomes a thief by sending a steal request to the other processors. We define an **event** as the time when a processor changes its state. This implies that the simulator has to simulate the events time instead of all the running times continuously. When an event occurs, the simulator uses the model instructions to execute it. For example, when a processor becomes a thief, the simulator chooses the victim using the strategy defined by the considered model and sends a steal request to the selected victim.

The execution of a simulation returns different statistical results (simulation time, number of steal requests, etc...). Other type of results are possible, for example, we

can generate some logs to show the Gantt execution chart. We can also display the DAG execution which delivers the execution in real time.

In our work, the simulator is used to experimentally analyze different variants of *Work Stealing*. Thus, it should handle the different variants described in Section-4.3. Moreover, the simulator needs to manage the different types of application, the different topologies and all other variants.

For all these reasons, our simulator is designed to be sufficiently flexible in order to simulate different *Work Stealing* models. Its flexibility aims to allow us to experiment with different *Work Stealing* algorithms, different topologies, different steal strategies and different types of application. The simulator should also generate a sufficient amount of logs for a detailed analysis each tested scenario.

We decompose the simulator into several independent engines. Each engine develops a part of the simulator and offers an operating interface which presents the main provided functionalities. The engines interact between them through these operating interfaces.

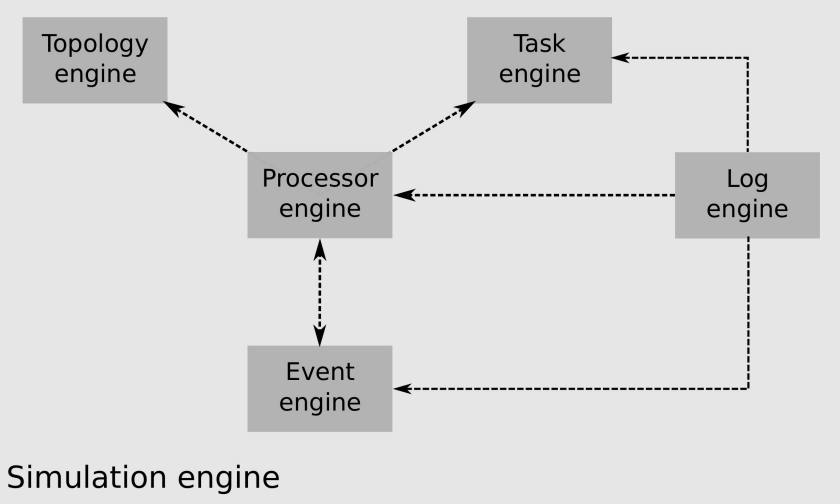


Fig. 4.4: The different engines of our simulator

The overall architecture of our simulator is composed of six main engines, as seen on Fig 4.4. The event engine is the core of our simulator, it manages the processors events during the time to run the simulation of a scenario. The events are executed through the processor engine which provides different functionalities to perform the *Work Stealing* algorithm. The processor engine uses the task engine to manage the execution of tasks and uses the topology engine to manage the interactions between the processors. During the execution of a simulation, the log engine keeps track of different information and generates different logs. The rest of this section details the role of each engine and explains the interactions between them.

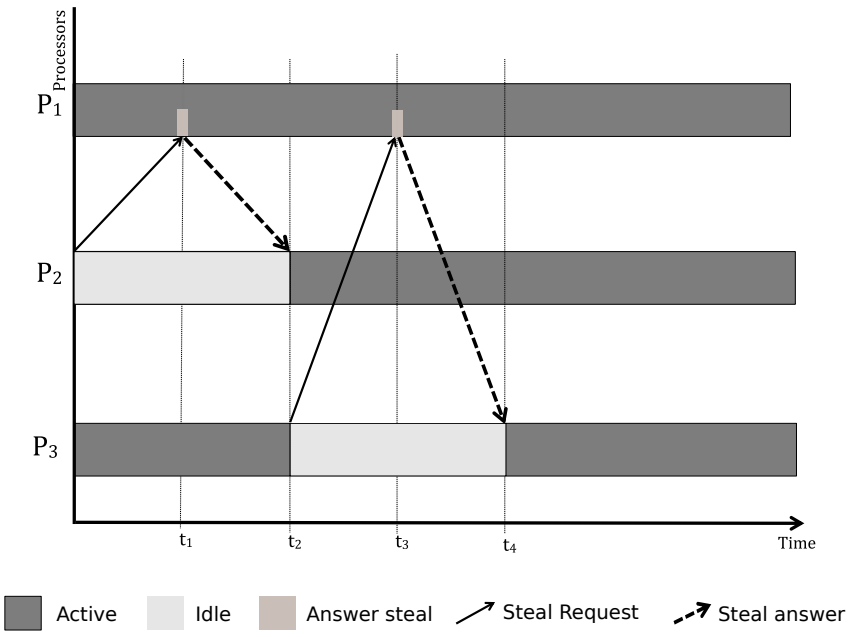


Fig. 4.5: Example of a *Work Stealing* execution

4.3.1 Event engine

The event engine represents the kernel of the simulator. In this section, we first explain the global idea to run the simulation of a scenario. Then we define the components used by the event engine to simulate an execution of an application defined by task engine on a platform defined by topology engine.

In the *Work Stealing* algorithm, a processor switches between different possible states. Fig 4.5 presents an example of *Work Stealing* execution, each processor interacts when it becomes idle (P_3 at t_2), when it receives a steal request (P_1 at t_3) or when it receives a steal answer (P_3 at t_4).

The global idea of our simulator consists in simulating a set of discrete **events** through time instead of simulating the whole execution time, where an *event* stands for changing the state of a processor at a specific time. For that, the event engine lists the available events on a heap and executes them sequentially according to their time. The execution of events follows different steps to update the system and creates new events in the global heap, these events will be executed following the same mechanism.

We define an event by its time, its related processors and its type. Based on the different states of a processor. We consider the three types of possible events :

- **Idle event**: a processor finishes its running task. When a processor has an idle event, it means that it is executing a task. Thus, the time of this event is defined by the execution time of the related task.
- **Steal request event**: a processor receives a steal request.
- **Steal answer event**: a processor receives the answer after a steal request.

The event engine offers two functions to manage the event heap, *next_event()* which pops the nearest event from the global events heap and *add_event()* which adds an event to the global event heap. The event engine controls also the global simulation time which starts at 0. All tasks type described below start with one big task. Thus, at the beginning of the simulation, the first processor executes the first task of the application, then it starts the simulation with the related *idle event*. All other processors start with an *idle event* that occurs at the beginning of the simulation (time 0). The event engine starts simulation with a global event heap that contains all the first events.

To run a simulation, the event engine call *next_event* to get the nearest event, then it updates the global simulation time according to this event time, and then it executes this event. The same processes will be used for other events. The event engine uses the task engine to detect the end of the simulation. (We will detail that in Section-4.3.2). The execution of an event interacts on the related processors and orders it to update its state and creates other events. The execution time is defined by the last executed event.

Before explaining the processor engine which performs the execution of the events. We present the task engine and topology engine that will be used extensively by the processor engine.

4.3.2 Task engine

The main objective of our simulator is to simulate the performance of different variants of the *Work Stealing* algorithm. The first variant consists in managing different types of applications. Where an application is defined by a set of tasks with or without precedence constraints. The task engine is used to handle everything related to the application during a simulation.

As stated, an application could be modeled as a divisible load or application with adaptive tasks. In these two types, the task could be divided during a steal request, Moreover, the adaptive task split the task into two subtasks and generates the merge task which depends on these two subtasks. Therefore, our idea is to define a method to **split** work during a steal request. Then, each application type defines this function

according to its characteristics. For instance, the split function in application with divisible load divides a task into two subtasks. In case of application with DAG of task where the steal is handled from the processor queue. Then, the split function return **None** since the tasks can not be splitted.

The execution of a task may activate one or more tasks as in case of DAG task or the application with adaptive tasks, where the execution of task may active the merge task if it exists. To manage this, the task engine defines a method to update the task dependencies when a task is completed.

For all these reasons, the task engine provides an operating interface which offers all the needed functionalities to manage tasks. It also controls the global application. Then, the implementation of a new type of application simply requires the redefinition of the operating interface functions.

We first describe what is needed to manage a task during an execution. Task management consists of controlling the execution time of each task, updating the dependencies when finishing the execution of a task, and splitting tasks between two processors during a steal request. Thus, the operating interface of task engine is based on the following functions:

- **init()** : used to create a new task during a simulation.
- **split()** : used to split the task during a steal and returns **Non** if the task can not be divided..
- **end_execute_task()** : used to update dependencies when a task is completed.
- **get_work()** : used to compute the execution time of the task.

The task engine offers the mechanism to detect the end of an execution. It uses two global variables, one to compute the number of created tasks in the system (updated each init() call)), and to compute the number of completed tasks (updated each *end_execute_task()* call). The execution finishes when the created tasks are equal to the completed tasks.

To simplify the simulation, the task engine offers different functions that automatically generate different application based on DAG tasks. It also offers a function to use a predefined application as input. For this, the predefined application must be described in JSON format that defines the tasks logs (Section 4.3.5).

4.3.3 Topology engine

We target to simulate the *Work Stealing* algorithm on platforms with different topologies. A topology defines the distribution of the processors on the platform and the communication characteristics between them. We explain in Section 4.2.2 the different type of topology. The topology engine is used to manage different platform topologies.

To simulate *Work Stealing* algorithm, the topology is used for knowing the communication time between two processors during a steal operation. Moreover, since the victim selection strategy depends on the processor topology, the topology engine is also used to manage different victim selection strategies. Thus, the engine defines the function *distance()* which returns the communication between two processors in the platform, and the *select_victim()* function which return the id of another processor based on specific strategy.

The topology engine is also used to manage different parameters which are used by the *Work Stealing* algorithm during an execution, for example, *is_simultaneous* is used to determine if a processor can send work to several processors at the same time. The mechanism used to manage this option is defined by the processor engine. It also defines *steal threshold* parameters which can be static or depend on the communication time.

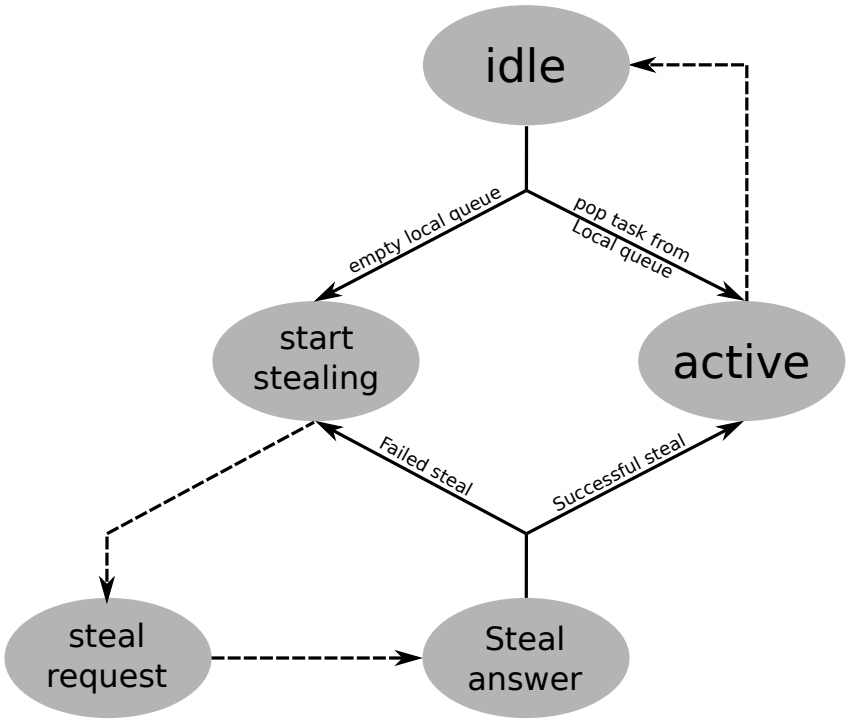


Fig. 4.6: States cycle of a processor

4.3.4 Processor engine

The processor engine manages the processors state on the simulator, it offers all necessary functionalities to update a processor when a related event occurs. These functions apply the mechanism of *Work Stealing* algorithm.

The process followed by a processor during the execution of *Work Stealing* algorithm is defined as shown in Fig 4.6. An active processor becomes idle when it finishes its running task. An idle processor becomes active if it finds tasks in its local queue, otherwise, it becomes a thief and it sends a steal request to another processor (called victim). Once the victim receives the request (*steal request event*), it answers by some of its tasks or failed. Once the thief processor receives the answer (*steal answer event*), it becomes active if the steal succeeds, or it becomes a thief again if the steal failures.

The processor engine provides for each processor different methods to process the *Work Stealing* algorithm. These functions are organizing as follows:

- **idle()** : used when a processor finishes its running task. Its main steps are as follows : It uses task engine to call *end_execute_task()* for the finished task. This operation may active other tasks on the processor local queue.

  Then, the function checks the processor local queue, if is not empty, the processor pops a task from it, and creates the *idle event* correspond. Otherwise, the processor performs a steal request (by calling the **start_stealing()** function).

- **start_stealing()** : used to perform the steal operation. This operation requires a victim selection, and produces a *steal request event*. The victim selection is issued by topology engine (by calling **select_victim()**). Once the victim is selected, then, it computes the communication time between to send the request to the selected victim, and finally it creates the corresponding *steal request event*.

- **answer_steal_request()** : used when a *steal request event* occurs. It performs the answer operation. An answer response moves (if it is possible) the work from the victim to the thief.

  In this function, **get_part_of_work_if_exist()** is used to compute the stolen tasks.

The steal failed in two cases : if there is no work to share, or if the processor is already busy with another steal answer and the topology does not allow simultaneous answer.

Once the stolen task is ready, this function uses topology engine to compute the communication time to answer this request in order to create the corresponding *steal answer event*.

- **get_part_of_work_if_exist()** : used to compute the stolen task. The processor checks its tasks queue, if it is not empty, this function returns a task from it, otherwise, the processor tries to split its running task using the **split()** function defined by the task engine, if the current task is split. It updates the *idle event* correspond to the running task before splitting.

- **steal_answer()** : used to trait the answer request which contains the stolen task. Two cases are possible, if the stolen task contains work, it creates the *idle event* corresponding the execution of the stolen task, Otherwise, the processor will try to steal work again by calling the **start_stealing()** function.

These functions are used by the event engine to execute the three event types as follows:

- The execution of an *idle event* call the **idle()** function.
- The execution of a *Steal Request Event* uses the victim to call the **answer_steal_request()** function.
- The execution of a *Steal Answer Event* use the thief to call the **steal_answer()** function.

4.3.5 Log engine

The simulator is used to experimentally analyze different models of the *Work Stealing* algorithm. It should therefore generate sufficient results that simplify the analysis of the execution of a scenario. For this reason, the log engine is used to provide different functionalities to keep trace of different information during the execution.

Several pieces of information are needed to analyze the execution of a scenario. For instance, we need the global execution information such as execution time and the number of steal requests. These results are presented in digital format. Other information are useful like the processes state over the whole execution or the final shape of the application executed, thus, the engine should log the different changes

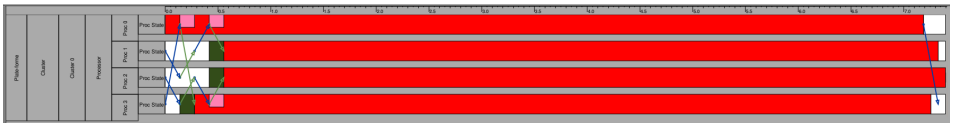


Fig. 4.7: Gantt Chart of the whole execution

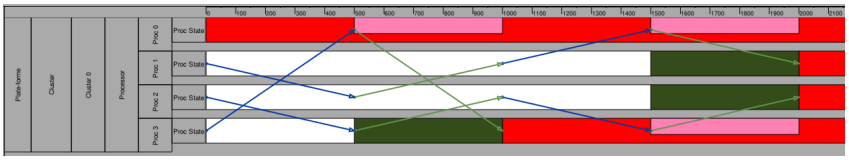


Fig. 4.8: Gantt Chart of the first step of the execution

on the application and on the processor states and their interaction during the simulation.

For these reasons, the log engine uses processor engine and task engine functionalities to keep track of simulation information. For instance, to account the global number of steal request, the log engine initialize the number of steal requests to 0 at the beginning and increments it each time the **Answer_steal_request()** function is called. In another example, the log engine captures the dependencies each time the **split()** defined in the task engine is called.

After a simulation, the overall results like (execution time, number of successful and failed steal requests, total work executed, etc...) are displayed in the console in digital format. Moreover, the simulator offers the possibility to generate other special logs that can be transformed into graphic format using standard trace analysis tools (*Paje* file format [69] and [36]).

For instance, Fig 4.14 depicts the Gantt chart of the processors during the execution simulation of a scenario generated by our simulator, and displayed using *Paje*. Through this presentation, we can analyze and understand what happened in the whole execution or a part of it. For instance, we can focus on the first phase of the execution to understand how the work is distributed as in Fig 4.8.

The simulator offers also the possibility to generates the executed application as output file with a *JSON* format. The *JSON* file store for each task of the application all information as the dependencies, the work , the start and finish execution time and the processor that executes it. The JSON file can be displayed using a *JSONTOSVG* tools developed by Frederic Wagner in [36]. Fig 4.9 depicts the execution graph of an application scheduled by our simulator. The colors present the processor, there are useful for understanding the impact of steals on the execution processes.

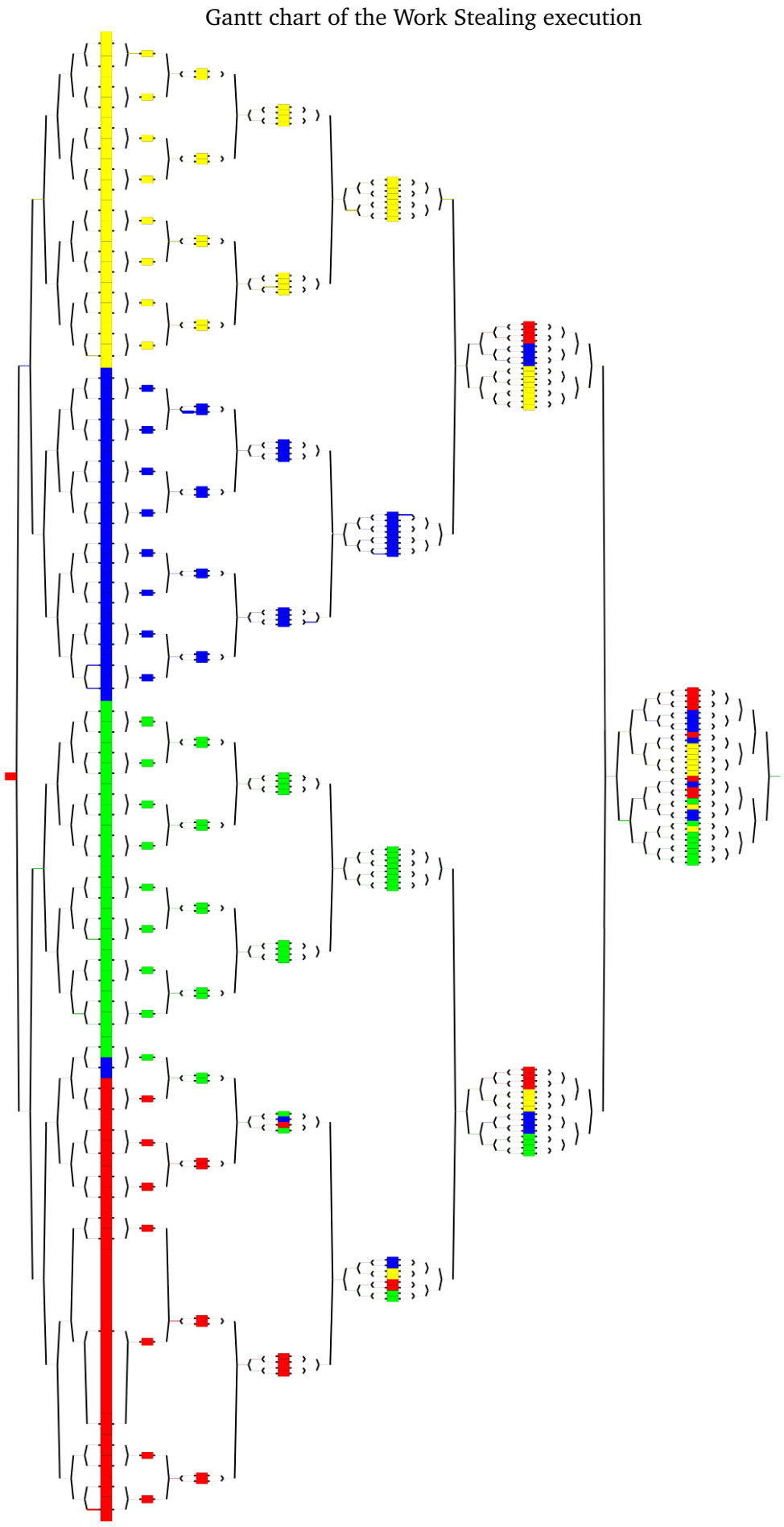


Fig. 4.9: Execution graph of a DAG task application (Merge sort)

4.3.6 Simulator engine

A simulation requires several configurations. We need to initialize and configure the application and the platform with its topology. Then, we need to configure different variants of *Work Stealing* algorithm. The Simulator engine is used to gather all engines to perform the different initialization before starting a simulation.

The principle of the experimental analysis is to obtain several execution results for a scenario in order to analyze the average or the limits. To analyze the impact of a variable, we need to simulate different scenarios for this variable to plot the results according to this variable. For instance, to analyze the impact of the communication latency on the Makespan, we need to run several scenarios for different value of the communication latency, then we analyze the average makespan according to latencies.

For these reasons, the simulator engine proposes for the users a control panel which allows the possibility to configure the different parameters of a scenario as the application and the platform. It allows the user to configure the number of executions for each scenario. It also allows the user to set the interval of configuration values. The simulator engine is developed to run several scenarios and simulation in the same time. This option allows users to save the execution time by running several experiments in common.

4.4 Use of the simulator

4.4.1 Validation and discussion of the theoretical analysis

In Chapter 3, we proved a new upper bound of the Makespan of the *Work stealing* algorithm with an explicit latency on a one cluster topology. The objective of this section is to use the simulator to experiment the *Work stealing* algorithm in order to confirm and discuss the theoretical results and to refine the constant γ defined in Section 4.4.2.

4.4.1.1 Configurations

We configure our this simulator to follow the model of independent tasks described in Section 3.2 to schedule $\mathcal{W}$ unitary independent tasks on a distributed platform composed of p identical processors in one cluster topology. Between each two

processors, the communication cost is modeled by a constant delay represents the latency. (denoted by λ in Chapter 3)

As said earlier in Section 3.3.1, each simulation is fully described by three parameters: $(\mathcal{W}, p, \lambda)$. For our tests, we vary the number of unit tasks $\mathcal{W}$ between 10^5 and 10^8, the number of processors p between 32 and 256 and the latency λ between 2 and 500. Each experimental setting has been reproduced 1000 times in order to compute median or interquartile ranges.

4.4.1.2 Validation of the bound and definition of the "overhead ratio"

As seen before, the bound of the expected Makespan consists of two terms: the first term is the ratio $\mathcal{W}/p$ which does not depend of the configuration and the algorithm, and the second term which represents the overhead related to work requests.

$$\mathbb{E}\left[C_{max}\right] \leq \frac{\mathcal{W}}{p} + 4\lambda\gamma \log_2 \frac{\mathcal{W}}{\lambda}$$

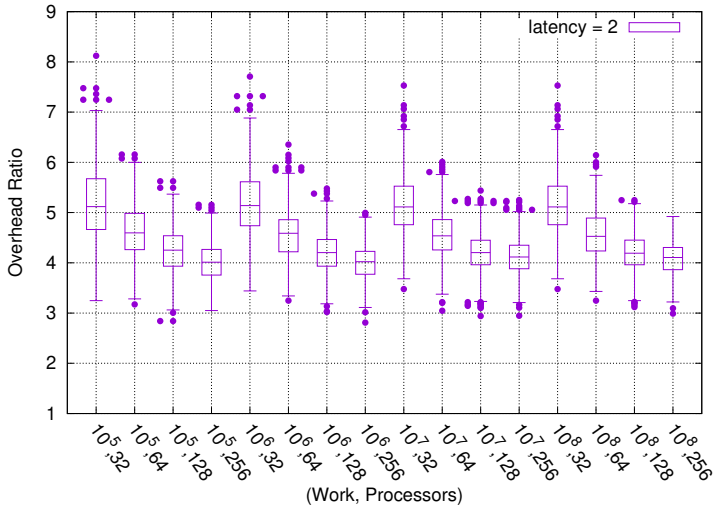
Our analysis bounds the second term to derive our bound on the Makespan. To analyze the validity of our bound, we define what we call the *overhead ratio* as the ratio between the second term of our theoretical bound ($4\gamma\lambda \log_2(\mathcal{W}/\lambda)$) and the execution time simulated minus the ratio $\mathcal{W}/p$: for a given simulation, we define

$$\text{Overhead\_ratio} = \frac{4\gamma\lambda \log_2(\mathcal{W}/\lambda)}{\text{Simulation\_time} - \frac{\mathcal{W}}{p}}$$

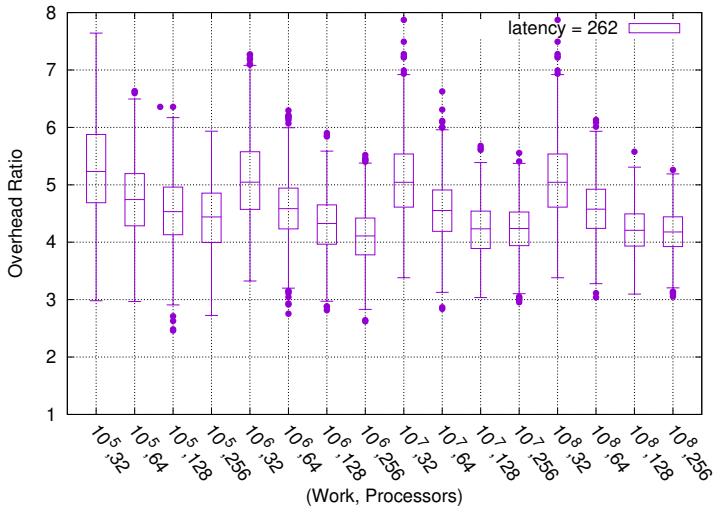
We study this overhead ratio under different parameters $\mathcal{W}$, p and λ.

Fig 4.10 plots the overhead ratio according to each couple $(\mathcal{W}, p)$, for different latency values $\lambda = \{2,\ 262,\ 482\}$ units of time. The x-axis is $(\mathcal{W}, p)$ for all values of $\mathcal{W}$ and p intervals and the y-axis shows the overhead ratio. We use here a BoxPlot graphical method to present the results. BoxPlots give a good overview and a numerical summary of a data set. The "interquartile range" in the middle part of the plot represents the middle quartiles where 50% of the results are presented. The line inside the box presents the median. The whiskers on either side of the IQR represent the lowest and highest quartiles of the data. The ends of the whiskers represent the maximum and minimum of the data, and the individual points beyond the whiskers represent outliers.

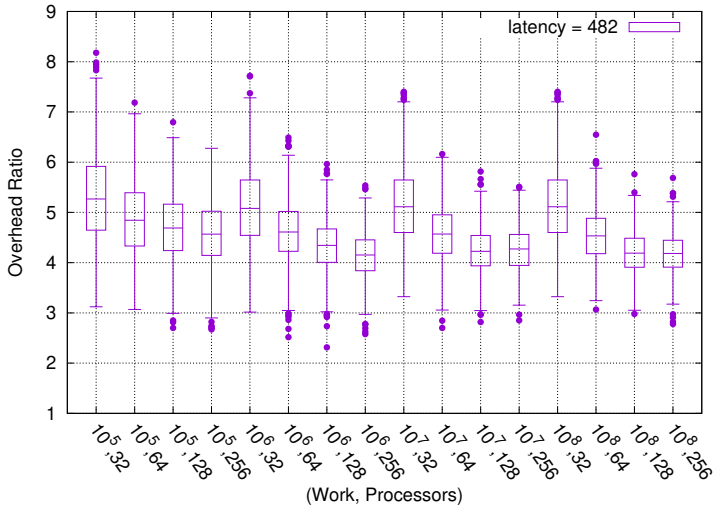
We observe that our bound is systematically about 4 to 5.5 times greater to the one computed by simulation (depending on the range of parameters). The ratio between the two bounds decreases with the number of processors but seems fairly independent to $\mathcal{W}$.



(a) Latency $\lambda = 2$



(b) Latency $\lambda = 262$



(c) Latency $\lambda = 282$

Fig. 4.10: Overhead ratio as a function of $(\mathcal{W}, p)$ for different values of latency λ

4.4.2 Discussion : where does the overhead ratio come from?

The challenge of this work is to analyze WS algorithm with an explicit latency. We presented a new analysis which derives a bound on the expected Makespan for a given $\mathcal{W}$, p and λ. It shows that the expected Makespan is bounded by $\mathcal{W}/p$ plus an additional term bounded by $4\gamma\lambda\log_2(\mathcal{W}/\lambda)$ with $4\gamma \approx 16$. As observed in Figure 4.10, the constant 4γ is about four to five times larger than the one observed by simulation. A more precise fitting based on simulation results leads to the expression $\mathcal{W}/p + 4\lambda\log_2(\mathcal{W}/\lambda)$ (the value 4 is a fitting computed on all our experiments). We explain below where does the discrepancy between the theoretical bound of 16 and the experimental result of 4 come from by looking at the different steps of the proof. Our analysis makes essentially three approximations: (1) The function $h(r)$ is an upper bound on the potential diminution (2) We consider a worst case scenario for the number of steal requests when we define $\gamma = \max_r g(r) = g(p-1)$; and (3) We bound γ by 4.03. We review below the contribution of each approximation to the overhead ratio.

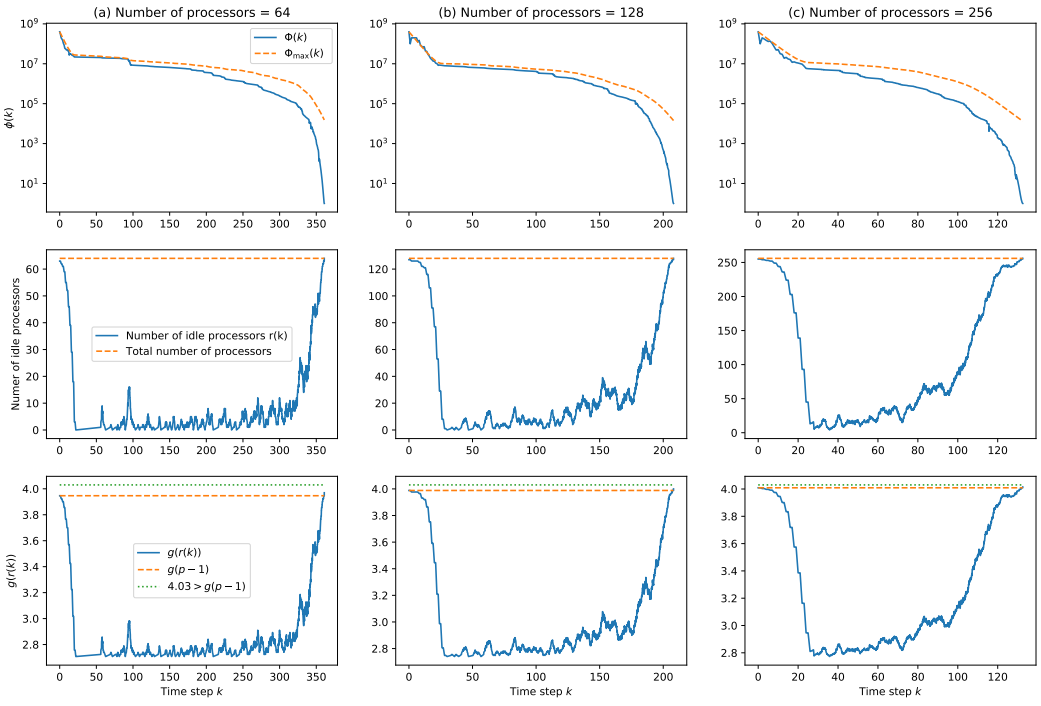


Fig. 4.11: Evolution of the state of the systems for different number of processors ($\lambda = 500,\ W = 10^7$). Each column corresponds to a number of processors (64, 128 or 256). The first line corresponds to $\Phi_{\max}$ (defined in Equation (4.2)). The second line displays the number of idle processors $r(k)$. The third line displays $g(r(k))$ (defined in Equation (3.8)). In all figures, the x-axis corresponds to the number of time steps k.

To illustrate our explanations, we run three simulations by using our simulator and report the results in Figure 4.11. These three executions have the same number of tasks $W = 10^7$ and latency $\lambda = 500$ and we consider three value of processors: $p \in \{64,\ 128,\ 256\}$. This figure illustrates the evolution with time of the potential $\phi(k)$, the number of idle processors and the value of γ defined in chapter 3 section 3.3.3 as:

$$\gamma \stackrel{\text{def}}{=} \max_{1 \leq r \leq p} \frac{r}{-p \log_2(h(r))}$$

Each column represents a value of p. The lines represent the results for each metric.

4.4.2.1 Impact of the bound $h(r)$

The first step of our analysis is to prove a bound on the decrease potential: We show in Lemma 3.3.1 that

$$\mathbb{E}\left[\Phi(k+1) \mid \mathcal{F}_t\right] \leq h(r(k))\Phi(k). \tag{4.1}$$

This bound is obtained by computing the diminution of the potential in the various cases of the proof of Lemma 3.3.1. This various cases makes different approximations. First, our bound $h(r)$ is in fact $h(r)$ the maximum between the ratio $2/3$ of Case 1 and the ratio $h(r)$ of Case 2a. Second, we assumed that we do not know when a work requests arrive in the interval. We therefore always took the worst case (arrivals at the end of the intervals). Third, we neglect the diminution of the potential due to working processors (Case 2a).

To see measure the impact of this approximation, we define a theoretical function $\phi_{\max}(k)$ that would corresponds to the potential of the system if the inequality of Equation (4.1) was an equality:

$$\phi_{\max}(k) = \begin{cases} \phi(0) & \text{if } k = 0 \\ h(r(k))\phi_{\max}(k-1), & \text{otherwise} \end{cases} \tag{4.2}$$

In Figure 4.11-(Line 1), we plot this theoretical function and the real potential function as a function of time step k. This figure indicates that the distance between the real potential and the theoretical bound is relatively small at the beginning of the execution, which makes sense since the diminution of the potential is dominated by the diminution related to Case 2a. The two function starts diverging slowly in the middle of the execution and this divergence is accentuated at the end: when the execution is close to finishing, the actual potential decreases much faster that its bound. We believe that this divergence is mostly due to neglecting the diminution

of potential due to working processors: At the beginning of the execution when all processors have many tasks to execute, neglecting working processor is negligible; At the end of the execution when the remaining work is small, the potential diminution is greatly affected by the working processors.

This analysis could probably be improved by taking a more complex potential function that will diminish the impact of working processors.

4.4.2.2 Evolution of the number of work requests

In our analysis, we study how the potential decreases a function of the number of steal requests $r(k)$. To obtain a bound, we then use a worst-case analysis and define γ as the maximal of a function $g(r) = r/(-p \log_2 h(r))$: $\gamma = \max_r g(r)$. As shown in Figure 3.2, $g(r)$ is between 2.8 where r is small to 4 when r is large. On the second Line 2 of Figure 4.11, we depicts how the number of idle processors evolve with time. We observe that an execution has essentially three phases: in the beginning, there is a high number of idle processors since the work has to be divided among processors; In the middle of the execution, the number of idle processors is small as everybody is working. In the final execution phase, it increases as finding work becomes harder.

In Figure 4.11-(Line 3), we plot $g(r(k))$ and γ as a function of the time step k. The figure shows that $g(r(k))$ is often about 2.8 (because the period where the number of idle processors is low is long). This suggests that our bound $\gamma = \max_r g(r)$ is about 1.4 times too high. Being able to capture more precisely how the number of idle processors evolves might lead to a bound that would be around 30% times smaller.

4.4.2.3 Bound $\gamma < 4.03$ and impact of the number of processors

In our analysis, we show that $\gamma = g(p-1)$ and we bound γ by 4.03. In fact, the value of 4.03 corresponds to what happens when p goes to infinity but smaller values of p leads to smaller values of γ. This explains why in Figure 4.10, we observe that overhead ratio decreases with the number of processors, from around 5 for 32 processors to around 4 to 4.5 for 256 processors.

In our simulation, we observe that the overhead ratio is between 4 to 5. Based on our experimental study of the evolution of the number of work requests with time, we believe that the worst-case analysis $\gamma = g(p-1)$ and the bound of 4.03 contributes to a factor of about 1.5 of the overhead ratio. The remaining factor (of about 3) is mostly due to the bound h that we obtained in Lemma 3.3.1. Refining this lemma, for example by being able to estimate the decrease of potential due

to the work execution or by using a different potential definition, would lead to a tighter bound.

4.4.3 Acceptable latency

The combination between the theoretical bound and the experiment fitting of the constant lend to the Makespan analytical expression $\mathcal{W}/p + 3.8\lambda \log_2(\mathcal{W}/\lambda)$. One of the first uses of this expression is to predict when a given $\frac{W}{p}$ and λ configuration will yield acceptable performances. Using the Makespan expression we observe that two parameters dominate: The $\frac{W}{p}$ ratio in the first term and λ which impacts the second term of the formula representing the overhead due to communication delays.

As stated before $\frac{W}{p}$ is a good lower bound on the best possible Makespan. A Makespan $C_{\max}$ is acceptable if the ratio $C_{\max}/C^*_{\max}$ is close to 1, where $C^*_{\max}$ is the best possible Makespan. In our analysis, we consider a Makespan $C_{\max}$ as acceptable if $\frac{C_{\max}}{(W/p)} \leq 1.1$ (overhead less than 10%). We study here which configurations allow us to obtain such an acceptable Makespan. Using the time estimation Formula we derive the equation below linking W, λ and p in order to get an acceptable Makespan.

$$\frac{W}{p} + 3.8 \log_2(\frac{W}{2\lambda})\lambda = 1.1\frac{W}{p}$$

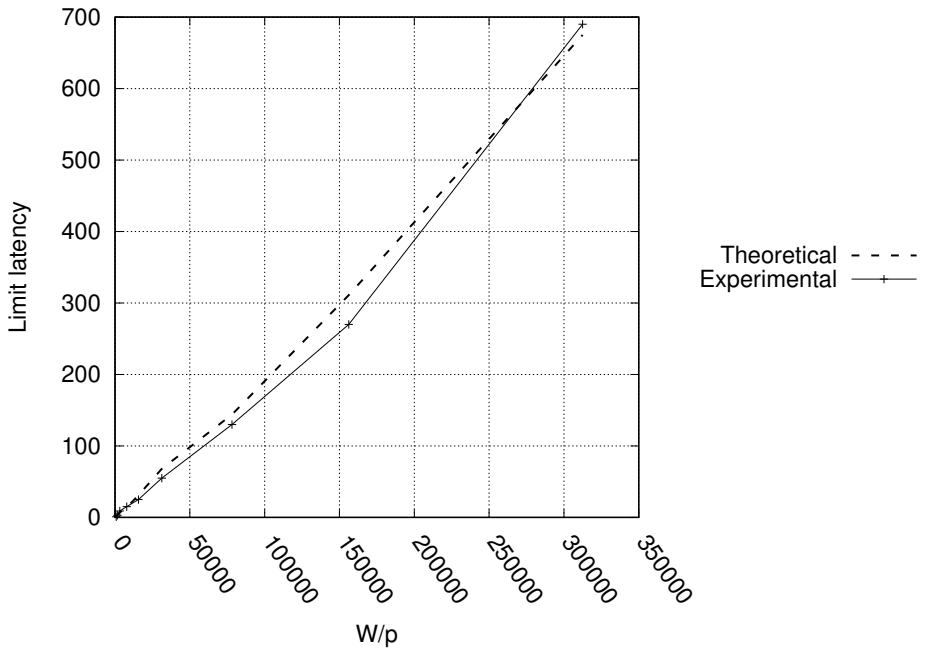


Fig. 4.12: Limit latency exhibiting an acceptable Makespan according to $\frac{W}{p}$

Using this equation we can easily predict when a given W, p and λ yields acceptable performance. Moreover for a specific W and a fixed λ we can easily choose the maximum number of processors applicable.

To verify the validity of this formula we solve numerically this equation for different $\frac{W}{p}$ to get the theoretical limit latency for an acceptable Makespan. We then verify experimentally the obtained solutions. So for a fixed W and p we test different λ and take the maximal one yielding an acceptable Makespan. We call this *the experimental limit latency*. With this result we are able to compare the theoretical and the experimental limit latency. Fig 4.12 plots the theoretical and experimental limit latency according to $\frac{W}{p}$. The x-axis is $\frac{W}{p}$ for W between 10^5 and 10^8 and p between 32 and 256 y-axis show the limit latency.

In Fig 4.12 we observe that the two curves overlap and conclude again on the good accuracy of our prediction. Moreover we can see that the relation between the latency limit and the $\frac{W}{p}$ ratio is close to linear. Using this figure we can derive the following equation: $\frac{W}{p} = 470\lambda$. Using this equation it is easy to evaluate performances for a given W, p and λ. In addition it allows us to compute easily for any configuration the maximal number of processors $\frac{W}{470\lambda}$ yielding an acceptable Makespan.

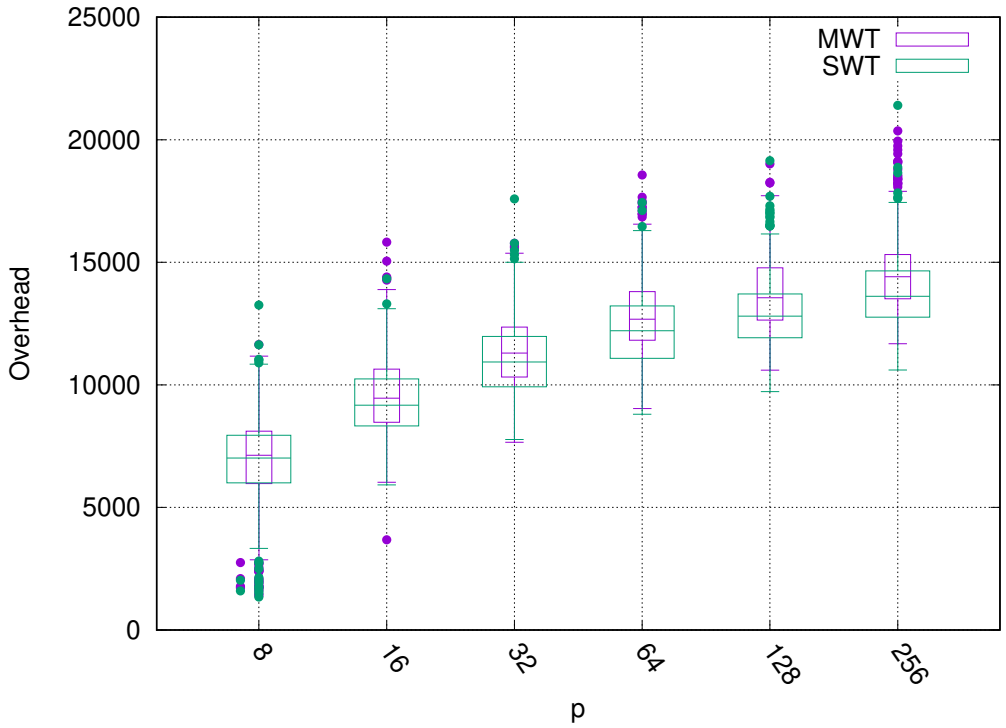
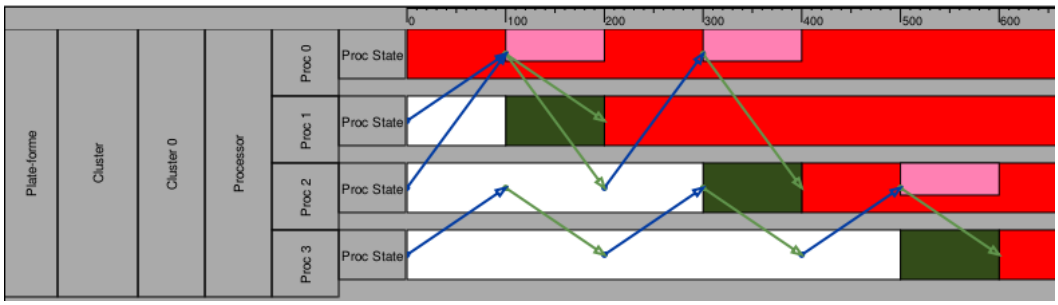


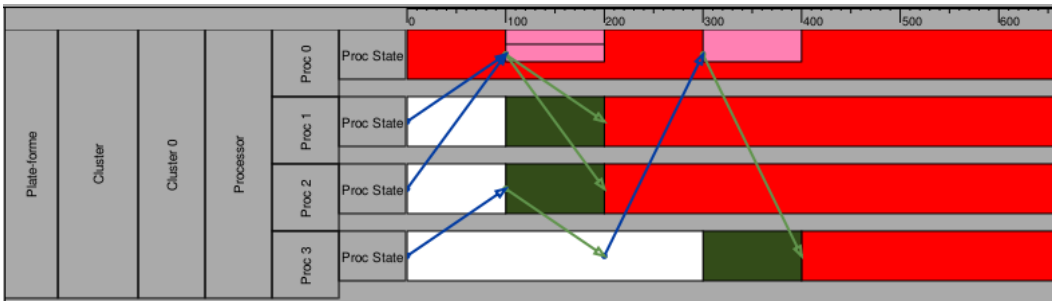
Fig. 4.13: The Overhead of the execution using *MWT* and *SWT* according to the number of processors ($\lambda = 262$ and $W = 10^8$)

4.4.4 The impact of simultaneous responses

We use in this section our simulator to study the influence of the multiple work transfers mechanism (*MWT*) on One cluster topology. In our experimental runs, we compare the results obtained using both variants: With multiple work transfers and with a single work transfer (*SWT*). Fig 4.13 depicts a comparison between LWR



a) execution with SWT



b) execution with MWT

Fig. 4.14: Gantt chart of the first phase of execution, comparison between *MWT* and *SWT*

and SWR showing the overload obtained using each mechanism according to the processor number. This show that the *MWT* mechanism does not bring a significant gain in the overall performances, which spurred us to analyze in detail the execution traces. In this analysis we remark that any execution using a *Work Stealing* algorithm decomposes into three phases. The first phase which is denoted by the startup phase, when all the processors try to have work. This phase finishes when all processors become active. The second phase corresponds to the situation in which all processors have work and just a few steal requests between processors happen. The last phase starts when there is little work and the majority of processors are inactive.

In practice, we observe that the *MWT* mechanism only impacts significantly the startup phase. Fig 4.14 depicts an example of two scenarios which clarify the impact of *MWT* and *SWT* on the Gantt chart of the first phase. As we see at time $t = 0$, the processors P_1 and P_2 send to steal the processor P_0, and P_3 sends a steal request to P_2, all steal requests arrive at the same time at $t = 100$. In the case of single work transfer *SWT* in Fig 4.14-a, the processor P_0 answers with some of its work

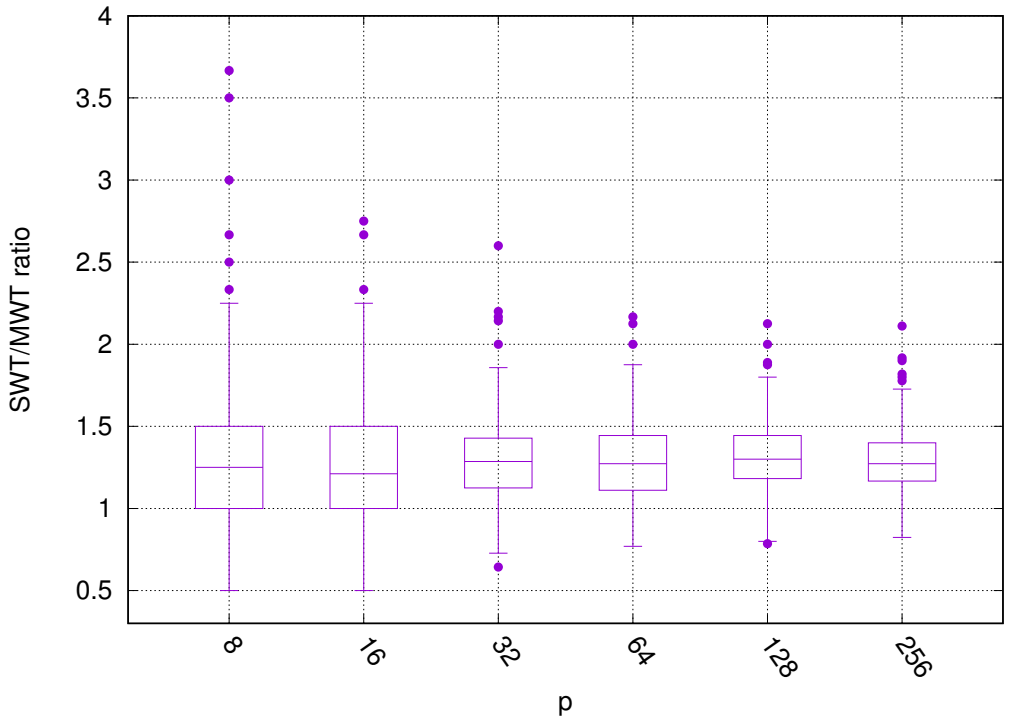


Fig. 4.15: The ratio between the duration of the startup phase of the execution using *MWT* and with *SWT* according to the number of processors ($\lambda = 262$ and $W = 10^8$)

to P_1 and answers P_2 with failed responses. At $t = 200$, P_1 receives the stolen work and becomes active, and P_2 and P_3 try to steal again. Which is not the case in the case with multiple work transfer *SWT* in Fig 4.14-b where the processor P_0 answer P_1 and P_2 at the same time. This act accelerates the increase in the number of active processors after each round trip (steal-answer), which is clear in the figure at $t = 300$.

Fig 4.15 presents in BoxPlot format the ratio between the duration of the startup phase using the *MWT* mechanism and using the *SWT* mechanism according to the number of processors. The x-axis is the number of processors and the y-axis is the ratio between the two durations of the startup phase using the *SWT* and *MWT* mechanisms for $\lambda = 262$ and $W = 10^8$. In this setting we see that *MWT* is reducing the duration of the startup phase for 75% of the runs with a gain larger than 200% for a small number of processors. The behavior of *MWT* is positive on the startup phase but the overall performance gains are small because the duration of the startup phase is small compared to the total execution time.

4.5 Conclusion

We present in this chapter our lightweight PYTHON simulator for experimentally analyze different model of *Work Stealing* algorithms. Our simulator is developed to be flexible enough to simulate different topologies and applications with different variants of *Work Stealing* algorithms. Using this simulator, we provided and discussed the theoretical bound on the Makespan execution of *Work Stealing* on one cluster topology founded in Chapter 3. We also experimentally study the impact of simultaneous responses on one cluster. In the next chapter, we will use our simulator to experimentally study the *Work Stealing* algorithm on more complex topologies. We also propose and experiment different victim selection strategies on two clusters topologies.

5 Work Stealing on multi clusters

Contents

We study in this chapter another model of *Work Stealing* algorithm on a more complex environment. In particular, we consider a two-cluster platform with two levels of communication. Light communication between the processors inside the same cluster, and more expensive communication between processors in different clusters. We start by describing the model and the difficulties arising from heterogeneous communication. Then we present different strategies to deal with communication. We use our simulator described in Chapter 4 to perform a large experimental analysis. Through this analysis, we discuss several strategies and we perform a comparison between them.

5.1 Introduction

Distributed-memory clusters consist in independent interconnected processing multi-cores owning each a private shared memory. The clusters are linked via an interconnection network and the communications between clusters are crucial since they highly influence the performances (as the trade-off between a good balance of the load and low communications is not that easy to determine). There are only limited

works dealing with optimized allocations and the relationships between allocation and scheduling are most often ignored. In practice, the impact of scheduling may be huge since the whole execution can be highly affected by the latency of interconnection networks. The most commonly studied objective in scheduling is to minimize the maximum completion time $C_{\max}$ (called makespan) assuming centralized algorithms. This assumption is not always realistic, especially for distributed memory allocations and an on-line setting. WS is an efficient decentralized scheduling mechanism targeting medium range parallelism of multi-cores for fine-grain tasks. Its principle is that each processor manages its own (local) list of tasks. When a processor becomes idle, it randomly chooses another one and if possible, it steals some work. The cost of the stealing mechanism is negligible in a shared memory context but it is not for distributed memory. Its analysis is probabilistic since the algorithm itself is randomized. Today, the research on WS is driven by the question on how to extend the existing bounds for new computing platforms like distributed memory clusters (our target). Notice that beside its theoretical interest, WS has been implemented successfully in several languages and parallel libraries including Cilk [37, 55], TBB (Threading Building Blocks) [72], the PGAS language [31, 64] and the KAAPI run-time [41].

5.2 Model of Work Stealing on two-clusters platform

We consider an underlying model of parallel platform composed of two clusters linked by an interconnection network. Each one contains m identical processors. Inside the clusters, the processors communicate through a shared memory, where each communication is immediate. Outside a cluster, the processors use an interconnection network to communicates between them. This communication takes a constant delay (the inter-cluster latency, denoted by λ). Thus, a communication outside clusters takes λ time steps. We consider that communication outside clusters takes much more than inside ($\lambda \gg 1$).

5.2.1 Bimodal Work Stealing

In WS, each processor either executes work or tries to steal some work. During a steal, If the thief selects a processor (the victim) on the same cluster, it takes 2 time steps (1 time step to send it another step to receive the answer). If the victim and the thief are not in the same cluster. The work request takes λ time steps to reach the victim and another λ to receive the answer (whatever it is, positive or negative). At the beginning, all the work is put on one processor. All the other processors

act as thieves. They try to find work by sending work requests to other processors randomly.

5.2.2 Victim Selection issue

Since work requests inside the cluster take much more time than outside, we prefer to steal inside the cluster where the thief is located. However, sometimes the work inside the clusters is not large enough to be divided. In this case, a steal outside the cluster is preferred to balance the work. For these reasons. Victim selection is not obvious and requires a smart method that intents to balance the work and minimize remote steals in the same time. We present in our work different victim selection strategies that try to balance the work and minimize the number of external steals in the same time. then, we experiment each strategy through an experimental campaign.

5.2.3 Amount of work per remote steal

The main objective of the Work Stealing algorithm is to balance the load between the processors. During each successful steal between two processors, the idle processor (thief) steals half of the work from the active processor (victim). In the model with two clusters, the steals inside a cluster intents to balance the load between the processes inside the cluster, and the steals outside the clusters try to balance the load between the clusters. The victim selection strategies aim to minimize the number of external steals because they take a long time. For these reasons, we propose in this variant to increase the amount of work stolen during remote steals. This means that, during each successful external steal, the thief sends more than half its work to the victim, which means that the victim cluster receives enough work during an expensive remote steal request. To show the impact of this this variant on the performance, We use simulation to experiment (in section 5.4) the impact of this variant on the Bimodal Work stealing with each strategies.

5.3 Victim Selection Strategies

We propose in this section three victim selection strategies, for each one, we configure our simulator (described in Chapter 4.1) to turn simulations with different parameters. To analyze the results of each simulation, we define what we call the

overhead as the execution time simulated minus the ratio $\mathcal{W}/p$,
For a given simulation, we define :

$$\text{Overhead} = \text{Simulation\_time} - \frac{\mathcal{W}}{p}$$

where $\mathcal{W}$ and p are the total amount of work and the number of processors respectively.

During the description of these strategies, We perform an experimental study to discuss each strategy and their configuration, we configure the simulator to follow each strategy to schedule $\mathcal{W}$ unitary independent tasks on a distributed platform composed of two clusters of $p/2$ identical processors each. The communication cost between two processors in different clusters is modeled by a constant delay represented the latency λ. In our simulation, we take several values of $\mathcal{W}$, p and λ to show the impact of each parameter. Each experimental setting has been reproduced 1000 times in order to compute median or interquartile ranges.

5.3.1 Baseline strategy

The baseline victim selection strategy consists of treating all processors with the same way. A thief chooses the victim from any available processor in the two clusters. This method does not take into account communication time between processors, which increases the time spent on work steal. Fig 5.1 shows an example of Work Stealing execution on two clusters using the baseline strategy. All works are on processor P_1 in cluster C_0. The pale color in the Gantt chart of the execution represents the steal time (include the answer) and the red color represents the execution time. It is clear that a lot of time is wasted looking for work at the beginning because of wrong victim selection. In Fig 5.2, we can observe that idle processors of cluster C_0 make the wrong decision when they choose the processor in the other cluster, the same problem is observed in the final phase.

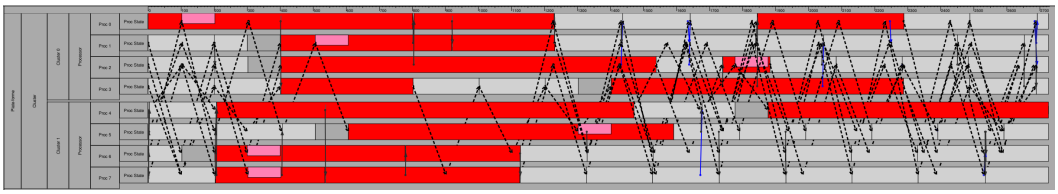


Fig. 5.1: Gantt chart of the Work Stealing execution on two clusters with 4 processors each, the victim selection follows the baseline strategy

It lacks a strategy that takes into account the communication time between processors, in order to avoid as much as possible the external steal. We should know that external steals are important to balance the load between the clusters. Therefore,

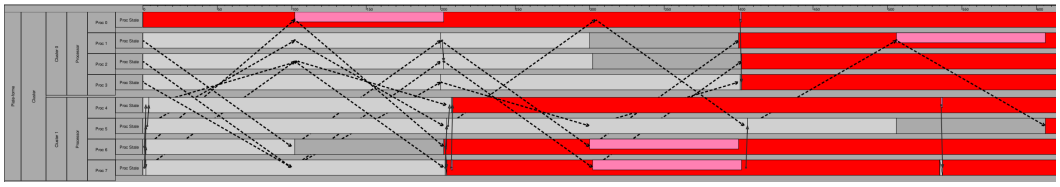


Fig. 5.2: A zoom on the beginning of the execution (same example as Fig 5.1)

we need a strategy that satisfies both sides, balance the load between clusters and minimize external steals. In the rest of this section, we propose three victim selection strategies that take this problem into account.

5.3.2 Systematic Victim selection (SVS)

The idea of this strategy is to control the external steals systematically, by fixing the number of local steal attempts before stealing remotely. Which means that each processor tries to steal locally (inside its cluster) several times, if none of the steals succeeds, the processor changes to steal from the other cluster. For this, we define the maximum internal steal attempts (denoted by isa) as the maximum number of steal attempts inside the cluster before sending an external steal. Thus, when a processor becomes idle, it attempts to steal randomly some work inside its cluster. If the number of failed steal attempts inside the cluster exceeds maximum isa, the processor chooses randomly another processor in the second cluster. If this request failed, the processor starts again to steal inside with the same strategy.

The challenge of this strategy is to find the reasonable value or (the interval) of the maximum internal steal attempts isa. For that, we use simulations to get an idea about the adequate value of isa and its impact on the performance of Work Stealing algorithm on a platform with two clusters. We study the overhead obtained by Work stealing algorithm using this strategy, according to different values of isa.

Fig 5.3 depicts the impact of the maximum internal steal attempts isa on the average overhead (defined in Section 5.3). Each figure plot curves for different $W = (10^7, 5.10^7, 10^8, 5.10^8)$ Each column corresponds to the number of processors $(16, 32$ or $64)$. Each line corresponds to the latency $\lambda = (64, 128, 256, 512)$.

The simulation results show that the curves follow the same behavior. The average overhead is high when isa is small ($isa < 4$). Then, there is an interval of isa where the overhead is stable and close to the minimum, Then, the average overhead increases again according to isa.

Fig 5.4 shows the curve for ($p = 32, W = 100000000, \lambda = 64$) with boxplot visualization. The "interquartile range" in the middle part of the plot represents the middle quartiles where 50% of the results are presented. This curve confirms the observation obtained by the average overhead.

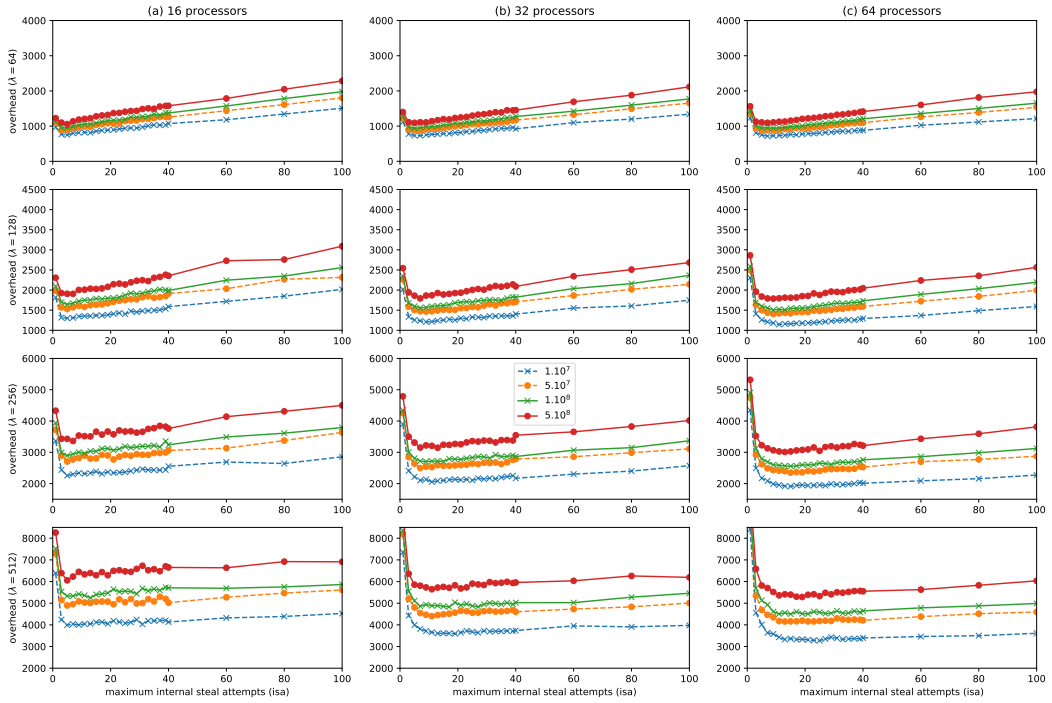


Fig. 5.3: Average overhead according to the maximum internal steal attempts isa (different W for each λ). Each column corresponds to the number of processors (16, 32 or 64). Each line corresponds to the latency $\lambda = (64, 128, 256, 512)$. In all figures, the x-axis corresponds to the maximum internal steal attempts isa.

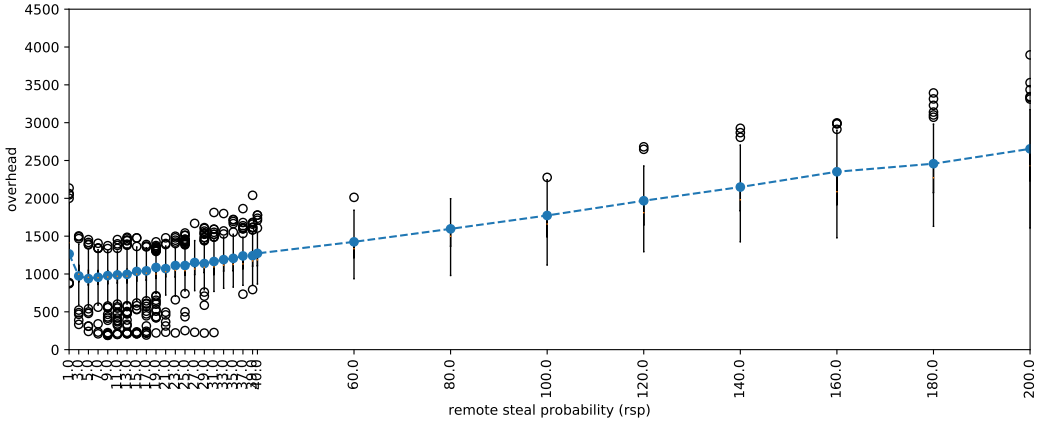


Fig. 5.4: Overhead according to the maximum internal steal attempts using boxplot visualization curve for $p = 32, w = 100000000, \lambda = 64$. The "interquartile range" in the middle part of the plot represents the middle quartiles where 50% of the results are presented.

As we observe in Fig 5.3 and Fig 5.4, it is difficult to see the best value of isa to obtain the minimum overhead, because there is an interval of values where the overhead is stable. For that, we propose to fix the value the maximum internal steal

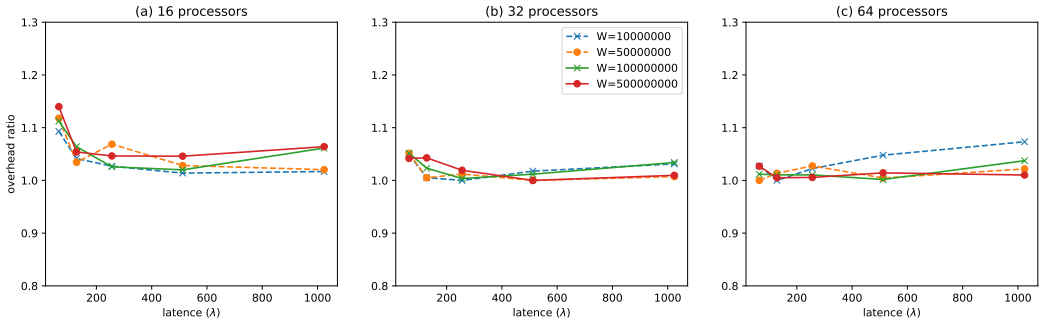


Fig. 5.5: Overhead ratio between the overhead obtained by $isa = 10$ and the best overhead minimum according to latency

attempts in isa in the middle of this interval. Moreover, if we choose $rsp = 0.5$ as the right value, we can obtain an overhead close to the minimum.

To confirm this proposition, we compute the ratio between the overhead obtained by $isa = 10$ and the best overhead obtained. Fig 5.5 depicts this ratio according the latency λ. the results show that the ratio is stable and almost next to 1, which confirm that by fixing the remote probability in 0.05 leads to a good performance.

5.3.3 Probabilistic victim selection (PVS)

In this strategy, the idea of this strategy is to use a probability to choose between an internal or external steal. Thus, when a processor becomes idle, it uses a probability to decide whether it sends a steal request inside or outside its cluster. Using the value of this probability, we are able to control the number of internal and external steal. We define a remote steal probability (denoted by rsp), which defines the probability to steal remotely (outside the cluster). Thus, an idle processor sends a steal outside its cluster with a probability of rsp, and inside its cluster with probability of $1 - rsp$. Once the cluster is chosen, the thief will select the victim randomly inside the selected cluster.

The challenge of this strategy is to find the reasonable value or (the interval) of the remote steal probability rsp. The value of rsp should logically be less than 0.5, otherwise, external steals are preferred. When $rsp = 0.5$, the probabilistic Victim selection become the baseline strategy. A small value of rsp (next to 0) will decrease the external steal requests, which can lead to an unbalanced between the clusters. Thus, the value of rsp could be between 0 and 0.5.

We use simulations to get an idea about the adequate value of rsp and its impact on the performance of Work Stealing algorithm on a platform with two clusters.

We study the overhead obtained by Work stealing algorithm using this strategy, according to different values of rsp.

Fig 5.6 depicts the impact of the remote steal probability rsp on the average overhead (defined in Section 5.3). Each figure plot curves for different $W = (10^7, 5.10^7, 10^8, 5.10^8)$ Each column corresponds to the number of processors (16, 32 or 64). Each line corresponds to the latency $\lambda = (64, 128, 256, 512)$.

The simulation results show that the curves follow the same behavior, when rsp is close to 0, the average overhead is high , then, there is an interval of rsp value which give an overhead close to the minimum. Then, the average overhead increases again according to rsp when the value of rsp is more than 0.1. Fig 5.7 shows the curve for $(p = 32, w = 100000000, \lambda = 64)$ with boxplot visualization. The "interquartile range" in the middle part of the plot represents the middle quartiles where 50% of the results are presented. This curve confirms that most of the results follow the same behavior as the averadge means. observations obtained average overhead.

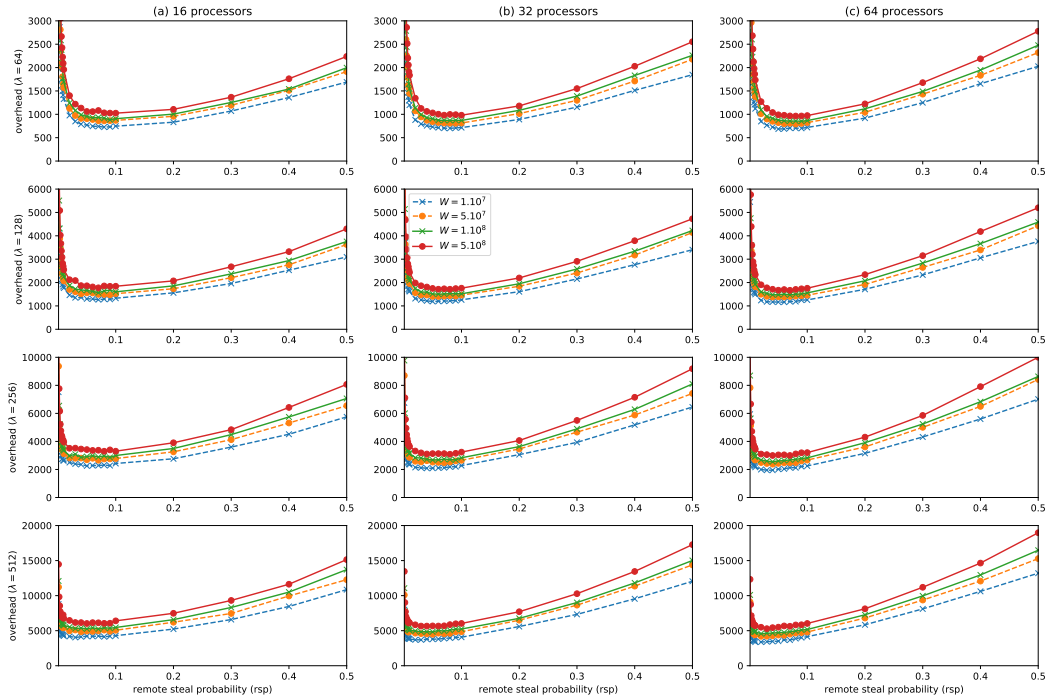


Fig. 5.6: Average overhead according to remote steal probability (different W for each λ) Each column corresponds to the number of processors (16, 32 or 64). Each line corresponds to the latency $\lambda = (64, 128, 256, 512)$ In all figures, the x-axis corresponds to the remote steal probability rsp.

As we observe in Fig 5.6 and Fig 5.7, it is difficult to see the best value of rsp to obtain the minimum overhead, but there is an interval of rsp values when the overhead is stable. This interval expands when the latency increase, but we can easily observe in (Fig 5.6) that is almost between 0.01 and 0.1 for all our configurations.

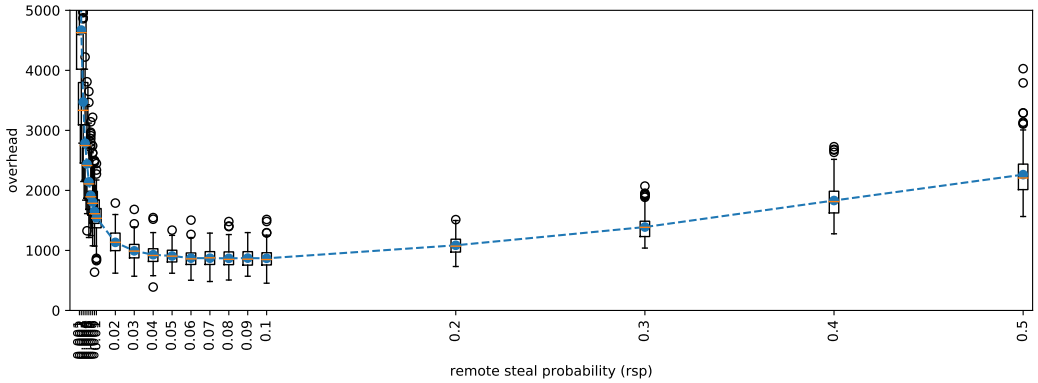


Fig. 5.7: Overhead according to remote steal probability using boxplot visualization curve for $p = 32, w = 100000000, \lambda = 64$

For that, we propose to fix the value of rsp in the middle of this interval, Moreover, if we choose $rsp = 0.05$ as the right value, we can obtain an overhead close to the minimum. To confirm this proposition, Fig 5.8 depicts the ratio between the overhead obtained by $rsp = 0.05$ and the best overhead obtained. the results show that the ratio is stable and almost next to 1, which confirm that using the dynamic probability victim selection and fixing the remote probability in 0.05 leads to a good performance.

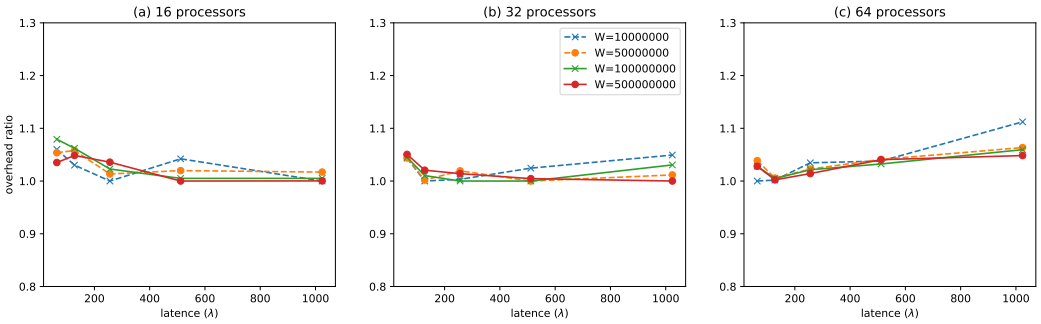


Fig. 5.8: Overhead ratio according to latency between the overhead obtained by $rsp = 0.05$ and the best overhead minimum

5.3.4 Dynamic probability victim selection (DPVS)

In the previous strategy (PVS), The remote steal probability is constant and independent of failed local steals, and we use a small remote steal probability to avoid as much as possible remote steals. In this strategy, we extend to made this probability dynamic and related to the local failed steals. Moreover, each processor starts the execution with a remote steal probability equal to 0, and when a local steal failed, the processor increments its remote steal probability to increase the chance of stealing remotely. Then, it tries to make another steal based on the new value of remote steal probability. Once a steal succeed or a remote steal failed, the processor resets its remote steal probability to 0. We define the remote steal probability step (defined by

rsp_{step}) as the additional value add to the remote steal probability after each failed local steal.

the challenge of this strategy is to find the adequate value of rsp_{step}. The remote steal probability should not increase quickly to avoid remote steal quickly. For that, we study the overhead obtained by the Work stealing algorithm using this strategy, according to different values of rsp_{step}.

Fig 5.9 depicts the impact of the remote steal probability step rsp_{step} on the average overhead. We observe in this figure that the overhead is better when the rsp_{step} is less than 0.1. And we also observe that is hard to detect the best value of rsp_{step}. For that, we propose to fix the value of rsp_{step} in a small value, Moreover, if we choose $rsp = 0.03$ as the right value of rsp_{step}, we can obtain an overhead close to the minimum for all our configuration.

To confirm this proposition, Fig 5.10 depicts the ratio between the overhead obtained by $rsp_{step} = 0.03$ and the best overhead obtained. the results show that the ratio is stable and almost next to 1, which confirm that using the dynamic probability victim selection strategy and fixing the remote steal probability step in 0.03 leads to a good performance.

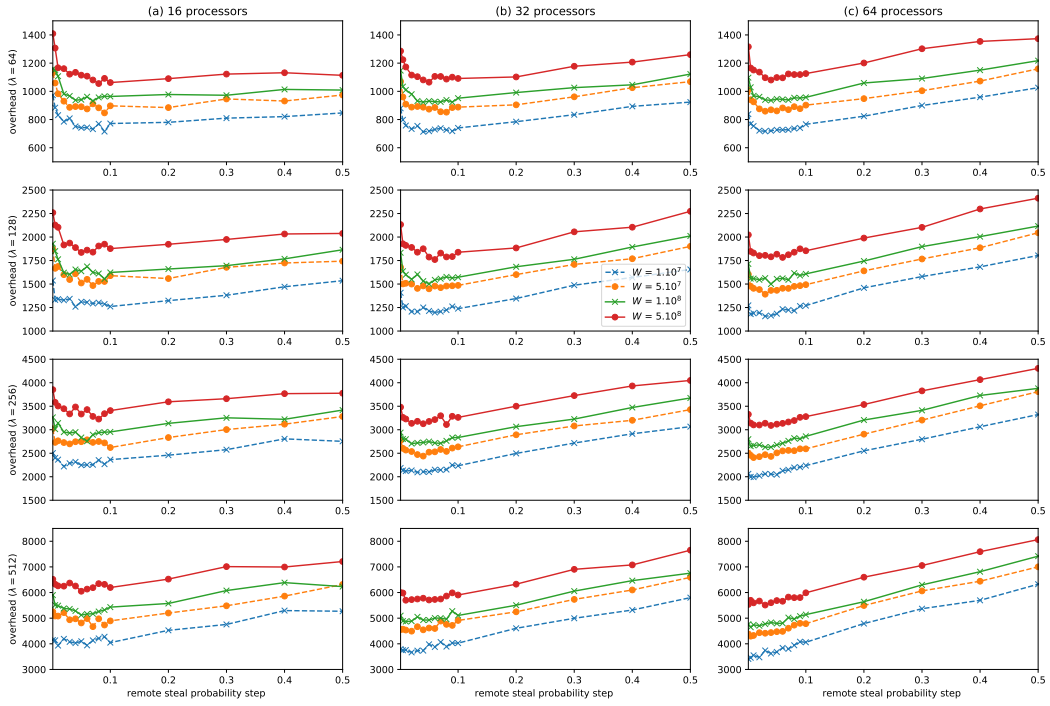


Fig. 5.9: Average overhead according to the remote steal probability step isa (different W for each λ). Each column corresponds to the number of processors (16, 32 or 64). Each line corresponds to the latency $\lambda = (64, 128, 256, 512)$. In all figures, the x-axis corresponds to the remote steal probability step rsp_{step}.

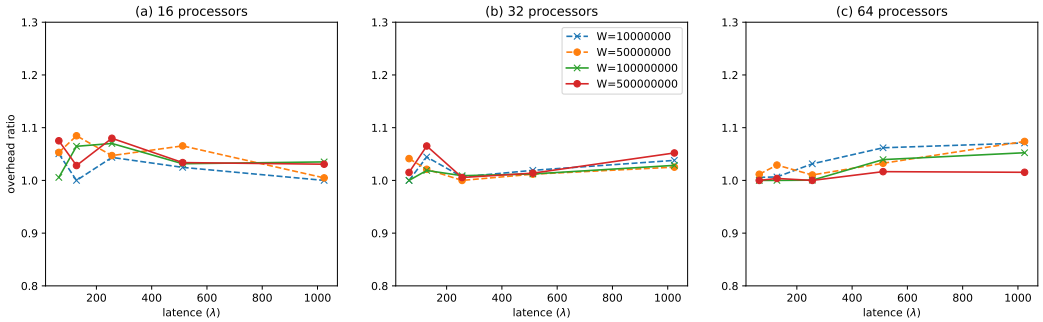


Fig. 5.10: Overhead ratio between the overhead obtained by $rsp_{step} = 0.03$ and the best overhead minimum according to latency

5.4 Impact of the amount of work per steal

We study in this section the impact of the amount of work per remote steal on each strategy. In our configuration, the amount of work per steal is represented by a percentage value, which means that during an internal steal, the thief receives 50% of the victim work. We adapt our simulator to control this percentage during the remote steals. then, we vary this percentage to steal 50%, 70%, 80%, and 90% of the victim work during a remote steal, then we compare the overhead obtained for each strategy. The steals inside the same cluster still fixed in 50% of work per steal.

Fig 5.11 shows a comparison between the average overhead obtained by the three strategies for different values of percentage of work per remote steal (50%, 70%, 80%, 90%). We observe that if we increase the percentage of work per remote steal, we improve the performance by decreasing the overhead for the three strategies. Moreover, the simulations show that steal between 70% and 80% gives the best performance for each strategy. Steal 90% also gives an overhead smaller than the classic percentage of steal 50%.

Besides the improvement obtained by the strategies, this variant could also important to improve performance of each strategy, Fig 5.11 shows in the right column an overview about the gain ratio of each value of the percentage of work per remote steal compared to the classical configuration that fixes the percentage in 50%.

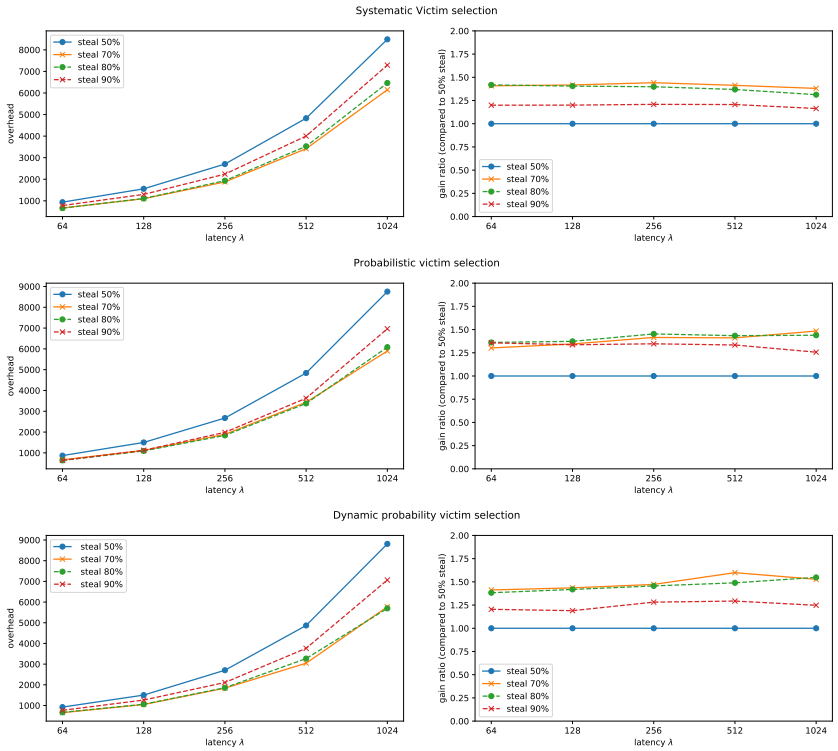


Fig. 5.11: Comparison between the average overhead obtained by the three strategies for different value of percentage of work per remote steal (steal 50%, 70%, 80%, 90%), we set other values in $p = 32$ and $W = 10^8$. The left column shows the ratio between the overhead obtained by each percentage and the classic percentage (50%). with the best configurations. ($\lambda = 10^8$, $p = 32$, best rsp_{step})

5.5 Comparison

In this section, we perform a general comparison between the different strategies using simulations, We compare the overhead obtained by each strategy with the adequate configuration of each strategy. To show the gain ratio, we compute the ratio between the overhead obtained by the baseline strategy and the overhead of each strategy with the adequate configuration. Then, we use the variant of the percentage of work per remote steal

Fig 5.12 (percentage per remote steal = 50%) and Fig 5.13 (percentage per remote steal = 70%) show the overhead ratio compared to the baseline strategy for different parameters, according to the latency λ. Each figure plot the ratio for each strategy compared to the baseline overhead. Each column corresponds to the number of processors (16, 32 or 64). Each line corresponds to the latency W=(10^7, 5.10^7, 10^8, 5.10^8).

The results show that the overhead obtained by the three strategies are not far from each other. we can observe that the overhead obtained from the *dynamic probability*

victim selection is a little bigger than the other strategies for the different latency, but it remains close to them compared to the baseline. We also observe that the *dynamic victim selection strategy* is better than the *static victim selection strategy* with a small latency ($\lambda < 256$), which is not the case when $\lambda > 256$, *static victim selection strategy* gives the best average overhead.

The results show also that the overhead gain ratio obtained by the proposed strategies increases when the latency increase, is between 2 and 4 times better than the baseline strategy. This ratio can be improved using the variant of percentage per remote steal. the gain ratio can be 6 time better for the *dynamic victim selection strategy* and *static victim selection strategy*.

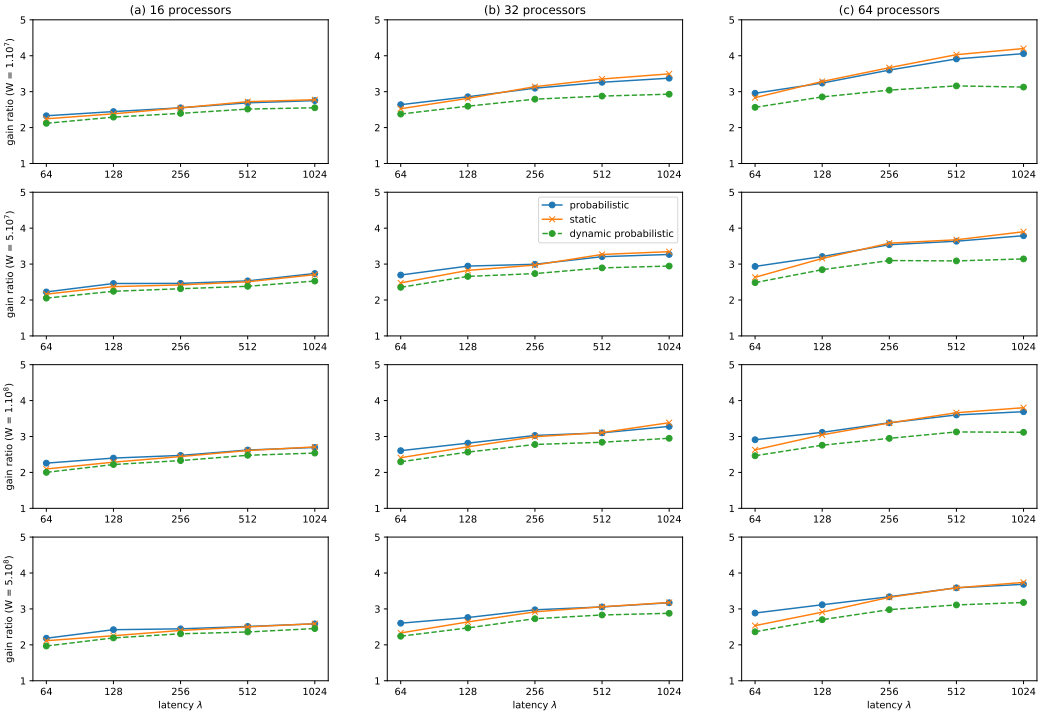


Fig. 5.12: The gain ratio compared to the baseline of the three strategies with the adequate configurations (percentage per remote steal = 50%)

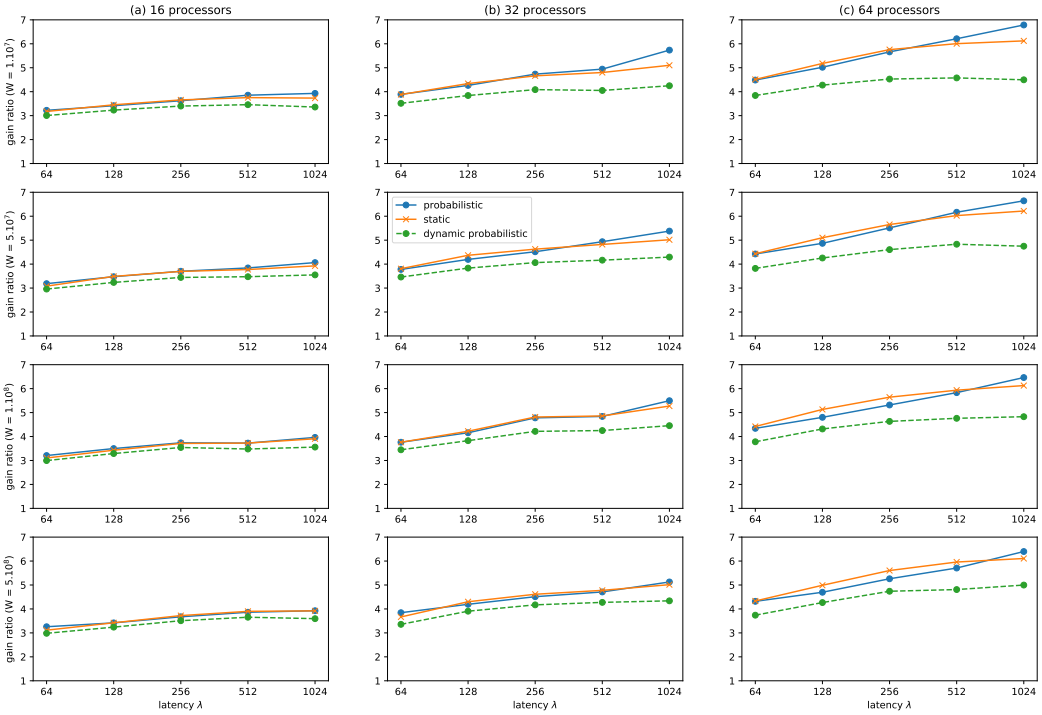


Fig. 5.13: The gain ratio compared to the baseline of the three strategies with the adequate configurations. (percentage per remote steal = 70%)

5.6 Conclusion

We presented in this chapter a new model of *Work Stealing* on a platform composed of two shared memory clusters linked via an interconnection network. the communication inside the cluster is much smaller than outside. This heterogeneity of communications creates new challenges for victim selection for *Work Stealing*. We proposed in this chapter different strategies to deal with the victim selection. The strategies are based on variants to control the steal outside the cluster. Using the simulation, we try to get an idea about the adequate variants to minimize the overhead. then, we introduce the variant to adapt the amount of work per remote steal, which can also reduce the average overhead with a good configuration, the average overheads could be reduced between 5 and 8 times better than the baseline overhead.

6 Reducing the interprocessors migrations of the EKG Algorithm

Contents

In this Chapter, we consider the scheduling problem of a set of k periodic implicit-deadline and synchronous tasks, on a real-time multiprocessor composed of m identical processors. It is known that the cost of migrations and preemptions has significant influence on global system performances. especially on modern machines where communications matters. The EKG algorithm which is optimal for $k = m$, can generate a great number of migrant tasks, but it has the advantage that each migrant task migrates between two processors only. Later, the EDHS algorithm has been proposed in order to minimize the number of migrant tasks of EKG. Although EDHS minimizes the number of migration compared to EKG, its drawback is the generation of additional preemptions caused by the migrations on several processors. In this paper we propose a new tasks allocation algorithm that aims to combine the advantages of EKG (migrations between two processors only) and those of EDHS (reduction of number of migrations).

This work has led to two publications [29, 30], and has been done in collaboration with El Mostafa Daoudi, Abdelmajid Dargham and Aicha Kerfali.

6.1 Introduction

In general, the uniprocessor scheduling is a one dimension problem because it deals with the temporal organization of tasks. That is to determine at what time to start,

pause (preempt) and resume execution of each task. As against the multiprocessor scheduling is a two-dimensional problem because in addition to the temporal organization of tasks it has to take care of their spatial organization. That is to determine when and on which processor execute each task. In both cases, the correct behavior of real-time systems depends not only on the operations they perform being logically correct, but also on the time at which they are performed. The real-time scheduling has to guarantee that all tasks are executed before their deadlines. In the multiprocessor/multi-core platforms filed, there are mainly two scheduling approaches for scheduling real-time tasks: global and partitioned scheduling.

Partitioned scheduling: In partitioned scheduling, tasks are organized in groups, and each task group is assigned to a specific processor. After their allocation to processors, the tasks are not allowed to migrate from one processor to another. When selected for execution, a task can only be dispatched to its assigned processor. On each processor the tasks are scheduled using standard known uniprocessor algorithms e.g. RM (Rate-Monotonic) or EDF (Earliest-Deadline-First) [58]. The main disadvantage of the partitioning approach is that the tasks allocation problem is analogous to the bin packing problem which is known to be NP-Hard [26, 39]. So a task cannot be assigned to any of the processors even if the total available capacity of the whole system is still large. When the individual task utilization is high, this waste could be significant, and in the worst-case only half of the system resource can be used.

global scheduling: unlike partitioned approaches that use a separate run-queue per processor, the global scheduling uses only a single queue for tasks that are ready to run. At each time, the m highest priority tasks are dispatched to any available processor according to a global priority scheme. The tasks can migrate from one processor to another which makes it possible to achieve a better use of the platform. Partitioned scheduling has gaps due to the absence of task migration from one processor to another. It is shown that a non schedulable system under partitioned policy can be scheduled if given the opportunity to unassigned tasks to run on multiple processors in global scheduling assuming that the cost of preemptions and migrations is neglected. This assumption is not realistic since this cost has an influence on global system performance. Several works have been proposed in the literature to reduce the number of preemptions and migrations [9, 68, 29].

Our contribution aims to combine the advantages of the EKG (migration between two processors only) and those of EDHS (reduction of migrant tasks). We proceed also into two steps to allocate the tasks: the first step is similar to the EDHS one, so we generate the same number of migrations as EDHS algorithm. In order to ensure the schedulability as EKG algorithm, our proposed algorithm avoid that a task migrates between more than two processors during the second step of the algorithm.

In order to achieve this goal, our key idea consists in reassigning the first allocated task of processors involved by migrations. In this case our algorithm achieves the optimality like EKG for k=m.

6.2 Presentation of the EKG algorithm

The system is composed of n periodic, implicit-deadline and synchronous tasks noted τ_1, τ_2,..., τ_n and m identical processors P_1,P_2,...,P_m. All the tasks cannot be executed in parallel and are independent. Each processor can't execute more than one task at any time. Each task τ_i has a period T_i (that is also the implicit deadline) and an execution time C_i. The ratio $C_i/T_i = U(\tau_i)$ defines the utilization rate of the task τ_i.

The EKG algorithm [8] cut the set of processors into groups each one is composed of k processors and limits migration within the same group. In addition, a task can migrate between two processors only. It allows scheduling optimally a set of periodic tasks with implicit deadline on m processors, when setting the parameter k equal to the number of processors ($k = m$).

Basic principle: Unlike partitioned algorithms, EKG allows tasks to run on two different processors (at different times, without parallelism). The algorithm is divided into two stages:

- Tasks allocation (offline): each task is assigned to one or two processors. The algorithm treats heavy and light tasks differently. A task τ_i is heavy if $C_i/T_i > SEP$, otherwise it is light where SEP is calculated as follows:

$$SEP = \begin{cases} 1, & if \ \ k = m \\ k/(k+1), & if \ \ k < m \end{cases}$$

  First, the algorithm assigns one heavy tasks to one processor where one processor is dedicated for one heavy (one per task). Then the lighter tasks are assigned to the remaining processors where several light tasks may be assigned to the same processor. To obtain a processor load of 1, some tasks can be split to run on two different processors (migrant task) belonging the same group. If a task is attempted to be assigned to the last processor in a group and it fails, then it is not split, but it is simply assigned to the first processor in a new group. This ensures that tasks in a group do not interact with tasks in another group.

- Tasks scheduling on processors (online): For each group, cutting the time into EKG intervals. An EKG interval is defined by two successive wake-up dates of tasks in the same group. It is similar to slots in the DP-Fair terminology [56], but limited to the tasks of the same group. Similar to DP-Fair, the work of migrant tasks should run for a time proportional to their utilization rate and duration of an interval [t_0,t_1] where t_0 denotes the time when a task arrives, and t_1 denotes the time when any task in that group arrives next. On an interval [t_0,t_1], if a task τ_i migrate between processors P_j and P_{j+1}, it will be splitted into subtasks τ_{11} and τ_{12} as shown in figure 6.1. At t_0 it runs on P_j for $U(\tau_{i1}) * (t_1 - t_0)$ time units and ends its execution at $timea$. Towards the end of the interval at $timeb$, the execution of the task restarts on P_{j+1} for a time duration of $U(\tau_{i2}) * (t_1 - t_0)$ units and ends its execution at t_1. The non migrant tasks are scheduled according to EDF on $]timea, timeb[$. After assignment of tasks, at runtime, our algorithm uses the same technique as the EKG algorithm to execute them on each processor.

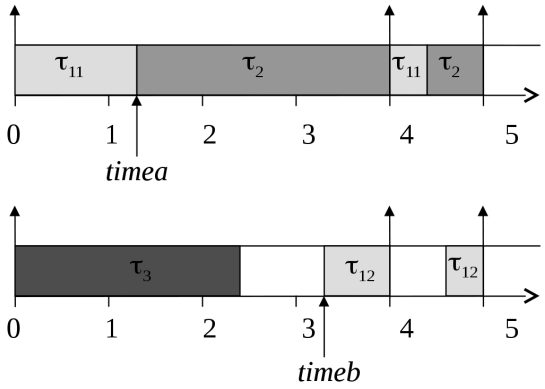


Fig. 6.1: Migration of the task τ_{11} between processors P_j and P_{j+1}

Reducing the number of preemption by mirroring: The mirror technique called (Mirroring) can be easily implemented by inverting simply τ_{i1} and τ_{i2}. This halves the number of preemptions. Figure 6.2 shows an execution with this technique. Note that it can be reused for other scheduling policies; it is the case for example DP-WRAP algorithm (similar to EKG)[11].

6.3 Presentation of EDHS Algorithm

EKG assigns migrant and non-migrant tasks simultaneously. This assignment produces several migrant tasks. To minimize the number of migrant tasks Kato et al. [51] have proposed the EDHS algorithm which proceeds into two separate steps: during the first one, the tasks are assigned according to a given partitioning algorithm in order to minimize the number of migrant tasks. The second step consists in

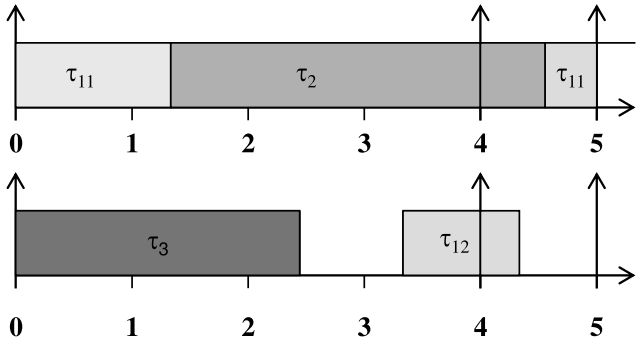


Fig. 6.2: The execution with mirroring technique

allocating, on multiple processors, migrant tasks (tasks that have not been allocated during the first step), according to a second algorithm, as shown in figure 6.3.

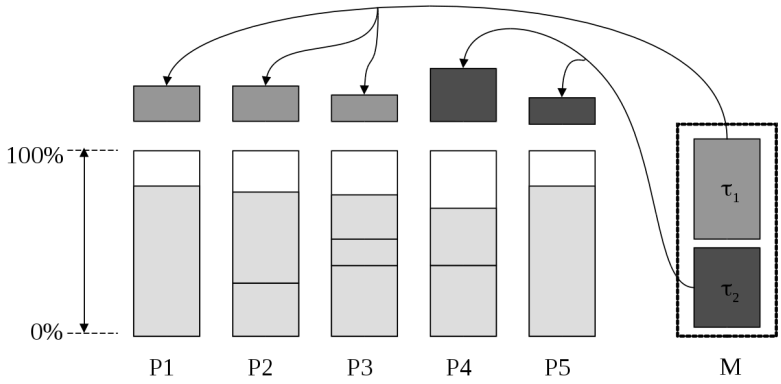


Fig. 6.3: Allocation of migrant tasks with EDHS algorithm

At runtime, the non-migrant tasks run according to EDF but migrant tasks run with high priority without overlap in time. When migrant task has exhausted its running time on a processor, it continues its execution immediately on the next processor and preempts the current task as shown in figure 6.4. Although EDHS minimizes the number of migration compared to EKG, its drawback is the generation of additional preemptions caused by the high priority of migrant tasks on several processors.

6.4 The proposed processor allocation heuristic

The system is composed of a periodic, implicit-deadline and synchronous tasks $\tau = \{\tau_1, \tau_2, \ldots \tau_n\}$. We assume that $\sum U(\tau_i) \leq m$ and $U(\tau_1) \geq U(\tau_2) \geq \ldots \geq U(\tau_n)$. The following notations are used in the remaining of the paper :

- M : denotes the set of the not allocated tasks. Initially $M = \tau$.

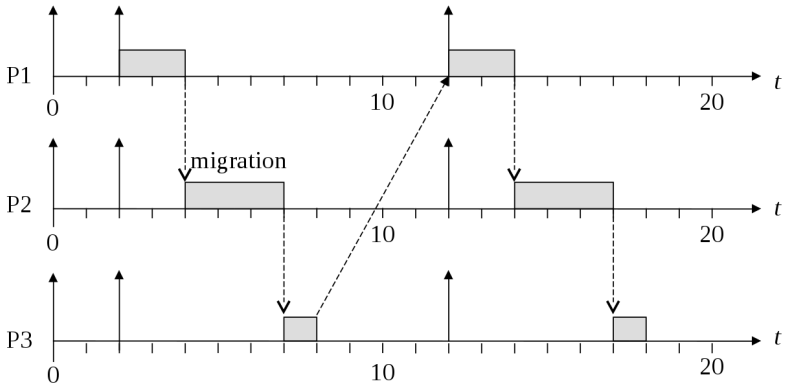


Fig. 6.4: Execution of migrant tasks with EDHS algorithm

- $\tau[j]$: denotes the set of allocated tasks to the j^{th} processor of the list of processors denoted by P_j, for $1 \leq j \leq m$. Initially, $\tau[j] = \emptyset$, for all j.

- $U[j]$: denotes the sum of the utilization rates of all tasks allocated to processor P_j.

- A processor P_j is full if $U[j] = 1$.

- $cap[j] = 1 - U[j]$: denotes the remaining capacity of processor P_j

In order to reduce the number of migrant tasks generated by the EKG algorithm, we proceed into two phases for allocating tasks to processors:

6.4.1 First phase

During the first phase, the allocation of tasks is done by applying one of the most known heuristics based on bin packing problem [49] as EDHS algorithm, namely First-Fit Decreasing, Best-Fit Decreasing and Worst-Fit Decreasing. These algorithms allocate the tasks by sorting them according to their utilization rates in the decreasing order.

- *First-Fit* This heuristic assigns the current task to the first processor in the list that has enough space. For all tasks, we begin the search from the first processor (algorithm 1).

- *Best-Fit*: This heuristic assigns the current task to most loaded processor with enough space. This heuristic is equivalent to the First-Fit algorithm but it sorts the processors according to their remaining capacities in increasing order.

- *Worst-Fit*. This heuristic selects the processor with the least loaded processor. This heuristic is equivalent to the First-Fit algorithm but it sorts the processors according to their remaining capacities in decreasing order.

Algorithm 1: First-Fit Decreasing heuristic

```
for each τ_i in M do
  j ← 1
  affecter ← 1
  while ( U[j] + U( τ_i ) > 1) and affecter = 1 do
    j ← j+1
    if j > m then
      affecter ← 0
    end if
  end while
  if affecter=1 then
    τ[j] ← τ[j] ⋃ {τ_i ; }
    U[j] ← U[j] + U(τ_i);
    M ← M\{τ_i } ;
  end if
end for
```

After this phase, the set of remaining tasks still not allocated (migrant tasks). To better explain the first phase, we take the following two examples:

- Example 1: In the following example, we consider the tasks system $\tau = (\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6)$ with U(τ_1)=0.7, U(τ_2) =0.6, U(τ_3) =0.6, U(τ_4) =0.4, U(τ_5) =0.4 and U(τ_6) =0.3. Figure 6.5 shows that the allocation of tasks using EKG algorithm gives rise to two migrant tasks τ_2 (τ_{21} and τ_{22}) and τ_4(τ_{41} and τ_{42}), but with the First-Fit Decreasing heuristic, there is no migrant task.

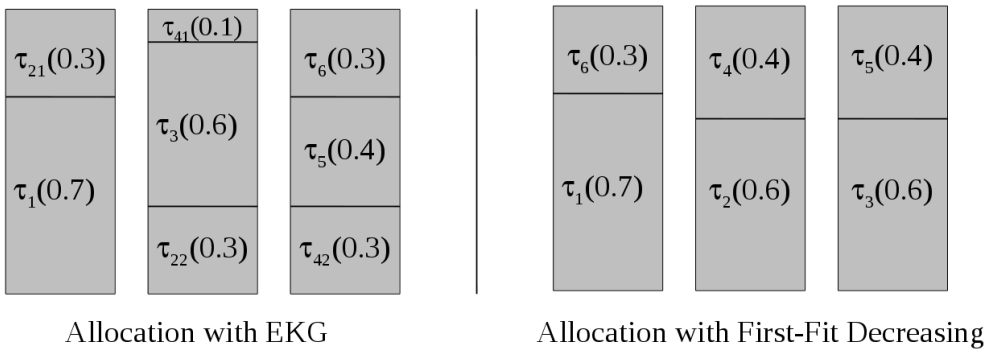


Fig. 6.5: Allocation with EKG and First Fit Decreasing heuristic on three processors

- Example 2: In the following example we will show that even if we apply the heuristics based on the bin packing problem, we cannot avoid the migration of

the tasks. We consider the system $\tau = (\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6, \tau_7)$ with U(τ_1)=0.9; U(τ_2)=0.8; U(τ_3)=0.5; U(τ_4)=0.3; U(τ_5)=0.3; U(τ_6)=0.15 and U(τ_7)=0.04). In figure 6.6, it is clear that the First-Fit Decreasing heuristic could not affect the task τ_5, so it must migrate on processors P_1, P_2 and P_3.

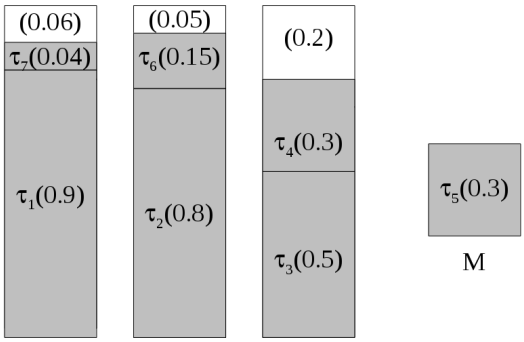


Fig. 6.6: Allocation with the First Fit Decreasing heuristic on three processors

6.4.1.1 Experimentations

On figure 6.7, we compare the number of migrations obtained with the EKG and the proposed algorithms by using the heuristics First-Fit, Best-Fit, Worst-Fit. For simulations, we have considered 10000 task systems and we have calculated the average of the number of migrations in a given interval, for each heuristic. The tasks are randomly generated with the respect of the schedulability condition that is $\sum U(\tau_i) \leq m$. Experimental results show that the heuristics First-Fit, Best-Fit, Worst-Fit reduce significantly the number of migrations. The reduction can reach 60% with the Best Fit and the First Fit heuristics.

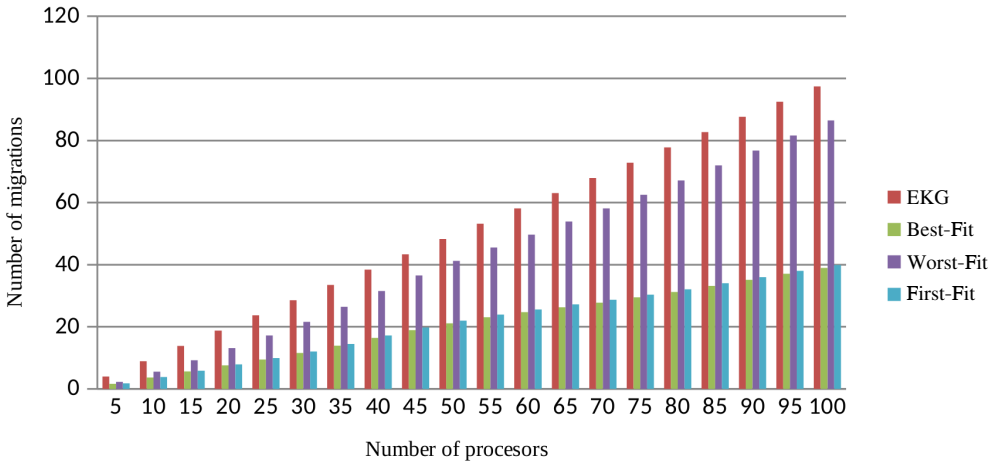


Fig. 6.7: Number of migration generated by each heuristic

6.4.2 Second phase

The second phase consists in allocating the set of remaining tasks (set of migrant tasks). Note that, by construction, the sum of utilization rates of migrant tasks is lower or equal to the sum of remaining processor capacities. Assume that, processors are sorted by decreasing order according to their remaining capacities, $cap[1] \geq cap[2] \geq \ldots \geq cap[m]$ and task τ_k can migrate on processors $P_1, P_2, \ldots, P_h$ which means that $cap[1]+cap[2]+\ldots+cap[h] \geq U(\tau_k)$ and $cap[1]+cap[2]+\ldots+cap[h-1] < U(\tau_k)$. the width of a time interval is denoted L.

Note that according to the first phase of the heuristic, the first task of each processor P_j, for $2 \leq j \leq h$, noted ρ_j, verifies $U(\rho_j) \geq U(\tau_k)$. The basic idea is to increase recursively the remaining processor capacities as follows:

- Subdividing the task ρ_2 into two subtasks ρ_{21} and ρ_{22}, such that U(ρ_{22}) = cap[1] and U(ρ_{21}) = U(P_2[1])-cap[1]

- Assigning ρ_{22} to P_1. In this case P_1 becomes full and the capacity of P_2 is increased with U(ρ_{22}) (cap[2]= cap[2]+ U(ρ_{22}).) Thus, task ρ_{21} becomes a migrant task on processors P_1 and P_2. At runtime:

  - Processor P_2 starts its execution by task P_2[1] during U(ρ_{21})*L.

  - Processor P_1 ends its execution by task P_2[1] during U(ρ_{22})*L.

- Recursively, the same process is repeated between processors P_{j-1} and P_j, for $2 < j < h$, where the task ρ_j is subdivided into two subtasks ρ_{j1} et ρ_{j2}, such that U(ρ_{j2}) = cap[j-1]. In this case cap[j]= cap[j]+ U(ρ_{j2}). At runtime:

  - Processor P_j starts its execution by task ρ_j during U(ρ_{j1})*L.

  - Processor P_{j-1} ends its execution by task ρ_j during U(ρ_{j2})*L.

- After this process, $cap[h-1] < U(\tau_k)$ and $cap[h]+cap[h-1] \geq U(\tau_k)$, then task τ_k migrates only between processors P_{h-1} and P_h in the following manner: τ_k is subdivided into two subtasks τ_{k1} and τ_{k2}, such that U(τ_{k1}) = cap[h-1] and U(τ_{k2}) = U(τ_k) - U(τ_{k1}). P_{h-1} starts its execution by task τ_{k1} during U(τ_{k1})*L and P_h ends its execution by task τ_{k2} during U(τ_{k2})*L.

With this reallocation, the number of migrations is still the same and each migrant task, migrates between two processors only. In this case our proposed algorithm generates lower migrant tasks than EKG.

Algorithm 2: allocation of migrant tasks

for each τ_k in M **do**
  sort in decreasing order the list of processors according to their remaining capacities
  calculate h such as $cap[1] + \ldots + cap[h] \geq U(\tau_k)$ and $cap[1] + \ldots + cap[h-1] < U(\tau_k)$.
  j $\leftarrow$ 1
  while $j < h - 1$ **do**
    Subdivide ρ_{j+1} into tow subtasks $\rho_{(j+1)1}$ and $\rho_{(j+1)2}$ such that $U(\rho_{(j+1)2}) = cap[j]$ and $U(\rho_{(j+1)1}) = U(\rho_{j+1}) - cap[j]$
    Assign $\rho_{(j+1)2}$ to P_j then $cap[j+1] = cap[j+1] + cap[j]$ and P_j becomes full.

    Processor P_{j+1} starts its execution by executing task ρ_{j+1} during $U(\rho_{(j+1)1}) * L$
    Processor P_j ends its execution by executing task ρ_{j+1} during $U(\rho_{(j+1)2}) * L$.
    j $\leftarrow$ j+1
  end while
  /* τ_k migrates only between P_{h-1} and P_h. */
  Subdivide τ_k into two subtasks τ_{k1} and τ_{k2}, such that $U(\tau k1) = cap[h-1]$ and $U(\tau_{k2}) = U(\tau_k) - U(\tau_{k1})$.
  Assign τ_{k1} to P_{h-1} and τ_{k2} to P_h
  Processor P_{h-1} starts its execution by task τ_{k1} during $U(\tau_{k1}) * L$
  Processor P_h ends its execution by task τ_{k2} during $U(\tau_{k2}) * L$.
end for

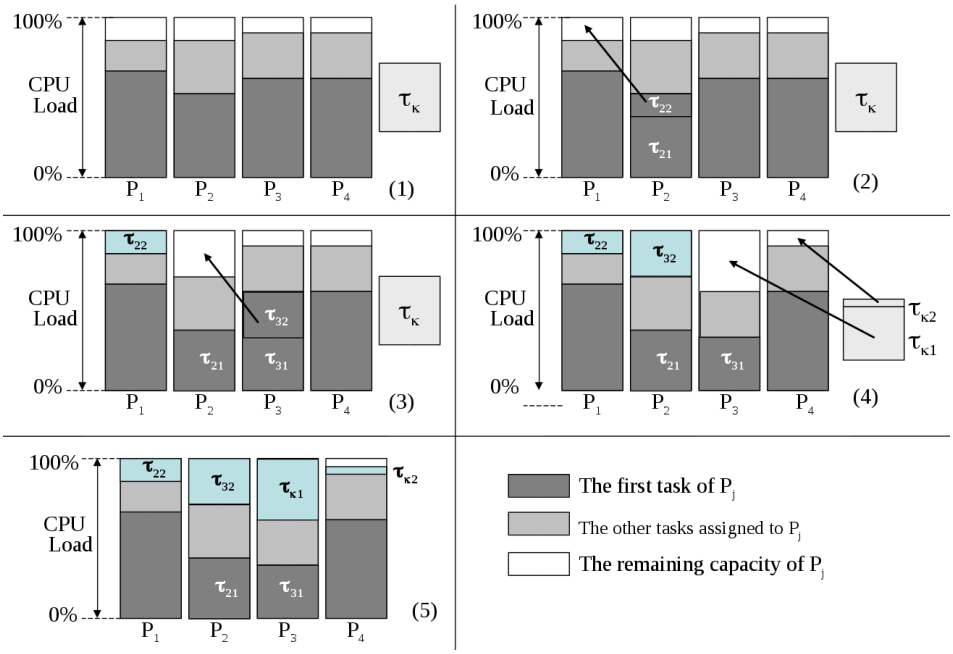


Fig. 6.8: Allocation of the migrant task τ_k with our proposed algorithm for h=4

Figure 6.8 shows the steps of the algorithm in order to allocate a migrant task τ_k to two processors only instead to allocate it to four processors.

6.5 Conclusion

In this work we have proposed a new semi-partitioned algorithm that reduces the number of migrations like EDHS and limits migrations between two processors only like EKG. The proposed algorithm is designed into two steps. During the first step we used the well-known bin packing heuristics [49] (the First Fit Decreasing, the Best Fit Decreasing, and the Worst fit Decreasing). This step is similar to the first step of the EDHS and it consists of reducing the number of migrant tasks compared to EKG. During the second step of the algorithm, we proposed a new technique that allocates the migrant tasks. Our key idea consists in increasing the number of migrant tasks, each one migrates on two processors, while keeping the same number of migrations: instead to migrate a task between h processors (h-1 migrations), we migrate (h-1) tasks each one between two processors. This reallocation has the advantage that we remain in the same condition of optimality of the EKG for m=k. Experimental simulations show that the number of migrations, compared to EKG, is significantly reduced. This reduction can reach 60%.

7 Microservice containers scheduling

Contents

The work presented in this chapter is the result of a two months internship at the University of São Paulo in collaboration with Alfredo Goldman and Denis Trystram.

7.1 Introduction

Microservices have been proposed recently in the field of Software Engineering for building efficient applications [52, 13]. The idea is to decompose the applications into independent modules (called the *microservices*) which are implemented and operated as light and thus, efficient sub-systems. Each one performs a specific task that may also be useful for other applications.

An application uses Microservices to offer several services to various users as shown in Figure 7.1. Each application is performed using one or more Microservices that communicate between them to accomplish a required functionality. The Microservices-based applications can be seen as a set of workflows as shown in Figure 7.2, each one being composed of sets of Microservices that work separately and communicate to perform a related service in the application. The application has also a user interface

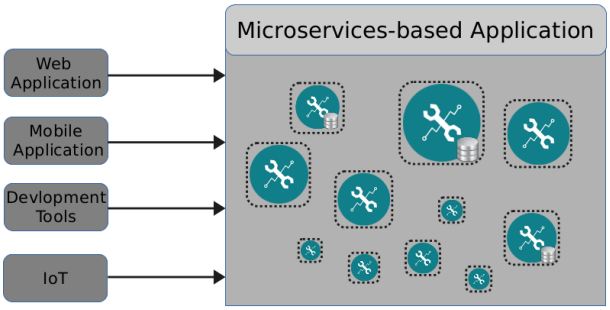


Fig. 7.1: Microservices-based application : cliens can be another applications (Web application, Mobile applications, IoT)

(for instance, a Web or a mobile application, a command line, etc.) which allows the users to call the application.

The user interface contains several types of requests, each one is linked to its specific workflow inside the application.

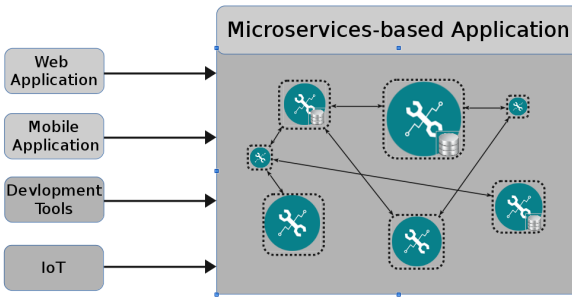


Fig. 7.2: Set of Microservices that work together inside an application

As pointed out in [32], Microservices-based applications have a lot of advantages. First, the Microservices may be implemented in different languages and technologies (Python, Java, XML, etc.) [13]. Since the Microservices are independent, each one may be updated separately, which facilitates the update of the whole application [13]. For the same reason, the application is more reliable since when a microservice breaks down, it does not affect the whole application. The second advantage is the scalability of Microservices-based applications. When several users use the application for the same service at the same time (it is *overload*). The application needs to be scaled up, which requires just scaling the set of Microservices that perform the concerned service. In case of overload on some Microservices (which means several requests at same time). The are two methods for scaling the microservice [59], namely vertical and horizontal scalings. Vertical scaling aims at adding more resources to

the overloaded Microsservice, while the horizontal scaling creates another instance of the overloaded Microservice. The load will be split between the instances of the microservice. While both methods accomplish the same goal, scaled applications using horizontal scaling is more simple, since the creation of new instances is not too complicated [21].

To benefit from the various advantages of Microservices, the application and their Microservices should be managed automatically. Currently, Kubernetes [54] and Docker [6] become the most popular managers of Microservices and Microservices-based applications. Docker is an open source software that isolates the Microservices inside a container. Containers are a mechanism of lightweight virtualization to isolate and control some processes with less overhead [15]. A container is lightweight because it does not emulate the physical hardware like a Virtual Machine. Kubernetes is also an open source software container cluster manager, it manages the different Microservices on the cluster. It facilitates the deployment of the application for managing the different components of the application. It offers also an auto-scaling mechanism that manages the different Microservices and resources. It uses the horizontal auto-scaling based on the creation of new instances if needed. Kubernetes also offers a simple scheduler to assigned different instances to available machines.

Auto-Scaling a Microservices-based application consists in two steps. First, controlling the load in order to detect the overload and deciding to create instances (or delete some instances). Second, determining a machine for these new instances. For the first step, Kubernetes is based on the average CPU utilization and a given threshold to scale application. When the average CPU utilization of a Microservices instance exceeds a given threshold, it creates the needed instances to adapt the average CPU utilization. The same thing occurs when the CPU utilization is very small, Kubernetes can delete some instances in order to free some resources.

The second step is to find a machine for executing those instances. Kubernetes offers a sample algorithm to assign new instances to available nodes. The algorithm selects a node for the instance in a 2-step operation, filter out nodes that doesn't have enough CPU or Memory, and choose one of the nodes that were not filtered out based on simple function (last loaded, less loaded ...) [54]. When the scheduler does not find an available machine for an instance, two cases are possible, either it refuses this instance and the Microservice remains overloaded, or the scheduler requests a new machine in addition in order to place the new instance.

The study conducted in this chapter aims at creating and analyzing new models for scheduling instances on machines. There are only few related studies since the domain is recent. We study in particular two variants of model: a static model that performs the first allocation of containers over machines, and a dynamic model to

update the scheduling when a scaling event (create or/and delete instances) occurs, The purpose is to propose a realistic preliminary model and we leave the scheduling analysis and practical implementations to future works. However, our models are based on several dedicated studies and experiences on actual systems done during a stay in Sao Paulo (Brazil).

7.2 Description

Applications based on Microservices have a specific deployment. Several steps are required to effectively run and manage the application. First, Microservices are isolated on a container [5]. The second step is to distribute Microservices on Pods with kubernetes, and finally manage different pods on machine on two step, auto-scaling and scheduling of the application. Each of these steps uses a specific technology. The objective of this section is to present these steps and the underling technologies.

7.2.1 Containerization of Microservices

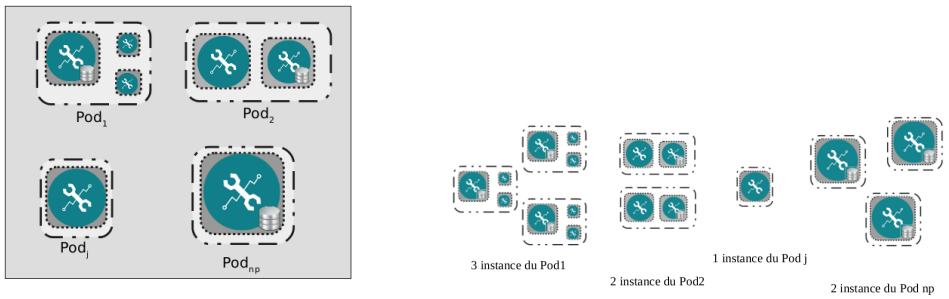
Each Microservice of the application must be isolated. The best solution is the containerization. Containers are a mechanism of lightweight virtualization to isolate and control some processes with less overhead. A container is lightweight because it does not emulate the physical hardware like a standard virtual machine [78] Currently, several applications provide containerization like OpenVZ [24], and Docker[1] [63]. The most popular at the high-level is Docker which is open source software [6].

7.2.2 Microservices manager

Manage the set of Microservice of an application consists in assigning a machine to each containerized Microservice, allowing the communications between the Microservices and auto-scaling them. Effectively managing an application requires a Container Manager. Kubernetes is an open source container cluster manager that offers the corresponding needed options.

Kubernetes use Pod which is the smallest deployable unit which can be created and managed by Kubernetes. Pod mechanism can be considered as another level of containerisation that contains one or more containerized Microservices. Containerized

[1] https://www.docker.com



(a) Containerized Microservices within a pod

(b) Set of instances of Pods

Fig. 7.3: (a) the distribution of Microservices over Pods, (b) Kubernetes create different instances of each Pod depend on the load or the initial configuration

applications within a pod can see each others' processes, access the same IP network, and share the same Node.

For each Pod, kubernetes creates one or more instance and assigned it to a machine. It also allows the communication between the different Pod instances, and offers an auto-scaling to manage the different Pod instances. Kubernetes uses a replication Controller to create, remove and manage different instance replicas for each Pod. Replication controller also interacts to perform the auto-scaling, it controls the CPU utilization to decide how many replica instances need to be created or deleted to keep the cluster on desired limits. When the Replication Controller creates a new instance, the scheduler assigns a machine for this new instance if available. We give the principle of the auto-scaling algorithm and scheduling as follows.

7.2.3 Auto-scaling and Scheduling

The idea of Microservices-based application is to separate the deployment of the application. Each Containerized Microservices within a pod is deployed separately. This option offers a more flexible auto-scaling. In case of an overload on such Microservices in the application, the solution is to create a new instance of the pod that contains this Microservice, then, find a machine for running this instance and dispatch the requests between the different instances in the system.

Kubernetes offers an auto-scaling to manage the different instances of Pod. Its algorithm is based on the average CPU utilization percentage to decide how many replica pods to create or to delete in order to keep it on desired limits.

The algorithm takes as input the threshold and the set of active instances for each Pod. At each interval τ, and for each Pod j, the algorithm computes the

sum of CPU utilization for all active instances of this Pod. Then, it computes the number of instances needed by dividing this sum by the threshold. Finally, the replication controller receives the number of instances needed, and compute how many instances should be created or deleted based on the number of running instances.

7.2.4 Scheduling

When the auto-scaler decides to create some instances, the role of the scheduler is to find for each instance a machine. The schedule should respect several configurations (one for each instance) in order to find sufficient CPU and Memory for running each instance. Kubernetes uses a sample algorithm that filters out nodes that does not have enough CPU or Memory, and choose one of the nodes that were not filtered out based on simple function (for instance, last loaded or less loaded or anything else that makes sense)

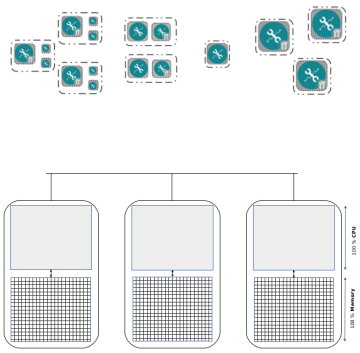


Fig. 7.4: Set of instances that need to be scheduler over machines

7.3 Model

We present below a modelization of the problem of scheduling microservices on a distributed platform composed of m identical processors.

We consider an application with k Microservices containerized on pods. We use in our model the sample distribution where each Pod contains one Microservice. Each Microservice has its own CPU request Q^{cpu} and its memory request Q^{mem} (we assume that Q^{cpu} and Q_{mem} take fractional value between 0 and 1). During the execution, kubernetes creates one or more instances for each container (Pod). Each instance inherits the same CPU and memory request from its container. The CPU and Memory of each node are considered as fractional. The idle CPU part (Resp. idle Memory) is denoted by A^{cpu} (Resp. A^{mem}).

The role of the scheduler is to schedule the first instances of all containers on the machine, and interact with each replication control decision (scale event) to put the new instances on the machines or remove it from a machine. The following section presents the static model to perform the initial placement and the dynamic model which re-schedules the instances when a scale event occurs.

7.3.1 Static Model

We present in this section the scheduling model to perform the first allocation of containers over machines. Our model is composed of:

- k Microservices containerized in p pods ($p = k$, one microservice per pod)
- n Containers instance indexed by i ($n \geq k$ since we can have several instances of the same microservice), (n could be the maximum number of all instances), each instance has :
  - Q_i^{cpu} : the CPU needed by n_i
  - Q_i^{mem} : the Memory needed by n_i
- m Machines indexed by j

The allocation of nodes over the machines is defined by the matrix x_{ij} as:

$$x_{ij} = \begin{cases} 1 & \text{if } n_i \text{ in on } m_j \\ 0 & \text{otherwise} \end{cases}$$

Then, we define A_j^{cpu} and A_j^{mem} for each machine j as follows (remember here that the A are expressed as the proportion of their need):

- A_j^{cpu} : the CPU available on m_j where:

$$A_j^{cpu} = 1 - \sum_i x_{ij} * Q_i^{cpu}$$

- A_j^{mem} : the Memory available on m_j where:

$$A_j^{mem} = 1 - \sum_i x_{ij} * Q_i^{mem}$$

The objective is to compute x_{ij} which satisfy the following constraints:

$$\forall\, j \leq m \quad A_j^{cpu} \geq 0 \quad \text{and} \quad A_j^{mem} \geq 0$$

while minimizing the number of machines.

7.3.2 Related models

There are several existing models that may be adapted for our problem. We briefly present below the most relevant ones.

- The first one is the famous paper of Graham [38] on bounds for multiprocessor scheduling with resource constraints. Here the additional memory requirements may be viewed as an extra resource constraint:

  Given a set of tasks $T = \{T_1, ..., T_r\}$ and a set of resources $R = \{R_1, ..., R_s\}$, let denote by $R_j(T_i)$ the resource R_j required by task T_i.

  The principle is to use a list algorithm (greedy) $L = \{T_{i_1}, ..., T_{i_r}\}$ that governs the order in which the tasks are chosen.

  The objective is to minimize $\omega(L)$, i.e. the maximum completion time for m processors executing T according to L.

  The main result is the approximation bound under the assumption of ar number of processors:

  $$\omega/\omega^* \leq s + 1$$

  In our case, we traget a 3-approximation considering both CPU and memory constraints as resources.

- Another direction is to refer to the classical vector-packing problem. Kellerer et al. established in [53] an approximation algorithm with absolute worst case performance ratio 2 for two-dimensional vector packing. Again, we can see our problem as a 2 dimensional packing (corresponding to CPU and memory). in this paper, the authors study the two-dimensional vector packing problem, they propose an $O(nlogn)$ time algorithm for two-dimensional vector packing with absolute performance guarantee 2. The idea of their algorithm to classify the different items in order to find the best allocation of items on bins. This problem has the same features as our model, and we can apply the same algorithm.

- In the same vein, we may use an improved approximation for vector bin packing [14]: Bansal studied the 2-dimensional vector bin packing problem, leading to an asymptotic approximation guarantee of $1 + ln(1.5) + \epsilon$ using a polynomial-time algorithm, they also derived an almost tight $(1.5 + \epsilon)$ absolute approximation guarantee.

7.3.3 Dynamic Model

All the previous works are limited to the static assumption. During the execution, x_{ij} is already defined, but the auto-scalar may decide (irregularly in time) to update the available instances, adding or deleting (or both) some instances. Then, the list of containers changes over time, and we need to update the allocation of the new containers list .

We model the costs of manage the containers on machines as:

- τ_c : the cost of placing a container on a machine.
- τ_s : the cost of deleting a container on a machine (equal to 0)
- τ_m : the cost of moving a container from a machine to another, since τ_s is 0, $\tau_m = \tau_c$

The objective is to reschedule the instances in order to **minimize the number of machines** and at the same time, **minimize the total overhead** related to instance management.

The mathematical formulation is as follows:

- Recompute x_{ij} for the new instance list (with : $\forall\ j \leq m \quad A_j^{cpu} \geq 0 \quad$ and $\quad A_j^{mem} \geq 0$).
- Minimize τ_c
- Minimize m (the number of machines)

7.3.4 Conclusion

We investigate in this chapter a preliminary analysis of the problem of microservices scheduling on cloud computing. We first present a detail description of microservice-based application, and their challenges, then we propose two models of scheduling of the microservices on cloud computing. The models simplify the problem and open different research directions able to provide effective solutions. The next step is to implement the different solutions in actual microservice-based application, and study the performances in order to identify the limit of each of them.

8 Conclusion

The High-Performance Computing (HPC) domain is evolving rapidly, many supercomputers and cloud computing systems are available at different levels of power and use. These systems are many common features, they are composed of multiple parallel and heterogeneous sub-systems, which are interconnected in through a complicated network. At the same time, the challenges of the HPC community are becoming increasingly complex (to cite only the most important ones: taking advantage of system power, efficient resource management, energy consumption , ...). One of the central and most complex challenge is the scheduling problem which aims at determining the allocation of the tasks to the different processors, and then, the local sequencing of these tasks.

In this thesis, we are interested in the question of scheduling problems taking into account communication in different contexts. The first context concerns an on-line scheduling problem on a distributed platform, including communication, with an extension to more heterogeneous platforms composed of two-clusters with different levels of communication. The communications between the processors influence significantly the scheduling and they require a careful study and analysis in order to understand the impact of these communications (when they should occur and what amount of data should be concerned). The second context concern the on-line real-time scheduling in a platform with communications. The communication reacts to the task migration level, which can impact the scheduling process. The third context dealt with a new scheduling problem that concerns the microservices-based applications on cloud platform.

In a first work, we were able to study the overall impact of the communication time on the Makespan objective using a theoretical analysis. Through this analysis, we obtained a novel expression of the expected Makespan of a *bag of independent tasks* and on a *task graph with precedences* scheduled by Work Stealing in a platform of processors including communication delays. We also extended this analysis one step further, by providing a bound on the probability to exceed the bound of the makespan. This work provided also a deep understanding of the work stealing in the basic homogeneous setting.

Then, We were interested in the *Work Stealing* algorithm on more complex environments, including non homogeneous ones. However, the mathematical analysis was not effective because of the heterogeneity (using a classical worst case analysis), and we had to rely on simulations to analyze the problem. For this reasons, we developed a lightweight **PYTHON** simulator. Our simulator is quite flexible and easy to use and update. Through this simulator, we were able to assess the tightness of the first analysis (on the homogeneous case). We showed by comparing the theoretical bound and the experimental results. We observed moreover that our bound (established on worst-case analysis) is four to five times greater than the simulation results and it is stable for all the tested values. By using traces of execution, we quantified the various approximations that are made in the analysis and swe uggested where the analysis could be made more accurate.

For the extension of *Work Stealing* on two-clusters, the challenge of the algorithm is how to select the victim (to steal work) and avoid the expensive external steals, while the external steals are mandatory to balance the work between the two clusters. In this sense, we proposed different strategies for solving this problem, the strategies are effective but their parameters must be carefully chosen and tuned. Through the simulation, we studied and compared these strategies.

The second work concern the study of two off-line scheduling problems on different distributed platforms. Firstly, we studied the off-line real-time scheduling problem in a multiprocessor including communications. In this work we proposed a new algorithm that reduces the number of migrations and limits migrations between two processors. Our key idea consists in increasing the number of migrant tasks, each one migrates on two processors. Experimental simulations show that the number of migrations, compared to other algorithms, is significantly reduced. This reduction can reach 60%. Secondly, we focus on the new scheduling problem of microservice-based application on cloud. The microservices (components of a large application) can also be executed (and duplicated) on different machines. each microservice instance requires a percentage of CPU and memory (depending on the micro-services). The allocation of these microservices instances is clearly a new challenge in the scheduling area. In our work, we proposed a model for microservices applications, which is used to perform some theoretical analyses in order to give directions to find approximations solutions. The focus of this part was put on the possible models, each one corresponding to a different optimization problem. We then discuss the most relevant existing literature and we proposed an adequate problem which consider the dynamic character of the problem.

Bibliography

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. „The Data Locality of Work Stealing“. In: *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '00. Bar Harbor, Maine, USA: ACM, 2000, pp. 1–12 (cit. on p. 10).

[2] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. „The Data Locality of Work Stealing“. In: *Theory of Computing Systems* 35.3 (June 2002), pp. 321–347 (cit. on p. 10).

[3] K. Agrawal, C. E. Leiserson, and J. Sukha. „Executing task graphs using work-stealing“. In: *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 2010, pp. 1–12 (cit. on p. 11).

[4] Susanne Albers. „Energy-Efficient Algorithms“. In: *Commun. ACM* 53.5 (May 2010), pp. 86–96 (cit. on p. 2).

[5] Marcelo Amaral, Jorda Polo, David Carrera, et al. „Performance evaluation of microservices architectures using containers“. In: *2015 IEEE 14th International Symposium on Network Computing and Applications*. IEEE. 2015, pp. 27–34 (cit. on p. 90).

[6] C. Anderson. „Docker [Software engineering]“. In: *IEEE Software* 32.3 (May 2015), pp. 102–c3 (cit. on pp. 89, 90).

[7] J. H. Anderson, J. P. Erickson, U. C. Devi, and B. N. Casses. „Optimal semi-partitioned scheduling in soft real-time systems“. In: *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*. Aug. 2014, pp. 1–10 (cit. on p. 12).

[8] B. Andersson and E. Tovar. „Multiprocessor Scheduling with Few Preemptions“. In: *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)*. Aug. 2006, pp. 322–334 (cit. on pp. 12, 77).

[9] Dalia Aoun, Anne-Marie Déplanche, and Yvon Trinquet. „Pfair scheduling improvement to reduce interprocessor migrations“. In: *16th International Conference on Real-Time and Network Systems (RTNS 2008)*. Ed. by Giorgio Buttazzo and Pascale Minet. Isabelle Puaut. Rennes, France, Oct. 2008 (cit. on p. 76).

[10] Humayun Arafat, James Dinan, Sriram Krishnamoorthy, Pavan Balaji, and P. Sadayappan. „Work Stealing for GPU-accelerated Parallel Programs in a Global Address Space Framework“. In: *Concurr. Comput. : Pract. Exper.* 28.13 (Sept. 2016), pp. 3637–3654 (cit. on p. 11).

[11]Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. „Thread Scheduling for Multiprogrammed Multiprocessors“. In: *In Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), Puerto Vallarta*. 2001, pp. 119–129 (cit. on pp. 10, 15, 18, 36).

[12]U. Awada and A. Barker. „Improving Resource Efficiency of Container-Instance Clusters on Clouds“. In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. May 2017, pp. 929–934 (cit. on p. 13).

[13]K. Bakshi. „Microservices-based software architecture and approaches“. In: *2017 IEEE Aerospace Conference*. Mar. 2017, pp. 1–8 (cit. on pp. 87, 88).

[14]Nikhil Bansal, Marek Eliás, and Arindam Khan. „Improved Approximation for Vector Bin Packing“. In: *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*. 2016, pp. 1561–1579 (cit. on p. 95).

[15]R. K. Barik, R. K. Lenka, K. R. Rao, and D. Ghose. „Performance analysis of virtual machines and containers in cloud computing“. In: *2016 International Conference on Computing, Communication and Automation (ICCCA)*. Apr. 2016, pp. 1204–1210 (cit. on p. 89).

[16]Michael A. Bender and Michael O. Rabin. „Online Scheduling of Parallel Programs on Heterogeneous Systems with Applications to Cilk“. In: *Theory of Computing Systems* 35.3 (2002), pp. 289–304 (cit. on pp. 3, 10).

[17]Petra Berenbrink, Tom Friedetzky, and Leslie Ann Goldberg. „The Natural Work-Stealing Algorithm is Stable“. In: *SIAM Journal on Computing* 32.5 (2003), pp. 1260–1279. eprint: `http://dx.doi.org/10.1137/S0097539701399551` (cit. on p. 10).

[18]Veeravalli Bharadwaj, Thomas G. Robertazzi, and Debasish Ghose. *Scheduling Divisible Loads in Parallel and Distributed Systems*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1996 (cit. on p. 36).

[19]Raphaël Bleuse, Safia Kedad-Sidhoum, Florence Monna, Grégory Mounié, and Denis Trystram. „Scheduling independent tasks on multi-cores with GPU accelerators“. In: *Concurrency and Computation: Practice and Experience* 27.6 (2015), pp. 1625–1638 (cit. on p. 2).

[20]Robert D. Blumofe and Charles E. Leiserson. „Scheduling Multithreaded Computations by Work Stealing“. In: *J. ACM* 46.5 (Sept. 1999), pp. 720–748 (cit. on pp. 10, 33).

[21]E. Casalicchio and V. Perciballi. „Auto-Scaling of Containers: The Impact of Relative and Absolute Metrics“. In: *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. Sept. 2017, pp. 207–214 (cit. on p. 89).

[22]Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. „Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms“. In: *Journal of Parallel and Distributed Computing* 74.10 (June 2014), pp. 2899–2917 (cit. on p. 35).

[23]Daniel Casini, Alessandro Biondi, and Giorgio Buttazzo. „Semi-Partitioned Scheduling of Dynamic Real-Time Workload: A Practical Approach Based on Analysis-Driven Load Balancing“. In: June 2017 (cit. on p. 12).

[24]J. Che, C. Shi, Y. Yu, and W. Lin. „A Synthetical Performance Evaluation of OpenVZ, Xen and KVM“. In: *2010 IEEE Asia-Pacific Services Computing Conference*. 2010, pp. 587–594 (cit. on p. 90).

[25]Lin Chen, Deshi Ye, and Guochuan Zhang. „Online Scheduling on a CPU-GPU Cluster“. In: *Theory and Applications of Models of Computation*. Ed. by T-H. Hubert Chan, Lap Chi Lau, and Luca Trevisan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–9 (cit. on p. 3).

[26]Edward G Coffman Jr, Michael R Garey, and David S Johnson. „An application of bin-packing to multiprocessor scheduling“. In: *SIAM Journal on Computing* 7.1 (1978), pp. 1–17 (cit. on p. 76).

[27]Michel Cosnard and Denis Trystram. *Algorithmes et Architectures Parallèles*. InterEditions, Collection IIA, 1993 (cit. on pp. 2, 16, 36).

[28]Van-Dat Cung, Vincent Danjean, Jean-Guillaume Dumas, et al. „Adaptive and Hybrid Algorithms: classification and illustration on triangular system solving“. In: *Transgressive Computing 2006*. Ed. by Jean-Guillaume Dumas. Grenade, Spain: Copias Coca, Madrid, Apr. 2006, pp. 131–148 (cit. on p. 37).

[29]E. M. Daoudi, A. Dargham, A. Kerfali, and M. Khatiri. „Reducing the inter processor migrations of the DP-WRAP scheduling“. In: *2015 IEEE/ACS 12th International Conference of Computer Systems and Applications (AICCSA)*. Nov. 2015, pp. 1–6 (cit. on pp. 75, 76).

[30]El Mostafa Daoudi, Abdelmajid Dargham, Aicha Kerfali, and Mohammed Khatiri. „Reducing the Interprocessors Migrations of the EKG Algorithm“. In: *Scalable Computing: Practice and Experience* 19.3 (2018), pp. 293–300 (cit. on p. 75).

[31]James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. „Scalable Work Stealing“. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC '09. Portland, Oregon: ACM, 2009, 53:1–53:11 (cit. on pp. 11, 16, 62).

[32]Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, et al. „Microservices: Yesterday, Today, and Tomorrow“. In: *Present and Ulterior Software Engineering*. Ed. by Manuel Mazzara and Bertrand Meyer. Cham: Springer International Publishing, 2017, pp. 195–216 (cit. on p. 88).

[33]Maciej Drozdowski and Paweł Wolniewicz. „Experiments with Scheduling Divisible Tasks in Clusters of Workstations“. In: *Euro-Par 2000 Parallel Processing*. Ed. by Arndt Bode, Thomas Ludwig, Wolfgang Karl, and Roland Wismüller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 311–319 (cit. on p. 36).

[34]Richard Durrett. „Probability: theory and examples“. In: (1996) (cit. on pp. 27, 28).

[35]I. Filip, F. Pop, C. Serbanescu, and C. Choi. „Microservices Scheduling Model Over Heterogeneous Cloud-Edge Environments As Support for IoT Applications“. In: *IEEE Internet of Things Journal* 5.4 (Aug. 2018), pp. 2672–2681 (cit. on p. 12).

[37]Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. „The Implementation of the Cilk-5 Multithreaded Language“. In: *SIGPLAN Not.* 33.5 (May 1998), pp. 212–223 (cit. on pp. 16, 62).

[38]M. R. Garey and Ronald L. Graham. „Bounds for Multiprocessor Scheduling with Resource Constraints“. In: *SIAM J. Comput.* 4.2 (1975), pp. 187–200 (cit. on p. 94).

[39]Michael Garey and David Johnson. „Computers And Intractability: A Guide to the Theory of NP-Completeness“. In: Jan. 1979 (cit. on pp. 9, 76).

[40]Nicolas Gast and Gaujal Bruno. „A Mean Field Model of Work Stealing in Large-scale Systems“. In: *SIGMETRICS Perform. Eval. Rev.* 38.1 (June 2010), pp. 13–24 (cit. on p. 10).

[41]Thierry Gautier, Xavier Besseron, and Laurent Pigeon. „KAAPI: A Thread Scheduling Runtime System for Data Flow Computations on Cluster of Multi-processors“. In: *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*. PASCO '07. London, Ontario, Canada: ACM, 2007, pp. 15–23 (cit. on pp. 16, 62).

[42]Ronald L Graham. „Bounds for certain multiprocessing anomalies“. In: *Bell System Technical Journal* 45.9 (1966), pp. 1563–1581 (cit. on p. 9).

[43]Ronald L. Graham. „Bounds on multiprocessing timing anomalies“. In: *SIAM journal on Applied Mathematics* 17.2 (1969), pp. 416–429 (cit. on p. 9).

[44]Y. Guo, J. Zhao, V. Cave, and V. Sarkar. „SLAW: A scalable locality-aware adaptive work-stealing scheduler“. In: *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. Apr. 2010, pp. 1–12 (cit. on p. 11).

[45]N. Hamid, R. Walters, and G. Wills. „Simulation and Mathematical Analysis of Multi-core Cluster Architecture“. In: *2015 17th UKSim-AMSS International Conference on Modelling and Simulation (UKSim)*. 2015, pp. 476–481 (cit. on p. 16).

[46]Norhazlina Hamid, Robert John Walters, and Gary Brian Wills. „An analytical model of multi-core multi-cluster architecture (MCMCA)“. 2015 (cit. on p. 16).

[47]Y. Hu, C. De Laat, and Z. Zhao. „Multi-objective Container Deployment on Heterogeneous Clusters“. In: *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. May 2019, pp. 592–599 (cit. on p. 13).

[48]Yang Hu, Huan Zhou, Cees de Laat, and Zhiming Zhao. „ECSched: Efficient Container Scheduling on Heterogeneous Clusters“. In: *Euro-Par 2018: Parallel Processing*. Ed. by Marco Aldinucci, Luca Padovani, and Massimo Torquati. Cham: Springer International Publishing, 2018, pp. 365–377 (cit. on p. 13).

[49]EG Co man Jr, MR Garey, and DS Johnson. „Approximation algorithms for bin packing: A survey“. In: *Approximation algorithms for NP-hard problems* (1996), pp. 46–93 (cit. on pp. 80, 85).

[50]J. A. S. Júnior, G. Lima, K. Bletsas, and S. Kato. „Multiprocessor Real-Time Scheduling with a Few Migrating Tasks“. In: *2013 IEEE 34th Real-Time Systems Symposium*. Dec. 2013, pp. 170–181 (cit. on p. 12).

[51]S Kato and N Yamasaki. „Semi-Partitioning Technique for Multiprocessor Real-Time Scheduling. In the 29th IEEE Real-Time Systems Symposium“. In: *Work-in-Progress Session (RTSS'08 WiP)* (2008) (cit. on pp. 12, 78).

[52]Gabor Kecskemeti, Attila Kertesz, and Attila Csaba Marosi. „Towards a Methodology to Form Microservices from Monolithic Ones“. In: *Euro-Par 2016: Parallel Processing Workshops*. Ed. by Frédéric Desprez, Pierre-François Dutot, Christos Kaklamanis, et al. Cham: Springer International Publishing, 2017, pp. 284–295 (cit. on p. 87).

[53]Hans Kellerer and Vladimir Kotov. „An approximation algorithm with absolute worst-case performance ratio 2 for two-dimensional vector packing“. In: *Oper. Res. Lett.* 31.1 (2003), pp. 35–41 (cit. on p. 94).

[54]*Kubernetes. [Online]*. `http://kubernetes.io/` (cit. on p. 89).

[55]Charles E. Leiserson. „The Cilk++ Concurrency Platform“. In: *Proceedings of the 46th Annual Design Automation Conference*. DAC '09. San Francisco, California: ACM, 2009, pp. 522–527 (cit. on pp. 16, 62).

[56]G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt. „DP-FAIR: A Simple Model for Understanding Optimal Multiprocessor Scheduling“. In: *2010 22nd Euromicro Conference on Real-Time Systems*. July 2010, pp. 3–13 (cit. on p. 78).

[57]S. Li, J. Hu, X. Cheng, and C. Zhao. „Asynchronous Work Stealing on Distributed Memory Systems“. In: *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. Feb. 2013, pp. 198–202 (cit. on p. 11).

[58]Chung Laung Liu and James W Layland. „Scheduling algorithms for multiprogramming in a hard-real-time environment“. In: *Journal of the ACM (JACM)* 20.1 (1973), pp. 46–61 (cit. on p. 76).

[59]Tania Lorido-Botrán, José Miguel-Alonso, and Jose Antonio Lozano. „Auto-scaling techniques for elastic applications in cloud environments“. In: *Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09* 12 (2012), p. 2012 (cit. on p. 88).

[60]Reinhard Lüling and Burkhard Monien. „A Dynamic Distributed Load Balancing Algorithm with Provable Good Performance“. In: *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '93. Velen, Germany: ACM, 1993, pp. 164–172 (cit. on p. 10).

[61]C. Maia, P. M. Yomsi, L. Nogueira, and L. M. Pinho. „Semi-Partitioned Scheduling of Fork-Join Tasks Using Work-Stealing“. In: *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*. Oct. 2015, pp. 25–34 (cit. on p. 12).

[62]Cláudio Maia, Patrick Meumeu Yomsi, Luís Nogueira, and Luis Miguel Pinho. „Real-time semi-partitioned scheduling of fork-join tasks using work-stealing“. In: *EURASIP Journal on Embedded Systems* 2017.1 (Sept. 2017), p. 31 (cit. on p. 12).

[63]Dirk Merkel. „Docker: lightweight linux containers for consistent development and deployment“. In: *Linux journal* 2014.239 (2014), p. 2 (cit. on p. 90).

[64]Seung-jai Min, Costin Iancu, and Katherine Yelick. „Hierarchical Work Stealing on Many-core Clusters“. In: *In: Fifth Conference on Partitioned Global Address Space Programming Models. Galveston Island*. 2011 (cit. on pp. 11, 16, 62).

[65]Michael Mitzenmacher. „Analyses of Load Stealing Models Based on Differential Equations“. In: *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '98. Puerto Vallarta, Mexico: ACM, 1998, pp. 212–221 (cit. on p. 10).

[66]El Mostafa Daoudi, Thierry Gautier, Aicha Kerfali, Rémi Revire, and Jean-Louis Roch. „Algorithmes parallèles à grain adaptatif et applications“. In: *Technique et Science Informatiques* 24 (May 2005), pp. 505–524 (cit. on p. 37).

[67] Stefan K. Muller and Umut A. Acar. „Latency-Hiding Work Stealing: Scheduling Interacting Parallel Computations with Work Stealing“. In: *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '16. Pacific Grove, California, USA: ACM, 2016, pp. 71–82 (cit. on p. 11).

[68] Falou Ndoye. „Ordonnancement temps réel préemptif multiprocesseur avec prise en compte du coût du système d'exploitation. (Multiprocessor preemptive real-time scheduling taking into account the operating system cost)“. PhD thesis. University of Paris-Sud, Orsay, France, 2014 (cit. on p. 76).

[69] B de Oliveira Stein, J Chassin de Kergommeaux, and G Mounié. *Pajé trace file format*. Tech. rep. Technical report, ID-IMAG, Grenoble, France, 2002. http://www-id. imag. fr . . ., 2010 (cit. on p. 48).

[70] S. Perarnau and M. Sato. „Victim Selection and Distributed Work Stealing Performance: A Case Study“. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. May 2014, pp. 659–668 (cit. on p. 11).

[71] Kochovski Petar, Sakellariou Rizos, Bajec Marko, Drobintsev Pavel, and Stankovski Vlado. „An Architecture and Stochastic Method for Database Container Placement in the Edge-Fog-Cloud Continuum“. In: To be presented at: 33rd IEEE International Parallel & Distributed Processing Symposium (IPDPS 2019). Zenodo, May 2019 (cit. on p. 12).

[72] A. Robison, M. Voss, and A. Kukanov. „Optimization via Reflection on Work Stealing in TBB“. In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. 2008, pp. 1–8 (cit. on pp. 16, 62).

[73] Jean-Louis Roch, Daouda Traoré, and Julien Bernard. „On-Line Adaptive Parallel Prefix Computation“. In: *Euro-Par 2006 Parallel Processing*. Ed. by Wolfgang E. Nagel, Wolfgang V. Walter, and Wolfgang Lehner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 841–850 (cit. on p. 37).

[74] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. „A Simple Load Balancing Scheme for Task Allocation in Parallel Machines“. In: *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '91. Hilton Head, South Carolina, USA: ACM, 1991, pp. 237–245 (cit. on p. 10).

[75] O. Runsewe and N. Samaan. „CRAM: a Container Resource Allocation Mechanism for Big Data Streaming Applications“. In: *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. May 2019, pp. 312–320 (cit. on p. 13).

[76] Peter Sanders. „Asynchronous Random Polling Dynamic Load Balancing“. In: *Algorithms and Computation: 10th International Symposium, ISAAC'99 Chennai, India, December 16–18, 1999 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 37–48 (cit. on p. 10).

[77] Jirí Sgall. „On-line scheduling| a survey“. In: *Dagstuhl Seminar on On-Line Algorithms (Schlo Dagstuhl (Wadern), Germany, June 24, 1996), to appear in LNCS. Springer-Verlag, Berlin-Heidelberg-New York*. Citeseer. 1998 (cit. on p. 3).

[78] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and YC Tay. „Containers and virtual machines at scale: A comparative study“. In: *Proceedings of the 17th International Middleware Conference*. 2016, pp. 1–13 (cit. on p. 90).

[79]Marc Tchiboukdjian, Nicolas Gast, and Denis Trystram. „Decentralized list scheduling“. In: *Annals of Operations Research* 207.1 (2013), pp. 237–259 (cit. on pp. 10, 18, 21, 36).

[80]Denis Trystram. „Les riches heures de l'ordonnancement“. In: *Technique et Science Informatiques* 31.8-10 (2012), pp. 1021–1047 (cit. on p. 2).

[81]Jixiang Yang and Qingbi He. „Scheduling Parallel Computations by Work Stealing: A Survey“. In: *International Journal of Parallel Programming* (2017), pp. 1–25 (cit. on p. 11).

Webpages

[36]frederic wagner. *rayon-logs*. 2018. URL: `http://datamove.imag.fr/frederic.wagner/category/rayon-adaptive.html` (cit. on p. 48).

List of Figures