Bridging a Gap Between Research and Production: Contributions to Scheduling and Simulation

Frédéric Suter

Numéro d'ordre : XXX Numéro attribué par la bibliothèque : YYY

ÉCOLE NORMALE SUPÉRIEURE DE LYON
Laboratoire de l'Informatique du Parallélisme

Habilitation à diriger des recherches

presented by

Frédéric SUTER

to obtain the HABILITATION À DIRIGER DES RECHERCHES
in *Computer Science* from École Normale Supérieure de Lyon.

Bridging a Gap Between Research and Production
Contributions to Scheduling and Simulation

Defense date: December 10, 2014

Defense committee:

Reviewers	Ms Ewa Deelman	Research Associate Professor, Univ. of Southern California, U.S.A
	Mr Johan Montagnat	CNRS Senior Researcher, Sophia Antipolis, France
	Mr Radu Prodan	Professor at University of Innsbruck, Austria
Members	Mr Michel Daydé	Professor at ENSEEIHT, Toulouse, France
	Ms Helen Karatza	Professor at Aristotle University of Thessaloniki, Greece
	Mr Yves Robert	Professor at ENS de Lyon, Lyon, France

Contents

Introduction **1**

1 Parallel Task Graph Scheduling **5**
1.1 Introduction . . . 5
1.2 Notations, Problem Statement, and Performance Metrics . . . 8
1.2.1 Platform and PTGs . . . 8
1.2.2 Problem Statement and Classification . . . 11
1.2.3 Performance Metrics . . . 12
1.3 Evaluation of Scheduling Algorithms . . . 15
1.4 Single PTG Scheduling . . . 17
1.4.1 On a Single Cluster . . . 17
1.4.2 On a Multi-Cluster . . . 30
1.5 Multiple PTG Scheduling . . . 42
1.6 Conclusion and Outlook . . . 53

2 Simulation of Distributed Systems & Applications **57**
2.1 Introduction . . . 57
2.2 The SIMGRID Framework . . . 59
2.2.1 User Interfaces . . . 60
2.2.2 Simulation Core . . . 60
2.3 SimDAG: an API for DAG Scheduling Simulation . . . 62
2.4 Simulation Input Generation . . . 71
2.4.1 Platforms . . . 71
2.4.2 Task Graphs . . . 81
2.5 Result Acquisition and Analysis . . . 85
2.6 Dimensioning Through Simulation . . . 89
2.6.1 Single Node On-Line Simulation of MPI Applications with SMPI . . . 89
2.6.2 Off-Line Simulation with Time-Independent Trace Replay . . . 95
2.7 Conclusion and Outlook . . . 106

3 Bridging the Gap Between Research and Production **109**
3.1 Introduction . . . 109
3.2 Buzzwords and Key Concepts . . . 112
3.2.1 Cloud Computing . . . 112
3.2.2 Big Data . . . 114
3.2.3 HPC . . . 115
3.3 Towards Production-Grade Simulation . . . 116
3.3.1 Realistic Simulation of DCIs for Scientific Experiments . . . 116
3.3.2 MPI as a Simulation . . . 121

3.3.3 Storage Dimensioning Through Simulation . . . 125
3.4 Applying Research in Scheduling to Production Applications . . . 129
3.4.1 Workflow Optimization for the Detection of Supernovae . . . 130
3.4.2 Multi-objective scheduling for large computing platforms . . . 132
3.5 Conclusion . . . 134

General Conclusion **135**

Acronyms, Figures, Tables, and External Links **136**

Publications Since the End of the Ph.D. **146**

Publications Related to the Ph.D. **151**

Bibliography **153**

Introduction

This document summarizes my research work since the defense of my Ph.D. in 2002. It covers more than ten years of work, successively done as a post-doctorate at the University of California, San Diego (UCSD) (2003), as an Attaché Temporaire d'Enseignement et de Recherche (ATER) at the Université Joseph Fourier, Grenoble 1 in the Informatique et Distribution, Informatique, Mathématiques et Applications de Grenoble (ID-IMAG) laboratory (2004), as an assistant professor at the Université Henri Poincaré, Nancy 1 in the AlGorille team of the Laboratoire lorrain d'informatique et de ses applications (LORIA) (2004-2008), and finally as a junior researcher (CR1) of the Centre National de la Recherche Scientifique (CNRS) at the IN2P3 Computing Center (CC IN2P3) (2008-present). Since January 2012, I am also a member of the Avalon team at the Laboratoire de l'Informatique du Parallélisme (LIP).

In this short introduction, I present the general context in which my research takes place as well as the motivations of my different works. Then I detail the organization of this document by giving a brief overview of the subsequent chapters.

General Context and Contributions

I am scientifically born with *the Grid*. Back in 1999, when I was starting my Master internship, the physics community was laying the foundations of a pan-European computing and storage distributed infrastructure that would become the Enabling Grids for E-sciencE (EGEE) project by the end of my Ph.D. Such an infrastructure was mandatory to store and analyze the tremendous amount of data produced by the forthcoming Large Hadron Collider (LHC). From a Computer Science point of view, the Grid brought new opportunities to application developers by increasing the scale of the available resource pool by orders of magnitude. It also came with a lot of challenging issues, the main one being having to deal with a large scale, heterogeneous, and hierarchical shared platform, distributed across several administrative domains. A large body of work has thus been proposed on the scheduling of applications and resource management on computing grids to tackle some of these issues. Another challenge that raised very early with computing grids is about the understanding of the dynamic behavior under load of such unprecedented infrastructures. Because of their scale, and the inherent time and resource usage constraints, researchers usually had to resort to simulation to analyze, understand, and optimize the usage of grids. Unfortunately, the respective interests and choices of the researchers on computing grids and the users of these infrastructures rapidly diverged and it became difficult to transfer research results, as interesting as they be, into production.

For the last ten years, I aimed at contributing to the resolution of these challenges related to computing grids. To achieve this goal, I developed two topics first addressed during my Ph.D. which was entitled *Mixed parallelism and performance prediction on heterogeneous networks of parallel computers*. The heterogeneous networks of parallel computers from then are the grids and clouds from now, but remained my principal object of interest and investigation since. The contributions of this document directly derive from the remaining of the title of my Ph.D. and are:

Exploit all the parallelism that is available in scientific workflow

During my Ph.D., I demonstrated that a simultaneous exploitation of both task- and data-parallelism was possible and led to interesting performance improvements. But this study was limited to a specific use case and all the scheduling decisions derived from a very good knowledge of both the application and the target platform. I found it frustrating to see a potential source of gain in terms of performance and not being able to apply similar techniques to a broader range of applications and execution environments. Then, I spent years proposing algorithms and heuristics to cover as much scenarios as possible. Part of this work corresponds to the Ph.D. of Tchimou N'Takpé (co-advised with Jens Gustedt on a co-funding by the department of education of Côte d'Ivoire and the Région Lorraine, 2005-2009).

Better understand the behavior of distributed systems and applications thanks to simulation

My first interest for performance prediction came from my need to understand how the specific mixed-parallel applications I played with behave and take good scheduling decisions. It was then limited to a specific context with almost no possibility of further extension. Consequently, I put it on hold for a long time as I was focusing on the design of scheduling heuristics. But this interest raised again as I was more and more deeply involved in the research and development around the SIMGRID toolkit. Finding the right tool to investigate the performance of parallel applications thanks to simulation allowed me to (re)develop a challenging research topic. Through the Ph.D. of Georgios Markomanolis (co-advised with Frédéric Desprez on an Inria CORDI-S funding, 2009-2014), I relied on simulation to study the performance of parallel applications and use the obtained results as objective indicators to help at the dimensioning of computing infrastructures.

Collaborate with members and users of the CC IN2P3

Since I have been hired as a researcher at CC IN2P3 in October 2008, on the uncommon position of being the only researcher in Computer Science surrounded by engineers and physicists, I tried to establish connections and foster collaborations between my community of origin and this new environment. Achieving an efficient cross-fertilization and finding win-win situations, is a long term objective. So is the opportunity to see my own research results be applied to concrete use cases and make the research made by others advance. However, these activities constitute a challenging and motivating aspect of my work, which is less driven by the production of immediate scientific results.

Organization of the Document

In this document, I reorganized my work done on parallel task graph scheduling and the simulation of distributed systems and applications to give a comprehensive view rather than presenting things in a historical way. Moreover I decided to focus on the ideas underlying the algorithms and developments instead of insisting on their performance results. They can be found in the corresponding publications. Those two research fields are by the way tightly intertwined. Indeed, validating the proposed scheduling algorithms often raised new needs with regard to the simulation framework and led to new developments. On the opposite, improvements made to the simulation kernel, the programming interfaces of SIMGRID, or to its surrounding ecosystem, offered new simulation opportunities that made possible the study of new scheduling problems. The third chapter of this document is more prospective and present some reflexions and efforts to establish bridges between two research communities: the one that studies and make progress distributed computing infrastructures such as Grids, and the one that actually uses such infrastructures in production to generate groundbreaking scientific results.

Chapter 1: Parallel Task Graph Scheduling

In this first chapter, I detail the dozen of original algorithms I have proposed to schedule Parallel Task Graphs (PTGs). Section 1.2 summarizes all the notations and concepts I used to design and evaluate these algorithms. This section also includes a taxonomy of the scheduling problems I have studied. Then, I explain in Section 1.3 how the evaluation of the proposed scheduling algorithms was conducted and how my own requirements in terms of experimental evaluation have evolved through the years. The algorithms themselves are fully described in Sections 1.4 and 1.5.

Chapter 2: Simulation of Distributed Systems & Applications

The second Chapter of this document is dedicated to my work within and around the SIMGRID project. Section 2.2 recalls the general organization and the founding concepts underlying SIMGRID. Then, I detail my contribution related to one the programming interfaces of SIMGRID, called SimDAG, that is dedicated to the study of task graph scheduling algorithms in Section 2.3. I also present work done upstream of a simulation on the generation of simulation inputs, such as execution environments or description of application task graphs, in Section 2.4 and downstream of a simulation by detailing how I improved over the years way I acquire and analyze simulation results in Section 2.5. Finally, Section 2.6 details my most recent work about the simulation of parallel applications using the message passing programming paradigm. It has the specificity to not be related to my research on scheduling at all.

Chapter 3: Bridging the Gap Between Research and Production

In this last Chapter, I present some research directions that I plan to follow in the next few years to bridge the gap that exists between research and production in the context of computing grids and clouds. First, I position my work with regard to the highly popular topics (a.k.a. buzzwords) that are Cloud Computing, Big Data, and High-(Performance/Throughput/...)-Computing in Section 3.2. Then, I detail two complementary approaches to bridge this gap. The first approach, described in Section 3.3, consists in bringing SIMGRID up to a production-grade level so that it can be trustfully used not only by researchers in Computer Science that study computing grids, but also by scientists for other domains that use these computing grids in production. The second approach, presented in Section 3.4, will be to apply the results, or the acquired knowledge and expertise, coming from my research on scheduling and resource management to applications and environments used in production.

General Conclusion

Each chapter has its own conclusion that sums up the work done and presents some research directions to investigate. The general conclusion of this document will thus come back on the challenges I faced and the solutions I proposed. It will also recall the future work scattered across the document and provide a higher level vision of what should be my research orientations in the next few years.

Chapter 1

Parallel Task Graph Scheduling

1.1 Introduction

Nowadays, many scientific applications from various disciplines, such as physics, biology, earth science, or even computer science, are structured as *workflows*. Informally, a workflow can be seen as the composition of a set of basic operations that have to be performed on a given input set of data to produce the expected scientific result. The interest for this kind of application structuring mainly comes from the need to build upon legacy codes that have been developed and used for decades and would be too costly to rewrite. Combining existing programs is also a way to lead to new results that would not have been found using each component alone. Finally, such program compositions are mainly done by hand by scientists, that have to run each program one after the other, manage themselves the intermediate data, and deal with the potentially tricky transitions between programs. The emergence of Grid Computing a decade ago and the development of complex middleware components to manage and exploit workflows on large scale distributed infrastructures, such as DAGMan [48], Pegasus [54], or Taverna [90], have allowed for the automation of this process.

Most of these scientific workflows compose serial kernels to form a graph structure. Nodes of such graphs are the compute kernels, or tasks, while the edges represent control dependencies, *i.e.,* a task cannot start before another completes, or flow dependencies, *i.e.,* a task requires data produced by another to proceed. Moreover, those graphs are usually acyclic and deterministic, even though some workflows include control nodes representing conditional branches or iterative structures.

In terms of computing infrastructures, the evolution of processors tends either towards the multiplication of cores of lower frequency or the use of graphical processors and hardware accelerators to keep pace with Moore's Law. This raises some concerns about the capacity of scientific workflows based on serial kernels to fully exploit these new computing architectures. First, the generalization of multi- and many-core processors implies that more and more parallelism becomes available at the processor level. However, a workflow may be able to exploit only a part of it, limited to the amount of task parallelism it exhibits. In other words, a workflow cannot use simultaneously more computing units than the width of the graph representing it. Some unconventional techniques, such as Dynamic Speculative Precomputation [46] or Inter-core Prefetching [98], can take advantage of extra-cores to run *helper threads* and accelerate the execution of purely serial applications, but it is unlikely that all the available parallelism can be exploited by traditional scientific workflows.

Second, we already mentioned that one of the advantages of scientific workflows was to reuse legacy codes to obtain new results. This prevents the rewriting of usually large source codes to port them on unconventional computing units such as Graphical Processing Units (GPUs) or hardware accelerators. To use such processors in an efficient way, specific techniques such as vectorization have to be implemented, new programming languages may be used, and memory management requires a special care.

Hence, it is not likely that scientific workflows can benefit of the large processing power offered by these new types of hardware without complex modifications. However, tools such as StarPU [10] or HMPP [58] may lighten this portability burden thanks to encapsulation or advance compilation.

A third issue for traditional scientific workflows is related to memory. Indeed, while the number of computing units, be they core or GPUs increases quickly, the amount of main memory grows at a slower pace. Consequently, the memory over computing unit ratio tends to stagnate or even decrease. Memory becomes the new bottleneck and serial applications will soon be limited in their use of the available computing resources. New strategies, mainly aiming at increasing the number of operations executed on each byte of information moved into memory, will thus have to be implemented.

All these issues motivate a disruptive shift in the way scientific workflows are programmed and executed. Changing the way scientific results are produced always faces a resistance from the users. Scientists from disciplines other than Computer Science usually prefer to keep their good old fashioned applications, at the risk of loosing performance, rather than making the effort to adapt their codes to new, and potentially abandoned soon, programming paradigms. However, some examples exist of software packages that followed the evolution of hardware and made the minds change in their user communities. The most striking example comes from a software suite to solve linear algebra problems. This work started in the 70's with the LINPACK library that was based on Level-1 Basic Linear Algebra Subroutines (BLAS) (vector-vector operations). In the 80's, LINPACK did evolve into LAPACK to include more memory and cache friendly operations thanks to the design of Level-3 BLAS (matrix-matrix operations). In the 90's, as the users' needs were ever growing, the use of distributed memory architectures became mandatory. The ScaLAPACK library was then proposed, based on the message passing paradigm. At that time, most users were reluctant to move from the serial version to the parallel one, although it was the only way to fully exploit the most powerful supercomputers at that time. A decade later, at the age of multi-core processors and GPU, ScaLAPACK is now the state-of-the-art library that no user wants to get rid off to use the new evolutions of the software suite that are Parallel Linear Algebra Software for Multicore Architectures (PLASMA) and Matrix Algebra on GPU and Multicore Architectures (MAGMA). Time will tell whether or not these evolutions will become the state-of-the-art packages of the next decade. This story tells us that applications have to adapt to disruptive technologies if they want to exploit all the available power. It also tells that users finally adopt software evolutions when it proves to be the only way to get more performance. Scientific workflows are now at such a crossroad: adopt and adapt or stay limited to current performance.

Since my Ph.D., I study one way of addressing the aforementioned issues. It consists in proposing scheduling algorithms for workflows not composed of serial but parallel kernels. This approach, introduced as *mixed-parallelism* in [41], as it allows for the simultaneous exploitation of both the task- and data-parallelisms exhibited by an application, is a promising way toward the full exploitation of modern architectures. According to the literature on job scheduling, parallel applications can be classified depending on who and when is decided the number of computing units onto which to execute a job. This classification, introduced in [68], is given in Table 1.1.

who decides number	when is it decided	
	at submission	during execution
user	*Rigid*	*Evolving*
system	*Moldable*	*Malleable*

Table 1.1: Classification of job types based on specifying number of processors used.

The most common type of parallel applications corresponds to *rigid* jobs. The user is responsible for deciding, prior to the execution, how many computing resources are needed. Once this number is fixed, it cannot be changed until the completion of the application. When the user application can be

decomposed in several phases that require different number of resources, one can specify these allocations beforehand. Then the job is said to be *evolving*. This distinction between fixed and dynamically changing number of resources can also be made when the system is responsible for taking the allocation decisions. Then the jobs are respectively classified as *moldable* or *malleable*.

In all my work on PTG scheduling, I considered applications represented by a Directed Acyclic Graph (DAG) in which vertices are moldable tasks. Then the whole applications can be seen as malleable jobs. Indeed, the number of computing resources allocated to their execution changes dynamically. Considering a workflow made of moldable tasks means that each node of the graph can be executed on a different number of processors. The number of computing units allocated to a task is called the task's *allocation*. It is generally assumed that the task execution time of a task is typically non-decreasing as its allocation increases. Figure 1.1 shows two possible configurations of an example five-task PTG, each configuration corresponding to different allocations. Each task's allocation is depicted by a "box" whose width corresponds to the number of resources and the height to the execution time.

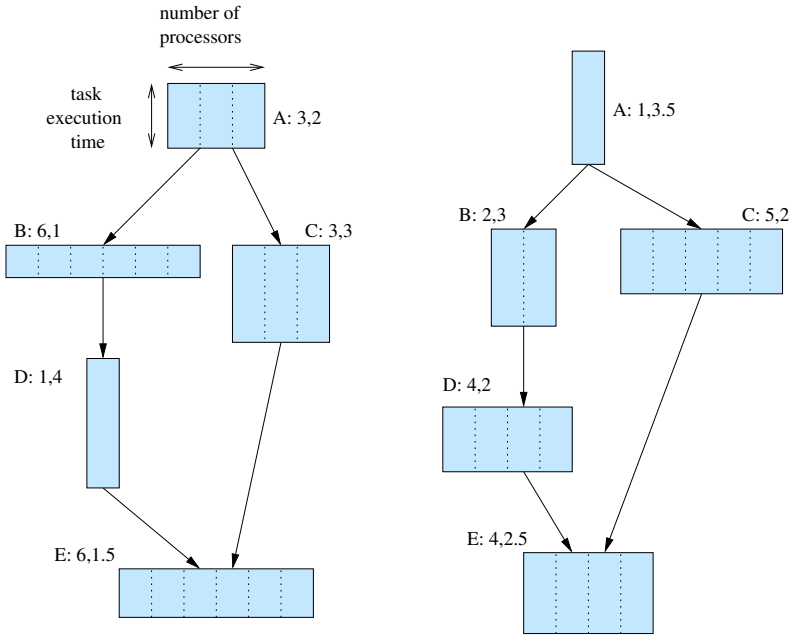


Figure 1.1: Example 5-task PTG, with two possible configurations. Each task is labeled as $X : x, y$, where X is the task's name, x is the task's allocation, and y is the task's execution time.

In this chapter, I will first detail in Section 1.2 all the notations related either to platforms and applications that I used in my work and describe the problem statement and the different metrics that allowed me to evaluate the performance of the proposed algorithms. Then, I explain in Section 1.3 how the evaluation of the proposed scheduling algorithms was conducted and how my own requirements in terms of experimental evaluation have evolved since the end of my Ph.D. I will review my different contributions to the resolution of the problem of scheduling either a single PTG or multiple PTGs simultaneously in Sections 1.4 and 1.5 respectively. Finally I conclude this chapter and provide some possible research perspectives in Section 1.6.

1.2 Notations, Problem Statement, and Performance Metrics

1.2.1 Platform and PTGs

We define a compute cluster as a homogeneous set of p processors. We use the term "processor" to refer to an individually schedulable compute resource. With this terminology, a "processor" may in fact be a physical compute node that is a multi-processor and/or multi-core computer. Processors are interconnected by a high-speed, low-latency network. Each processor is able to execute a certain amount of floating operations (or flop) per second that represents its computing speed. We denote this computing speed by s. A processor can communicate simultaneously with several other processors under the *bounded multi-port model*. All the concurrent communication flows share the bandwidth of the communication link that connects this processor to the remaining of the cluster. By extension a *multi-cluster* platform is a heterogeneous collection of homogeneous clusters. The target computing platform then consists of c clusters, where cluster c^i , $i = 1, \ldots, c$ contains p^i identical processors. A processor in cluster c^i computes at speed s^i. We also define as r^i the ratio between the processor's computing speed of cluster c^i to that of the slowest processor over all c clusters, which we call the *reference* processor speed s^{ref}. Clusters may be built with different interconnect technologies and are interconnected together via a high-capacity backbone. Each cluster is connected to the backbone by a single network link. Inter-cluster communications happen concurrently, possibly causing contention on these network links. The backbone connecting the different clusters can be composed by several links separated by routers. Figure 1.2 illustrates our definitions of cluster and multi-cluster platforms.

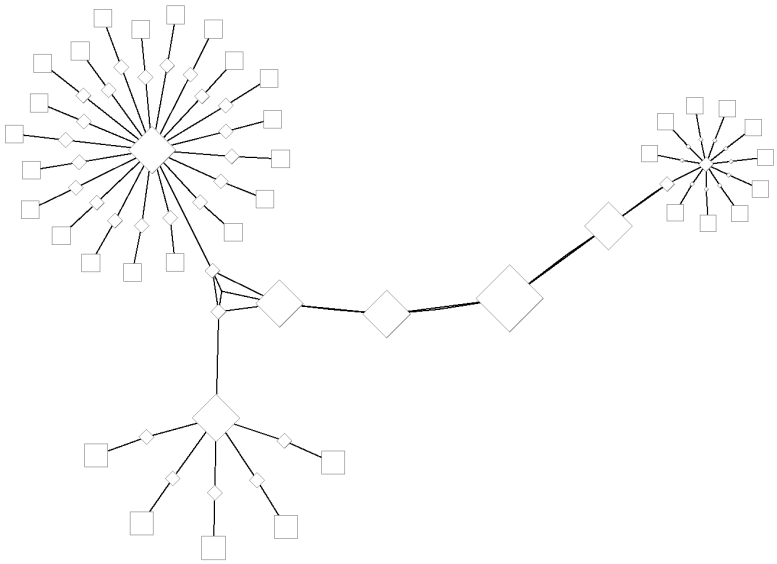


Figure 1.2: Illustration of a heterogeneous multi-cluster platform made of three homogeneous clusters.

This figure shows a computing platform made of three homogeneous clusters. Squares represent the computing capacities of the processors, while diamonds shows the capacity, in terms of bandwidth, of the network links. A larger square (resp. diamond) indicates a higher computing speed (resp. network capacity.) The commodity clusters comprise respectively 5, 10, and 20 nodes. Each cluster computes at its own speed, and may have different network interconnect. For instance, the cluster on the right hand side of Figure 1.2 is interconnected through a slower network (smaller diamonds). Moreover this cluster is located in a different site as the other two clusters, hence the longer route toward it.

We denote by P the total number of processors in the considered platform. In the case of a single cluster, $P = p^1 = p$, while in a multi-cluster platform, $P = \sum_{i=1}^{c} p^i$. Table 1.2 summarizes the notations related to computing platforms that will be used in this chapter.

Notation	Definition
c	Total number of compute clusters in the platform
c^{ref}	slowest cluster in the platform
c^i	i^{th} cluster in the platform
s^{ref}	processing speed of a processor of the slowest cluster
s^i	processing speed of a processor of the i^{th} cluster
r^i	processing speed ratio for the i^{th} cluster, *i.e.*, $r^i = s^{ref}/s^i$
p^{ref}	number of processors in the slowest cluster
p^i	number of processors in the i^{th} cluster
P	Total number of processors in the platform

Table 1.2: Summary of notations related to computing platforms.

A PTG can modeled as a DAG $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_i \,|\, i = 1, \ldots, V\}$ is a set of vertices representing moldable tasks, or "tasks" for short, and $\mathcal{E} = \{e_{i,j} \,|\, (i,j) \in \{1, \ldots, V\} \times \{1, \ldots, V\}\}$ is a set of E edges representing precedence constraints between tasks. We denote by $pred(v_i)$ (resp. $succ(v_i)$) the set of predecessors (resp. successors) of task v_i. A predecessor of task v_i is a task v_j such as $e_{j,i} \in \mathcal{E}$, while v_j is a successor of v_i if $e_{i,j} \in \mathcal{E}$. Without loss of generality we assume that $\mathcal{G}$ has a single entry task and a single exit task.

Since the tasks composing the PTG are moldable, they can be executed on various numbers of processors. Then we denote by $T^j(v_i, p)$ the execution time of task v_i if it were to be executed on p processors of cluster c^j. When the execution platform is made of a single cluster, we reduce this notation to $T(v_i, p)$. The question of the determination of this $T(v_i, p)$ for any given task and number of processors is a research problem in itself. In practice, $T(v_i, p)$ can be measured via benchmarking for several values of p, or it can be calculated *via* a performance model. Both approaches have drawbacks that were investigated in [HCS11] in the particular context of PTG scheduling.

Several performance models of parallel, and so forth moldable, tasks have been proposed in the literature, mainly for homogeneous compute clusters. These models quantify the speedup an application, or a task in our case, can achieve when parallelized. The speedup of a task v is defined as $S(v) = T(v,1)/T(v,p)$, *i.e.*, its execution time on one processor divided by its execution on p processors. The most prevalent speedup model was introduced by G. Amdahl in 1967 and is known as Amdahl's Law [6]. This model claims that the speedup of a parallel application is limited by its strictly serial part. By denoting this strictly serial part by α, it is possible to estimate the parallel execution time of a task on p processors by:

$$T(v,p) = \left(\alpha + \frac{(1-\alpha)}{p}\right) \times T(v,1). \tag{1.1}$$

While this law has been refined or augmented for more than forty years, its fundamental principle is still valid. Other alternate speedup models exist, such as those proposed by Downey [62] or Cirne and Berman [45], that derive from a finer characterization of parallel applications. However in the different scheduling studies detailed in the remaining of this chapter, we relied on the simple but generic Amdahl's Law to estimate parallel execution times as defined by Equation 1.1.

By extension we can define the *work* associated to the execution of a task. This quantity, denoted by $W^j(v_i)$ corresponds to the product of the execution time on cluster c^j by the number of resources used

on this cluster to complete this execution, *i.e.*, $W^j(v_i) = T^j(v_i, p) \times p$. As for the execution time, when the platform comprises only one cluster, we reduce this notation to $W(v_i)$.

The edges of a PTG may have a weight related to communication costs. The existence of such a weight directly depends on how the PTG will be actually executed on the target computing platform. At runtime, tasks are no more moldable as a decision has been taken about their allocation, *i.e.*, the number of processors onto which they will be executed. A first scenario is to execute the whole application within a single large resource reservation that encompasses the executions of all the tasks composing the PTG. Typically, such a reservation should cover the maximum number of processors simultaneously used by the computed schedule for the total execution time of the PTG. In this case, data transfers between tasks can be done as communications over the network. Indeed, every processor in the global reservation can reach every other processor. Each edge $e_{i,j}$ is then weighted by the amount of data (in bytes) that task v_i must send to task v_j. A fundamental assumption in this model is that if two subsequent tasks are mapped on the same set of resources, no network communication occurs.

The second possible scenario is to see the execution of a PTG as a set of distinct resource reservations, one per task, submitted independently to a Job and Resource Management System (JRMS). In this scenario it may happen that a task completes well before the beginning of one or more of its successors. Then it precludes the use of network communication between tasks as processors in different resource reservations can not communicate together. One solution is then to implement communications using files on disks, and relying on a shared file system. The overhead of such communication is then included in the task performance model (as a sequential overhead) and the edges of the graph are zero-weighted. Then we do not model any communication network or data transfer between tasks.

The *bottom level* of a task v_i, denoted as $bl(v_i)$, is defined as the length of the path from the task to the exit task, that is the sum of execution times of the tasks along this path. This value includes the execution time of task v_i itself. Conversely, the *top level* of a task v_i, denoted as $tl(v_i)$, is defined as the length of the path from the entry task to the considered task. As for bottom level, it corresponds to the sum of the execution times along this path. The execution time of task v_i is excluded from the computation of its top level. For any given task, adding its bottom level and top level values gives the length of the longest path, in terms of execution times, to which this task belong. The tasks with the highest $bl(v_i) + tl(v_i)$ values compose the *critical path* of the application. Note that more than one critical path may exist in very regular task graphs. Finally, the *precedence level* of a task v_i, denoted as $pl(v_i)$, is the value such that $pl(v_j) < pl(v_i), \forall v_j \in pred(v_i)$ and $\exists v_k \in pred(v_i)$ such that $pl(v_k) = pl(v_i) - 1$. The precedence level of an entry node is set to 0. Table 1.3 summarizes the notations related to PTGs.

Notation	Definition
$\mathcal{G}$	Directed Acyclic Graph describing the application
$\mathcal{V}$	Set of tasks composing the application
$\mathcal{E}$	Set of edges connecting tasks and representing flow and control dependencies
v_i	A moldable computing task
$e_{i,j}$	Edge representing a dependency between tasks v_i and v_j
$pred(v_i)$	Set of predecessors of task v_i
$succ(v_i)$	Set of successors of task v_i
$T^j(v_i, p)$	Execution time of task v_i on p processors of cluster c^j
$W^j(v_i)$	Work associated to the execution of task v_i on p processors of cluster c^j
$bl(v_i)$	Bottom level of task v_i
$tl(v_i)$	Top level of task v_i
$pl(v_i)$	Precedence level of task v_i

Table 1.3: Summary of notations related to Parallel Task Graphs.

1.2.2 Problem Statement and Classification

The classical DAG scheduling problem consists in, for each of the serial tasks that compose the DAG, to determine onto which machine to execute it. A schedule also sets the order in which independent tasks, *i.e.*, tasks without any precedence relationship, mapped on the same machine have to be executed.

Dealing with PTGs adds another degree of liberty, or complexity from a more pessimistic standpoint, to this already NP-hard problem [75]. Indeed, a scheduler first has to determine the number of computing resources to *allocate* to each moldable task before even deciding how to *map* them on the available resources.

The PTG scheduling problem can then be divided into two distinct subproblems, namely the *allocation* and *mapping* problems, that are illustrated by Figure 1.3. Informally the allocation step takes care of deciding of the shape of the "boxes" that represent the different tasks, *i.e.*, how many resources are used for how much time by each task. The mapping step aims at finding the best arrangement of these boxes on the target execution environment.

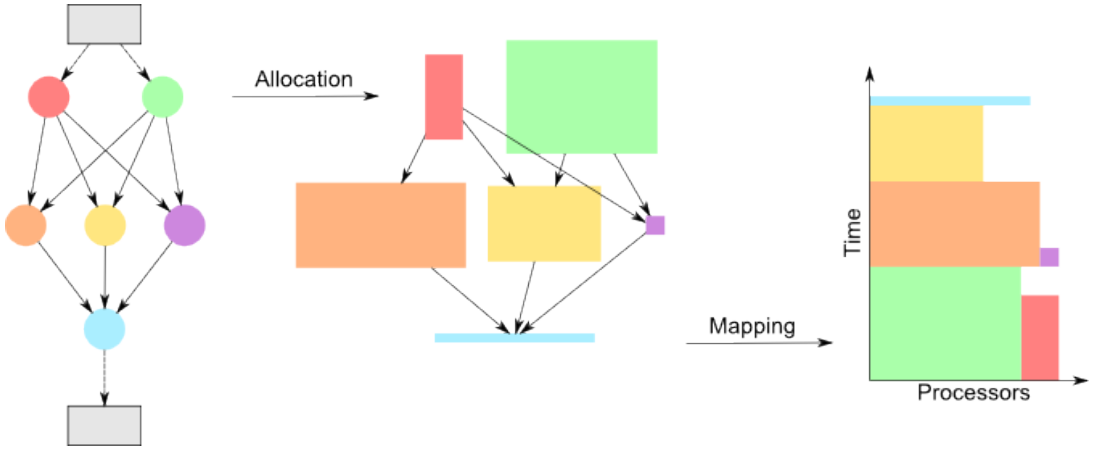


Figure 1.3: Example of the two-step schedule of a PTG.

One of the main advantages of the two-step approach is its relative simplicity allowed by the decoupling of the allocation and mapping processes. This simplicity also comes with a potential drawback related to inter-task communications that may be badly handled and hinder the performance of a given scheduling algorithm. A possible solution is to do the allocation and the mapping in a tightly coupled way. In this case, we talk about one-step algorithms. While such algorithms better handle communications, they are either much more complex [150, 151] or limited to very specific execution platforms [BDS03]. Most of the work presented in this chapter in then based on the two-step approach. Nevertheless some techniques to cope with inter-task communications are proposed.

PTG scheduling is a very general problem whose instances can be classified according to the taxonomy proposed in Figure 1.4. First, we have to distinguish the scheduling of a single PTG that has a dedicated access to an execution environment from the scheduling of multiple PTGs that have to share, or most likely compete for, resources. For both categories, the scheduling issues are different whether the targeted execution environment is composed of a single, and thus homogeneous, compute cluster or of multiple, and likely heterogeneous, clusters. When scheduling a single PTG, algorithms may aim at optimizing only one performance metric, *e.g.*, minimizing the completion time of the scheduled application, or several metrics simultaneously, *e.g.*, also try to minimize the resource usage in the process. As mentioned earlier, such objectives are reached in one or two steps. Conversely, scheduling multiple PTGs is an intrinsically multi-criteria optimization problem. Indeed, a scheduling algorithm has not

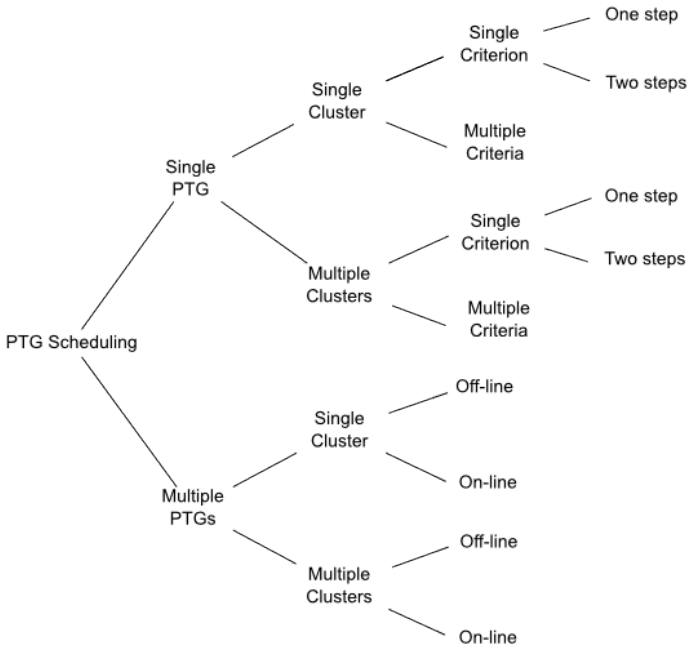


Figure 1.4: A taxonomy of Parallel Task Graph scheduling problems.

only to minimize the completion time of each scheduled application, but also to ensure a certain fairness between them. A sound algorithm has to prevent an application to be too severely unfavored with regard to the others. In this context, a distinction can be made on the way the different applications are handled by the scheduler. In the off-line case, the algorithm knows all the applications before starting to build its schedule. In the on-line case, new applications are submitted to the scheduling algorithm while it is taking decisions for those already in the system.

At the end of my Ph.D. in 2002, only a few works were addressing the PTG scheduling problem. The earliest propositions were coming from programming languages in a view to integrate both task and data parallelisms in the same language but did not investigate scheduling issues. The works that fit in the taxonomy given by Figure 1.4 are those proposed by Ramaswamy *et al.* [126], Radulescu *et al.* [124, 125], and Rauber and Rünger [128]. All of them considered the scheduling of a single PTG on a homogeneous cluster and optimized a single performance metric, the makespan of the application. All the other branches of the taxonomy remained a open field.

1.2.3 Performance Metrics

The performance metric that is the most commonly used to evaluate scheduling algorithms is the *makespan*. This metric, also called completion time, schedule length, or denoted by C_{max}, corresponds to the time elapsed between the beginning of the first task of the application and the completion of the last application's task. Makespan is generally measured in seconds. Formally, the makespan of a PTG is defined as

$$C^*_{max} = \max_i C(v_i), \tag{1.2}$$

where $C(v_i)$ is the completion time of task v_i. Makespan minimization can be seen as a user-centric performance metric. Indeed, a vast majority of users of high performance computing centers want their jobs, i.e., application executions, to finish as early as possible. However, this is not the concern of every

user. One may be more interested by the total completion of a large set of jobs as every produced data is needed for subsequent analysis. Such cases are more related to high throughput computing than high performance computing. While makespan reduction remains an important concern, it goes in the background in favor of reliability, *i.e.*, the capacity to guarantee that every single job will complete.

A second popular performance metric is the *total work* needed to execute an application. Unlike makespan, this metric is more resource provider-centric. A system administrator, through the use of a JRMS, will aim at using its resources as efficiently as possible, and thus at minimizing the total work. Using less resources can also be one of the user's goals. A less resource consuming schedule will indeed be "greener". It can also lead to a lower bill if resource consumption is accounted. The total work is usually measured as the product of the number of computing resources needed to execute the application and the makespan, and denoted by W_{max}. The formal definition of W_{max} depends on how inter-task communications are handled. When data are passed from one task to another through a shared file system, the total work corresponds to the sum of the work $W(v_i)$ needed to execute each task v_i. The formal definition is then

$$W_{max}^{i/o} = \sum_i W(v_i). \tag{1.3}$$

Conversely when data transfers are done through the network, the schedule has to be executed within a single resource reservation. Graphically it corresponds to a box whose width is equal to the number of resources (a.k.a the "size" of the job) and height is equal to the application makespan. This job size corresponds to the maximal number of computing resources simultaneously used during the execution. The total work is then defined as the area of this box

$$W_{max}^{net} = C_{max}^{*} \times \texttt{job\_size}. \tag{1.4}$$

A corollary metric is to measure the *efficiency* of a schedule. Many definitions of the parallel efficiency of a schedule or a scheduling algorithm exist in the literature. Here we define the efficiency as a ratio of the work needed to execute a PTG on a single processor, denoted by W_{seq}, on the work (as defined by Equation 1.3 or 1.4) needed to execute the same PTG on a cluster or a multi-cluster. The efficiency of a given schedule is then defined as

$$E = \frac{W_{seq}}{W_{max}}. \tag{1.5}$$

A value close to one for this metric means that computing resources are used in a rational way during the parallel execution of the application. In other words, the schedule gives the "highest bang for the buck" to the user. Conversely a low efficiency value means that too many resources are used when compared to the gain in terms of makespan reduction. Then a slightly longer schedule but using less computing resources should be found.

When scheduling not only one PTG but N PTGs simultaneously, the important performance metrics change. In this context, applications have to compete for resources. The definition of the makespan given by Equation 1.2 assumes that the target platform is dedicated to the execution of the scheduled PTG. This assumption does not hold anymore when considering more than one PTG. Depending on the size of the platform, it is likely that all the individual schedules cannot be applied without experiencing an increase of the respective makespan of some applications.

Then three metrics can be considered to quantify the quality of as schedule. For PTG $i = 1, \ldots, N$, we use $C_{max_i}^{*}$ to denote the ideal execution time on the dedicated platform, and C_{max_i} to denote the execution time in the presence of competition with the other PTGs.

First, we can quantify the overall performance of the PTGs by using the sum of completion times divided by the sum of completion times on a dedicated platform, which we call *scaled sum completion*

time, and is defined as:

$$\frac{\sum_{i=1}^{N} C_{max_i}}{\sum_{i=1}^{N} C^*_{max_i}}. \tag{1.6}$$

While this sum completion time captures a notion of average performance, we also need a metric for the performance of the whole batch of PTGs, *i.e.*, the *overall makespan*, defined as $\max_{i=1,\ldots,N} C_{max_i}$.

Finally, there is a need to evaluate the ability of scheduling algorithms to produce fair schedules. A popular metric to evaluate the level of performance achieved by a job that competes with other jobs is the *stretch*, also called slowdown. In our case, jobs are PTGs, and the stretch of PTG i is defined as $C_{max_i}/C^*_{max_i}$. For instance, if a PTG could have run in two hours using the entire cluster, but instead ran in six hours due to competition with other PTGs, then its stretch is three. This is the most widely accepted definition in the literature, with a lower value denoting better performance.

A perfectly fair schedule can then be defined as a schedule in which all PTGs experience the same stretch. One possibility is to define *unfairness* as the difference between the maximum stretch and the minimum stretch, or the average absolute value of the difference between the stretch of each PTG and the overall average stretch. Another natural definition would be to define unfairness as the standard deviation or the coefficient of variance of the stretches. Yet another possibility, is to quantify unfairness as the *maximum stretch*, defined as

$$max\_stretch = \max_{i=1,\ldots,N} C^*_{max_i}/C_{max_i}.$$

If the maximum stretch is optimally minimized, then all PTGs have the same stretch and fairness is optimal. Minimizing the maximum stretch has long been known to be a good approach to improve performance as well as fairness [20]. This metric has the interesting property to be not completely agnostic to performance. Consider schedule A in which all PTGs have a stretch of 1,000, while a schedule B exists in which all PTGs can have stretches between 10 and 20. With the aforementioned fairness metrics, schedule A would be deemed preferable because the unfairness would be equal to 0. However, schedule B is preferable to all users, which is clearly indicated by the maximum stretch metric.

Note that the stretch can also be used to define a performance metric as an alternative to the scaled sum completion time. For instance, performance can be quantified by the sum stretch or *average stretch* [20]. Nevertheless, we opt for scaled sum completion time, based on the following rationale. Consider a 100-processor cluster on which to schedule 101 independent, serial jobs. 100 of these jobs run in 1 time unit, and one runs in ε time units. There are two reasonable types of schedules: either the task with execution time ε runs before a task with execution time 1 (schedule A), or it runs after such a task (schedule B). The standard average stretch definition gives $101 + \varepsilon$ for schedule A, and $101 + 1/\varepsilon$ for schedule B. As ε becomes small, in spite of the schedules becoming virtually identical, these stretches diverge and schedule B has an infinite average stretch. Using the scaled sum completion time definition, schedule A obtains $(100 + 2\varepsilon)/(100 + \varepsilon)$ and schedule B obtains $(101 + \varepsilon)/(100 + \varepsilon)$. As ε goes to 0, these two schedules have roughly equivalent stretches. We thus deem the scaled sum completion time more stable than the average stretch.

The last two metrics used in this chapter on PTG scheduling are independent of the scheduling context. They are directly inherent to the scheduling algorithms themselves. The first metric is the asymptotic complexity of an algorithm. It allows us to compare algorithms in more general settings and analyze their behavior with regard to parameters such as the number of tasks and dependencies composing the application to schedule and the number of processors and clusters comprised in the target platform. The second metric is the time needed to build a schedule. It indicates how practical is a given algorithm. Indeed, some algorithms that leads to optimal or near-optimal schedules rely on very time consuming methods, such as linear programming. The scalability of such algorithms is generally limited to small problem instances and often prevent their actual implementation in runtime environments. Moreover, non-guaranteed heuristics are generally faster to produce schedules that are good enough for the vast majority of execution scenarios.

1.3 Evaluation of Scheduling Algorithms

In this section, I briefly detail how the different algorithms presented in the next sections have been evaluated and also explain how I tried to improve my evaluation methodology through the years. Much more details will be given in the next chapter.

Since 2004, and the first scheduling algorithm I proposed in [CDS04], I use simulation for evaluating the performance of algorithms. Simulation allowed me to perform a statistically significant number of experiments for a wide range of application configurations in a reasonable amount of time. Moreover, simulation allows for an objective comparison of several algorithms that is not always made possible by experiments on real platforms. Indeed, simulation ensures that the experimental conditions, *i.e.*, the topology and static and dynamic characteristics of the execution environment, input parameters, execution time of each task composing a PTG, etc., remain exactly the same from one simulation run to another. The only variable component is then how a given algorithm takes scheduling decisions and how it influences the execution of the scheduled application(s).

All the simulators I have developed were based on the same simulation toolkit, that is SIMGRID [39]. SIMGRID provides the required fundamental abstractions for the discrete-event simulation of parallel applications in distributed environments, and was specifically designed for the evaluation of scheduling algorithms. Basing my work upon the same toolkit for more than 10 years, allowed me to benefit of all the improvements, in terms of scalability, validity, and usability, of SIMGRID. I was even able to influence and contribute to some developments that were suited to needs raised by my work on PTG scheduling. For instance, my first simulators were developed using the original Application Programming Interface (API) of SIMGRID, which was removed of the main development branch since the third major release of SIMGRID due to a major rewriting of the simulation kernel itself. I was then forced to use another API, less adapted to my studies, until I have initiated the revival of the original API on top of the new kernel. Another example is the addition of a new simulation model specifically designed to handle moldable tasks that I have initiated. The SIMGRID toolkit and my contributions to the domain of the simulation of distributed systems and applications will be presented in more details in Chapter 2.

I also tried to improve my analysis methodology from article to article. The first algorithms I have proposed [CDS04, NS06, NSC07, NS07] were evaluated by analyzing the average makespan or work achieved by the algorithms versus the problem size, the platform size, or the platform heterogeneity. While such a basic analysis gives a general idea on the respective performance of each algorithm, considering only average values does smooth the results and may hide some pathological instances. This is why I have considered in later works [Sut07, HRS08a, HRS08b] another ways to compare algorithms. First, I counted the number of times that each scheduling algorithm produced better, equal or worse schedule length compared to every other considered algorithm to obtain a pair-wise comparison of algorithms. It allowed me to detect whether an algorithm was better than another for a large part of the simulated scenarios or the good average result was only an artifact coming from a small number of instances for which very good performance was achieved. Second, I studied the *degradation from best* metric. It allowed me to determine the relative quality of the schedules produced by an algorithm when these schedules are not the best ones. The degradation from best is computed by dividing the percent relative difference between the makespan achieved by an algorithm and the best makespan achieved for a given simulation scenario by the total number of experiments. This analysis is a good complement the count of occurrences of better quality schedules. However, it still implies an average over the entire range of experiments and may hide extreme cases. Then, in my last publications to date [DS10, CDS10, CDS10], I have analyzed the distribution of the studied metrics across the range of simulation scenarios and relied on "box-and-whiskers" graphs in my articles. The box represents the Inter-Quartile Range (IQR), *i.e.*, all the values comprised between the 1st (25%) and 3rd (75%) quartiles, while the whiskers show the minimal and maximal values. An horizontal line within the box also indicates the median, or 2nd quartile, value. Such graphs allowed me to analyze more deeply the relative performance of the proposed algorithms by removing any bias introduced by the computation of aver-

age values. A next step towards evaluations reaching the standards used in other sciences would be to conduct a proper statistical analysis. This is unfortunately not a common practice in Computer Science and requires advanced knowledge in statistics that I still have to acquire.

To evaluate the performance of a scheduling algorithm in various conditions, the models of platform and PTG described in Section 1.2.1 have to be instantiated. In my early work on PTG scheduling, I have developed a simple generator of synthetic compute platforms. The generated environments were similar in structure to what is depicted in Figure 1.2. While this approach allows for the generation of a large set of scenarios, ensuring that this set is representative of real distributed infrastructures is a difficult issue. Then I decided in 2007 to use descriptions of existing multi-cluster platforms instead. The set of tested scenarios is obviously smaller, but this is balanced by a gain in terms of realism of the simulation results. As my main aim is to design scheduling algorithms that can be implemented and used, realistic multi-cluster platforms overcome a large number of randomly generated platforms. However, Section 2.4.1 in Chapter 2 will detail all my reflexions and contributions to this complex problem of the generation of synthetic platforms in a simulation context.

To instantiate the PTG model, we need to define specific models for execution times of moldable tasks and for the structure of the task graph. What is detailed hereafter was used in most of my publications. We take a simple approach for modeling moldable task execution times based on Amdahl's law. We assume that a task operates on a dataset of d double precision elements (for instance a $\sqrt{d} \times \sqrt{d}$ square matrix). We arbitrarily assume that processors have at most 1GByte of available memory and thus $d \leq 121M$. We also assume that d is above $4M$ (if d is too small, the moldable task should most likely be fused with its predecessor or successor). We model the computational complexity of a task, in number of operations, with one of the three following expressions, which are representative of common applications: $a \cdot d$ (*e.g.*, a stencil computation on a $\sqrt{d} \times \sqrt{d}$ domain), $a \cdot d \log d$ (*e.g.*, sorting an array of d elements), $d^{3/2}$ (*e.g.*, a multiplication of $\sqrt{d} \times \sqrt{d}$ matrices). For the first two types of complexity a is picked randomly between 2^6 and 2^9, to capture the fact that some of these tasks often perform multiple iterations. We consider four scenarios: three in which all tasks have one of the three computational complexities above, and one in which task computational complexities are chosen randomly among the three. Beyond this model for serial task execution, Amdahl's law specifies that a fraction α of a task's serial execution time is non-parallelizable. We pick random α values uniformly between 0% and 25%. With this model, task execution time strictly decreases as the number of processors increases.

We also use five commonly used parameters to define the shape of a PTG: width, regularity, density, and jumps. The width determines the maximum parallelism in the PTG, that is the number of tasks per level. A small value leads to "chain" PTGs and a large value leads to "fork-join" PTGs. The regularity denotes the uniformity of the number of tasks in each level. A low value means that levels contain very dissimilar numbers of tasks, while a high value means that all levels contain similar numbers of tasks. The density denotes the number of edges between two levels of the PTG, with a low value leading to few edges and a large value leading to many edges. These three parameters take values between 0 and 1. In our experiments we use values 0.2, 0.5, and 0.8 for width, and 0.2 and 0.8 for regularity and density. This leads to PTGs with mean maximum task parallelism of $V^{0.2}$, $V^{0.5}$, or $V^{0.8}$, where V is the total number of tasks, and coefficients of variance 20% or 80%. As a result, we generate PTGs that are close to chain graphs and PTGs that are close to fork-join graphs, with a spectrum of configurations in between. Furthermore we add random "jumps edges" that go from level l to level $l + jump$, for $jump = 1, 2, 4$ (the case $jump = 1$ corresponds to no jumping "over" any level). More details on the graph generation algorithm will be given in Section 2.4.2. In some of my publications [Sut07, HRS08a, DNSC09], I had to adapt this generation method to produce *layered* PTGs with the particularity that all the tasks in a given precedence level have the same cost. Then all the transfers between the same two levels share the same communication cost. For such PTGs the *jump* parameter is always set to 1.

While the above specifies a way to generate a population of synthetic PTGs, I also used in the evaluations real PTGs from the Strassen matrix multiplication and from the Fast Fourier Transform (FFT)

applications. Both are classical test cases for PTG scheduling algorithms. These PTGs are more regular than the aforementioned synthetic PTGs, which are more representative of workflow applications that compose arbitrary operators in arbitrary ways. However, only the structure of the original applications was preserved and, as for the random PTGs, I used different computational complexity scenarios. This is to explore scenarios beyond those corresponding to the actual FFT and Strassen applications.

1.4 Single PTG Scheduling

In this section, I detail the different heuristics I proposed over the last decade to solve the problem of scheduling a single PTG. These contributions are not presented in a chronological order, but according to the taxonomy given by Figure 1.4. Then, I first consider only one homogeneous cluster as a target platform and distinguish heuristics that aim at optimizing only one performance metric from those that optimize more criteria. In a second part, I present heuristics designed for multi-cluster platforms.

1.4.1 On a Single Cluster

Mono-criterion Optimization

The scheduling of a single PTG on a single homogeneous cluster while aiming at optimizing a single performance metric is the simplest problem instance in this field. Nevertheless, this problem is still challenging and has been vastly investigated over the last decade. Generally the proposed algorithms aim at minimizing the makespan of the scheduled application.

This problem has been studied from a theoretical standpoint in a view to find performance guarantees in worst case scenarios. Lepère *et al.* have proposed in [108] an algorithm whose guarantee is $3+\sqrt{5}$. This means that, even in the worst case, the makespan achieved by this algorithm cannot be more than 5.236 times greater than the optimal makespan. Jansen and Zhang [96] have improved this result with a $100/43 + 100(\sqrt{4349} - 7)/2451 \approx 4.731$ approximation. The main issue with such theoretical algorithms is that they ignore the data transfers that occur between dependent tasks. Thus it hinders their applicability to data-intensive applications as data transfers cannot be neglected in this case. Moreover such algorithms usually rely on complex problem solving methods, such as linear programming, that are time consuming. Building a guaranteed schedule takes a lot of time. These guaranteed but long to execute scheduling algorithm are then difficult to use in practice. For this reason many heuristics, *i.e.*, non-guaranteed but fast to execute scheduling algorithms, have been designed.

The pioneering heuristics for PTG scheduling on homogeneous clusters have been proposed by A. Radulescu and A. van Gemund in 2001. They designed two algorithms respectively named Critical Path Reduction (CPR) [124] and Critical Path Area-Based scheduling (CPA) [125].

The CPR algorithm uses an iterative allocation procedure to reduce the makespan of the scheduled application. Each iteration allocates an extra processor to a selected task while it leads to a reduction of the overall makespan. The selected task v_i always belongs to the critical path of the application, *i.e.*, the sum of its top level and bottom level ($tl(v_i) + bl(v_i)$) is maximal. The rationale is that allocating more processors to the critical path is the directest way to reduce the makespan of an application. However, the CPR algorithm does not perform any further selection within the current critical path. Then this procedure is likely to increase the allocation of the same task until it goes out of the critical path and proceed with the next task. The procedure stops when the critical path length cannot be further reduced. This lack of distinction in the critical task selection process may lead to a saturation of the target cluster with the tasks of the critical path. This would lead to the postponing of some tasks that do not belong to it and hinder the performance of the algorithm. Moreover the CPR algorithm is completely focused on makespan reduction. Then the produced schedules can use the whole cluster for a limited gain of performance, which is not efficient.

Algorithm 1 Allocation Procedure of CPA

1: **for all** $v_i \in \mathcal{V}$ **do**
2: $p_i \leftarrow 1$
3: **end for**
4: **while** $T_{CP} > T_A$ **do**
5: $v_i \leftarrow$ task $\in CP \mid \left(\frac{T(v_i,p_i)}{p_i} - \frac{T(v_i,p_i+1)}{p_i+1}\right)$ is maximum
6: $p_i \leftarrow p_i + 1$
7: **Update** T_A **and** T_{CP}
8: **end while**

Algorithm 2 Mapping Procedure of CPA

1: Sort tasks in decreasing order of bottom level
2: **while** not all tasks are scheduled **do**
3: Schedule v_i on the first p_i free processors
4: **end while**

The same authors proposed a second algorithm a few months later that solves some issues of CPR thanks to a different task selection process in the allocation procedure. The CPA algorithm aims at finding the best compromise between the length of the *critical path*, and the *average area* which measures the mean processor-time area required by the application. The allocation procedure of CPA considers that the makespan of an application can be approximated by $T_p^e = \max\{T_{CP}, T_A\}$, where T_{CP} is the execution time of the application critical path and T_A the average area of the application, defined as:

$$T_{CP} = \max_{v_i \in \mathcal{V}} (bl(v_i)), \text{ and} \tag{1.7}$$

$$T_A = \frac{1}{P} \sum_{i=1}^{V} W(v_i). \tag{1.8}$$

The objective of CPA is to minimize T_p^e during the allocation step. Note that T_{CP} decreases whereas T_A increases when more processors are allocated to a task. The procedure then starts by allocating only one processor to each task. This initial allocation leads to a maximal value for T_{CP} and a minimal value for T_A. Then each iteration allocates one more processor to the most critical task while $T_{CP} > T_A$. This selected task is the task belonging to the critical path that benefits the most of the addition of an extra processor. This benefit is measured as the difference between the execution time over allocation ratios with the current and potential allocations, *i.e.,*

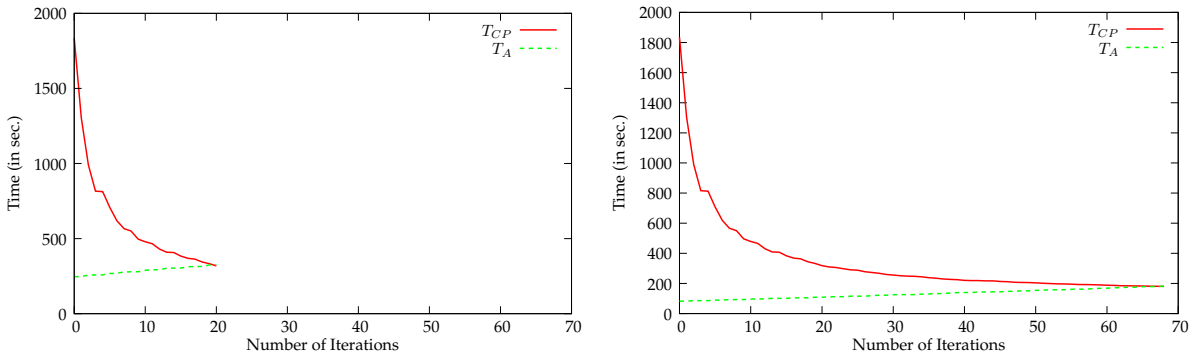
$$\frac{T(v_i, p_i)}{p_i} - \frac{T(v_i, p_i + 1)}{p_i + 1}.$$

When verified, the stopping condition ($T_{CP} \leq T_A$) implies that T_p^e will be very close to its minimal value ($T_p^e \approx T_{CP} \approx T_A$) provided that a good mapping is performed.

In the mapping step, the ready task with the highest bottom level is considered at each iteration. This step includes data redistribution costs to determine the start and end dates of each scheduled task and the set of processors allowing this task to finish as early as possible. Algorithm 1 presents the pseudo-code of the allocation procedure of the CPA algorithm, while Algorithm 2 shows the pseudo-code of the mapping procedure.

Despite the good performance claimed by its authors, CPA suffers from two glaring drawbacks that hinder the quality of the produced schedules. First, the experiments conducted by Radulescu *et al.* only involve PTGs with a small number of tasks and clusters made of a small number of nodes. Though, the computation of the average area given by Equation 1.8 becomes less pertinent when the number of computing resources is well over the number of tasks, *i.e.,* $P >> V$. This unsuspected behavior is illustrated by Figure 1.5.

Figure 1.5(a) shows the evolution of T_{CP} and T_A when CPA determines allocations for the PTG displayed in Figure 1.3 on a cluster comprising 10 processors. We see that the convergence point is quickly



(a) Cluster of 10 processors.

(b) Cluster of 30 processors.

Figure 1.5: Evolution of T_{CP} and T_A, throughout the allocation procedure of CPA for a random PTG of 6 tasks on clusters of 10 (a) and 30 (b) processors.

reached, after only 20 iterations. Figure 1.5(b) shows what happens when CPA determines allocations for the same PTG, but on a larger cluster comprising 30 processors. In such a configuration, more iterations are required (68) to make T_{CP} and T_A reach close values. While the evolution of T_{CP} remains unchanged, T_A grows more slowly as the total work is divided by a greater number of processors.

The direct consequence of this greater number of iterations of the procedure is the allocation of more processors to the different tasks of the PTG. This may prevent the potential concurrent execution of independent tasks for a very limited gain on the reduction of the critical path length. For instance, the three tasks of the second level of the PTG are respectively allocated 24, 8, and 1 processors. As the target cluster only comprises 30 processors, one task necessarily has to be postponed. This in turn increases the application makespan.

The second drawback of the CPA algorithm is due to its focus on the reduction of the critical path length. The allocation procedure thus ignores the potential task parallelism exhibited by the PTG structure and the possibility to have several paths of similar length in the application. This drawback becomes even more glaring for very regular applications. Let consider the example of a Strassen matrix multiplication algorithm to illustrate this. In this application, the tasks with the highest computing needs are the seven inner matrix multiplications. Then the allocation procedure of CPA will preferentially allocate extra processors to these tasks. Moreover all the paths in the PTG are almost equivalent. They differ at most by a single matrix addition. Figure 1.6 shows the resulting schedule on a cluster of 20 processors.

On this particular example, the allocation procedure stops when the critical path length is equal to 78.2 and the average area is equal to 78.7. But we see that it is impossible to concurrently execute the seven inner products tasks (the big blue tasks labeled from 11 to 17) although they are independent. Indeed the sum of the allocations for these tasks is 23 while the target cluster is only made of 20 processors. Then one product (task 12) is postponed and the schedule length dramatically increases to 122.3 seconds. It is interesting to note that an allocation allowing to execute the seven products in parallel is possible and leads to a shorter schedule and a smaller total work. For instance, if three processors are allocated to six of the inner products and only two to the last one (for a total of 20 processors), the schedule length depends on the execution time of this less well served task. In our example, executing a matrix multiplication on two processors would take 86.9 seconds while the sequential execution of tasks 11 and 12 in Figure 1.6 takes 106.5 seconds. Moreover less work would be required to execute the inner products and then some extra processors could be allocated to some other tasks to further reduce

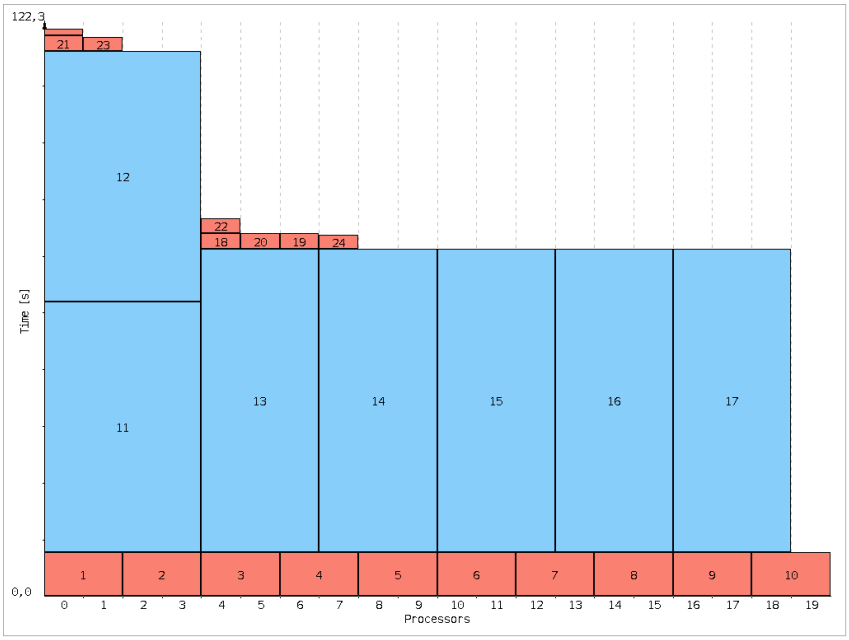


Figure 1.6: Schedule of a Strassen matrix multiplication produced by the CPA algorithm on a cluster of 20 processors.

the makespan.

These two simple examples have illustrated some issues raised by the CPA algorithm. They have been addressed by several heuristics since the publication of the original article by Radulescu and Van Gemund. In [NSC07] we proposed, with T. N'Takpé and H. Casanova, to accelerate the convergence of the allocation procedure of CPA by modifying the computation of the average area T_A. The rationale was to avoid large allocations that offer only a little gain in terms of makespan while hindering the exploitation of task parallelism. We saw in Figure 1.5 that the number of iterations was directly related to P as it divides the total work in the computation of T_A. To solve the potential issues arising when $P >> V$ we proposed the following computation of the average area:

$$T_A^{geo} = \frac{1}{\min\left(P, \sqrt{V \times P}\right)} \sum_{i=1}^{V} W(v_i). \tag{1.9}$$

Using the geometric average of V, the number of tasks in the PTG, and P, the number of processors in the target cluster, to compute the average area is an empirical choice. We determined many schedules for various values of P and V and found out that it was a good compromise between makespan and work, especially for $P >> V$. Moreover the use of P as a lower bound was introduced to prevent bad schedules in the opposite configuration, *i.e.*, when $V >> P$. Indeed, if $V \geq P$, *i.e.*, there are more tasks to schedule than available processors, T_A^{geo} gives exactly the same value as the original T_A. Then the produced schedules are the same too. Figure 1.7 shows the evolution of T_{CP}, T_A and, T_A^{geo} in a configuration such that $P > V$ (6 tasks to schedule on a cluster of 20 processors).

We see that the use of T_A^{geo} leads to a higher initial value that, combined with a steeper slope, reduces the number of iterations of the allocation procedure by a factor greater than two. However, such a stringent reduction has only a moderate impact on the length of the critical path. With T_A^{geo} the value of T_{CP} when convergence is reached is 272.58 seconds while it was 180.10 seconds with the original T_A. However, if we consider the final schedules that are obtained after the execution of the mapping

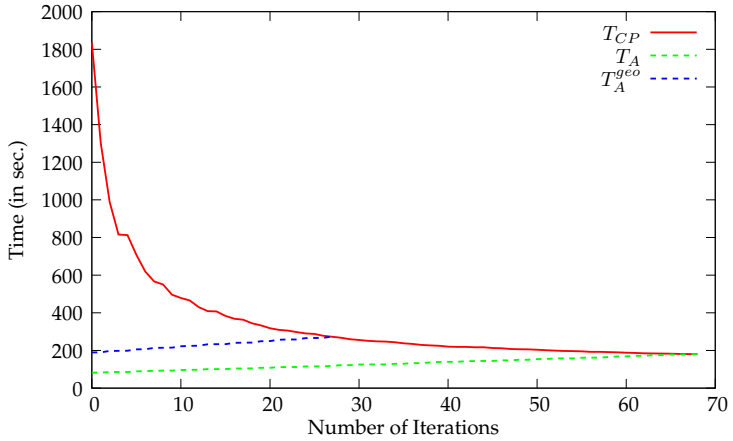


Figure 1.7: Evolution of T_{CP}, T_A, and T_A^{geo} throughout the allocation procedure for a random PTG of 6 tasks on a cluster of 30 processors.

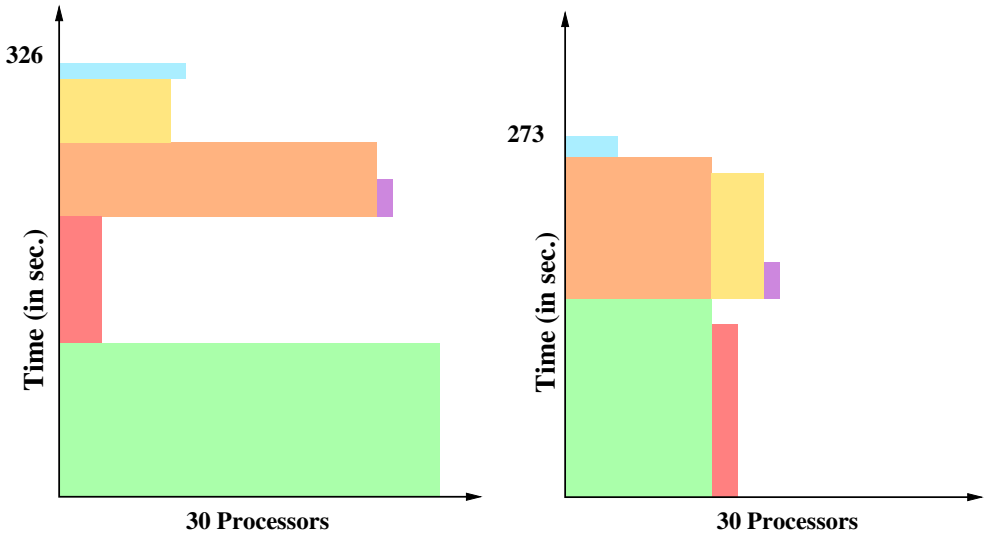
step, we see on Figure 1.8(a) that the schedule length is far from the estimated value. Indeed, the large allocations determined in the first step prevent possible concurrent executions in the second and cause the postponing of some tasks.

On the contrary Figure 1.8(b) shows that the use of T_A^{geo} has several advantages. First, the scheduling length is more conform to the estimation made in the allocation procedure, and much shorter than that achieved with T_A. Second, less processors are used (16 instead of 29). This schedule is then more efficient in terms of resource usage.

The main drawback of T_A^{geo} is that it has no more the same unit as T_{CP}. Indeed, the original T_A is a sum of areas, *i.e.,* a product of time by a number of resources, divided by a number of resources. Then it is homogeneous with T_{CP} that is a time. As we introduce V, the number of nodes in the PTG, in the formula we break this homogeneity. This may make questionable a comparison of two values expressed in different units. This issue was addressed in another work that will be detailed in Section 1.4.1.

The second drawback of CPA, *i.e.,* only considering the critical path in the allocation process, was first addressed in 2006 by Bansal *et al.* in [15]. They proposed the Modified Critical Path Area-Based scheduling (MCPA) algorithm which is, as its name says, a straightforward modification of the original CPA algorithm. To prevent the postponing phenomenon illustrated by Figure 1.6, this algorithm proposes to add an extra condition to the allocation procedure of CPA algorithm. At each step of the allocation procedure a task has now to respect two conditions to be selected. It still has to be the task belonging to the current critical path that benefits the most of an extra processor allocation. The additional condition is that the number of processors currently allocated to critical tasks at the same *precedence level* as the candidate task is strictly lower than the total number of processors in the target cluster. In other words, once an extra processor has been allocated to the selected task, the algorithm ensures that all the important tasks at this precedence level can still be executed simultaneously without postponing.

This approach that considers the precedence levels of tasks was also followed by the algorithm proposed in [91]. This algorithm relies on a more complex selection criterion to deal with heterogeneous precedence levels. Indeed, if tasks that belong to the same precedence level have very different computational requirements, it may be preferable to give more processors to the most consuming tasks even tough it means to allocate more than P processors at this level. This will lead to the creation of a second virtual layer of tasks in a way to reduce the overall execution time of this level. To determine which levels have to be *extended*, the algorithm in [91] defines two ratios, *i.e.,* the *cover* ratio and the *width* ratio. The former is defined as the ratio between the total work of the level and P times the largest execution



(a) With T_A. (b) With T_A^{geo}.

Figure 1.8: Gantt chart of the schedule of a six-task PTG on a cluster of 30 processors using the original T_A (a) and T_A^{geo} (b).

time in this level with the current allocation. The latter ratio allows the procedure to distinguish the levels that comprise only a small number of tasks. Indeed, such levels are not likely to be affected by concurrency issues.

The different aforementioned works aim at improving the first step dealing with resource allocations. Optimizations can also be made during the second step that maps allocated tasks on the target cluster. All these algorithms rely on a classical list scheduling heuristic. The tasks are ranked in decreasing order of bottom level. This ensures the respect of data dependencies between tasks and give higher priority to the tasks that are the farthest from the end of the application. Then the procedure aims at mapping each task in order on the first set of processors of the appropriate size, *i.e.*, that of the determined allocation, that becomes available. A first optimization is to take data transfers costs in account in the processor set determination process. Then the best candidate processor set will be the one minimizing the data movements between all parent tasks and the currently scheduled task.

One potential issue is that the mapping of tasks can be made more difficult due to rigid processor allocations computed in the first step. In particular, a task may be delayed unnecessarily just because its allocation is (perhaps only slightly) larger than the number of processors available when the task is ready for execution. In practice, we observed "holes" in schedules due to such a phenomenon. Figure 1.9 illustrates this situation and the solution that we proposed in [NSC07].

The proposed optimization is the following. Consider a task to be scheduled, whose original processor allocation was computed in the first step of the algorithm, *e.g.*, task 3 in Figure 1.9. The allocation of this task prevents it to start as soon it is ready. Task 3 has to wait for the completion of task 2 so that enough resources are available to satisfy its allocation. As stated before, this creates an idle time and an unnecessary delay. In such a case, we determine if, by using a smaller allocation, the task could be started earlier and finish no later than when using its original allocation. If so, we use the smallest such allocation, otherwise, the packing is not allowed. This is illustrated by the right part of Figure 1.9. This optimization was called *allocation packing* in [NSC07]. While simple, this allocation packing strategy has the good property to not degrade the original schedule. As an allocation is modified if and only if it results in an earlier finish time for the considered task, the whole schedule length can only be shortened.

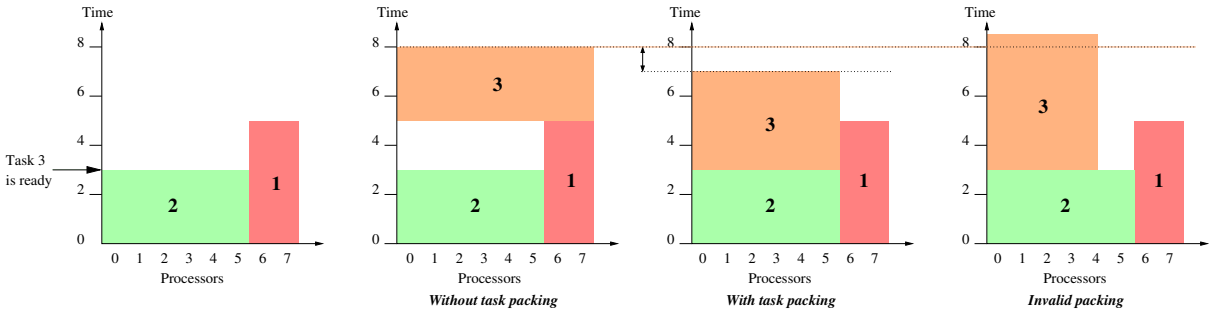


Figure 1.9: Illustration of the *allocation packing* performed during the mapping step.

This technique was further developed with S. Hunold and T. Rauber in [HRS08a] in which we allowed us to modify allocations during the mapping. The main goal of the proposed optimization was to save some unnecessary or costly data redistributions that may have an impact on the overall performance. This is particularly true for data intensive applications. Indeed, the totally decoupled allocation and mapping procedures of two-step scheduling algorithms may cause important data redistributions to satisfy data dependencies and favor network contention. As we assume that there is no data redistribution if two subsequent tasks are mapped on the same set of processors, modifying a task allocation to match that of one of its predecessor does not lead to a data redistribution anymore.

Two different strategies can be applied as shown by Figure 1.10. The former consists in *packing* a task, *i.e.*, reducing its allocation, to obtain the same number of processors as its parent task while the latter *stretches* the allocation, *i.e.*, allocates more processors to the task. Both strategies were implemented in the Redistribution-Aware Two-Steps (RATS) algorithm in [HRS08a] and are detailed hereafter.

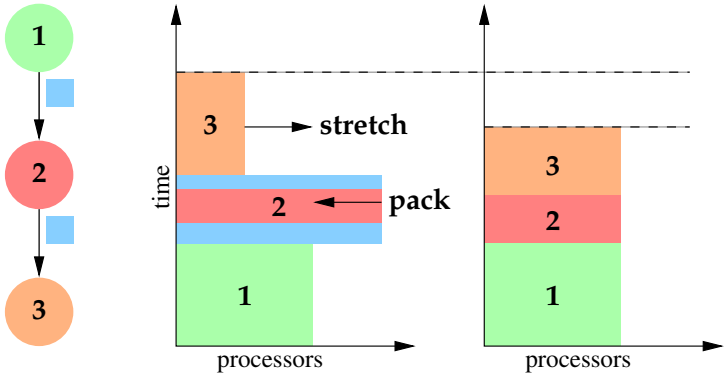


Figure 1.10: Motivating example to stretch or pack task allocations to save some data redistributions.

Stretching an allocation may lead to a double gain, as it avoids a data redistribution and reduces the execution time of the task, but at the price of a higher resource usage. It may also prevent concurrent execution of ready tasks. Moreover tasks allocated on small sets of processors by the allocation procedure of CPA are not critical. Conversely, packing the allocation of a task increases its execution time, as the performance model is assumed to be monotonically decreasing. But this can be compensated by two consequences of reducing the number of allocated processors. First, it may allow a task to start earlier as it has to wait for the availability of less processors. Then using a smaller allocation leaves more room for the execution of other potentially concurrent tasks and thus increase the exploitation of task parallelism. In both cases, the number of processors that can be added or removed to a determined

allocation has to be bounded. We propose the two following options.

The *delta* strategy is only concerned by avoiding a redistribution. We define δ_i as the minimal difference between the allocations of task v_i and that of one its predecessors:

$$\delta_i = \min_{v_j \in pred(v_i)} (abs(p_j - p_i)) . \tag{1.10}$$

Note that the difference $p_j - p_i$ can be either positive or negative. A positive value means that the allocation of the predecessor is larger and that the allocation of task v_i has to be stretched. Conversely a negative value means that the allocation of v_i can be packed. Then we determine δ_i^{max}, the maximal allowed value for δ_i on a per task basis as a parameter of the mapping procedure. This `maxdelta` parameter describes the fraction of the number of processors of the original allocation that can be added or removed. For example, if $p_i = 6$ and `maxdelta = 0.5`, this means that a stretched allocation can comprise at most 9 processors ($6+0.5\times6$) and that a packed allocation can comprise at least ($6-0.5\times6$). Then $\delta_i^{max} = 3$ for this task. According to these definitions, when a task v_i is ready to schedule, the *delta* mapping procedure:

1. Checks if $\delta_i \leq \delta_i^{max}$,
2. Finds the predecessor(s) of v_i corresponding to that δ_i. Keep the original allocation if there is no corresponding predecessor,
3. Maps v_i on the same processors as those of the selected predecessor (if found).

When two predecessors have the same δ_i but one with a smaller allocation than that of v_i and the other with a larger allocation, we break the tie by stretching the allocation of v_i.

The *time-cost* strategy takes care of the additional work implied by the stretching of an allocation. Indeed, if an allocation is packed, the work is automatically reduced and the time can also be reduced provided that the finish time of the task is not worse than before packing. We consider the ratio between the work corresponding to the original allocation of a task and the work achieved if this task were to be executed on one of its parents' allocation:

$$\rho_i = \max_{v_j \in pred(v_i)} \left(\frac{T(v_i, p_i) \times p_i}{T(v_i, p_j) \times p_j} \right) . \tag{1.11}$$

This parameter takes values in the $]0 \cdots 1]$ interval. The closer ρ_i is to 1, the better it is, as this means a better balance between the reduction of the execution time of a task and the augmentation of the work needed for its execution. In addition to this ratio, we also define a threshold, ρ_{min} to determine the candidate allocations. According to this definition, when a task v_i is ready to schedule, the *time-cost* mapping procedure:

1. Checks if $\rho_i \geq \rho_{min}$,
2. Finds the predecessor(s) of v_i corresponding to that ρ_i. Keep the original allocation if there is no corresponding predecessor,
3. Maps v_i on the same processors as those of the selected predecessor (if found).

Another important issue is related to the order in which the ready tasks are considered for mapping. Indeed, when a task finishes its execution, more than one of its children may become ready. This raises the following question: "Which of these tasks has to be handled first?" As the different candidates have at least one predecessor in common, taking an allocation modification decision for one of them can have a negative impact on the others. For instance, stretching the allocation of a task may cause the postponing of potentially concurrent tasks by not leaving enough resources available.

As mentioned earlier, the CPA mapping procedure sorts the list of ready tasks by decreasing bottom level values. We keep this ordering of the list of ready tasks but apply a secondary stable sort to order tasks of same priority. We apply two different sorting strategies that occur before mapping a ready node. The delta sorting strategy orders tasks by increasing δ_i values. The rationale is to prioritize tasks which require less modifications of their initial allocation. In the time-cost strategy we compute the maximal gain in terms of execution time for each ready task. It corresponds to the time that could be saved if this task were to be executed on one of its parents' processor set. We define this gain as:

$$gain(v_i) = \max_{v_j \in pred(v_i)} \left(T(v_i, p_i) - T(v_i, p_j) \right), \tag{1.12}$$

and use it to sort the ready tasks by decreasing $gain(v_i)$ values. Algorithm 3 summarizes the application of these two strategies by the RATS algorithm.

Algorithm 3 Redistribution-Aware Two-Step Scheduling

```
1: compute allocation
2: while not all nodes scheduled do
3:     for each ready node do do
4:         compute delta / estimate execution time
5:     end for
6:     sort ready nodes
7:     while list of ready nodes is not empty do
8:         node = pop from list of ready nodes
9:         if a parent allocation matches delta or time-cost conditions then
10:            map node onto parent's allocation
11:            recompute the values delta or execution time for all ready nodes only if they have been computed
    using this parent allocation
12:            resort ready nodes if necessary
13:        else
14:            map on original allocation
15:        end if
16:    end while
17: end while
```

As an alternate or a complement to these algorithms, note that it is also possible to reduce the makespan of the scheduled application once the complete schedule has been built. The list scheduling heuristic common to all the algorithms derived from CPA enforces the order in which tasks are mapped onto the target cluster. As described by Algorithm 2, allocations are satisfied by the first free processors. But a processor is considered as free once it has completed all the tasks assigned to it. Idle slots created by the schedule are then not considered as candidate although some tasks may fit in. Such a situation is illustrated by Figure 1.11. For the sake of simplicity tasks are labeled from the order in which they are considered for mapping.

As v_3 is mapped after v_2, it has to wait for the completion of this task to find enough free processors to be executed. However, v_3 is small enough to be executed on the processors let available by the execution of v_1 and not used by v_2. A *backfilling* post-processing step will look for all the idle slots that exist, and try to find an appropriate slot for each task. Some conditions have to respected to move a task earlier in the schedule. First, the control dependencies have to be respected, *i.e.,* a task cannot start before the completion of all its predecessors. Second, this backfilling post-processing step has to be *conservative*, *i.e.,* a backfilled task should not delay the start time of any other task. These packing and backfilling optimizations have also been used in the mapping step of the algorithm proposed in [91].

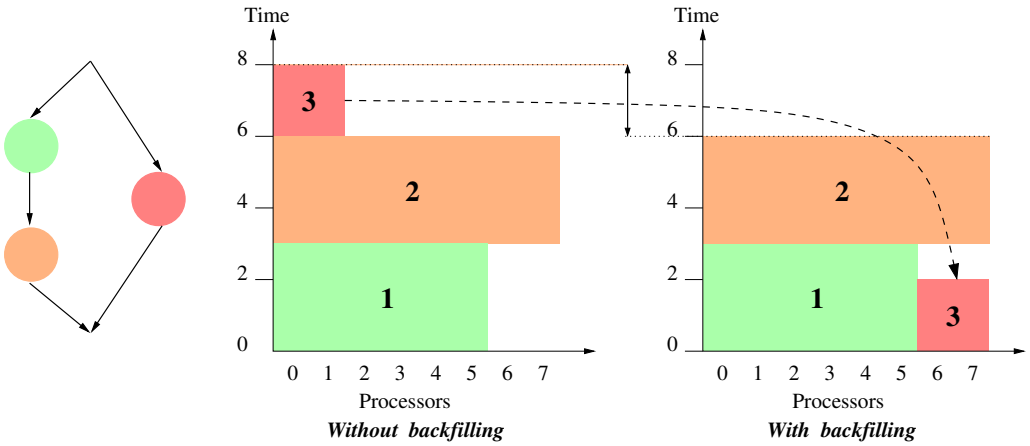


Figure 1.11: Illustration of the benefits of a *backfilling* post-processing step.

As mentioned earlier, it is also possible to schedule PTGs on a homogeneous cluster in a single step. This is the approach followed by the authors of the iterative Coupled processor Allocation and Scheduling algorithm with Lookahead and Backfill (iCASLB) [150] and its improved version, the Locality Conscious Mixed Parallel processor allocation and Scheduling algorithm (LoC-MPS) [151]. This last algorithm starts with an initial allocation that aims at giving its best allocation to each task, *i.e.*, the allocation that minimizes its execution time, while taking the allocations of subsequent dependent tasks into account. From this initial allocation pseudo-edges are inserted in the original PTG to represent *induced dependencies* caused by resource limitations.

Then an iterative procedure is applied to reduce the critical path length of this modified task graph. The improvement of LoC-MPS with regard to iCASLB comes from the distinction it makes about the component that dominates the makespan, *i.e.*, computation or communication costs. If the makespan is dominated by computation costs, LoC-MPS proceeds as CPA by adding an extra processor to the allocation of a task in the critical path. This task is selected according to the potential improvement of its execution time and its concurrency ratio, *i.e.*, how many other tasks could be executed in parallel of that task. When communication costs dominate the critical path, another optimization strategies are used by LoC-MPS to further reduce the makespan. First, the algorithm may increase the allocations of source, destination, or both communicating tasks in order to exhibit more parallel data transfers. Then this algorithm favors processor reuse, *i.e.*, tries to maximize the overlap between the processor sets allocated to the sending and receiving tasks.

Finally, both the iCASLB and LoC-MPS algorithms resort to two additional techniques to improve the produced schedule at each iteration. A look-ahead technique is implemented to avoid a premature stop of the algorithm due to local minima. This look-ahead is bounded in terms of number of iterations to prevent a dramatic complexity increase. Then a backfilling step is applied after each decision to reduce idle times as much as possible as it was explained earlier. In the case of LoC-MPS, this backfilling is locality conscious, *i.e.*, it tries to maximize the overlap between processor groups.

Such an integrated approach is appealing as it may solve the main potential drawback of two-step algorithms. However, the complexity of algorithms such as iCASLB and LoC-MPS is much higher than that of their two-step contenders. Their authors claim that this high complexity can be afforded as many PTGs comprise only a small number of tasks. But experiments conducted in [DS10] have shown that the scheduling time of iCASLB is also impacted by the size of the target cluster and can become greater than the execution time of the application. Some simpler techniques such as those proposed in [HRS08a] or [91] may then be preferred.

Bi-criteria Optimization

All the algorithms in the previous section aimed at reducing the application makespan. Some of them also consider work as a secondary objective, but more as some kind of upper bound to respect than as a priority performance criterion. In [DS10], we introduce, with F. Desprez, the Bi-Criteria Critical Path Area-Based scheduling (biCPA) algorithm that is able to optimize these two performance metrics either simultaneously or separately. This algorithm was built upon the same observation as for the definition of previously introduced T_A^{geo}. But we saw that this new definition did not preserve the homogeneity of the relation between T_{CP} and T_A. On the contrary, the biCPA algorithm allows for a faster convergence of the allocation procedure while relying on a homogeneous stopping criterion.

Figure 1.12, which is similar to Figure 1.5, shows the evolution of the critical path length T_{CP} and the average area T_A during the allocation procedure of CPA for a PTG of 50 tasks on a cluster that comprises 20 processors. We see that 96 iterations are needed to reach the desired trade-off between T_{CP} and T_A.

We already mentioned that the number of iterations of the allocation procedure strongly depends on how T_A and T_{CP} evolve. Due to the chosen performance model, the gain on T_{CP} tends to decrease as more processors are allocated. The slope of the evolution curve of T_A depends on the number of processors P. By computing the average over a value P' smaller than P, this slope will then be steeper. As a consequence the allocation procedure will converge faster. For instance, if $P' = 10$ (instead of $P = 20$) in Figure 1.12, the allocation procedure stops after 34 iterations. The question is thus to choose the best value for this P'. The improvement of CPA detailed in the previous section used $P' = \min\left(P, \sqrt{V \times P}\right)$.

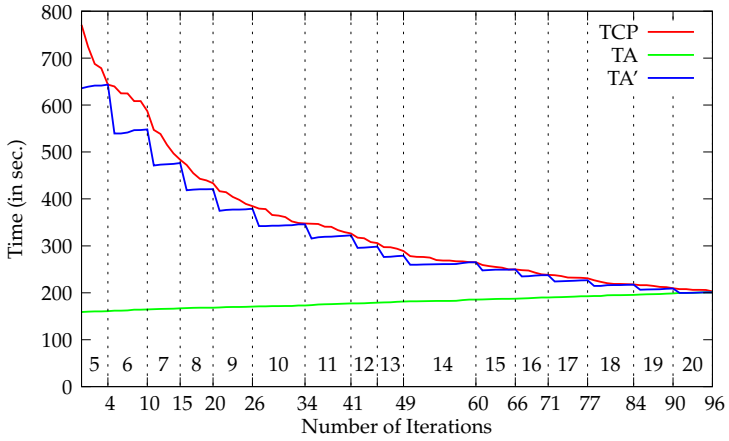


Figure 1.12: Evolution of T_{CP}, T_A, and T'_A throughout the allocation procedure of CPA for a random PTG of 50 tasks on a cluster of 20 processors.

The biCPA algorithm bases its allocation procedure on a new definition of the average area denoted as T'_A. This variant of the average area is defined by

$$T'_A = \frac{1}{P'} \sum_i W(v_i), \tag{1.13}$$

Figure 1.12 also shows the evolution of T'_A throughout the allocation procedure. The value of P' is incremented each time T'_A becomes larger than T_{CP}. The evolution of P' is depicted by the labels at the bottom of Figure 1.12.

An interesting fact is that each time P' is incremented, the current task allocations correspond to those that would have been determined by CPA if the cluster has comprised P' processors. Moreover

all these intermediate allocations can be determined during the execution of the original allocation procedure of the CPA algorithm. These intermediate allocations are the key information needed by the biCPA algorithm to find the best compromise between makespan and work.

Algorithm 4 presents the allocation procedure of the biCPA algorithm which relies on this definition of T'_A. The main difference with the allocation procedure of the CPA algorithm lies in the outer for loop (lines 4-14). This loop sets the value of T'_A that will be used in the inner loop (lines 6-10). Note that this inner loop actually corresponds to an interval of iterations of the seminal allocation procedure. Each time $T_{CP} \leq T'_A$, the current allocation is stored for each task (lines 11-13). At the end of this procedure, P distinct allocations are then associated with each task in the PTG.

Algorithm 4 The biCPA allocation procedure

1: **for all** $v_i \in \mathcal{V}$ **do**
2: $p_i \leftarrow 1$
3: **end for**
4: **for** $j = 1$ to P **do**
5: $T'_A = \frac{1}{j} \sum_i W(v_i)$
6: **while** $T_{CP} > T'_A$ **do**
7: $v_i \leftarrow$ task $\in CP | \left(\frac{T(v_i,p_i)}{p_i} - \frac{T(v_i,p_i+1)}{p_i+1} \right)$ is maximum
8: $p_i \leftarrow p_i + 1$
9: **Update** T'_A **and** T_{CP}
10: **end while**
11: **for all** $v_i \in \mathcal{V}$ **do**
12: Store $p_i^j \leftarrow p_i$
13: **end for**
14: **end for**

The second step of the biCPA algorithm consists in getting an estimation of the makespan and total work that can be achieved with each of these P allocations. To obtain these performance indicators, the biCPA algorithm relies on the same list scheduling algorithm as the CPA algorithm, *i.e.,* Algorithm 2. Once a mapping has been found for each task of the PTG, we determine the makespan, $C_{P'}$, and total work, $W_{P'}$ obtained with P'. We also denote as C_p and W_p the makespan and total work achieved when $P' = P$, that is with the original allocation procedure of the CPA algorithm. From these makespan and total work estimations, the biCPA algorithm is able to output four interesting schedules among all the computed schedules.

The first two schedules aim at optimizing both metrics simultaneously. A first step is to determine, for each candidate allocation, the gain it offers with regard to each metric. This gain is measured by dividing the makespan and work achieved with the considered allocation respectively by the makespan and work obtained for P processors. A value smaller than one indicates a shorter completion time or less required work. Conversely, a ratio greater than one shows a performance degradation. Note that these relative makespan and work also tell us if the schedule produced by the CPA algorithm can be improved. Then we can determine which allocations lead to non-dominated solutions. The best trade-off between the two objectives can be found among these allocations.

In a multi-objective optimization problem, there are at least two ways of defining what should be a good trade-off. A first definition is to find a solution that leads to the same improvement on each criterion. In our particular context this means an allocation that reduces the makespan and work with regard to the allocation of the CPA algorithm in the same proportion formalized by Equation 1.14

$$\frac{C_{P'}}{C_P} = \frac{W_{P'}}{W_P}, \tag{1.14}$$

If several solutions satisfy to Equation 1.14, the best one will be the one with the smallest relative

makespan. Another definition of a good trade-off is to maximize the sum of the improvements that a solution achieves on each criterion. In our context, small values for relative makespan or work mean better performance. The best trade-off will be the allocation that minimizes

$$\frac{C_{P'}}{C_P} + \frac{W_{P'}}{W_P}. \tag{1.15}$$

Ties produced by this equation are broken by selecting the allocation that leads to the smallest work.

As a side effect, two schedules that optimize one metric each are also produced. Indeed, we can select the allocation that leads to the shortest estimation of the makespan. This allocation is found by sorting the candidate allocations by increasing makespan and picking out the first element. The last schedule produced by the biCPA algorithm is the one that requires the smallest amount of work to execute the PTG. As for the schedule that minimizes the makespan, the corresponding allocation is found by sorting the allocations, this time by increasing total work. Recall that our main aim is to design a bi-criteria scheduling algorithm. We then discard the solutions that leads to an improvement of one criterion but degrades the other one, *i.e.*, such that $C_{P'} > C_P$ or $W_{P'} > W_P$. Such a situation occurs when trying to minimize the work needed to execute a PTG as only a few processors are used.

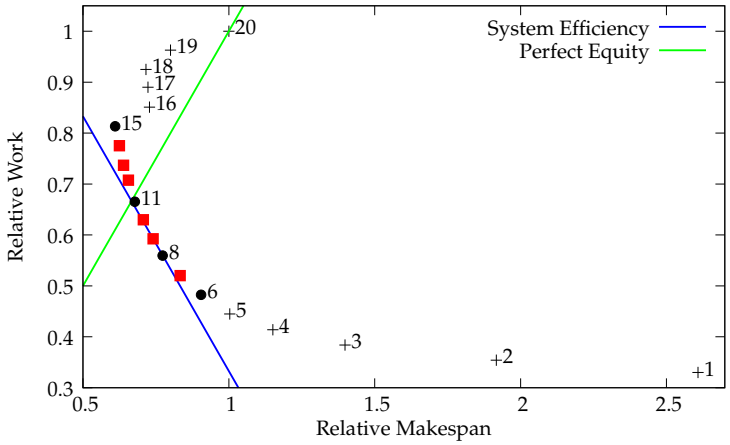


Figure 1.13: Evolution of $C_{P'}/C_P$ and $W_{P'}/W_P$ when P' varies for a random PTG of 20 tasks on a cluster of 20 processors.

Figure 1.13 illustrates and summarizes the different allocations selected by the biCPA algorithm. This figure shows the relative makespan ($C_{P'}/C_P$) and work ($W_{P'}/W_P$) when P' varies for the scheduling a random PTG of 20 tasks on a cluster of 20 processors. The crosses depicts the discarded options, either because they are dominated (from 16 to 20) or degrading one of the criterion (from 1 to 5). The red squares correspond to the non dominated solutions while the black circles are the four values selected by the biCPA algorithm: (i) the best makespan improvement is achieved with $P' = 15$; (ii) the best work without makespan degradation is obtained when $P' = 6$; (iii) with $P' = 11$, Equation 1.14 (whose solutions are depicted by the green line) is satisfied; and (iv) the sum of the two relative values is minimized when $P' = 8$. The blue line shows the system efficiency that corresponds to this minimum.

A seminal result in the area of PTG scheduling from a theoretical standpoint is the guaranteed two-step algorithm proposed in [107]. It relies on a earlier work by Skutella [138] that gives a linear program to find the task allocations that lead to the best possible trade-off between the length of the critical path and the average work per processor. This problem is very similar to a classical time-cost trade-off problem (see [53]) in which we have two lower bounds on the metric to be optimized: the length of

the critical path and the average work per processor. Reducing the number of processors allocated to any task affects this trade-off in favor of a smaller average work, while it may increase the critical path. Conversely, increasing the number of processors used to compute a task is likely to shorten the critical path and to increase the average work per processor. The goal is to achieve the best trade-off between the two, *i.e.*, minimizing their maximum. Following the approach in [138], we hus have to solve a discrete optimization problem, *i.e.*, finding the optimal trade-off by picking for each task a particular allocation among a finite set of possible allocations.

As with many problems, solving the discrete problem is strongly NP-hard, while solving a continuous version of the problem is easy. The idea here is then to first solve a larger, but continuous problem, in which each task v is replaced by a set of $m-1$ "activities". Each activity has a continuous linear cost function defined based on execution time. To each activity corresponds a variable and a cost that are used to define a rational linear program. Solving this linear program, detailed in [DNSC09] and in [138], allows us to determine the minimal cost necessary to achieve any critical path length $\overline{C_{max}}$ and the corresponding work $\overline{W}$. The objective is to find the optimal trade-off between these two values. The $\overline{C_{max}}$ and $\overline{W}$ values are continuous and not necessarily integers. Since $\frac{\overline{W}}{m}$ and $\overline{C_{max}}$ have opposite behavior, there are two possible scenarios. If one is always larger than the other, one can use straightforward extreme allocations (each task uses one processor if $\frac{\overline{W}}{m}$ is always larger, or all processors if $\overline{C_{max}}$ is always larger). Otherwise, an optimal trade-off can be approached by binary search. In the latter scenario, the values obtained are lower bounds of the optimal discrete trade-off, that is of the discrete values of the critical path length and of the total work so that the maximum of the critical path length and of the average work per processor is minimized. The work in [138] uses a rounding technique to turn the continuous solution into a solution of the discrete problem. This techniques leads to discrete C_{max} and W values that are at most a factor $\frac{1}{1-\mu}$ and $\frac{1}{\mu}$ larger than the optimal discrete values, respectively, where μ is a parameter that can be chosen arbitrarily between 0 and 1.

Based on the linear programming approach in [138], the work in [107] focuses on how to schedule the tasks efficiently while preserving most of the allocations so that one can obtain a performance ratio derived from the lower bounds on the critical path and the average work per processor. The difficulty comes from the fact that the allocations are computed in a setting where an infinite number of processors can be used at the same time, since there are no constraints in the linear program on simultaneous execution of data-parallel tasks. With tasks with different execution times there is no simple geometrical transformation to transform a schedule for an unbounded number of processors into one for a fixed number of processors. The schedule has to be reconstructed from scratch, only keeping the allocation information. The algorithm proposed in [107] is derived from the classical list scheduling algorithm. However, list scheduling cannot be used directly as it can be arbitrarily far from the optimal schedule. Consider for example an instance with m pairs of tasks where the first task has to be executed on all processors for a very short amount of time, while the second task has to be scheduled afterwards on a single processor for a long time. The worst case for list scheduling is to schedule all pairs one after the other, while the optimal is to schedule all the first tasks, and then all the second tasks in parallel resulting in a schedule without idle time. To avoid the problem of having lots of ready tasks requiring too many processors, the solution is to enforce a maximum number of processors per task noted b. Simply put, the algorithm inserts a bounding step between allocation (derived from the time cost linear-program with parameter μ set to $\frac{1}{2}$) and placement (according to a list scheduling algorithm). The optimal b can then be computed depending on the total number of processors, and the expected performance ratio. See [107] for all details on the algorithm achieving a performance ratio of $3+\sqrt{5}$.

1.4.2 On a Multi-Cluster

Going from a single homogeneous cluster to a multi-cluster increases the complexity of the scheduling process. Indeed multi-cluster platforms are inherently heterogeneous. At best, this heterogeneity lies in

the network interconnect as the route connecting two processors belonging to different clusters is obviously longer than one between processors within the same cluster. At worst, the processing capabilities of the different clusters are also heterogeneous. In this case, the execution time of a given task differs from one cluster to another.

Two approaches can be followed to schedule PTGs on heterogeneous platforms. The first approach consists in adapting the aforementioned algorithms for PTG scheduling on homogeneous platforms and making them amenable to heterogeneous platforms. The second approach consists in adapting list heuristics that were specifically designed for scheduling DAGs on heterogeneous platforms [26, 86, 119, 131, 137, 147] and making them amenable to moldable tasks. For both approaches, a common assumption is to not span the allocation of any task across more than one cluster. This restriction is a convenient way to ensure that we can reuse the speedup models traditionally employed in the literature. Moreover, the performance of the execution of a task is likely to be badly impacted by communications over an inter-cluster network with a higher latency.

In [NS06], we followed, with T. N'Takpé, the first approach. We adapted the already introduced CPA algorithm [125] to propose the Heterogeneous Critical Path Area-Based scheduling (HCPA) algorithm. To remove the difficulty related to the different processor speeds in a heterogeneous platform, the HCPA algorithm reasons about allocations using an equivalent *virtual homogeneous cluster*. For each task in the PTG, this allocation will stand for the c potential allocations on the different clusters of the multi-cluster platform. This simplification allows us to apply a classical two-step algorithm based on these virtual allocations. But two questions first have to be answered: (i) "how to define this virtual homogeneous cluster?"; and (ii) "how to translate a virtual allocation into an actual allocation on a given cluster?"

We define the virtual homogeneous cluster so that it has a total computing power equivalent to that of the original heterogeneous platform. Moreover each processor of the virtual cluster computes as fast as a processor of the slowest cluster in the actual platform. Consequently, the virtual cluster comprises more resources than the original multi-cluster. We denote by P^v the total number of processors in the virtual homogeneous cluster, that is defined as:

$$P^v = \sum_{i=1}^{c} \frac{p^i}{r^i}, \tag{1.16}$$

where r^i is the processing speed ratio of cluster c^i (made of p^i processors) to the speed of the slowest cluster. In the following description of the HCPA algorithm, we denote by p_i^v the allocation of task v_i on the virtual homogeneous cluster, and by $T^v(v_i, p_i^v)$ the corresponding virtual execution time.

To determine this allocation for each task in the PTG, we first have to redefine the critical path length T_{CP} and the average area T_A used by the allocation procedure of the CPA algorithm. The first version of the HCPA algorithm that was proposed in [NS06] derived the average area directly from the definition given by Equation 1.8. In [NSC07] an improved version of HCPA was proposed that derived instead from T_A^{geo}, as defined by Equation 1.9. The critical path length is now defined as:

$$T_{CP}^v = \max_{v_i \in \mathcal{V}} bl^v\left(v_i\right), \tag{1.17}$$

where $bl^v(v_i)$ is the bottom level of task v_i using the virtual allocations for the other tasks, while the average area is now:

$$T_A^v = \frac{1}{\min\left(P^v, \sqrt{V \times P^v}\right)} \sum_{i=1}^{V} \left(T^v\left(v_i, p_i^v\right) \times p_i^v\right). \tag{1.18}$$

Thanks to these new definitions, it is possible to apply the allocation procedure presented in Algorithm 1 with almost no modification. The only necessary change is related to the effective translation from the virtual allocation into the allocation on an actual cluster. This translation aims at preserving

the execution time of the task, that is:

$$T^j(v_i, p_i^j) = T^v(v_i, p_i).$$

To reach such an equality on cluster c^j, we rely on the Amdahl's law to define the following translation function.

$$f(v_i, p_i^v, c^j) = \frac{(1-\alpha)T^j(v_i, 1)}{T^v(v_i, p_i^v) - \alpha T^j(v_i, 1)}. \tag{1.19}$$

This function of the task, the virtual allocation, and the target cluster gives us a rational approximation of p_i^j, the actual allocation of task v_i on cluster c^j. As all allocations have to be in the integer space and as an allocation cannot exceed the number of processors available in a cluster, the proper definition of p_i^j is then:

$$p_i^j = \min\left(p^j, \left\lceil f(v_i, p_i^v, c^j)\right\rceil\right).$$

The extra condition added by the HCPA algorithm to the original allocation procedure is to check if there still exists at least one cluster onto which the translated allocation can be increased. In other words there must exist a task v_i belonging to the critical path and a cluster c^j such that $p_i^j < p^j$. If no such task exists, it means that the critical path is *saturated*, *i.e.,* its length cannot be reduced any further by giving an extra processor to any task. In this case, the allocation procedure must stop regardless of the expected compromise between T_{CP}^v and T_A^v. Algorithm 5 summarizes this modified allocation procedure.

Algorithm 5 Allocation Procedure of HCPA

1: **for all** $v_i \in \mathcal{V}$ **do**
2: $\quad p_i^v \leftarrow 1$
3: **end for**
4: **while** $T_{CP}^v > T_A^v$ and not-saturated critical path **do**
5: $\quad v_i \leftarrow$ task $\in CP \mid (\exists\, c^j \mid p_i^j < p^j)$ and $\left(\frac{T^v(v_i, p_i^v))}{p_i^v} - \frac{T^v(v_i, p_i^v+1)}{p_i^v+1}\right)$ is maximum
6: $\quad p_i^v \leftarrow p_i^v + 1$
7: $\quad$ **Update** T_A^v **and** T_{CP}^v
8: **end while**

As the allocation step, the mapping step of the HCPA algorithm has been adapted to translate a single allocation on the virtual homogeneous cluster into c candidate allocations for each task. This modified procedure follows the same basic principle that orders tasks by decreasing bottom level values. Once a task has been selected for mapping, its completion time on *each* cluster is estimated using the corresponding translated allocation. The cluster that achieves this earliest completion time is selected and the procedure proceeds with the next task. The allocation packing optimization, illustrated by Figure 1.9, is also applied during the mapping step of the improved version of the HCPA algorithm presented in [NSC07].

In [DNSC09], we also proposed, with P.-F. Dutot, T. N'Takpé and H. Casanova, a guaranteed algorithm for scheduling a single PTG on an "almost" homogeneous multi-cluster, *i.e.,* a collection of homogeneous clusters. This work draws inspiration from the work in [63] that targets a homogeneous cluster of Symmetric Multi-Processor (SMP) nodes. There are thus two key differences:

1. In [63] all nodes have the same number of processors (because they are homogeneous SMP nodes), while clusters can have different numbers of nodes (there are small clusters and large clusters).

2. In [63] moldable tasks are allowed to run over multiple nodes, while we restrict a task to run within a single cluster.

We call this new algorithm Multi-Cluster Guaranteed Allocation Scheduling (MCGAS). MCGAS, like the algorithm in [63] for scheduling PTGs on clusters of SMPs, relies heavily on the works in [138] and [107]. The first step of the algorithm is the allocation phase from [138], which rounds off the solution of a rational linear program corresponding to a time-cost trade-off problem. Let us denote by C^*_{max} and W^* the values of C_{max} and W that correspond to the optimal trade-off. The allocations produced in this first phase ensure that C_{max} and W are at most $1/(1-\mu)$ and $1/\mu$ as large as C^*_{max} and W^*, respectively, where μ is a parameter between 0 and 1. Note that this approach has been repeatedly presented in the theoretical literature over the last decade. One contribution was that, to the best of our knowledge, we presented the first practical implementation of the time-cost trade-off linear program. Therefore, for the first time, we were able to evaluate its efficacy in practice.

Once the initial processor allocation is determined, the schedule is produced via a modified list scheduling algorithm as in [107]. We perform a bounding of task allocations so that these allocations are at most b, where the value of b is to be defined. A large value favors data parallelism, while a small value favors task parallelism. The goal for setting b to a value lower than, say, the number of processors of the largest cluster is to avoid ill-advised stalling of the critical path. Indeed, the allocation phase of MCGAS, albeit leading to a performance guarantee, does not attempt to balance data and task parallelisms, and may thus lengthen the critical path in ways that could be avoided. Once all allocations have been bounded, tasks can then be mapped to processors using a list-scheduling algorithm. Algorithm 6 summarizes the steps of the MCGAS algorithm.

Algorithm 6 Main steps of the MCGAS algorithm

1: *Input:* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, $T(v, p)$ for each $v \in \mathcal{V}$ and c, μ, b
2: *Output:* An allocation for each task and a schedule
3: *Steps:*
4: 1) Construct an instance of the continuous time-cost trade-off problem, based on $\mathcal{G}$ and $T()$
5: 2) Solve the continuous time-cost trade-off problem to obtain the optimal continuous trade-off
6: 3) Round off the solution of the continuous problem, so that C_{max} is a factor $1/(1-\mu)$ from optimal and W is a factor $1/\mu$ from optimal, while computing corresponding integer task allocations
7: 4) Bound all allocations to be at most b
8: 5) Use any list scheduling algorithm to schedule the tasks on the platform

The efficacy of the scheduling algorithm and the performance guarantee are both contingent upon a good choice for the values of the μ and b parameters. To compute MCGAS's performance guarantee as a function of μ and b, we consider a multi-cluster homogeneous platform. Let c be the number of clusters in the platform, and p^i, $i = 1, \ldots, c$, the numbers of processors in these clusters. We refer to p^i as the *size* of cluster c^i. Without loss of generality we assume that the clusters are sorted by non-increasing sizes ($p^1 \geq p^2 \geq \cdots \geq p^c$), so that p^1 is the size of the largest cluster. Given the clusters sizes, we define

$$S = \sum_{i=1}^{c} \max\left(0, p^i - b + 1\right), \tag{1.20}$$

which is a quantity that we will use in what follows. Intuitively, S represents the minimum number of allocated processors so that no cluster has b idle processors.

For a given PTG we can categorize each time step in the resulting MCGAS schedule into three kinds of time intervals according to the following rules:

- T_1: intervals where at most $b-1$ processors are used;

- T_2: intervals where at least b and at most $S-1$ processors are used; and
- T_3: intervals where at least S processors are used.

The definitions of these intervals are adapted from those used in [107], and use the newly defined constant S. For the sake of simplicity t_i denotes the sum of the lengths of all intervals of type T_i.

The goal of this classification is to bound the contribution of each time step to C_{max} and to W. We know from [138] that after the rounding phase C_{max} is at most $1/(1-\mu)$ larger than C^*_{max}, and that W is at most $1/\mu$ larger than W^*. After the allocation bounding step, during which tasks that were allocated more than b processors are reduced to exactly b processors, W does not increase. Indeed, a smaller processor allocation for a task does not increase the task's work because we assume that tasks have parallel efficiencies lower than 1. For the same reason, the reduction to b processors causes C_{max} to increase by at most a factor p^1/b.

During intervals of type T_1, no task has seen its allocation reduced to exactly b processors (since fewer than b processors are used). Therefore for each interval T_1 there is a task that is on the critical path and whose allocation has not been reduced during the allocation bounding step. During intervals of type T_2, there is at least one cluster where b processors are idle, which means that no task is ready to be scheduled, which means again that there is a task in each of these intervals that belongs to the critical path. However, in this case the task may have seen its allocation reduced from p^1 processors to b processors. With this reduced allocation the task's contribution to C_{max} is at least b/p^1. For intervals of type T_3, there is no cluster with at least b idle processors, which means that there might be an unscheduled ready task that is on the critical path.

Consequently, on the one hand C_{max} is not smaller than $t_1 + b/p^1 \times t_2$, and on the other hand W is larger than $t_1 + b \times t_2 + S \times t_3$. Since the schedule length, C_{max}, is the sum $t_1+t_2+t_3$, we can now write a complete set of inequalities leading to the performance guarantee for the MCGAS algorithm, using C^*_{max} to denote the optimal schedule length:

$$\begin{array}{rl} & C_{max} = t_1 + \quad t_2 \quad + \; t_3 \;, \\ \frac{C^*_{max}}{1-\mu} \geq & C_{max} \geq t_1 + \frac{b}{p^1}\, t_2 \;, \\ \frac{C^*_{max}}{\mu} \sum_{i=1}^{c} p^i \geq & \quad W \quad \geq t_1 + \; b\, t_2 \; + S\, t_3 \;. \end{array}$$

Let us define $P = \sum_{i=1}^{c} p^i$, and introduce a new parameter $\alpha \in [0,1]$. This parameter does not have any concrete interpretation, but is used as an algebraic device to combine the two above inequalities. More specifically, multiplying the first inequality by α, the second by $1-\alpha$, and adding them together, we obtain

$$C^*_{max} \geq \left(\alpha(1-\mu) + \frac{(1-\alpha)\mu}{P}\right) t_1 + \; b\left(\frac{\alpha(1-\mu)}{p^1} + \frac{(1-\alpha)\mu}{P}\right) t_2 + \frac{(1-\alpha)\mu S}{P} t_3 \;. \tag{1.21}$$

Let us now define β as the minimum of the three following quantities:

$$\begin{aligned} \beta_1(\alpha,\mu,b) &= \alpha(1-\mu) + \frac{(1-\alpha)\mu}{P}, \\ \beta_2(\alpha,\mu,b) &= b\left(\frac{\alpha(1-\mu)}{p^1} + \frac{(1-\alpha)\mu}{P}\right), \\ \beta_3(\alpha,\mu,b) &= \frac{(1-\alpha)\mu S}{P}. \end{aligned}$$

Using the fact that $C_{max} = t_1 + t_2 + t_3$, we obtain

$$C^*_{max} \quad \geq \quad \beta C_{max} \;.$$

The guaranteed performance ratio is thus equal to $1/\beta$, which is minimized when β is maximized. Finding a closed form for the α, b, and μ values that maximize β given the $(p^1, \ldots, p^c)$ values seems very challenging. But it turns out that it is possible to determine a good approximation of the solution.

The three quantities β_1, β_2, and β_3 are of the form $AX + BY$ with A equal to $\alpha(1-\mu)$, B equal to $(1-\alpha)\mu$, and with both X and Y greater than or equal to zero. Then the function $f(\alpha, \mu) = \alpha(1-\mu)X + (1-\alpha)\mu Y$ reaches its maximum when $\alpha = 1-\mu$ and we can remove the α parameter from the equations:

$$\begin{aligned}\beta &= \min(\beta_1(\mu, b), \beta_2(\mu, b), \beta_3(\mu, b)) \\ &= \min\left((1-\mu)^2 + \frac{\mu^2}{P}, b\left(\frac{(1-\mu)^2}{p^1} + \frac{\mu^2}{P}\right), \frac{\mu^2 S}{P}\right)\end{aligned}$$

Let us compute the values of b and μ that maximize this quantity (recall that P and p^1 are characteristics of the platform and are fixed, while S is a piecewise linear function of b). Note that $\beta_1(\mu, b)$ depends only on μ. Also, β_1 decreases from 1 to $1/P$ when μ increases from 0 to 1, and β_3 increases from 0 to S/P when μ increases from 0 to 1. Therefore, the largest minimum of β_1 and β_3 is achieved when $\beta_1(\mu, b) = \beta_3(\mu, b) = \beta_{1,3}$, that is when $(1-\mu)^2 = \mu^2(S-1)/P$. β is then maximized when $\beta_2(\mu, b)$ is equal to $\beta_{1,3}$, that is when $b(S - 1 + p^1) = Sp^1$. We obtain the best value for b as an integer approximation of the non-integer solution of this simple equation, and can then compute the best value of μ. Recall that by "best values" we mean the values that lead to the tightest performance guarantee.

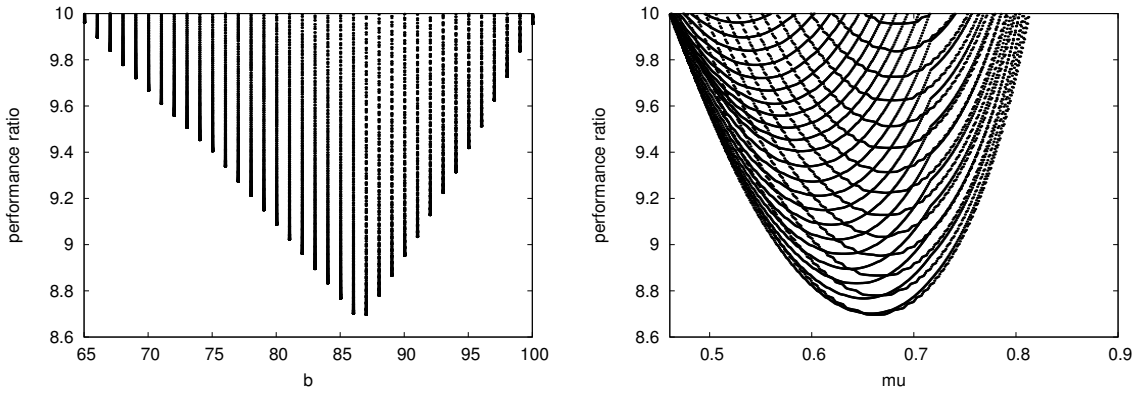


Figure 1.14: 2D projections of all $(\mu, b, 1/\beta)$ triplets with performance ratio $1/\beta$ lower than 10.

For given values of $(p^1, \ldots, p^c)$, we can easily plot the different guaranteed performance ratios, $1/\beta$, each for given values of b and μ. Figure 1.14 shows, for a particular platform configuration detailed in [DNSC09], the two projections of all triplets $(b, \mu, 1/\beta)$ along the μ and the b axes, for performance ratios at most 10. The left graph shows the projection along the b axis. The right graph shows the projection along the μ axis. In both graphs we see that the performance ratio increases more sharply as b or μ become larger than their optimal values, and more moderately when they become smaller than their optimal values. Figure 1.15 shows the domain of the μ and b values in which one is guaranteed that the performance ratio is lower than 10.

Such graphs provide good guidance for tuning the values of μ and b. Indeed, the values of μ and b that lead to the tightest performance guarantee may not lead to the best average application performance in practice. Therefore, one may wish to tune them to ensure a reasonable performance guarantee while leading to good average observed performance over a range of relevant application configurations.

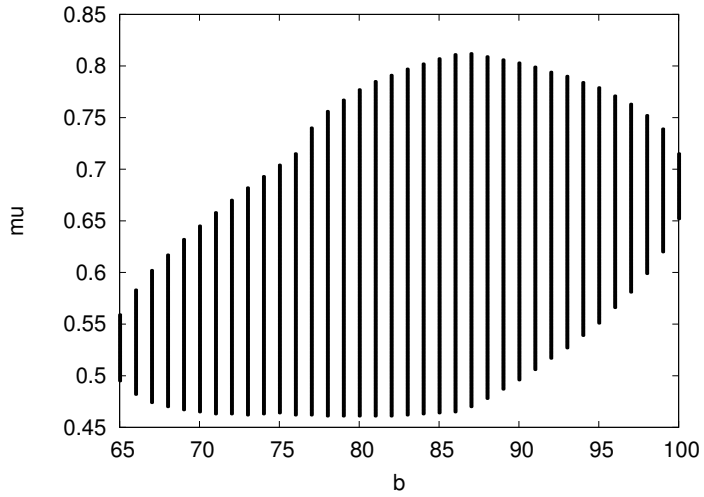


Figure 1.15: Domain of b and μ values for which MCGAS's performance ratio is lower than 10, for a particular platform configuration.

In the evaluation conducted in [DNSC09], we have determined the values of the μ and b parameters that lead to the tightest performance guarantee both analytically and in practice. Our key finding was that MCGAS outperforms the nonguaranteed HCPA algorithm, on average, over a large range of application configurations. However, this guarantee comes at a the price of the larger scheduling time, up to 1,600 times longer for PTGs with 30 tasks, due to solving a rational linear program. The application of this algorithm is then better suited for applications with long execution times to justify such a scheduling time.

A second and complementary approach to adding the support of heterogeneity in heuristics designed for homogeneous platforms is to make heuristics designed to schedule DAGs on heterogeneous platforms amenable to PTG scheduling. To some extent, it "simply" consists in a shift from a serial execution to a parallel one. Indeed, classical DAG scheduling heuristics aim at finding which *serial* (and thus homogeneous) computing resource is the best suited for the execution of each *serial* task in the DAG. Similarly, scheduling a PTG on a multi-cluster consists in determining which *set of* homogeneous computing resources is the best suited for the execution of each *parallel* task in the PTG. This kind of adaptation was at the origin of the Mixed-parallel HEFT (MHEFT) algorithm that we proposed, with H. Casanova and F. Desprez, in [CDS04]. As its name says, MHEFT uses the Heterogeneous Earliest Finish Time (HEFT) [147] algorithm as a starting point. HEFT is one of the very popular algorithms for scheduling a DAG onto a heterogeneous set of processors. As many other list scheduling algorithms, HEFT is based on two components: a *priority function*, which is used to order all nodes in the task graph at compile time; and an *objective function*, which must be optimized. The priority function of HEFT uses a notion, called *upward rank*, similar to the bottom level of a task. The difference between the upward rank and the bottom level of a task is that the length of the longest path to the exit node is based on average values. It is defined as the sum of the average computation time of each task and the average communication time of each communication edge along the path. These averages are computed over all processors and network links. Tasks are scheduled by HEFT in order of decreasing upward ranks. The objective function that is optimized by HEFT is the finish time of a task. The algorithm will then, for each task, look for the processor that minimizes the completion time of this task, accounting for time spent in communication. Indeed a task can be executed on a processor only once this processor is

available and all the input data of the task have been transferred to this location.

The proposed MHEFT algorithm is a very straightforward extension of HEFT to the case of moldable tasks and a heterogeneous collection of homogeneous clusters. The priority function, and its underlying metric, are kept unmodified. The main difference lies in the objective function. The earliest finish time can now be achieved by any subset of any cluster in the platform. To determine the best set of resources, MHEFT exhaustively computes an estimation of the completion time of the currently scheduled task for each possible subset size in each cluster. Availability dates of the processors, *i.e.*, the moment at which a processor can process a new task, and data transfer costs are taken into account in this process.

MHEFT was the first proposed algorithm to schedule PTGs on heterogeneous multi-clusters. This was also my first contribution to this field. To some extent, it explains the simplicity of this algorithm and its tight links with a popular algorithm such as HEFT. However, many of the limitations of MHEFT have been highlighted and addressed since its publication in 2004.

A first problem with MHEFT is that it tends to produce very large processor allocations for most of the data-parallel tasks. Indeed, allocations are chosen to minimize the completion time of each task, and, in turn, the overall makespan of the algorithm. By contrast, the HCPA algorithm attempts to achieve a trade-off between makespan reduction and work augmentation. Then HCPA often lead to smaller allocations in the same circumstances. To remedy this problem with MHEFT we have proposed in [NSC07] three simple methods to bound a task's processor allocation:

MHEFT-IMP – A task's allocation is increased by one processor only if that task's execution time is improved by more than some given threshold percentage.

MHEFT-EFF – A task's allocation is increased by one processor only if that task's parallel efficiency is improved by more than some given threshold percentage.

MHEFT-MAX – No task allocation on a cluster can be larger than some fraction of the total number of processors in that cluster.

Each of these variants trades some makespan reduction to increase the potential to execute some other tasks in parallel. Figure 1.16 illustrates their relative impacts, when associated to common sense thresholds, on the execution time and the determined allocation of a single moldable task. The considered task has 1 billion of floating point operations to compute with 10 percent of them that cannot be executed in parallel, *i.e.*, the α parameter of Amdahl's law is equal to 0.1. The red line in Figure 1.16 shows the evolution of the execution time of this task with regard to the number of allocated processors. The target cluster comprises 30 processors.

We see that the performance model limits the reduction of the execution time past a certain number of processors. In this particular example, the execution time on one processor is divided by less than a factor 8 when using the whole cluster. More than half of this gain is already obtained with only 6 processors, while using only half of the cluster leads to more than 80% of the maximum gain. Drastically reducing the allocation of this task will then have a moderate impact on its execution time, while other tasks could find enough resources to be executed concurrently. The proposed variants of MHEFT reduce the allocation of the task in this illustrative example by a factor of two at least. The MHEFT-MAX-50 variant does not consider allocations that uses more than half of the available resources, leading to an allocation of 15 processors. Then the MHEFT-EFF-50 variant guarantees that the work needed to execute the task is not more than twice the sequential work. In this particular case, that limit is reached with 12 processors. Finally the MHEFT-IMP-5 stops to increase the allocation is the execution time is not 5% better than with the current allocation. Here it means that the potential gain on execution time becomes negligible as soon as the task is allocated on 10 processors.

These techniques may lead to clearly suboptimal makespan in some cases. For instance, when the PTG is a simple "chain", the best makespan is achieved by allocating all processors to each task (which is the schedule computed by the original MHEFT algorithm). However, the objective here is to not aim solely for the best makespan, but to take into account the efficiency of the schedule too.

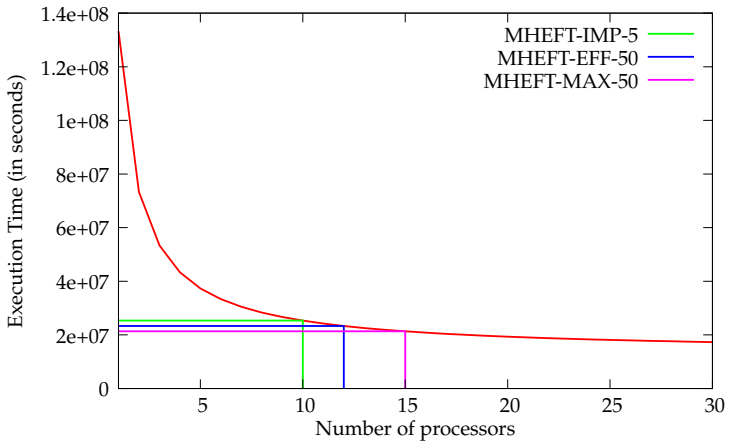


Figure 1.16: Illustration of the allocation reduction strategies added to MHEFT for a 1 Gflops task to be scheduled on a cluster of 30 processors.

Another reason why MHEFT determines large allocations for the tasks composing a PTG is that a task's processor allocation is chosen "blindly" so that the task's completion time is minimized. In other words, the scheduling totally ignores the other ready tasks when taking a decision for the current task. Then it has no reason to reduce the allocation in order to let enough resources available for the next scheduling round. In [Sut07], I proposed a scheduling algorithm that takes the other ready tasks into consideration, especially those having the same (or a close) bottom-level priority. Such tasks are indeed as critical as the first task. The selfish, and potentially large, allocation of the first task may delay the other tasks and could have a negative impact on the overall execution time of the application.

Depending on the structure of the PTG, the number of tasks with exactly the same bottom level value varies. For instance, some parallel applications, *e.g.*, Strassen's matrix multiplication or one dimensional FFT algorithms, can easily be decomposed into regular precedence levels and tasks in a given level have the same bottom level priority. When the decomposition into levels is more complex, *e.g.*, with execution path of different lengths or costs, the maximal number of tasks with *exactly* the same bottom-level value is likely to be one. The set of critical tasks then has to include tasks with a bottom-level value greater or equal to that of the most critical task minus a certain Δ. A natural upper bound for this Δ is the average execution time (as estimated in the computation of the bottom level) of the first task added into the set of Δ-Critical Tasks. Otherwise we may include tasks that depend on that most critical task and thus cannot be executed concurrently. Experiments have shown that shorter schedules were produced by setting Δ to the half of the average execution time of the first task added. Based on this idea I proposed the Δ-Critical Tasks Scheduling (Δ-CTS) heuristic described by Algorithm 7.

This algorithm first computes the bottom level of each task in the PTG. Then it sorts the tasks in a scheduling list by decreasing bottom level values. Then it builds a set of Δ-Critical Tasks of size d. For each task in this set, the objective is to minimize its Earliest Finish Time (EFT) while taking the other critical tasks into account. We denote as $EFT(v_i, c^j, p_i^j)$ the EFT of task v_i if were to be executed on p_i^j processors of cluster c^j. The maximal number of processors that can be allocated to a task v_i belonging to the set of Δ-Critical Tasks on a cluster c^j, $j = 1, \cdots, c$, is then limited to $a^j = p^j/(\lfloor d/c \rfloor + b^j)$, where b^j correspond to the repartition of the $d \bmod c$ remaining tasks among the c clusters, with regard to the relative cumulative power of the clusters. For instance, if $d = 8$ and $c = 3$, there are two tasks to dispatch among three clusters. If the heterogeneity of the platform is low, the first two clusters will respectively determine a^1 and a^2 considering one more task than the third cluster *i.e.*, $b^1 = b^2 = 1$, and $b^3 = 0$. If the heterogeneity is higher and the first cluster is twice as fast as the slowest cluster of the platform,

Algorithm 7 Δ-Critical Tasks Scheduling

```
for all v_i ∈ V do
    Compute bl(v_i)
end for
Sort tasks by decreasing bl(v_i) values
while there are unscheduled tasks in the list do
    Build a set of Δ-Critical Tasks from the first tasks of the list
    for all task v_i ∈ this set of Δ-Critical Tasks do
        for all Cluster c^j do
            Compute a^j the maximal allowed number of processors to allocate to v_i on c^j
            Find 1 ≤ p_i^j ≤ a^j such as EFT(v_i, c^j, p_i^j) is minimal
        end for
        Assign v_i on the p_i^j processors of cluster ^j that minimize EFT(v_i, c^j, p_i^j)
    end for
end while
```

the two remaining tasks will only influence the computation of the maximal number of processors that can be allocated to a task on the first cluster, *i.e.*, $b^1 = 2$ and $b^2 = b^2 = 0$. The rationale is to favor the scheduling of more concurrent tasks on faster clusters.

The experimental evaluation conducted in [Sut07] showed that the Δ-CTS produces better or equal schedule lengths when compared to the MHEFT algorithm and its variants in 80% of the broad range of investigated scenarios. Moreover, when not the best, the Δ-CTS algorithm is always close to its contenders (less than 6% on average.) However, while the gain is clear for very regular PTGs, there is still room for improvement with more irregular applications. Indeed even with considering tasks with a bottom-level value as close as Δ, the size of concurrent tasks is often too small to see a gain with regard to the original MHEFT algorithm. A solution could be to use a more flexible bound for very irregular applications to increase the size of the set of critical tasks.

In collaboration with S. Hunold and T. Rauber, we proposed another variation of the MHEFT algorithm to schedule dynamically generated DAGs onto a heterogeneous collection of clusters [HRS08b]. Indeed, the complete structure of some PTGs is not known before their execution. For instance, in a recursive algorithm, a task may spawn new sub-tasks if certain criteria are satisfied. As for static PTGs, assigning too many processors to one task may prevent other tasks from being executed. A first scheduling algorithm leading to satisfying makespans for dynamic PTGs, namely the Reuse Processors (ReP) algorithm, was proposed in [92]. The ReP algorithm assumed that the data distribution of tasks is unknown to the scheduler and therefore the data transfer costs cannot be estimated. Since the ReP algorithm does not account for these communication costs and attempts to use all idle processors in a heterogeneous system, long running tasks could get scheduled to a small set of processors and delay the execution of subsequent tasks. The Dynamic MHEFT (DMHEFT) algorithm extends the ReP by considering the time to perform data transfer between subsequent tasks and proposing a postponing strategy to avoid assigning a small number of processors to computation intensive tasks.

First, let recall the principle of the ReP algorithm. It is executed each time a task finishes its execution and new tasks become ready. It manages a sorted queue of ready tasks according to their amount of computation and precedence level. Considering the precedence level prevents in most cases starvation for tasks with small computational costs. Indeed, successors of such tasks will take a long time becoming, thus hindering the exploitation of task parallelism. Sorting tasks by decreasing computational costs is a common practice in list scheduling. "Big" tasks have to be scheduled first and "small" tasks are used to "fill the holes". The ReP algorithm determines the allocation of each task in three steps:

1. Determining the cluster that is the most suitable for executing the current task. This is the cluster

with the most available computational power, *i.e.,* that minimizes the completion time of the task;

2. Determining the number of processors which are assigned to this task. This number takes into account the computational power of the cluster and the computational costs of the other unscheduled tasks. As a result, a task is assigned to a fair share of the available processors while leaving enough room for the other tasks;

3. Selecting this number of processors among the available ones. There, the ReP algorithm favors processors which were assigned to a parent task to reduce the communication overhead for data transfers, hence its name.

The investigation of the cases in which the schedules produced by the ReP algorithm could be improved highlighted the fact that neglecting the data transfer costs when selecting a cluster can lead to a big communication overhead, especially when data has to be moved across cluster borders. It was also observed that the strategy of using all processors at all time has a big impact on the overall makespan. As we consider moldable tasks, the processors which are assigned to a task are determined before starting the task and cannot be changed during the execution. An illustration of a problematic setup is depicted in Figure 1.17(a). At time t_r, task v_i becomes ready. In this case, the ReP algorithm attempts to schedule the biggest task, *i.e.,* the one with the most operations to perform, on the available processors. As one can see, as a consequence of this decision V_i will be executed on this cluster for a long time. If the PTG does not exhibit a high degree of task parallelism and many tasks are dependent on the result of v_i , the schedule will contain large idle times. A possible solution might be to postpone v_i until more processors become available as shown in Figure 1.17(b).

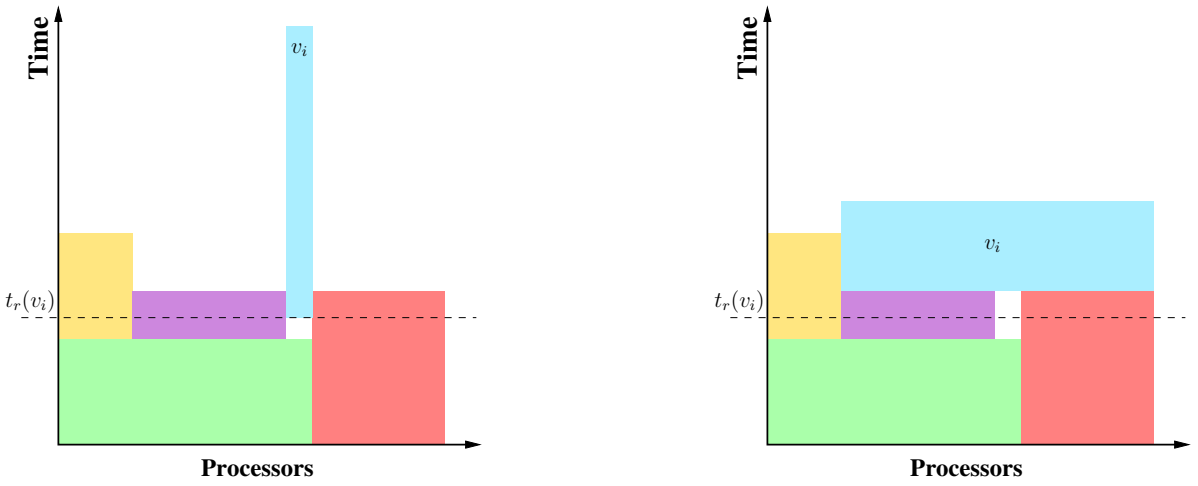


Figure 1.17: Common problem for dynamic schedulers: placing and executing a task v_i at time $t_r(v_i)$ may lead to an unfavorable schedule if the number of available processors is small.

The general principles of the DMHEFT and ReP algorithms are similar, as both allocate processors to task following the same three steps. However, two improvements have been made in the DMHEFT algorithm to overcome the highlighted drawbacks of the ReP algorithm.

The first improvement is to consider the communication costs associated to data transfers. To achieve this, we have to consider all clusters having at least one idle processor which are potential candidates for the task allocation. For each cluster, the DMHEFT algorithm still estimates the execution

time of the considered task but also the communication time to move data between this possible task allocation and parent task allocations. Then, the algorithm selects the cluster leading to the smallest sum of the task execution time and the longest data transfer.

The second optimization is to add a postponing strategy to the allocation process. As in the ReP algorithm, the task with the largest amount of computation (or the smallest precedence level) is selected and assigned to the cluster that minimizes its finish time (accounting for data transfer costs). As shown in Figure 1.17(a), executing a computationally intensive task on a small set of processors may dramatically increase the overall makespan of the schedule. This issue can be solved by delaying the beginning of the execution of the task to wait for more available processors, as shown in Figure 1.17(b). The proposed postponing strategy relies on three parameters illustrated by Figure 1.18.

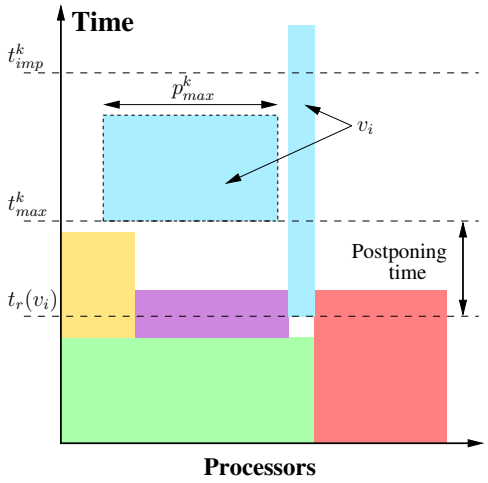


Figure 1.18: Postponing strategy of the DMHEFT algorithm for task v_i that becomes ready at time $t_r(v_i)$ on cluster c^k.

The first two parameters are times defining an interval for the potential execution of the postponed task. The boundaries of this interval correspond respectively to the maximum amount of time a task can be postponed and the time by which a postponed task must be completed to justify its postponing. On cluster c^k, these two times are directly related to the sum of $t_r(v_i)$, *i.e.*, the time at which v_i becomes ready, and the estimation of the EFT of the task on that cluster. This estimation is computed using the number of processors currently available on cluster c^k, denoted by p^k_{avail}. In the previously used notation, that estimation is then given by $EFT(v_i, c^k, p^k_{avail})$.

The computations of the boundaries thus only differ by the use of one of the two tunable factors, f_{max} and f_{imp}, that define the fraction of the EFT used to determine each boundary. More formally, we define t^k_{max} and t^k_{imp} the interval boundaries on cluster c^k as:

$$t^k_{max} = t_r(v_i) + f_{max} \times EFT(v_i, c^k, p^k_{avail}), \text{with } f_{max} \in [0; 1[\text{ and} \quad (1.22)$$

$$t^k_{imp} = t_r(v_i) + f_{imp} \times EFT(v_i, c^k, p^k_{avail}), \text{with } f_{imp} \in]0; 1]. \quad (1.23)$$

The third parameter used by our postponing strategy is a number of processors. More precisely, p^k_{max} represents the number of processors of cluster c^k that can be allocated to task v_i at time t^k_{max}. It is a tunable fraction of the number of processors available on this cluster at that time. The rationale of the existence of this third parameter is that v_i would unlikely be the only ready task at time t^k_{max}, even though the scheduling algorithm has no such knowledge. Allowing v_i to be allocated only a fraction f_p of the available processors ensure that other ready tasks can be scheduled too.

In the experimental evaluation of the DMHEFT algorithm conducted in [HRS08b], the values of the three tunable parameters f_{max}, f_{imp}, and f_p have been respectively set to 0.2, 0.8, and 0.4. These values are of common sense and express that a task cannot be delayed more than 20% of its estimated execution time, that its completion time should be reduced by at least 20% to justify the introduced delay, and that no more than 40% of the available processors can be used to execute this task.

In this section, I have detailed the different heuristics designed to schedule a single PTG. They are summarized by Table 1.4 that distinguish heuristics that target a single cluster (in green) from those who consider a multi-cluster. Among the latter, this table makes a distinction related to the followed approach. Heuristics in red were designed by adapting classical DAG scheduling algorithms to the case of PTG, while those in blue have introduced a support of heterogeneity in two-step algorithms designed for homogeneous platforms.

		Platform	
		Single Cluster	*Multi-cluster*
Type of tasks	*Serial*	list scheduling algorithms	⇒ list scheduling algorithms
	Moldable	⇓ RATS [HRS08a] biCPA [DS10] two-step algorithms	⇓ MHEFT[CDS04, NSC07] DMHEFT [HRS08b], Δ-CTS [Sut07] ⇒ HCPA [NS06, NSC07], MCGAS [DNSC09]

Table 1.4: Summary of the contributions to the problem of scheduling a single Parallel Task Graph on a single cluster or a multi-cluster.

Many of these heuristics have been compared together, depending on when they have been designed. While no clear winner has emerged, it was often possible to identify which heuristic was the most suited for a specific scheduling scenario, *i.e.*, a combination of an application class and characteristics of the target platform.

1.5 Multiple PTG Scheduling

In this section, we consider the problem of the simultaneous scheduling of a batch of N PTGs on a single homogeneous cluster. As stated in Section 1.2.2, this scheduling problem is intrinsically a multi-criteria optimization problem. Indeed, a scheduling algorithm still aims at minimizing the completion time of each individual application, but also has now to ensure that this objective is achieved in a fair way. Moreover, the overall completion time of the batch of PTGs has to be minimized too. Here, we will focus on platforms that comprise a single cluster even though some of our early work on this topic has been designed for multi-cluster platforms. To evaluate the quality of the schedules produced by the different heuristics presented hereafter, we relied in the associated publications on the *overall makespan*, *scaled sum completion time*, *average stretch*, and *maximum stretch* metrics as defined in Section 1.2.3.

The concurrent scheduling of multiple PTGs has been studied by several authors. In [157] four algorithms are proposed that combine multiple task graphs into a single composite task graph, which is then scheduled using a standard task graph scheduling algorithm, and two algorithms that perform task-by-task scheduling over all tasks in all task graphs in a view to optimize fairness. In [43] a two-level distributed scheduling algorithm for multiple task graphs is proposed. The first level is a WAN-wide distributed scheduler responsible for dispatching the different task graphs (viewed at this level as a single task) to several second level schedulers that are LAN-wide and centralized. The focus of that work is more on environment-related issues (such as machine failure rates and queue waiting times) than on

scheduling concerns (such as promoting fairness among applications). In [95] the authors propose a hierarchical competitive scheduling heuristic for multiple tasks graphs. A restrictive assumption, which we do not make in this section, is that each application is responsible for its own scheduling and has no direct knowledge of the other applications. All the above focus on task graphs of sequential tasks and not on PTGs. Consequently, they do not address the difficult issue, which arises in PTGs, of how many processors should be given to each task. However, some ideas developed in [157] are extended in this section to handle PTGs.

From a theoretical standpoint, the problem of scheduling multiple PTGs can be seen as a special case of the problem of scheduling independent moldable jobs, given that PTGs are inherently moldable. This problem has been studied for homogeneous clusters and a guaranteed algorithm for makespan optimization with a $3/2 + \epsilon$ approximation ratio is given in [65]. Particularly relevant for the content of this section is the work in [64], which builds on the algorithm in [65] and proposes algorithms for solving a bi-criteria scheduling, with the two criteria being makespan and weighted average completion time. We build on the ideas in [64] to develop some algorithms for scheduling PTGs rather than generic moldable jobs.

Our first contribution related to this scheduling problem, made in the context of the Ph.D. of T. N'Takpé, was to focus on the allocation step [NS07]. In this work, we introduced the use of a constraint on the amount of computing resources that can be used to schedule each concurrent application to ensure a fair sharing of the execution environment.

The determination of such a constraint on resource usage can be done either by the provider of each application or by a global scheduling entity. Leaving the responsibility of the constraint determination to users may lead to selfish behaviors, *i.e.*, a loose constraint for each application. Then it may compromise the efficient execution of the concurrent applications. Conversely, a global scheduling entity will be responsible to adapt the resource constraint on each independent schedule depending on the global load of the environment. Moreover, the formal definition of a resource constraint can take different forms, especially for heterogeneous multi-clusters such as those targeted in [NS07]. For instance, a resource constraint can be expressed in terms of a number of processors that cannot be exceeded during the execution of the schedule. This number of processors can be either a maximal value, *e.g.*, the schedule never uses more than X processors at the same time, or an average value, *e.g.*, the schedule cannot allocate more than X processors per task on average. But on heterogeneous platforms, reasoning solely in terms of number of processors is not relevant as scheduling a PTG onto 100 processors computing at 1 GFlop/sec is not the same as onto 100 processors computing at 4 GFlop/sec. The resource constraint can also be expressed in terms of a ratio of the processing power (maximum or average) that can be used to build the schedule over the globally available processing power. This expression of a resource constraint is more adapted to heterogeneous platforms and is equivalent to reasoning in terms of number of processors for homogeneous clusters. Then the allocation procedures proposed hereafter define $\beta \in]0; 1]$ as a fraction of the available resources used by a given application. Formally, $\beta = (used\ power)/(total\ power)$.

The next issue is to determine how to dispatch this usable processing power between the different tasks of the PTG while respecting the constraint. We propose two different strategies, respectively called Self-Constrained Resource Allocation Procedure (SCRAP) and SCRAP-MAX. The driving principle of SCRAP is to determine the allocation of each task while ensuring the respect of the global usage constraint β, starting from an initial allocation of one processor per task. As in the CPA algorithm, one extra processor is allocated to the task in the critical path that benefits the most of it in each iteration. This iterative process stops if a violation of the resource constraint is detected. To do so, we estimate the amount of resources consumed by the allocation of the current iteration by dividing the total work W_{max} associated to this allocation and by the time spent executing the critical path of the PTG. Thus we define β' as the ratio of W_{max}/C_{max} over the total processing power of the platform. This β' can be seen as a dynamic expression of the resource constraint β that evolves along with the allocation. If β' exceeds β, this mean that a violation of the initial resource constraint occurred. Then, the allocation

process must be stopped and the last processor addition canceled. If $\beta' > \beta$ with the initial allocation, SCRAP does not allocate more processors to any task of the PTG. It may also happen that the length of the critical path cannot be further reduced, *i.e.*, all the tasks that belong to the critical path are allocated on the whole cluster, before the resource constraint is violated. Then we add a second stop condition, called *saturated critical path*. This first allocation procedure is described by Algorithm 8.

Algorithm 8 Self-Constrained Resource Allocation Procedure

1: **for all** $v_i \in \mathcal{V}$ **do**
2: $p_i \leftarrow 1$
3: **end for**
4: **while** $\beta' < \beta$ and $\neg$ (saturated critical path) **do**
5: $v_i \leftarrow \text{task} \in CP \mid \left(\frac{T(v_i,p_i)}{p_i} - \frac{T(v_i,p_i+1)}{p_i+1}\right)$ is maximum
6: $p_i \leftarrow p_i + 1$
7: **Update** β'
8: **end while**

The second allocation procedure includes the precedence level of tasks in the respect of the resource constraint. The idea is to restrain the amount of resources allocated at any precedence level to β. The rationale behind this variant is that ready tasks candidate to a concurrent placement often belong to the same precedence level. If all these tasks can be executed concurrently, the allocation has to ensure that the *maximum* processing power required at this level is less than $\beta \times P \times s$. This impacts the selection of the critical task to which allocate one extra processor. To be candidate, a task must show a maximal benefit of the additional processor as in SCRAP but now the sum of the processing power alloted to the tasks, including itself, in its precedence level ($pl\_alloc(v_i) \times s$) has also to be less than $\beta \times P \times s$. Algorithm 9 details this second allocation procedure, called SCRAP-MAX.

Algorithm 9 SCRAP-MAX Allocation Procedure

1: **for all** $v_i \in \mathcal{V}$ **do**
2: $p_i \leftarrow 1$
3: **end for**
4: **while** $\beta' < \beta$ and $\neg$ (saturated critical path) **do**
5: $v_i \leftarrow \text{task} \in CP \mid (pl\_alloc(v_i) \times s) < (\beta \times P \times s)$ and $\left(\frac{T(v_i,p_i)}{p_i} - \frac{T(v_i,p_i+1)}{p_i+1}\right)$ is maximum
6: $p_i \leftarrow p_i + 1$
7: **Update** β'
8: **end while**

Experiments conducted in [NS07] have shown that the resource constraint is respected in 99% by both allocation procedures even for stringent values, such as $\beta = 0.2$. Moreover, when the constraint is violated, the extra amount of used processing power is rather small (3% on average and less than 8% at worst.) Then, we developed this approach based on imposing constraints to determine the allocation of each PTG in [NS09]. Indeed, SCRAP and SCRAP-MAX are allocation procedures, and thus only address issues in the first phase of two-step algorithms. However, some challenges and possible optimizations exist in the second step too, in which the allocated PTGs are actually mapped on resources. Simple list scheduling heuristics are generally used in this second step. Most of them order the list of tasks by decreasing bottom level values. In the case of a single PTG, this order guarantees the respect of the precedence relations and favors the task that is the farthest from the end of the application when several tasks are simultaneously ready.

When scheduling multiple PTGs, the ordering of tasks becomes more complex. The different applications can be aggregated into a single PTG as discussed in [157]. However, such a global ordering

of tasks coming from different applications may have a strong impact on the fairness of the produced schedule. Indeed, the entry tasks of a small PTG will have small bottom level values and be close to the end of the ordered scheduling list. Then, this PTG will experience a high delay as its entry tasks are ready as soon as it is submitted. Several strategies can prevent such a postponing issue. A classical approach, used by batch schedulers, is to use conservative backfilling strategies [69] that try to fit some waiting tasks into schedule holes to improve resource usage without delaying already mapped tasks.

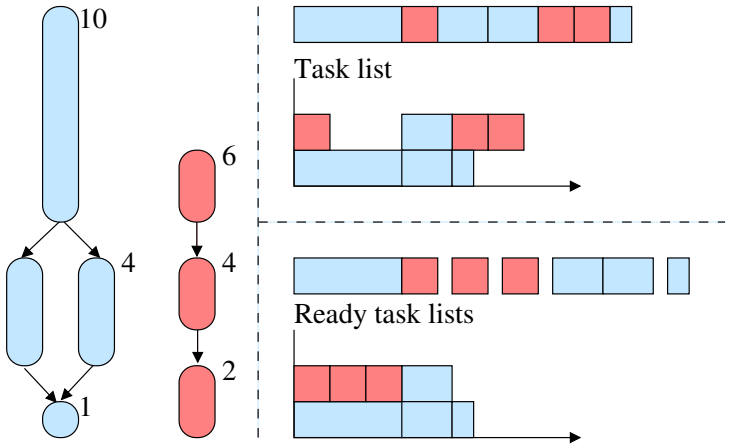


Figure 1.19: Impact of an ordering limited to the list of ready tasks (bottom right) on the schedule length with regard to a global ordering (top right).

In [NS09], we proposed a simple mapping procedure that prevents the postponing issue mentioned in [157]. This approach orders tasks according to their bottom level, but only those that are ready, *i.e.*, whose predecessors have finished their executions. Let consider two PTGs, as shown in the left part of Figure 1.19, with one that can complete during the execution of the first task of the other PTG. Let also assume that each PTG is allowed to use a half of the available power, *i.e.*, one processor each in that simple example, and that there is no backfilling available. The top right part of Figure 1.19 shows the schedule resulting from a global ordering while the bottom right part presents the schedule obtained by ordering only the ready tasks. We can see that with the global ordering some tasks of the smallest PTG are postponed. The resulting schedule is thus unfair, as the smallest application has to wait and also inefficient as it contains idle times. Conversely with the ordering of the ready tasks only, there is no postponing leading to a fairer and more efficient schedule.

The advantage outlined by this simple example is not enough to ensure that postponing will not occur. As the mapping decisions for the first task of the smallest PTG will be taken once all the entry tasks of the other PTGs have been mapped, it could happen that not enough resources are still available thus leading to postponing. Yet, thanks to the respect of the resource constraints, this small PTG should have its share of the resources available when its entry task(s) will be considered for mapping.

Finally, the procedure will select the first task of the list and determine the processor set that achieves the earliest finish time. It may happen that a task is delayed because its allocation is (perhaps only slightly) larger than the number of processors available when the task is actually ready for execution. Then, we include an *allocation packing* mechanism similar to that of the HCPA algorithm. If a task has to be delayed because all the processors it needs are not available, we reduce its allocation if and only if the task can start earlier and finish no later than on its original allocation.

With the implementation of this mapping procedure, we now have a fully functional algorithm that allows us to investigate different strategies to determine the resource constraint assigned to each concurrent PTG. This was not addressed by the SCRAP and SCRAP-MAX procedures that assumed the

constraints were provided by the user. It defines a family of heuristics whose names are prefixed by `CRA_`, standing for Constrained Resource Allocation (CRA).

A first strategy consists in allowing each PTG to use all the available resources. In other words, each application has a selfish behavior and all compete for resources. The β constraint is then fixed to 1 for each application. The rationale is to have an indication on the fairness of schedules built by two-step heuristics designed for the scheduling of a single PTG. We denote this first strategy by `SELFISH`.

By opposition the second strategy relies on a simple assumption. The fairest repartition of the resources, which does not imply the fairest schedule, is to allow each PTG to use an equal share of resources to build its own schedule. For instance if ten PTGs have to be scheduled simultaneously, each of them will be associated to a resource constraint $\beta = 0.1$ and will be thus allowed to use only one tenth of the processing power of the platform. More generally if $\mathcal{A}$ is the set of applications to schedule, each PTG $\in \mathcal{A}$ will have to respect a resource constraint $\beta = 1/|\mathcal{A}|$. We call this strategy `CRA_NDAGS`.

The previous strategy aims at increasing the fairness, but may lead to longer schedules. Indeed some PTGs may not fully exploit the allocated resources while some others are limited by a too constrained allocation. A solution would be to unbalance the resource sharing so that each PTG is constrained proportionally to its contribution relative to a particular metric inherent to the structure of a PTG. A first characteristic could be the length of the critical path. If a PTG has a long critical path, more resources would help to reduce its execution time. Conversely, a PTG with a short critical path may not complete really earlier with more resources. A second characteristic could be the maximal width of each PTG, *i.e.*, the size of the precedence level comprising the most tasks. A large PTG, or at least with one large level, can exploit more task parallelism than a chain-like PTG. Moreover, constraining the allocation of a large PTG may create a bottleneck and cause the postponing of some tasks. Allocations for a large level may also have to be reduced to favor the concurrent execution of other PTGs. Finally, a last option would be to consider the respective amount of work of each application. If one PTG has only a little amount of work to do, less resources than allowed may be needed. These unused resources could benefit to other PTGs with more work that require more than their share. These three considered characteristics respectively define the `CRA_CP`, `CRA_WIDTH`, and `CRA_WORK` heuristics.

Regardless of the chosen characteristic, *i.e.*, length of the critical path, width, or amount of work, unwanted situations with a negative impact on fairness may occur. For instance, if the relative contribution of a PTG is very small, it will be forced to build its schedule on a few resources only. Then its makespan will be much longer than what it would have achieved on a dedicated platform. To ensure that each PTG can use a reasonable share of resources, we mitigate this proportional sharing of resources by including the number of concurrent PTGs in the computation of the constraint. While in [NS09], a tunable parameter taking its value in $[0; 1]$ was used, experiments have shown that a perfect balance between the two components of the equation led to better results. The resource constraint of the i^{th} PTG is then given by

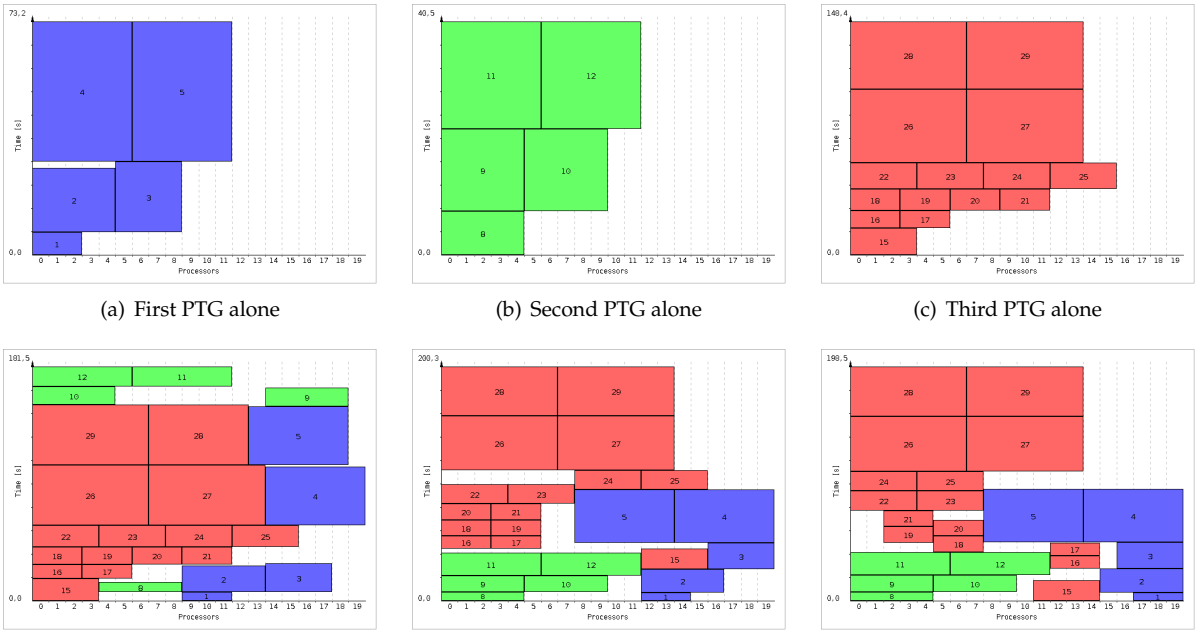
$$\beta_i = \frac{1}{2\,N} + \frac{1}{2\,\sum_{j=1}^{N} \gamma_j}, \tag{1.24}$$

where γ_i represents the relative contribution of the i^{th} PTG with regard to the complete set of N applications. This formula ensures that each PTG can use enough resources to achieve a good makespan while preventing the wasting of resources by PTGs having a small contribution.

In [CDS10] and [CDS10], besides two original heuristics, we proposed, with F. Desprez and H. Casanova, optimized versions of the heuristics I just detailed. The first optimization was to implement a backfilling post-processing phase similar to that used in [150] and inspired by the "conservative backfilling" technique used by batch schedulers [115]. Tasks are considered in the order in which they were scheduled and each task is started as early as possible as long as no other task is delayed. We reuse this technique as a way to compact schedules at the last step of all the algorithms and simply call it "backfilling." We have found it to be beneficial for all our performance metrics for all algorithms.

Then, we have identified a clear weakness of SELFISH, that does not differentiate between "short" and "long" PTGs. Since it schedules all tasks of all PTG together, by decreasing bottom-level values, the completion of a short PTG could be postponed, leading to a high stretch. Recall that $C^*_{max_i}$ represents the makespan achieved by the i^{th} PTG on a dedicated cluster. We also denote by $bl^i(v_j)$ the bottom level of task v_j belonging to the i^{th} PTG. We propose two simple enhancements to SELFISH to ensure that tasks belonging to PTGs with short execution times are given higher priority. SELFISH_ORDER is similar to SELFISH, but instead of sorting tasks by bl^i_j values it sorts them by increasing $C^*_{max_i}$ values, and then by decreasing order of bl^i_j values. This simply amounts to schedule short PTGs before long PTGs. SELFISH_WEIGHT instead sorts the tasks by decreasing $bl^i_j/(C^*_{max_i})^2$. This heuristic attempts to weigh the bottom level of a task by the makespan of its PTG so as to give priority to tasks belonging to short PTGs. The use of the power 2 does not have a theoretical justification, but in our experiments led to much improved results. In all these algorithms all ties are broken randomly.

Figure 1.20 illustrates the impact of these modifications of the way tasks are ordered on the produced schedule for a simple example. Let consider three PTGs with different execution times. Figures 1.20(a), 1.20(b) and 1.20(c) respectively show the schedule obtained for each PTG on a dedicated cluster of 20 processors and drawn with Jedule [HHS10]. The first PTG (in blue) completes in 73.2 seconds and use 12 processors at most. The second PTG (in green) uses also 12 processors to finish after 40.5 seconds. Finally, the third PTG (in red) needs 16 processors to be executed in 148.4 seconds.



(a) First PTG alone (b) Second PTG alone (c) Third PTG alone

(d) SELFISH (e) SELFISH_WEIGHT (f) SELFISH_ORDER

Figure 1.20: Illustration of the impact of task ordering in the mapping step of the SELFISH heuristic on the resulting schedule.

Figure 1.20(d) shows the schedule obtained when scheduling these three PTGs concurrently with SELFISH. The achieved makespan is 181.5 seconds and all the processors in the cluster are used. The highlighted problem here is that the longest PTG (in red) completes exactly as if it were alone on the cluster while the shortest PTG (in green) is the most delayed. This schedule is then particularly unfair, even though the overall makespan is good. The SELFISH_WEIGHT (Figure 1.20(e)) and

SELFISH_ORDER (Figure 1.20(f)) variants that act on the ordering of the scheduling list solve this fairness issue by allowing the tasks of the shortest PTG to be scheduled first. The direct consequence is that the longest PTG is now the most delayed. Its completion time respectively becomes 200.3 seconds and 198.5 seconds. While the overall makespan is increased by around 20 seconds, the impact on fairness is much lower. Indeed, the measured stretch for the red PTG is close to 1.34 with both variants and is the maximum stretch. With the original SELFISH strategy, the maximum stretch, achieved for the green PTG, was more than three times larger, around 4.5. We also see that the PTG of intermediate length (in blue) benefits of these variants too and completes earlier. These two variants of the prioritization of tasks in the mapping step have also been applied to the CRA_* heuristics.

The main contribution in [CDS10] was to propose another family of algorithms that derive from a work by Dutot *et al.* [64] that addresses a more general problem than ours. They proposed algorithms for scheduling moldable independent jobs on a homogeneous cluster. PTGs are just one kind of moldable jobs. Indeed, since the tasks in a PTG are themselves moldable, a PTG can be executed on any number of processors. As a result, the algorithms in [64] produce coarse-grain schedules that cannot take advantage of the fine-grain structure of PTGs and lead to schedule fragmentation.

We proposed algorithms based on the ideas in Dutot *et al.*, but that schedule multiple PTGs and focus on optimizing makespan and fairness. We called this family of algorithms Coarse-grain Allocation Fine-grain Mapping (CAFM), since allocations are computed assuming generic moldable jobs, but task mapping is done with respect to individual PTG tasks.

Let describe the contribution made in [64] first. All their algorithms rely on the $3/2 + \epsilon$ approximation algorithm in [65]. This algorithm computes an approximation of the optimal makespan, C_{max}, computes an allocation for each moldable job, and produces a schedule. This schedule is structured as two phases, or "shelves", with jobs scheduled in either one of the two shelves (larger jobs in the first longer shelf, and smaller jobs in the second shorter shelf). The first algorithm in [64] simply uses the two-shelf schedule produced by the approximation algorithm. Two other algorithms were proposed that use only the allocations produced by the approximation algorithm and use two well-known list scheduling algorithms to schedule the jobs with these allocations. The first is Longest Processing Time First (LPTF), which gives priority to the job that has the longest processing time. The second is Smallest Area First (SAF), which gives priority to the job that has the smallest product of the number of processors allocated to it by its execution time.

The last algorithm proposed by Dutot *et al.* uses only the approximation of C_{max} computed by the approximation algorithm. It then partitions the time from 0 to C_{max} in K phases, or "shelves," where K depends on C_{max} and the smallest possible execution time over all jobs. By contrast with the schedule produced by the approximation algorithm, there can be more than two shelves, and shelves increase in duration throughout the schedule. The goal is then to determine which jobs are scheduled within each shelf. This is done by solving a knapsack problem, via dynamic programming, for each shelf, from the smallest to the largest shelf. The goal is to maximize the "weights" of the jobs packed into each shelf, where the weights are those used for computing the weighted average completion time, which is one of the two criteria to optimize. The resulting schedule is then compacted via a number of optimizations (*e.g.*, shuffle the order of shelves randomly, use a list scheduling heuristic to schedule jobs while respecting a given shelf order).

The algorithms we designed in the CAFM family all start by computing the execution time of each PTG assuming that p processors are available, with p varying from 1 to P. This is done using the allocation procedure from the biCPA algorithm [DS10], given by Algorithm 4, to schedule the PTG on the given number of processors. In this way we obtain a specification of each PTG as a moldable job. We can then use the approximation algorithm in [65] to compute an approximation of C_{max}, and an allocation for each job. Note that, unlike in [64], we do not attempt to reuse the two-shelf schedule produced by this algorithm for two reasons. First, the second shelf contains mostly smaller jobs, which is detrimental to fairness. Second, even if the two shelves were to be swapped, we propose below a K-shelf algorithm that subsumes the two-shelf approach.

Once allocations have been computed for each job, like in [64], we can use standard list scheduling techniques to schedule the jobs. Each job is then assigned a rectangular "box" in the schedule, that spans a number of processors and a number of time units. Within this box we can then schedule the individual tasks of the job, which is really a PTG. Once this is done for all PTGs, we obtain a schedule for all tasks of all PTGs. This schedule is likely very fragmented, and we compact it using a backfilling step. For list scheduling we use the SAF and LPTF approaches as in [64]. However, we note that LPTF, unlike SAF, is likely detrimental to fairness since it gives priority to longer jobs and risks postponing the execution of shorter jobs. For this reason we also use a Shortest Processing Time First (SPTF) approach. We obtain three algorithms: `CAFM_LPTF`, `CAFM_SPTF`, and `CAFM_SAF`.

The K-shelf idea in [64] is attractive from the perspective of fairness, as smaller jobs can be placed in smaller, earlier shelves. We reuse the algorithm from [64] to compute a K-shelf schedule for moldable jobs. One difference is that the packing of jobs into shelves solves a knapsack problem in which the weights of all the jobs are equal to 1, as opposed to an arbitrary weight. This is because our objective is to maximize an unweighted notion of fairness, rather than maximizing weighted average completion time. Note that, unlike in [64], we do not shuffle shelf order as shelves of non-decreasing durations promote fairness. Once a K-shelf schedule has been produced, as for the three CAFM algorithms described earlier, we schedule the tasks of each PTG within its box, and use backfilling to compact the schedule. We name this algorithm `CAFM_K_SHELVES`.

Finally, we propose an algorithm that combines the ideas from algorithms in the CRA family and those by Dutot *et al.*. Recall that the CRA algorithms simply attempt to constrain the number of processors used for each PTG. While these algorithms use a number of heuristics to compute those constraints, it is possible to base them on the allocations computed by the approximation algorithm in [65]. These allocations are known to provide a strong guarantee on the overall makespan, and may therefore provide a good basis for producing a desirable overall schedule. Using the same procedure as for the other algorithms in this section we execute the approximation algorithm and obtain a resource constraint for each PTG. Using this constraint, we can now compute an allocation for each task of each PTG. We can then schedule all tasks together, using the standard list scheduling approach of prioritizing tasks by decreasing bottom-levels. Like for the other CAFM algorithms, we compact the schedule using backfilling. We name this algorithm `CAFM_CRA`. We also implemented a `CAFM_CRA_WEIGHT` variant of `CAFM_CRA` while a `CAFM_CRA_ORDER` variant was found to be always outperformed.

Based on our experiments, three out of all the studied algorithms have emerged. We have found that `CAFM_K_SHELVES` is the best algorithm considering its performance in terms of overall makespan, scaled sum completion time and maximum stretch. Another algorithm that achieves performance close to that of `CAFM_K_SHELVES` is `CRA_WORK_WEIGHT`. This algorithm uses a much simpler allocation and mapping procedure. It may therefore be a good choice for large problem instances, for which the time needed by `CAFM_K_SHELVES` to compute the schedule may be prohibitively large. `CRA_WORK_WEIGHT` would also be a good choice for practitioners that prefer a less involved implementation of the scheduling algorithm. Finally, the even simpler `SELFISH_ORDER` outperforms both these algorithms in terms of maximum stretch and scaled sum completion time, but performs poorly in terms of makespan. If makespan is not a metric under consideration, then `SELFISH_ORDER` is the algorithm of choice.

In [CDS10], we investigated whether better schedules can be obtained by relaxing the resource constraint applied to each PTG. Our rationale was that, to increase fairness, it should be beneficial to allocate more processors to short PTGs so that they can complete earlier. These processors can then be later redistributed among other PTGs The allocation of each PTG is thus *malleable*, *i.e.*, decomposed in several periods with a potentially different number of allocated processors in each period. Based on this idea, we proposed a novel algorithm, called Malleable Allocations with Guaranteed Stretch (MAGS), that determines such periods and computes allocations within them these periods. The periods and allocations are based on a relaxation of a perfectly fair schedule assuming that PTGs are ideal malleable jobs. This relaxation comes with a guarantee relative to both performance and fairness.

As stated earlier, CAFM_K_SHELVES and CRA_WORK_WEIGHT were the best algorithms in [CDS10]. They both schedule each PTG within a rigid rectangular "box." On the contrary, the MAGS algorithm reasons on malleable allocations and structures the execution of the batch of PTGs as a sequence of time periods. We detail the steps of this algorithm hereafter, starting with the determination of the periods.

To structure the schedule in periods we use a perfectly fair schedule as a starting point. In a perfectly fair schedule all PTGs experience the same stretch, S, and the best perfectly fair schedule is the one that leads to the smallest value for S. We first compute a lower bound on S, called S^*. This lower bound is computed under the (unrealistic) assumption that each PTG is an ideal malleable job, *i.e.*, composed of an infinite number of independent, infinitesimal tasks. It is therefore possible to allocate any number of processors to each job at any instant in time.

Recall from Section 1.2.3 that $C^*_{max_i}$ denotes the makespan of PTG i when scheduled on the dedicated cluster. Let us assume, without loss of generality, that $C^*_{max_i} \leq C^*_{max_{i+1}}$ for $i = 1, \ldots, N-1$. In a perfectly fair schedule that achieves a stretch S, job i completes exactly at time $S \times C^*_{max_i}$. We can now write simple constraints on the total work, *i.e.*, execution time multiplied by number of allocated processors, of each job. For each i, all job j for $1 \leq j \leq i$ must be completed by time $S \times C^*_{max_i}$. Otherwise a PTG would have a stretch higher than S. Therefore, the sum of the works of each job j, for $1 \leq j \leq i$, must be lower than or equal to $P \times S \times C^*_{max_i}$. More formally:

$$\forall i = 1, \ldots, n \quad \sum_{j=1}^{i} p \times C^*_{max_j} \leq p \times S \times C^*_{max_i} \ , \tag{1.25}$$

and we obtain the lower bound on the stretch as:

$$S^* = \max_{i=1,\ldots,n} \frac{1}{C^*_{max_i}} \sum_{j=1}^{i} C^*_{max_j} \ . \tag{1.26}$$

We wish to structure the schedule as a sequence of time periods. Within each period a job is allocated a number of processors. One possibility would be to have the period boundaries coincide with the job finish times. This would lead to N periods, with the boundaries at times $S^* \times C^*_{max_i}$, $i = 1, \ldots, N$. There could therefore be many periods (up to N), and, more importantly, some of these periods could be short when two jobs have similar $C^*_{max_i}$ values. Short periods are not a problem under the assumption that jobs are ideally malleable. However, in our schedule, each task of a PTG must be scheduled entirely within a period. We impose this constraint because, as we will see, it makes task allocation and task scheduling tractable. Unfortunately, this constraint also means that, in general, a short period may not be usable: all ready tasks of a PTG may simply have execution times larger than the period duration even for large allocations. Consequently, we propose a relaxation of the perfectly fair schedule so that we can reduce the number of periods and avoid short periods. This relaxation, described hereafter, comes with the guarantee that the maximum stretch over all PTGs is not more than a fixed factor away from the bound S^*.

We structure the schedule as M time periods. Period $i = 1, \ldots, M$ lasts from time t_{i-1} until time t_i, with $t_0 = 0$. t_i, $i = 1, \ldots, M$, and M are to be determined. We denote by i_j the index of the period during which job j completes in the perfectly fair schedule. More formally, $t_{i_j-1} \leq S^* \times C^*_{max_j} < t_{i_j}$. We set $t_1 = S^* \times C_{max_1}$ so that only job 1 may complete during the first period. We use geometrically increasing periods, relaxing the perfectly fair schedule so that each job $j > 1$ completes at time t_{i_j} rather than at time $S^* \times C^*_{max_j}$. Periods are defined geometrically by $t_{i+1} = t_i \times (1 + \lambda)$ for $i = 2, \ldots, M$ and some $\lambda > 0$. Therefore, $t_i = S^* \times C_{max_1} \times (1 + \lambda)^{i-1}$ for $i = 1, \ldots, N$. Let use write the completion time of job $j > 1$ in the perfectly fair schedule as $t_{i_j-1} + \varepsilon$, for $\varepsilon \geq 0$. The stretch of job $j > 1$ in the relaxed schedule, S_j, is computed as:

$$S_j = \frac{t_{i_j}}{C^*_{max_j}} = \frac{t_{i_j-1}(1+\lambda)}{C^*_{max_j}} = \frac{t_{i_j-1}(1+\lambda)}{\frac{1}{S^*}(t_{i_j-1}+\varepsilon)} = (1+\lambda)S^* \frac{t_{i_j-1}}{t_{i_j-1}+\varepsilon} \leq (1+\lambda)S^* \tag{1.27}$$

We have therefore obtained a set of periods so that each job completes at the end of a period, with the guarantee that no job experiences a stretch higher than $S^*(1+\lambda)$.

Our algorithm uses $\lambda = 1$, meaning that the duration of period $i+1$ is twice the duration of period i, and the maximum stretch is $2 \times S^*$. Note that our initial goal was to avoid short periods. We allow the specification of a bound on the smallest period, π, so that we do not generate any period of duration shorter than π. If $\pi > S^*C^*_{max_1}$, then we merge the first two periods into one of duration $S^*C^*_{max_1}(1+\lambda)$, leading to $\lambda = \max(1, \pi/(S^*C^*_{max_1}) - 1)$. If instead $\pi \leq S^*C^*_{max_1}$, then the smallest period may be the second one, which is of length $S^*C^*_{max_1}\lambda$. We obtain $\lambda = \max(1, \pi/(S^*C^*_{max_1}))$. In summary, given the $C^*_{max_i}$ values and a specification of the smallest allowable period π, we can generate a set of periods for which, under the perfectly malleable parallel job assumption, the stretch of a job is guaranteed not to be more than a factor two away from S^*.

Once the periods are determined, our algorithm computes allocations within each period in the following way. At each period, available processors are allocated in a round-robin fashion to all jobs that must complete during the period. Jobs are then considered in order of increasing completion time. The algorithm allocates processors to each job greedily considering periods in order, until enough processors have been allocated to the job. Enough processors have been allocated once the sum of the works of the job at all periods (*i.e.*, the number of allocated processors during a period multiplied by that period's duration) is equal to the job's total work.

Figure 1.21 illustrates this allocation scheme on an example. Six PTGs have to be scheduled concurrently on a cluster that comprises 47 processors. The values of $C^*_{max_i}$ are respectively 45, 76, 93, 102, 221 and 232 seconds. According to Equation 1.25, S^* is equal to $(45+76+93+102+221+232)/232 = 3.31$. This means that, in a perfectly fair schedule, the stretch of a job is at least 3.31.

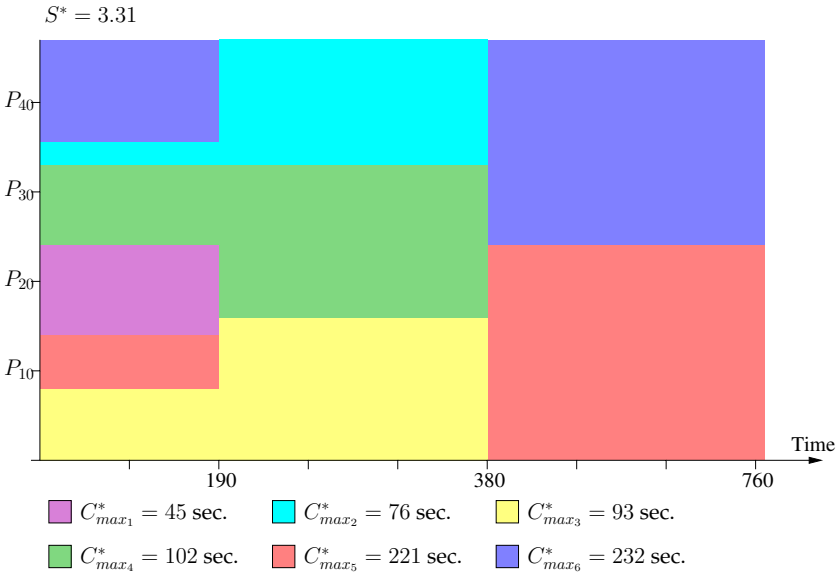


Figure 1.21: Example of period and malleable allocations for the concurrent scheduling of 6 PTGs on a cluster of 47 processors.

Our algorithm generates periods ($[0; 190]$, $[190; 380]$, and $[380; 770]$). The smallest PTG ($C^*_{max_i} = 45$) is the only one to finish in the first period. Note that the two largest PTGs use no processors in the second period to let PTG 2, 3, and 4 complete earlier. Assuming perfectly malleable jobs, the respective stretches of the PTG in the relaxed schedule are at worse 4.2, 5.0, 4.08, 3.43, 3.48 and 3.31. Expectedly, these values are greater than S^* and lower that $2 \times S^*$.

Given these allocations, we must now schedule the tasks of the PTGs. The main objective is that PTG i completes no later than $S^* \times C^*_{max_i}$. To achieve it, we consider the periods in reverse order and schedule the tasks of each PTG in a bottom-up fashion, *i.e.*, starting from the exit tasks and moving towards the entry task. Then the exit task of each PTG finishes exactly at the end of the last period at which the PTG's allocation is non-zero. Implementing this scheme requires a "reversed" copy of each PTG, obtained by changing the direction of each edge $e_{i,j}$ in $\mathcal{E}$. Successors of a task become predecessors and conversely. For each period, each PTG is allowed to use a certain number of processors. We compute allocations using the allocation procedure of the biCPA algorithm. We then select the task with the highest bottom level in the reversed PTG and map it on the set of processors that minimizes its finish time.

Depending on the periods and the PTGs, it may not be possible to schedule all tasks. Indeed, the allocations are computed assuming that the jobs are ideally malleable. But in practice, PTG tasks are not infinitesimal. Therefore, some tasks may not be able to complete before the end of a period and may be postponed to the subsequent period. Tasks may then remain unscheduled after the procedure finishes, meaning that the schedule is not feasible. If this situation arises, we introduce *slack* in the schedule, meaning that the guarantee on the maximum stretch is no longer $2 \times S^*$ but instead $slack \times 2 \times S^*$, where $slack \geq 1$. If $slack > 1$, then the schedule is still perfectly fair, but PTGs experience lower performance than when $slack = 1$. In such a schedule periods are longer and possibly more numerous. For a $slack$ value large enough, one is guaranteed to be able to compute a feasible schedule.

Our algorithm searches for the smallest $slack$ value that leads to a feasible schedule. It starts with $slack = 1$ and doubles it until a feasible schedule is found. A binary search is then used between this slack value and 1 to find the smallest $slack$ value that leads to a feasible schedule. Once such a schedule has been found, it is compacted using backfilling. Algorithms 10 and 11 summarize these steps.

Algorithm 10 The MAGS algorithm

```
for all PTG i do
    Compute C*_{max_i}
    Make a reversed copy of PTG i
end for
Sort the PTGs by increasing values of C*_{max_i}
slack = 1
while schedule(PTGs, slack) != ok do
    slack = slack × 2
end while
maxslack = slack
Binary search for the smallest slack between 1 and maxslack for which schedule() returns ok
Apply backfilling
```

Algorithm 11 schedule(PTGs, $slack$)

```
Determine periods and malleable allocations
for all period do
    for all reversed PTG i do
        for all unscheduled task v of PTG i do
            Determine t's allocation using and schedule t if it fits in the current period
        end for
    end for
end for
return (no task remains unscheduled)
```

Our experiments showed that MAGS achieves better results for these three considered metrics than SELFISH_ORDER. More importantly, MAGS achieves performance on par with that of CAFM_K_SHELVES and CRA_WORK_WEIGHT, but leads to a significant improvements on fairness. Finally, MAGS produces schedules in a reasonable amount of time, which renders it usable in practice.

In this section, I have detailed the different heuristics designed to schedule a batch of PTGs on a single compute cluster. Unlike in the previous section on single PTG scheduling, a single branch of the taxonomy in Figure 1.4, and hence a single scheduling problem, was studied. Moreover, all the proposed heuristics derive from the same common idea of splitting the available resources among the PTGs to prevent competition. This part of my work was then to follow a lead and propose improvements step by step. It ended up with the proposition of the MAGS algorithm [CDS10] that leads to the best compromise between the studied metrics while remaining usable in practice.

1.6 Conclusion and Outlook

The easiest way to wrap up this chapter on Parallel Task Graph scheduling and summarize my contributions to this topic is to come back to the taxonomy proposed at the beginning of this chapter. Figure 1.22 shows the same classification as Figure 1.4 but also lists the different algorithms I proposed where they do belong. We see on that figure that the single PTG branch is almost totally covered, except for two cases. The Single Cluster – Single Criterion – One step case has been addressed by the work in [150] and [151]. Results in [DS10] showed that the biCPA algorithm outperforms iCASLB both for performance and scheduling time. Then, no algorithm was proposed for the Multiple Clusters – Multiple criteria scheduling problem. However, it would be easy to rely on the concept of virtual homogeneous cluster proposed in [NS06] to extend the biCPA algorithm to this context.

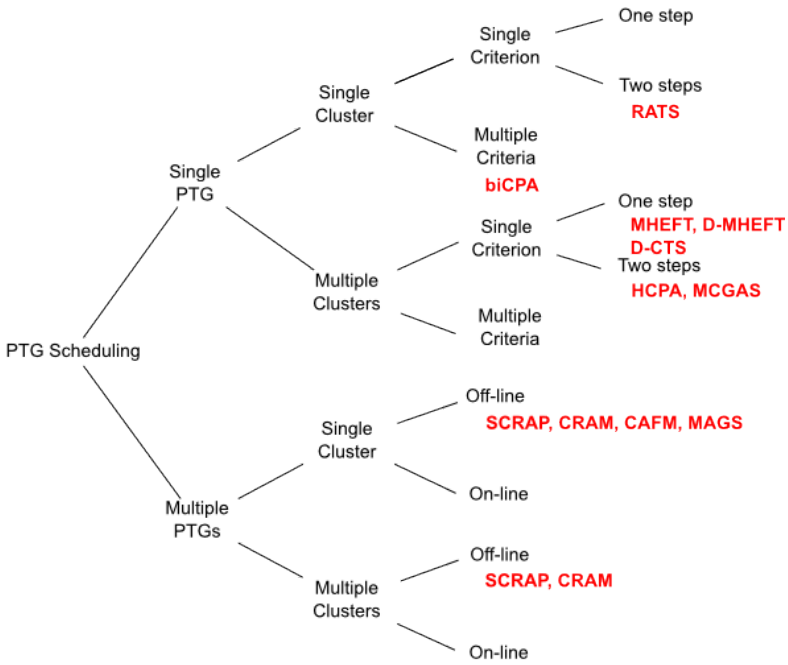


Figure 1.22: A taxonomy of Parallel Task Graph scheduling problems with the proposed heuristics.

As mentioned in the conclusion of the previous section, my recent work on the scheduling of multiple PTGs has been focused on the Single Cluster – Off-line case, even though the SCRAP and CRAM algorithms have been originally designed with multiple clusters in mind. Studying the on-line scheduling of multiple PTGs is a very challenging and interesting problem. Faced with the features that some JRMS such as OAR [34] offer, *e.g.*, inter-dependent jobs or container jobs, we can see the opportunity to propose innovative solutions for the scheduling of large applicative workflows on large distributed computing infrastructures using PTGs.

Another research direction comes from some kind of frustration that is twofold. It was actually very hard, or nearly impossible, to find existing applications onto which apply the algorithms I have designed. Indeed, applications are usually represented either by a (very) complex DAG whose nodes are purely sequential computations, or by a very simple PTG that comprises moldable jobs. In the former category we can cite the task graphs that represent high level linear algebra operations and are exploited in the Directed Acyclic Graph Unified Environment (DAGue) project [25]. The objective of this project is to decompose a complex parallel application into an orchestration of more simple computations that fit well on a single core. This aim is then quite contradictory to the use of moldable tasks. In the latter category we found some image processing applications that have a very simple fork-join structure. Each child of the fork corresponds to a chain of filters applied to a part of the initial image. As parallel implementations of image filters exists, we can consider such applications as PTGs, but applying complex scheduling algorithms on such simple graphs would be overkill. Finding an application that is a PTG **and** would truly benefit of the use of a scheduling algorithm looks like the quest for the Holy Grail [42]. This lack of validation of the algorithms on real applications running in production is the origin of the second source of frustration. Because of it, the articles describing the algorithms were always in a sort of in-between situation when it came to the peer-review evaluation. For theoreticians, our algorithms were too pragmatic and lack of NP-completeness or worst-case complexity analysis. For application or middleware developers, our algorithms were too theoretic and lack of a "true" *in-situ* experiment with a "true" application to be fully validated. The consequence was that critics arose from both sides, often disregarding the merits of the approach.

Two ongoing works are good candidates to (eventually) apply all the algorithms detailed in this chapter in a production context. The first one is the subject of the Ph.D. of S. Gault that I am co-advising. It concerns the study and implementation of advanced scheduling for MapReduce applications. MapReduce [52] is a parallel programming paradigm specifically designed to analyze massive amounts of data. A MapReduce application is usually composed of two basic user-provided functions applied in two steps. First the data is divided into chunks that are passed as input of the *Map* function. One instance of this function is created for each chunk. All the maps are scheduled across the available resources. It generates intermediate data, *i.e.*, key/value pairs that are (potentially shuffled and) distributed to several instances of the *Reduce* function. Figure 1.23 shows the typical workflow representation of a MapReduce application.

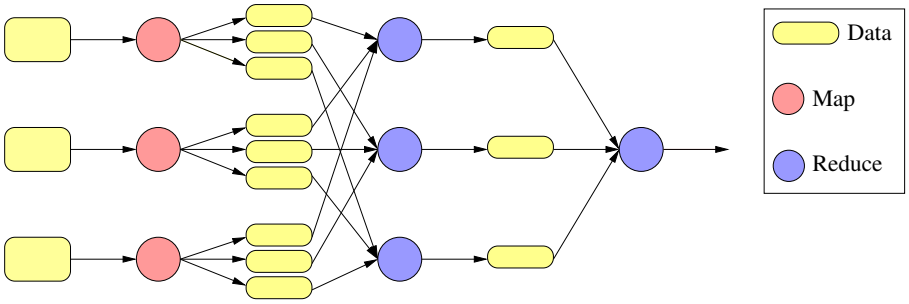
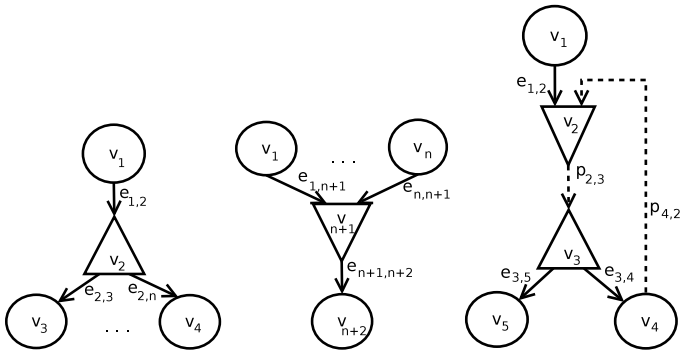


Figure 1.23: Typical workflow a MapReduce application.

Even though this workflow is not as complex as those of linear algebra operations, it is not as simple as image processing workflows either. Moreover, what makes MapReduce applications a good candidate for PTG scheduling is that the Map (and/or Reduce) function can be complex enough to be a moldable task. Another option is to constitute groups of sequential Maps (and/or Reduces) to form a parallel task. This approach is interesting as computing and data location are tightly linked in such applications. Then, applying advanced scheduling techniques makes sense. Finally, compute clusters dedicated to data analysis are often shared by multiple MapReduce users. All the work on multi-PTG scheduling could then be applied. Some preliminary work has been published in [ABB+12, ACB+13] as part of the MapReduce project[1] funded by the Agence Nationale de la Recherche (ANR).

The second ongoing work that aims towards real applications considers scientific workflows that have a *non-deterministic* structure. Among such applications, we can cite the problem of gene identification by promoter analysis [5], or the Grid ENabled Integrated Earth (GENIE) project that aims at simulating the long term evolution of the Earth's climate [112]. In this context, the classical PTG model is augmented with special semantics [149] that allow for exclusive diverging control flows, *e.g.*, conditional structures, or repetitive flows, *e.g.*, cycles. Adding this additional degree of freedom enlarges the scope of target applications but also increases the complexity of the scheduling process.

In [CDMS12], with E. Caron, F. Desprez and A. Muresan, we made a first step towards the scheduling of such non-deterministic PTGs by focusing on the allocation step. We added the following control nodes to the existing PTG model. A **OR-split** node has a single predecessor and any number of successors, that represent mutually-exclusive branches of the workflow. When the workflow execution reaches an OR-split node, it continues through only one of the successors. The decision of which successor to run is taken at runtime. Then in the scheduling phase, all the sub-workflows deriving from an OR-split node have to be considered as equally potential execution paths. Conversely an **OR-join** node has any number of predecessors and a single successor. If any of the parent sub-workflows reaches this node, the execution continues with the successor. A **Cycle** construct is an edge joining an OR-split node and one OR-join ancestor. Figure 1.24 gives a graphical view of these control nodes and constructs.



(a) OR-split (b) OR-join (c) Cycle

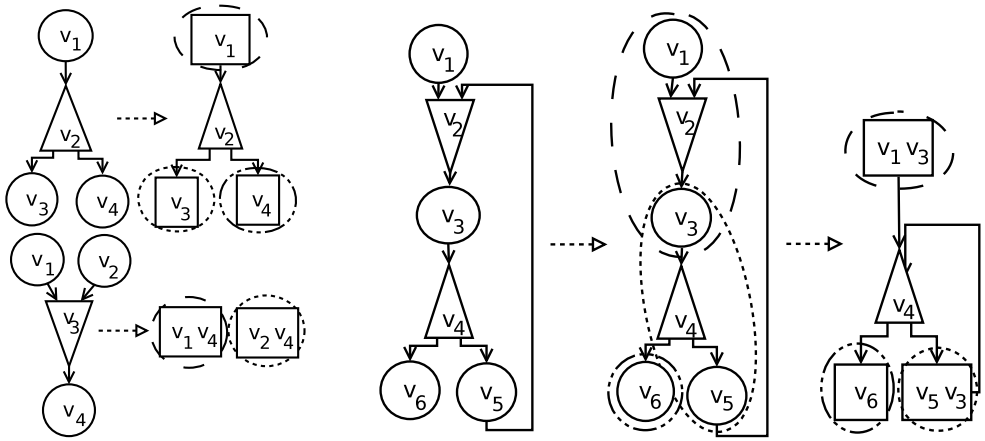
Figure 1.24: Non-deterministic workflow control nodes and constructs.

Figure 1.24(c) is a simple representation of the Cycle construct. $p_{2,3}$ and $p_{4,2}$ are not edges of the workflow, but paths leading from v_2 to v_3 and from v_4 to v_2 respectifely. These paths are a weak constraint that ensure the creation of a cycle in the graph, in combination with the OR-join and OR-split nodes v_2 and v_4. However, a Cycle can contain any number of OR-split or OR-join nodes and even an unbound number of edges leading to other parts of the workflow.

[1] http://mapreduce.inria.fr

In [CDMS12], we target Infrastructure as a Service (IaaS) cloud platform on which using compute resources has a cost. Executing an application is then constrained by a budget that has to be respected by the chosen allocation. The algorithm is thus divided in three steps. First, we split the non-deterministic workflow into a set of deterministic PTGs. It allows us to fall back to a well studied scheduling problem and to reuse existing algorithms. In a second step, we divide the overall budget among the resulting PTGs. Finally we determine an allocation, under a specific budget constraint, for each PTG.

Transforming a non-deterministic workflow into a set of PTGs amounts to extract all the sequences of nodes free of any non-deterministic construct. The OR-split nodes define the boundaries. An OR-split node leads to $n + 1$ sequences, one ending with its predecessor and n starting with each of its successors. The OR-join nodes do not actually lead to the creation of new sequences since they do they do not lead to non-deterministic transitions. They just preserve the number of sequences coming from their inwards transitions. These principles are illustrated in Figure 1.25.



(a) OR-split and OR-join (b) Cycle construct

Figure 1.25: Extracting sub-workflows from OR-splits, OR-joins and Cycles.

Extracting PTGs from a Cycle construct is simple if we consider that a Cycle is actually composed of one or more OR-split nodes and a feedback loop. Then we fall back to the rules of splitting OR-split nodes, as shown in Figure 1.25(b). Here we extract three PTGs containing two instances of task v_3. One is a result of executing task v_1 and the other derives from following the cycle branch. Task v_5 is the predecessor of this second instance.

Once the allowed budget has been distributed among PTGs, we apply to each of them modified versions of the allocation procedures of the CPA [125] and biCPA [DS10] algorithms. The modifications mainly concern the definition of the average area (see Equation 1.8) to take the specifics of an IaaS cloud into account. The procedure based on CPA determines a single allocation for the whole PTG while the second procedure acts as that of biCPA, *i.e.*, determines candidate allocations under tighter constraints in an iterative way. The makespan and cost of each of these candidate allocations are then estimated and the algorithm selects the one that achieves the best makespan while respecting the budget constraint. More details on these procedure are given in [CDMS12]. We found that the second procedure is more suited to small applications or small budgets, while the first one has to be preferred for large applications or large budgets. However, more work is still needed before obtaining of a complete scheduling algorithm that could be integrated into a cloud management stack. Indeed, the non-deterministic transitions are solved at runtime. Then the offline decisions taken at scheduling time may have to be reconsidered. This important part of my research work may not be totally closed after all ...

Chapter 2

Simulation of Distributed Systems & Applications

2.1 Introduction

During my PhD, my work on parallel task graph scheduling was centered around a specific application, the Strassen matrix multiplication algorithm. Then it was possible to assess the relative performance of a given scheduling strategy through *in situ* experiments, *i.e.,* running the real application on a real platform. During my post-doctorate at the University of California, San Diego, I aimed at extending the scope of my scheduling studies. Considering more applications and more target computing infrastructures made it more difficult to perform time and resource consuming experiments. Consequently, I decided to resort to simulation to evaluate the proposed scheduling heuristics. Simulations may not be as realistic as *in situ* experiments but come with attractive features such as the reproducibility of results, an objective basis for application comparison, and the capacity to explore a broad range of experimental scenarios in a reasonable amount of time. This decision was eased by the fact that my adviser, Henri Casanova, was the founder of the SIMGRID project. To be clear from the outset, SIMGRID is not a simulator. It provides all the required fundamental abstractions to design simulators of parallel applications in distributed environments. Moreover the first version of SIMGRID was specifically designed for the evaluation of scheduling algorithms. Then it seemed natural to rely on this existing framework to build my own simulators. However, the common practice in Computer Science is alas to develop a complete *ad-hoc* simulator, *i.e.,* from specific user code to models of resources, for each new study. Most of the times the underlying details of such simulators are not even detailed in publications. This is contradictory with what is commonly done in other experimental sciences. For instance, every article in biology or physics starts with a thorough description of the materials and methods used in the article. Such a description makes references to previously published works from which techniques or inspirations were borrowed. This ensures a certain pledge of quality of the contents of the article. It also allows the reader to be able to reproduce the conducted experiments to confirm, infirm, or extend the presented results. In other words, giving as many details as possible on a simulator is mandatory to ensure the reproducibility of simulation results obtained with it.

Changing the common practices in the field is a long term objective. While I do not pretend to achieve this alone, I tried to do my own share of the process over the last decade. It started by basing my simulation studies on an existing tool rather than developing yet another undocumented-unvalidated-undistributed simulator from scratch. Later, I happened to contribute to this framework at different levels. These contributions will be described in the remaining of this chapter, but it is important to describe what is meant by the term "simulation" first.

Figure 2.1 describes the different components of a simulation environment and their interactions. First, a simulator comprises an *application* to test, *e.g.,* a scheduling algorithm, and a *simulation kernel*.

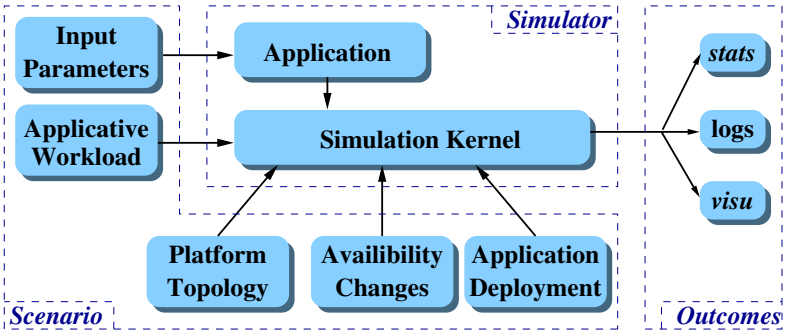


Figure 2.1: Components of a classical simulation environment.

The application is generally very specific to a given simulation study. This is a variable part of a simulator whose rewriting is acceptable. The simulation kernel, however, is the core of a simulation toolkit such as GridSim [29], OptorSim [19], or SIMGRID [39]. For some of them, a lot of efforts has been put to assess the validity of the models underlying the simulation kernel. Except from very simplistic studies, it would be foolish or arrogant to ignore a kernel with an important validation background and develop a simple kernel on his/her own with too many abstractions.

In addition to the simulator itself, a simulation also implies the definition of an experimental scenario whose complexity may vary. A fundamental component is a model of the *platform topology*, *i.e.,* an interconnection of computing elements through a network. The *input parameters* of the simulated application, *e.g.,* the jobs to schedule, are also part of the scenario. The two remaining components add the capacity to inject external dynamic conditions that impact the application (the *workload*) or modify the infrastructure (the *availability changes*). During or at the end of a simulation run, the kernel can output several kinds of information. It can be raw data such as *logs* of all the events that occurred. These data can also be post-processed to produce higher level information such as *statistics* or *visualization*. All these components of a single simulation lead to more complex tools when it comes to the execution of a *simulation campaign*. Indeed, sound generators of platforms and applications abstractions are needed to build a comprehensive set of scenarios. Advanced analysis and visualization techniques are also mandatory to extract the most pertinent information from the whole set of produced results. Finally, the management of a campaign itself requires some adapted tools to efficiently launch the simulations, retrieve, and store the results. A simulation framework is then a whole ecosystem of tools that allows its users to conduct sound performance evaluations of parallel and distributed applications.

Since 2003, I have contributed to the SIMGRID ecosystem at the interface between the simulated applications and the simulation kernel through my work on the SimDAG API detailed in Section 2.3, but also upstream of a simulation with contributions on the generation of simulation inputs presented in Section 2.4 and downstream of a simulation by improving over the years way I acquired and analyzed simulation results as explained in Section 2.5. These contributions granted me access to the core team of the SIMGRID project in 2009. This coincides with the beginning of the large projects funded by the ANR to make SIMGRID evolve. From 2008 to 2011, the Ultra Scalable Simulation with SIMGRID (USS SIMGRID) project[1] aimed at addressing three challenges related to simulation: **accuracy**, *i.e.,* obtaining simulation results that match results that would be obtained with real-world application executions, or that introduce a quantifiable bias; **scalability**, *i.e.,* simulating large-scale applications on large-scale platforms with low time complexity and low space complexity; and **usability**, *i.e.,* providing users with ways to instantiate simulation models, to augment or develop simulation models, to implement simulations for various application scenarios, to analyze/visualize simulation results, so as to enable

[1] http://uss-simgrid.gforge.inria.fr

reproducibility of simulation results by others. In this project, I was in charge of the work package on the application of simulation to some specific use cases, and more precisely on cluster dimensioning through simulation as described in Section 2.6.

Then, starting in January 2012, the Simulation Of Next Generation Systems (SONGS) ANR project [2], will continue to enlarge the scope of applicative domains in which SIMGRID can be used. The concepts, models, and APIs, needed to simulate IaaS Clouds, and High Performance Computing (HPC) systems will be added to the current framework. With the existing support of Computing Grids and Peer-to-Peer (P2P) and Volunteer Computing systems, SIMGRID will cover the whole spectrum of Large Scale Distributed Computing (LSDC) systems. This project will also continue to investigate issues related to the pillars of simulation methodology that are the design of an efficient simulation kernel and of sound concepts and models, *e.g.*, storage, memory, energy, or High Performance Network (HPN), the analysis and visualization of simulation results, design of experiments and campaign management. As stated earlier, there is a need to change the common practices in Computer Science in terms of experimental evaluation. One of the main objectives of the SONGS project will be to promote Open Science in the context of the simulation of large scale distributed systems and applications. In this project, I am in charge of the work package related to Grids. The focus is made on Data Grids through a collaboration with the Organisation Européenne pour la Recherche Nucléaire (CERN) in order to model the distributed data management system of A Toroidal LHC ApparatuS (ATLAS), which is one of the six particle detector experiments constructed at the LHC. This task requires to add missing concepts, models, and API related to storage to SIMGRID.

2.2 The SIMGRID Framework

SIMGRID is a 12-year old open source project whose domain of application has kept growing since its inception. It was initiated in 1999 as a tool for studying scheduling algorithms for heterogeneous platforms. The first version of SIMGRID [37] made it easy to prototype scheduling heuristics and to test them on a variety of abstract applications (expressed as task graphs) and platforms. In 2003, the second major release of SIMGRID [38] extended the capabilities of its predecessor in two major ways. First, the accuracy of the simulation models was improved by transitioning the network model from a wormhole model to an analytical fluid model. Second, an API was added to simulate generic Concurrent Sequential Processes (CSP) scenarios. The third major release SIMGRID was distributed in 2005, but major new features appeared in version 3.3 in April 2009. These features include a complete rewrite of the simulation core for better modularity, speed and scalability, the possibility to attach traces to resources to simulate time-dependent performance characteristics as well as failure events, and two new user interfaces.

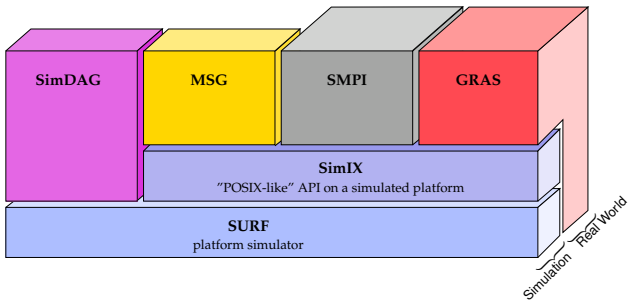


Figure 2.2: SIMGRID components.

[2] http://infra-songs.gforge.inria.fr

The current software stack with its relevant components is depicted in Figure 2.2. The four components on the top of the figure, SimDAG, MSG, SMPI, and GRAS are *user interfaces*. The two components below, SimIX and SURF, form the *simulation core*. A last component, not shown in the figure but used throughout the software stack up to user-space, is the eXtended Bundle of Tools (XBT), a general-purpose toolbox that implements classical data containers (*e.g.,* FIFO, dynamic arrays, hash maps, ...), logging and exception mechanisms, and support mechanisms for configuration and portability. The next two sections describe the user interfaces and the simulation core in details.

2.2.1 User Interfaces

SIMGRID provides four APIs. Two of these APIs are designed for simulating executions of applications based on an abstract specification of them. The **SimDAG** API allows for the simulation of parallel applications structured as DAGs. Vertices represent (sequential or parallel) tasks and edges represent task dependencies and optional data transfers between tasks. A large literature is devoted to the study of DAG scheduling algorithms, and SimDAG provides the necessary abstractions to quickly implement and evaluate such algorithms. The **MSG** API is intended for CSP simulation and provides classic CSP abstractions (processes, mailboxes, channels, etc.). It is therefore generic and, to date, it is the most widely used SIMGRID API, providing bindings for C, Java, Ruby, and Lua.

The remaining two APIs are designed for simulating the execution of applications based on the actual application source codes. The **SMPI** API [CSG+11] targets the on-line simulation of Message Passing Interface (MPI) applications. Actual application code is executed and MPI calls are intercepted so that communication delays can be injected based on network simulation. The **GRAS** API [123] makes it possible to use SIMGRID as a development framework for implementing full-fledged distributed applications. It provides two back-ends, allowing the same application source code to be executed either in simulation or deployed on a real-world platform. GRAS can thus bypass the simulation core entirely, as depicted in Figure 2.2. Consequently, application developers benefit from an enhanced development cycle in which the application can be quickly tested over arbitrary simulation scenarios as it is being developed.

2.2.2 Simulation Core

SURF and SimIX

The component that implements all simulation models available in SIMGRID is called **SURF**. It provides an abstract interface to these models that exposes them as *resources* (*i.e.,* network links, workstations) and *activities* that can consume these resources. For convenience, an additional layer is provided, called **SimIX**. It provides POSIX-like services including *processes*, Inter-Process Communication (IPC), locks, and *actions*. SimIX processes correspond to execution contexts (*e.g.,* threads) for the simulated application, that run code written by the SIMGRID user using one of the provided APIs. All APIs but SimDAG are written using SimIX. This exception is because SimDAG only allows the user to simulate centralized algorithms without independent processes, removing the need for the SimIX layer.

SimIX acts as a virtual operating system that provides a system call interface through which processes place *requests*. These requests are used for all interaction between the user program and the simulated platform. SimIX actions connect the requests from the user programs (expressed through the APIs) and the activities on the simulated resources in SURF. These activities are used by SURF to compute the delays incurred by the user actions (*e.g.,* computations and communication operations). Each activity is represented by a data structure that stores the total amount of "work" to be done (*e.g.,* number of bytes to transfer, number of compute operations to perform) and the amount of work remaining. A process blocks on a request until it is answered when the delay corresponding to all the activities currently performed by the process have expired. Simply put, if a process issues a request that would

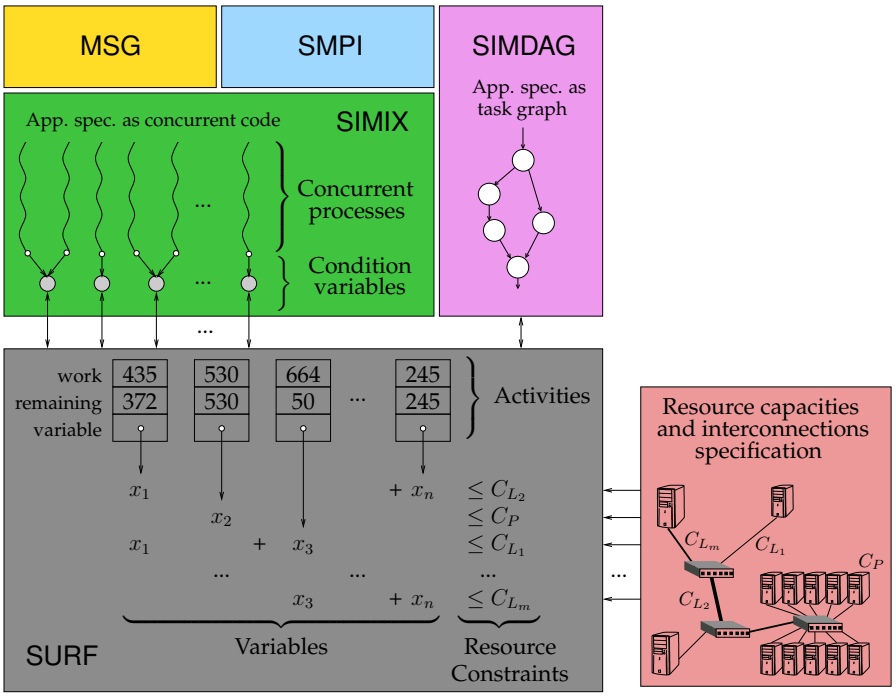


Figure 2.3: Design and internals of SIMGRID.

initiate an activity that should take x time units, this process blocks on the request until SimIX answers it, *i.e.,* once the simulated clock has advanced by x time units. This scheme is depicted in Figure 2.3. For instance, the third activity depicted in the figure corresponds to a total amount of work of 664 units, and 50 of these units remain until completion of the activity. When the activity completes, the request depicted above the action associated to the activity is answered, allowing the corresponding process to continue execution. SIMGRID is designed so that the simulation state can only be updated in the SimIX layer. In SIMGRID, when an activity completes, a task dependency is resolved and other activities can be simulated by SURF.

The Main Simulation Loop

SimIX implements a "simulation loop" through which the simulation makes progress, as shown in pseudo-code in Algorithm 12. The algorithm maintains a set of ready processes, *i.e.,* new processes or those processes whose requests have been answered. The loop executes the processes in the ready set each in turn, and ends when there are no such processes in the ready set (which is either the end of the simulation or a detected deadlock). At the beginning of each iteration, SimIX lets all ready processes execute (line 3). By default, all processes are run in mutual exclusion and in round-robin fashion. Each process runs until completion or until it issues a request. Next, the SimIX maestro handles the set of requests issued by the processes, possibly creating or canceling activities (line 4). These requests are processed in deterministic order based on process ids to ensure simulation repeatability. For each simulated resource with at least one pending activity, SURF determines when each pending activity will complete (note that an activity may use more than one resource). The minimum of these completion dates is then computed and the set of activities that complete at that minimum date is determined (line 5). SIMGRID allows users to attach "traces" to simulated resources. These traces are time-stamped lists of resource states. They are used for instance to simulate time-dependent resource availability for an out-of-band

Algorithm 12 Main simulation loop

```
1: readyset ← all processes
2: while readyset ≠ ∅ do
3:     requests ← run_processes(readyset)
4:     handle_requests(requests)
5:     (t1, activities) ← compute_next_activity_completions()
6:     t2 ← compute_next_resource_state_change()
7:     t ← min(t1, t2)
8:     update_simulation_state(t)
9:     readyset ← answer_requests(activities)
10: end while
```

workload that causes fluctuations in the performance delivered by the resources. SURF computes the earliest resource state change date (line 6), and then the minimum of this date and of the minimum activity completion date (line 7). The state of the simulation is then advanced to this date, updating activity states and resource states (line 8). Finally, based on those activities that have completed, the corresponding requests are answered thus unblocking the relevant processes and updating the ready set (line 9).

Simulation Model Formalization and Implementation

During the main simulation loop, SURF determines execution rates and completion times of activities on resources. This determination is based on the various simulation models implemented in SIMGRID, several of which are discussed in upcoming sections. It turns out that most of these models can be formalized in a unified manner as a multi-variate optimization problem subject to linear constraints. As depicted at the bottom of Figure 2.3, a variable is associated to each activity that quantifies the activity's execution rate. SIMGRID implements an efficient sparse representation of the set of linear constraints, and solves the optimization problem with time complexity linear in the number of activity variables and the number of resources.

2.3 SimDAG: an API for DAG Scheduling Simulation

Before detailing the basic concepts of the SimDAG API and how I have contributed to this part of the SIMGRID toolkit, it is necessary to come back to the early years of SIMGRID to understand where SimDAG does come from. From 1999 to 2003, before the addition of the MSG API by A. Legrand to study CSP scenarios, SIMGRID had only one API designed by H. Casanova. The primary purpose of SIMGRID was then to factor the concepts and models that were common to *ad hoc* simulators developed by each and every student and faculty in the lab.

At that time, SIMGRID focused on algorithms that have to assign a set of *tasks* on a set of *resources* in a way that optimizes some performance metric, typically that minimizes the application makespan. In other words, SIMGRID was primarily designed to study algorithms or heuristics that build schedules. A task can represent different types of basic operations, such as a computation, a data transfer, or an I/O operation. Control or flow dependencies may exist between such tasks to form a DAG. Similarly, a resource can be either a computing unit, a network link, or a storage device.

Comparing the relative performance of several algorithms on an objective basis is almost impossible with actual experiments. It would be too time and resource consuming and variations in resource load over time would prevent obtaining repeatable results. Simulation then appears as the most viable approach to compare scheduling algorithms, provided that a sound simulation framework is used.

The early versions of SIMGRID provided a set of core abstractions and functionalities to easily build *event-driven* simulators. Resources are modeled by two types of basic entities. A *host* represents a single computing unit that is associated to a certain *service rate*, *i.e.*, the capacity to perform a certain number of work units per time unit. The most commonly used unit for this service rate is a number of floating point operations executed per second, or *flop/s*. Then a *link* is a simple network device that connects two entities in a point-to-point fashion. A link can be connected to a host or to another link to form a more complex route. Each link has its own service rate, which corresponds to a network *bandwidth*, *i.e.*, a number of bytes transferred per second, and a *latency*, *i.e.*, the time in second to access the resource. These basic entities can be aggregated to form a *workstation* that comprises a host and at least one link.

The second main concept underlying the first version of SIMGRID is that of a *task*. SIMGRID makes no distinction between computations and data transfers. The user is in charge to ensure that the former are scheduled on computing units while the latter are scheduled on network links. This slightly modifies the way applications structured as a DAG are described, but greatly simplifies SIMGRID implementation and API. Figure 2.4 illustrates how a simple application made of four computational tasks and four data transfers is represented in the scheduling literature and in SIMGRID.

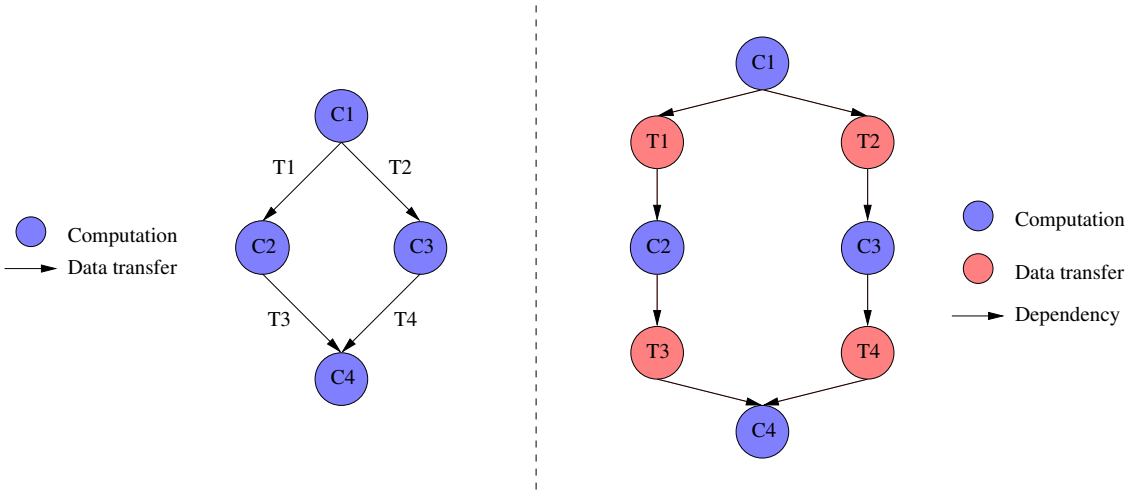


Figure 2.4: Scheduling literature and SIMGRID representations of an application structured as a DAG.

Thanks to these basic concepts, writing a simulator with the first version SIMGRID [37] amounts to:

- Create instances of computing and network resources to form the execution environment with the `SG_createHost` and `SG_createLink` functions. As mentioned earlier, in addition of a name, a host is represented by its service rate while a link has a latency and a bandwidth;
- Create task instances and add dependencies between some of them to form the application DAG respectively with the `SG_newTask` and `SG_addDependent` functions. When created, a SIMGRID task is given a name and a *cost*, *i.e.*, an amount of work to be executed;
- Schedule tasks on resources. This step is the one that allows the user to implement his/her scheduling algorithm. A single function, `SG_scheduleTaskOnResource`, is provided and the user is in charge to ensure that computations are scheduled on hosts and data transfers on links;
- Run the simulation with the `SG_simulate` function. A simulation can be stopped either when all the tasks have completed, upon the completion of a given task, or after a certain amount of simulated time. It allows the user to dynamically take scheduling decisions during the simulation.

While SIMGRID had been specifically designed to implement DAG scheduling algorithms, using it for my own research work on PTG scheduling raised several issues since 2004 and [CDS04]. Indeed, my work considered the problem of scheduling DAGs made of *moldable* tasks, as explained in Chapter 1. Then scheduling a task that belongs to a PTG becomes much more complex than just calling the `SG_scheduleTaskOnResource` function. Once allocated, a moldable compute task corresponds to a set of SIMGRID tasks that have to be scheduled on a set of SIMGRID hosts. For instance, if an algorithm decides that a moldable task has to be executed on five processors, four new SIMGRID tasks have to be created in addition to the one that already exists in the PTG. Then the amount of work of the initial moldable task has to distributed among these five tasks. Finally, each of these five SIMGRID tasks has to be scheduled on a distinct SIMGRID host.

Scheduling a data transfer between two moldable tasks is even more cumbersome. Now we have to determine a *communication matrix* first. It describes what share of the data to transfer each processor in the source allocation has to send to each processor in the destination allocation. For each non-zero element in this matrix, a new SIMGRID task has to be created and scheduled accordingly on the network link or route that connects source and destination hosts.

It is quite easy to see that writing a simulator for a PTG scheduling algorithm with the original API of SIMGRID is a tedious and error-prone process because of the management of these newly created tasks. Moreover, multiplying the number of SIMGRID tasks this way puts an extra burden on the simulation kernel and greatly slows down the simulation speed.

Then I faced a certain dilemma as I wanted to benefit from the quality of the models underlying SIMGRID, but was limited by an API that was not suited to my needs. In August 2005, two releases of SIMGRID were made that changed that situation. The former tended to simplify the way PTG scheduling simulators could be written while the latter complicated it.

With A. Legrand, we added a new concept to SIMGRID, that of a *parallel task*, in release 2.96. Depending on the parameters given upon creation, a parallel task can represent a fully parallel computation, *i.e.,* each host executes its share of the parallel task without any communication, a data redistribution, *i.e.,* a data transfer from n sources to m destinations without any computation involved, or a sort of Bulk Synchronous Parallel (BSP) task that alternates computation and communication phases. More precisely, a parallel task is created with the following arguments:

1. A *name*, that is a user-level information and can be null;
2. The *number of hosts* involved in the execution of the parallel task;
3. The *list of hosts* that will execute the parallel task;
4. An *array of computation amounts* that denotes the amount of work each host has to perform;
5. A *communication matrix* that describes how much data has to be transferred between each and every pair of hosts;
6. Some *user data* attached to the task, if needed.

The most important parameters are the array of computation amounts and the communication matrix that define together the type of a parallel task, *i.e.,* fully parallel, data redistribution, or BSP task. A SIMGRID parallel task is a very flexible entity as heterogeneity is handled seamlessly by the simulation kernel. A user can define a homogeneous parallel task, *i.e.,* each host has the same amount of work to execute, and map it onto a heterogeneous set of hosts. In that case, the execution of the task will progress according to the speed of the slowest host. Conversely, the work to be done or the amount of data to transfer can be unbalanced among hosts. Whatever the processing and network characteristics of the execution environment, each host involved in the execution of a parallel task has to wait for the completion of the task on *all* the hosts to continue the simulation. When both computation array and

communication matrix are filled, the simulation kernel makes each component of the task progress in turn and at their own rate. Communications are overlapped by computations when possible. Finally, if a resource is slowed down during the execution, because of its sharing policy or external load injection, it impacts the task progress rate on all the involved resources.

This addition of a model to the kernel and specific functions to the API totally solves the aforementioned issues related to the management of moldable tasks. Thanks to it, transforming a classical SIMGRID task coming from some DAG description (see Section 2.4.2) into a parallel task becomes relatively easy at scheduling time. In my different works on PTG scheduling, a compute task is initially described by an amount of work and a strictly sequential part (see Equation 1.1). Once its allocation has been determined, *i.e.*, the number of hosts onto which execute the task is known, it is possible to fill the array of computation amounts by applying the Amdahl's law to that initial amount of work. The work is thus evenly distributed among the participating hosts. Similarly, a data transfer task initially corresponds to a global amount of data. Once the allocations of both sending and receiving tasks are known, the communication matrix can be built. There, a major assumption is made for the sake of simplicity, that is a one dimensional block distribution of the data. Figure 2.5 shows an example of a data redistribution from a compute task allocated on five hosts to another mapped on three hosts. With the chosen data distribution, filling the communication matrix is straightforward. When some hosts are common to the source and destination allocations, we first have to build a list of distinct hosts. Then if a host has to keep some data, a communication occurs on a local loopback link.

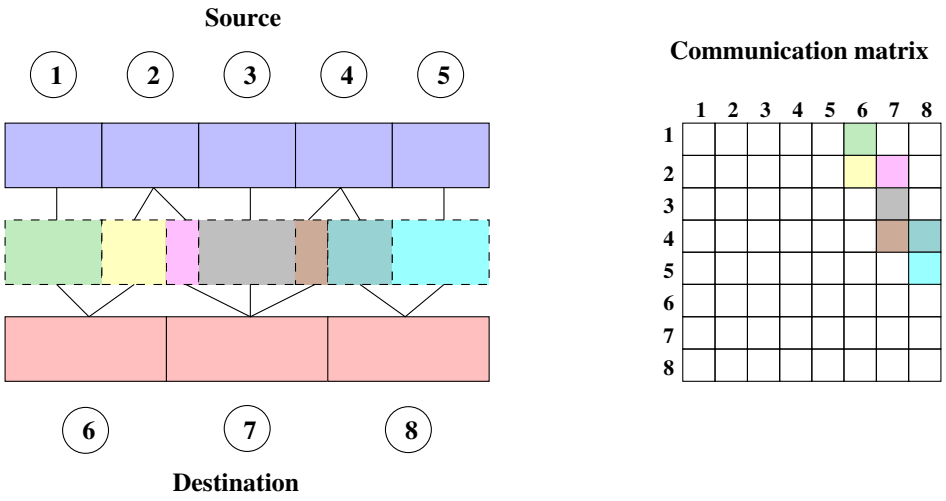


Figure 2.5: Communication matrix of a data redistribution between a source allocation on five hosts to a destination allocation on three hosts.

Unfortunately, the next version of SIMGRID that was released less than two weeks later was the 3.0 major revision in which the historical API was removed to give room to the SURF optimized simulation kernel and the MSG API. Consequently, the functions to manage moldable tasks more easily were part of MSG only, that was not an API adapted to the scheduling of task graphs. Then the simulated execution of a schedule had to be coded as a CSP application instead of a simple call to the `SG_simulate` function. The basic idea was that a set of processes running on some hosts are in charge not only of the execution of a compute task assigned to this set, but also of spawning new processes that will handle the successor(s) of the task and the afferent data redistribution(s). This way of implementing the simulated execution of a schedule was designed only once and reused in several simulators. Nevertheless, it was much more complex than what was allowed by the former API and became a great source of bugs.

To benefit from the addition of parallel task handling **and** the historical API that was tailored to my needs, I initiated the revival of that API on top of the new SURF simulation kernel. With some available funding from the Action de Recherche Collaborative (ARC) INRIA Ordonnancement de Tâches Parallèles en milieu Hétérogène (OTaPHe)[3] that I led at that time, I hired C. Thiery for this job. Thanks to his internship and with the great help of M. Quinson, SimDAG was born and became part of the SIMGRID toolkit in July 2006 with the release of SIMGRID 3.1.

The main aim was to preserve the historical API as much as possible. For most functions, the prefix in the function name simply changed from `SG_` to `SD_`. However, some major semantic modifications have been introduced to remain compliant with the new simulation kernel and coherent with the prevailing MSG interface. Figure 2.6 presents a simple code to schedule and simulate the execution of the DAG from Figure 2.4 on two processors connected by a single network link. This code sample was given in Appendix of the first paper on SIMGRID [37]. Figure 2.7 shows an equivalent code written using the SimDAG API. Both simulation codes can be divided in five sections. For each of these sections, major differences can be highlighted. First, the execution environment, *i.e.,* the set of computing and network resources, has to be declared. Early versions of SIMGRID allowed (but also forced) the user to explicitly create the resources (Fig. 2.6, lines (12-14)). In SimDAG, as in the other APIs, such a declaration is externalized in an eXtended Markup Language (XML) file that is loaded by the `SD_create_environment` function (Fig. 2.7, line 12). In this example, the `platform.xml` file is

Platform file loaded in Fig. 2.7 code

```
1  <platform version="3">
2    <AS  id="AS0"  routing="Full">
3      <host id="p1" power="1e9"/>
4      <host id="p2" power="2E9"/>
5
6      <link id="link" bandwidth="1.25E8" latency="1.0E-4"/>
7
8      <route src="p1" dst="p2"> <link_ctn id="link"/> </route>
9    </AS>
10 </platform>
```

A second difference related to resource declaration is the definition of the processing power of hosts. In Figure 2.6, the processing power is unitless and relative, while in Figure 2.7, hence in SimDAG, it is measured as a number of floating point operations per second and is absolute. Finally, the capacity to attach trace files that describe the dynamically changing latency and bandwidth of a network link and processing power of a host still exists in SimDAG. Indeed, the `<link>` tag accepts `latency_file` and `bandwidth_file` attributes while the `<host>` tag may have an `availability_file` attribute.

The second part of these simulation codes consists in creating the computation and data transfer tasks to simulate. There again the major change is about units. While computation tasks were formerly described by a duration in seconds, they are now associated with an amount of work (in floating point operations) to perform. Moreover the amount of data transferred by a communication task is now expressed in bytes instead of bits. Finally, the `SD_task_create` function has two additional parameters, *i.e.,* a comprehensive name and a field dedicated to user data.

As illustrated by Figure 2.4, forming the application DAG is done by adding dependencies between tasks. The original `SG_addDependent(x,y)` function specifies that task `x` depends on task `y` (Fig. 2.6, lines 27-30)). This way of adding a dependency between two tasks may seem quite counter-intuitive if we refer to the orientation of the edges in the application DAG. Then, in SimDAG, a dependency is created from a source to a destination (Fig. 2.7, lines 25-28)) which is more natural with regard to the execution flow. As for task creation, a comprehensive name and user data can be given as input.

[3] `http://www.loria.fr/~suter/OTAPHE/`

```
#include "simgrid.h"
int main() {
  SG_Resource p1,p2,link;
  SG_Task c1,c2,c3,c4; /* Computations */
  SG_Task t1,t2,t3,t4; /* Data Transfers */make
  SG_Task *tasks done; /* List of completed tasks */
  double clock;

  SG_init(); /* Simgrid initialization */

  /* 2 processors (P2 twice as fast as P1) and 1 link */
  p1 = SG_createHost("p1",1.0,"./cpu1");
  p2 = SG_createHost("p2",2.0,"./cpu2");
  link = SG_createLink("link", "./latency","./bandwidth");

  /* Create the tasks */
  c1 = SG_newTask("c1",50.00);    /* 50 s */
  c2 = SG_newTask("c2",100.00);   /* 100 s */
  c3 = SG_newTask("c3",200.00);   /* 200 s */
  c4 = SG_newTask("c4",80.00);    /* 80 s */
  t1 = SG_newTask("t1",1000.00);  /* 1 Kb */
  t2 = SG_newTask("t2",10000.00); /* 10 Kb */
  t3 = SG_newTask("t3",200000.00);/* 200 Kb */
  t4 = SG_newTask("t4",200000.00);/* 200 Kb */

  /* Set the dependencies */
  SG_addDependent(t1,c1);SG_addDependent(t2,c1);
  SG_addDependent(c2,t1);SG_addDependent(c3,t2);
  SG_addDependent(t3,c2);SG_addDependent(t4,c3);
  SG_addDependent(c4,t3);SG_addDependent(c4,t4);

  /* Schedule computations to hosts */
  SG_scheduleTaskOnResource(c1,p1);
  SG_scheduleTaskOnResource(c2,p1);
  SG_scheduleTaskOnResource(c3,p2);
  SG_scheduleTaskOnResource(c4,p2);

  /* Transfers t1,t4 within a single host */
  SG_scheduleTaskOnResource(t1,SG_LOCAL_LINK);
  SG_scheduleTaskOnResource(t4,SG_LOCAL_LINK);

  /* Transfers t2,t3 scheduled on the link */
  SG_scheduleTaskOnResource(t2,link);
  SG_scheduleTaskOnResource(t3,link);

  /* run for 100 seconds of virtual time */
  SG_reset();
  tasks done=SG_simulate(100.0,NULL,SG_VERBOSE);

  /* run until at least the simulation completes */
  SG_reset();
  tasks done=SG_simulate(-1.0,NULL,SG_VERBOSE);
  clock=SG_getClock();

  /* Free memory used by Simgrid */
  SG_shutdown();
  exit(0);
}
```

Figure 2.6: SimGrid v1 code sample [37].

```c
#include "simdag/simdag.h"
int main(int argc, char **argv) {
  SD_workstation_t *host_list;
  SD_task_t c1,c2,c3,c4; /* Computations */
  SD_task_t t1,t2,t3,t4; /* Data Transfers */
  xbt_dynar_t tasks_done; /* dynamic array of completed tasks */
  double *comp_array, *comm_matrix, clock;

  SD_init(&argc, argv); /* SimDAG initialization */

  /* 2processors (P2 twice as fast as P1) and 1 link */
  SD_create_environment("platform.xml");

  /* Create the tasks */
  c1 = SD_task_create("c1", NULL, 50e9);   /* 50 s */
  c2 = SD_task_create("c2", NULL, 100e9);  /* 100 s */
  c3 = SD_task_create("c3", NULL, 200e9);  /* 200 s */
  c4 = SD_task_create("c4", NULL, 80E9);   /* 80 s */
  t1 = SD_task_create("t1", NULL, 1000);   /* 1 Kb */
  t2 = SD_task_create("t2", NULL, 10000);  /* 10 Kb */
  t3 = SD_task_create("t3", NULL, 200000); /* 200 Kb */
  t4 = SD_task_create("t4", NULL, 200000); /* 200 Kb */

  /* Set the dependencies */
  SD_task_dependency_add("c1-t1",NULL,c1,t1); SD_task_dependency_add("c1-t2",NULL,c2,t2);
  SD_task_dependency_add("t1-c2",NULL,t1,c2); SD_task_dependency_add("t2-c3",NULL,t2,c3);
  SD_task_dependency_add("c2-t3",NULL,c2,t3); SD_task_dependency_add("c3-t4",NULL,c3,t4);
  SD_task_dependency_add("t3-c4",NULL,t3,c4); SD_task_dependency_add("t4-c4",NULL,t4,c4);

  comp_array=(double*)malloc(sizeof(double)); comm_matrix=(double*)malloc(sizeof(double));
  host_list = (SD_workstation_t*) malloc (sizeof(SD_workstation_t));

  /* Schedule computations to hosts */
  host_list[0]=SD_workstation_get_by_name("p1"); comp_array[0] = SD_task_get_amount(c1);
  SD_task_schedule(c1,1,host_list,comp_array,NULL,-1);
  comp_array[0] = SD_task_get_amount(c2);
  SD_task_schedule(c2,1,host_list,comp_array,NULL,-1);
  host_list[0]=SD_workstation_get_by_name("p2"); comp_array[0] = SD_task_get_amount(c3);
  SD_task_schedule(c3,1,host_list,comp_array,NULL,-1);
  comp_array[0] = SD_task_get_amount(c4);
  SD_task_schedule(c4,1,host_list,comp_array,NULL,-1);

  /* Transfers t1,t4 within a single host */
  host_list[0]=SD_workstation_get_by_name("p1"); comm_matrix[0] = SD_task_get_amount(t1);
  SD_task_schedule(t1,1,host_list,NULL,comm_matrix,-1);
  host_list[0]=SD_workstation_get_by_name("p2"); comm_matrix[0] = SD_task_get_amount(t4);
  SD_task_schedule(t4,1,host_list,NULL,comm_matrix,-1);

  /* Transfers t2,t3 scheduled on the link */
  comm_matrix = (double*) realloc (comm_matrix, 4*sizeof(double));
  host_list = (SD_workstation_t*) realloc (host_list,2*sizeof(SD_workstation_t));
  host_list[0]=SD_workstation_get_by_name("p1");
  host_list[1]=SD_workstation_get_by_name("p2");

  comm_matrix[1] = SD_task_get_amount(t2);
  SD_task_schedule(t2,1,host_list,NULL,comm_matrix,-1);
  comm_matrix[1] = SD_task_get_amount(t3);
  SD_task_schedule(t3,1,host_list,NULL,comm_matrix,-1);

  tasks_done=SD_simulate(100.0); /* run for 100 seconds of virtual time */
  tasks_done=SD_simulate(-1);    /* run until the simulation completes */
  clock=SD_get_clock();

  free(comp_array); free(comm_matrix); free(host_list);
  SD_exit();  /* Free memory used by Simgrid */
  return 0;
}
```

Figure 2.7: SimDAG code sample.

The next step is to schedule the different tasks on the computing and network resources. With regard to the historical API, the fact that the request for developing SimDAG came from me, with my work on PTG scheduling in mind, had an unwanted yet important side effect. Indeed, tasks in SimDAG are parallel by default, and the default underlying model is the one designed for parallel tasks. More precisely, the parallel nature of a SimDAG task does not appear when it is created but only when it is scheduled. Before calling the `SD_task_schedule` function, a user has to fill the list of hosts to use, the array of computation amounts (Fig. 2.7, line 34), and/or the communication matrix (Fig. 2.7, line 44). This has to be done even though tasks are not parallel at all and to be executed on a single host as in this simple example. The designed API, tailored to suit my own needs, could thus be too complex and non natural for users aiming solely at studying DAG scheduling algorithms. Such SimDAG users do not want to deal with the extra burden of dealing with parallel tasks.

Before detailing how the SimDAG API has been extended to reintroduce the concepts of sequential task and point-to-point data transfers, let mention that the main simulation function (`SG_Simulate`) has been preserved. Its prototype has only be simplified as logging mechanisms have been moved to the XBT module. The original function also allowed the user to explicitly indicate upon the completion of which task the simulation should be suspended. This *watch point* mechanism that enables the implementation of dynamic scheduling algorithms has been automatized in SimDAG. The `SD_Simulate` functions stops each time a watched task completes.

In December 2009, typed tasks were introduced in the SimDAG API with the release 3.3.4 of SIMGRID. The objective was to remove the burden of managing parallel tasks when there are not needed. Two new task creation functions were added, that are actually wrappers on the `SD_task_create` function. The `SD_task_create_comp_seq` and `SD_task_create_comm_e2e` functions allow the user to create tasks that correspond to a sequential computation and a point-to-point data transfer respectively. The input parameters are exactly the same as for the generic task creation function.

Two functions were also added that considerably ease the scheduling of tasks on resources. Their use is currently limited to the aforementioned typed tasks but might be extended to parallel typed tasks. The main idea is to totally decouple the declaration of the amount of work to do or data to transfer between the creation and the scheduling of a task. For parallel tasks, the amount given as input of the creation function has to be distributed among the involved resources to simulate a parallel execution or a complex data redistribution. Then some intermediate functions allowing the user to create the computation array and/or the communication matrix will have to be added to the API so that the scheduling function has a semantic as simple as in the historical SIMGRID API. However, for sequential computations and point-to-point data transfers, this is straightforward. Indeed, there is no need to modify the arguments passed to the creation functions at scheduling time. Then such tasks can already benefit from the newly added `SD_schedulel` and `SD_schedulev` functions. Both functions require three parameters: the task to schedule, the number of workstations onto which execute the task, and the list of involved workstations. They only differ by the way the list of workstations is passed to the function. `SD_schedulev` accepts a vector of workstations (`const SD_workstation_t *`) as input, while the third parameter of `SD_schedulel` is a variable argument list.

An interesting side effect of these two functions is that they allow for *auto-scheduling*. Once the current task is scheduled, the function scans the lists of its predecessors and successors to inform them about the used set of resources. This information may allow some tasks to be automatically scheduled. For instance, we see in Figure 2.4 that a computation task is generally preceded and followed by data transfer tasks. In this example, when task $C2$ is scheduled, two data transfer tasks, $T1$ and $T3$, are candidate for auto-scheduling. If task $C1$ (resp. $C4$) has already been scheduled, all the necessary information, *i.e.,* source and destination of the point-to-point communication, is known and the transfer can be scheduled in turn. If task $C1$ (resp. $C4$) is not scheduled yet, the destination (resp. source) of data transfer task $T1$ (resp. $T3$) is set. When $C1$ (resp. $C4$), will be scheduled, $T1$ (resp. $T3$) is automatically scheduled. Another potential situation is that a control dependency exists between two computation tasks, *e.g.,* task Cy depends on task Cx. Scheduling task Cx with one of the new schedul-

ing function may automatically set off the actual scheduling of task Cy. This happens when Cx is the last predecessor Cy is waiting for.

```
1  #include "simdag/simdag.h"
2  int main(int argc, char **argv) {
3    SD_task_t c1,c2,c3,c4; /* Computations */
4    SD_task_t t1,t2,t3,t4; /* Data Transfers */
5    xbt_dynar_t tasks_done; /* dynamic array of completed tasks */
6    double clock;
7
8    SD_init(&argc, argv); /* SimDAG initialization */
9
10   SD_create_environment("platform.xml");
11
12   /* Create the tasks */
13   c1 = SD_task_create_comp_seq("c1",NULL, 50e9);   /* 50 s */
14   c2 = SD_task_create_comp_seq("c2",NULL, 100e9);  /* 100 s */
15   c3 = SD_task_create_comp_seq("c3",NULL, 200e9);  /* 200 s */
16   c4 = SD_task_create_comp_seq("c4",NULL, 80E9);   /* 80 s */
17
18   t1 = SD_task_create_comm_e2e("t1",NULL, 125);   /* 1 Kb */
19   t2 = SD_task_create_comm_e2e("t2",NULL, 1250);  /* 10 Kb */
20   t3 = SD_task_create_comm_e2e("t3",NULL, 2500); /* 200 Kb */
21   t4 = SD_task_create_comm_e2e("t4",NULL, 2500); /* 200 Kb */
22
23   /* Set the dependencies */
24   SD_task_dependency_add("c1-t1",NULL,c1,t1); SD_task_dependency_add("c1-t2",NULL,c2,t2);
25   SD_task_dependency_add("t1-c2",NULL,t1,c2); SD_task_dependency_add("t2-c3",NULL,t2,c3);
26   SD_task_dependency_add("c2-t3",NULL,c2,t3); SD_task_dependency_add("c3-t4",NULL,c3,t4);
27   SD_task_dependency_add("t3-c4",NULL,t3,c4); SD_task_dependency_add("t4-c4",NULL,t4,c4);
28
29   /* Schedule computations to hosts, transfers are auto-scheduled */
30   SD_task_schedulel(c1,1,SD_workstation_get_by_name("p1"));
31   SD_task_schedulel(c2,1,SD_workstation_get_by_name("p1"));
32   SD_task_schedulel(c3,1,SD_workstation_get_by_name("p2"));
33   SD_task_schedulel(c4,1,SD_workstation_get_by_name("p2"));
34
35   tasks_done=SD_simulate(100.0); /* run for 100 seconds of virtual time */
36   tasks_done=SD_simulate(-1); /* run until the simulation completes */
37   clock=SD_get_clock();
38
39   SD_exit(); /* Free memory used by Simgrid */
40   return 0;
41 }
```

Figure 2.8: SimDAG code sample using typed tasks and auto-scheduling features.

Figure 2.8 shows a code equivalent to those presented in Figures 2.6 and 2.7 but that uses the typed tasks (Fig. 2.8, lines 13-21) and auto-scheduling (Fig. 2.8, lines 30-33) features. It is worth noting that the data transfers are not explicitly scheduled anymore thanks to auto-scheduling. There is also no more need to manage a computation array and a communication matrix.

My last contributions with regard to SimDAG intend to help its use and adoption by new users of SIMGRID. I have written a tutorial to introduce the basic concepts underlying SimDAG [4] and developed a set of stock implementations of some popular DAG scheduling algorithms[5].

[4] http://simgrid.gforge.inria.fr/tutorials/simdag-101.pdf
[5] svn://scm.gforge.inria.fr/svn/simgrid/contrib/trunk/DAGSched

2.4 Simulation Input Generation

2.4.1 Platforms

When resorting to simulation to study parallel and distributed applications, every researcher is faced with the question: "which platform configurations should I simulate?" Depending on the research field in which simulation is planned, the answer to this question may not be the same. Indeed each research community expresses a different set of requirements that we identified in [QBS10], with L. Bobelin and M. Quinson. In what follows, we express these different requirements through simple use cases. This analysis was refined in [BLM+12], with the same authors and A. Legrand, D. Marquez, P. Navarro and C. Thiéry, to expose requirements related to network modeling raised by community specific use cases.

Networking In the networking community simulation is used to assess the behavior and the efficiency of algorithms and protocols at differing scales, ranging from local networks to the full Internet. The underlying platforms thus have to reflect the fundamental properties of the actual structure of the Internet, such as the presence of power-laws [66]. In this community a synthetic platform has to be fully connected and some qualitative information is needed. For instance, link bandwidths and network delays can have an impact on the studied protocols. However, this community shows only little interest in the description of the computing resources located at the edges of the network.

Large Scale Distributed Systems This domain covers the study and design of peer-to-peer algorithms and protocols. While the platforms needed by this community are as network-oriented as those required by the networking community, their characteristics are however different. Indeed the topology is here less important than the scale, typically hundreds of thousands entities. Hence the most important structural property is now the the geographic diversity of peers. Studies on P2P Distributed Hash Tables (DHTs) involves exchange of small messages. Contention can thus be somehow ignored as the measure of interest is generally the amount of exchanged messages. Conversely, P2P streaming uses large messages but the key characteristics are connection asymmetry and interference between concurrent connections on the borders of the network. A detailed modeling of the topology is then not important. In this context, network is commonly modeled according to the distance between entities. This distance can be represented by network latency while some recent studies show the impact of link bandwidth on peer-to-peer algorithm design [18]. Classical information on network links then has to be provided. Details on non-network resources may also be needed. For instance, a peer-to-peer storage application will base its strategies on the amount of available disk space of each host.

Volunteer Computing Initiated by the SETI@Home project and generalized with the BOINC framework [7], Volunteer Computing is now a common way to solve large scale computing problems. In this community, simulation is used to study fault-tolerance and scheduling to improve the reliability, fairness, and throughput of the computing platforms [60]. Volunteer computing studies imply the simulation of clients and/or servers. The key characteristics to account for are the volunteer dynamic availability, computing and storage properties, *e.g.,* CPU speed, number of cores, or disk space, and reliability. In some cases it may be important to model the characteristics of the connection of the peers to the Internet as well. Then the same requirements as for P2P streaming simulations apply.

Cluster computing Simulation is also used in High Performance Computing, *e.g.,* to compare batch scheduling algorithms [32, 100]. Here, the network topology has only little interest. Indeed batch schedulers usually manage only one or a few clusters. The platform descriptions are thus simpler. But this community also requires access to descriptions of real production clusters. Moreover adding flexibility to these descriptions would allow users to easily study the behavior of their algorithms on smaller, larger, or upgraded versions of the initial cluster. Then simulation can be used to replay some workload traces in different what-if scenarios, *e.g.,* with twice as much processors, with a high performance network interconnect, or to assess the impact of switching off some machines to save energy.

Grid Computing Grid resources are often interconnected either through a private network or National Research and Education Networks (NRENs). This is the case of production infrastructures such as the European Grid Infrastructure (EGI)[6] which relies on the pan-European GÉANT network that interconnects European NRENs. Research grids such as the French Grid'5000 initiative[7] also have their own private network. This leads to less complex network topologies than what can be found on the Internet. The compute resources typically deployed in Grids are commodity clusters.

There is a strong need for resource descriptions in this research field. Indeed many scheduling algorithms map jobs thanks to match-making techniques. Various properties, such as the operating system, the available memory and disk space, installed libraries, are thus part of the platform description. While such properties are usually not handled by simulation toolkits they have, however, to be described.

Another important requirement in the domain of grid computing is to run simulations on an experimental environment as close as possible to reality. Ideally, descriptions of deployed grid infrastructures should be available. In a production context, this would help application developers to prepare an experiment campaign in a controlled and realistic setting. As for cluster computing, performance assessments can be done by introducing some variations in the experimental environment.

Cloud computing Modeling an IaaS Cloud in terms of network requires require a mixture of the previous requirements. For instance, it may be important to precisely model what happens within data centers. To reflect a hierarchical organization, high-end compute nodes are distinguished from low-end compute nodes and storage nodes. When the infrastructure comprises several sites, the wide area network connection between sites has also to be accounted for. Then it is possible to study the different charging mechanisms involved when going from one site to another. While a precise modeling of client connectivity may not be required, it is important to consider their geographic diversity at least. Moreover the accurate description of the resources and services available on each site is mandatory.

Community	Desired Topology	Computing Resources	Network Resources	Properties
Networking	Similar to Internet	(none)	Latency	LAN/WAN
Large Scale Distributed Computing	Large scale (not realistic)	Single nodes	Latency Bandwidth	Disk
Volunteer Computing	Similar to Internet	Single hosts	Latency Bandwidth	Computing power
Cluster Computing	Simple (flat)	Cluster	Latency Bandwidth	Computing power
Grid Computing	Simple (hierarchical)	Cluster	Latency Bandwidth	Power Services
Cloud Computing	Similar to Internet	Cluster	Latency Bandwidth	Power, Storage Services

Table 2.1: Requirements for synthetic experimental environments per research community.

Table 2.1 summarizes the requirements in terms of network topology, type of computing and network resources, and additional properties, expressed by the different research communities.

A first approach to synthesize such experimental environments, is to generate random platform configurations. In my early works, I applied this approach to the specific fields of cluster and grid computing. The random generation was then about the number of clusters, the number of nodes per cluster, or the nodes' compute speeds. Simple uniform probability distributions were used. Then I

[6] http://www.egi.eu/
[7] http://www.grid5000.fr/

happened to develop a simple generator of such platform configuration producing very compact files in a specific text format. The outputs of this generator were used in [CDS04, NS06, NSC07, NS07, HRS08b]. While this approach was simple and made it possible to generate large numbers of platform configurations it was not clear that many of the generated platforms were representative of the real world. Moreover the proposed description format was not compliant with the one used by SIMGRID and required to bypass the parser provided by the toolkit.

More complex synthesizing tools have been designed over the last decade. Tiers [30] follows a top-down approach to generate N-Level topological graphs. It starts from a connected graph and replace a node by another connected graph at each step. Tiers also adds a semantic to the edges of the graph to distinguish local networks from metropolitan or wide-area networks. Unfortunately, this semantic is not exploited to derive network link latency and bandwidth values. Moreover Tiers does not consider non-network resources. The Boston university Representative Internet Topology gEnerator (BRITE) [113] provides a unified framework for the generation of network topologies with particular emphasis on topologies reflecting the structural properties of the Internet, *e.g.,* hierarchical structure and degree distribution. As in Tiers, the computing resources located at the edges of the network are ignored by BRITE. However it allows users to label communication links with bandwidth values. While these two tools focus on network topologies, some work exists about the generation of synthetic computing resources. From the observation that grids are principally made of clusters, a commodity cluster synthesizer has been proposed in [99]. The authors examined 114 production clusters comprising more than 10,000 processors in terms of processor architecture, clock frequency, cache size, number of processors, network interconnect, disk capacity, or release date. They came up with statistical models to allow users to extrapolate for future configurations. By contrast with Tiers and BRITE that generate network topologies with no computing resources, this commodity cluster synthesizer can produce multiple cluster-like resources, but does not interconnect them. Furthermore this synthesizer does not include information such as computing power while it is a fundamental information for simulation kernels. The GridG [110] project is a computational grid synthesizer. It relies on structured topologies obtained with Tiers. The routers, hosts, and links of the produced topologies are then annotated with attributes such as memory size, number of computing units, disk size, or bandwidth. The GridG annotation mechanism also supports user-supplied conditional probability rules to define correlations among the attributes. A main drawback of GridG is the limitation to hierarchical network topology supported by Tiers. This prevent the users of GridG to test their algorithms or protocols on other network topologies.

In conclusion no existing execution environment synthesizer is able to cover all the requirements expressed by the different research communities. Each of them is indeed specific to a given community while simulation frameworks such as SIMGRID are now able to span across several communities.

An alternate approach to produce synthetic environments for simulation purposes is to opt for using real-world platform configurations directly. For instance, in [Sut07, HRS08a, NS09, DNSC09, DS10, CDS10, CDS10], descriptions of Grid'5000 clusters were used. Such descriptions based on the actual hardware configuration and performance of the clusters are obviously more realistic than randomly generated cluster configurations. However, describing a cluster in the XML format of SIMGRID was cumbersome and verbose before the release of SIMGRID 3.3. In the previous versions of the toolkit, describing a cluster implied to separately declare each node of the cluster. Then all the network links connecting a node to the interconnection switch also had to be declared. Finally, all the routes detailing the complete set of network links a communication has to go through from one node to another were also described. For instance, the following XML file describes a homogeneous compute cluster made of one hundred identical nodes, or *hosts*, interconnected through a single switch. It is easy to see the verbosity of such a declaration of a hundred hosts, followed by a hundred network links, and worst of all, by 4,550 routes between pairs of hosts. Hopefully, it is assumed that routes are symmetrical so that half of the declarations can be saved.

Cluster description prior to SimGrid 3.3

```
<platform version="1">
 <host id="bob-1.hamburger.edu" power="1E9"/>
 <host id="bob-2.hamburger.edu" power="1E9"/>
                      [...]
 <host id="bob-100.hamburger.edu" power="1E9"/>

 <link id="link_bob-1" bandwidth="1.25E8" latency="5E-5"/>
 <link id="link_bob-2" bandwidth="1.25E8" latency="5E-5"/>
                      [...]
 <link id="link_bob-100" bandwidth="1.25E8" latency="5E-5"/>

 <link id="BobSwitch" bandwidth="1.25E8" latency="5E-5" sharing_policy="FATPIPE"/>

 <route src="bob-1.hamburger.edu" dst="bob-2.hamburger.edu">
   <link:ctn id="link_bob-1"/>
   <link:ctn id="BobSwitch"/>
   <link:ctn id="link_bob-2"/>
 </route>
 <route src="bob-1.hamburger.edu" dst="bob-3.hamburger.edu">
   <link:ctn id="link_bob-1"/>
   <link:ctn id="BobSwitch"/>
   <link:ctn id="link_bob-3"/>
 </route>
                      [...]
 <route src="bob-99.hamburger.edu" dst="bob-100.hamburger.edu">
   <link:ctn id="link_bob-99)"/>
   <link:ctn id="BobSwitch"/>
   <link:ctn id="link_bob-100"/>
 </route>
</platform>
```

This format being obviously not scalable at all, we proposed, with M.-E. Frincu and M. Quinson, a new description format for SimGrid in [FQS08]. The main objective was to dramatically reduce the size of the XML files thanks to the factoring of all the redundant information. As a side effect, the parsing time of these description files was also reduced. To this end, we proposed a new XML tag to declare a homogeneous cluster with a switched network interconnect in a more concise way. In the description format used by SimGrid from its 3.3 version, the previous example can be written as follows.

Cluster description from SimGrid 3.3

```
<platform version="2">
 <cluster id="bob-cluster" prefix="bob-" suffix=".hamburger.edu"
          radical="1-100" power="1E9" bw="1.25E8" lat="5E-5"
          bb_bw="1.25E8" bb_lat="5E-5"/>
</platform>
```

The optimizations proposed in [FQS08] allowed SimGrid's users to describe large computing platforms and parse these descriptions in a reasonable time. For instance, with the Document Type Definition (DTD) of SimGrid 3.2 the description of the whole Grid'5000 platform (that comprised 1,300 nodes scattered across 23 clusters at that time) required 425 MiB of disk space and was impossible to parse. With the DTD of SimGrid 3.3, the same platform was fully described in a file of 78.8 KiB and was parsed in 31 seconds. This description format has since been improved, with the extra help of L. Bobelin, A. Legrand, P. Navarro, and D. Marquez. The way the routing was expressed has been simplified and corrected. But the compactness and expressiveness of the existing format was preserved.

The main contribution in [BLM+12] was to allow for the description of a synthetic platform as a hierarchical aggregation of networks. On the Internet, most aggregated networks are named Autonomous Systems (ASs), as they behave independently from each other and may have very different structures. Each AS is connected with lower or higher level ASs by a set of entry points. This hierarchy is often bypassed by direct connections between ASs in the same level. Within an AS, a routing policy is applied, most of the time based on shortest paths algorithms such as Open Shortest Path First (OSPF) and Routing Information Protocol (RIP). In our proposal, we opt for static routing. From a performance point of view, higher hierarchy sub-networks may use traffic aggregation and dynamic routing to perform load balancing. However studies have shown that no change may occur for 80% of the paths in a 24 hour period [28]. Moreover, such changes may especially affect load balancing on backbone links, that are usually not bottlenecks. Then these changes can be ignored without any significant impact on simulation accuracy. SIMGRID comes with stock implementations of the Dijkstra (with or without cache) and Floyd routing algorithms, both described in [51]. A classical flat representation, in which each route is completely defined by the set of equipments belonging to it, is also implemented. Finally we propose a rule-based routing model that relies on regular expressions to exploit regular structures. Figure 2.9 shows an example of such a hierarchical network representation.

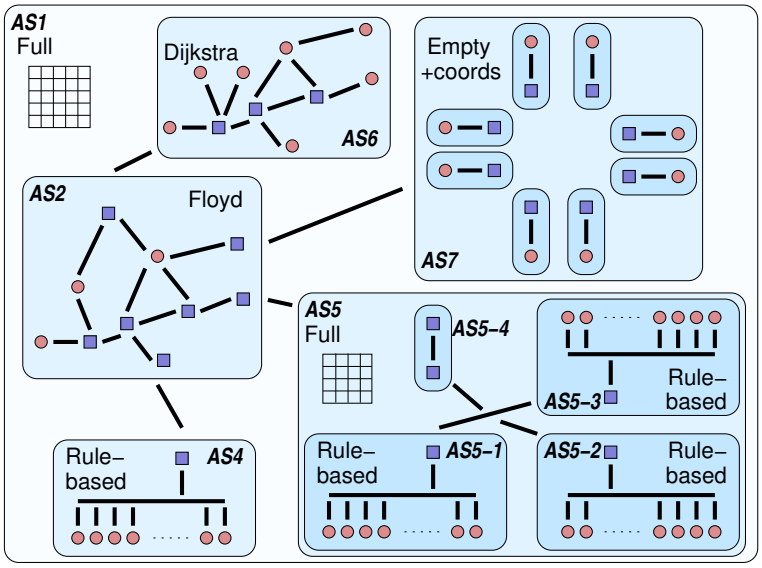


Figure 2.9: Illustration of hierarchical network representation. Circles represent processing units and squares represent network routers. Bold lines represent communication links. AS2 models the core of a national network interconnecting a small flat cluster (AS4) and a larger hierarchical cluster (AS5), a subset of a LAN (AS6), and a set of peers scattered around the world (AS7).

Each AS has one or more gateways, which are used to compute routes between ASs included in an AS of higher level. With this mechanism, the simulator can determine routes between hosts belonging to different ASs by looking for the first common ancestor in the hierarchy (see Figure 2.10). Routing is then solved recursively using the hierarchy. It allows us to represent hierarchical platforms in a very compact and effective way. However, as real platforms are not strictly hierarchical, we also define bypassing rules to manually declare alternate routes between ASs.

This extension of SIMGRID's description format also includes the definition of a peer tag. This tag allows users to easily create P2P overlays by defining at the same time a host and a connection to the rest of the world. With such a tag we benefit both from the compactness of coordinate-based models that

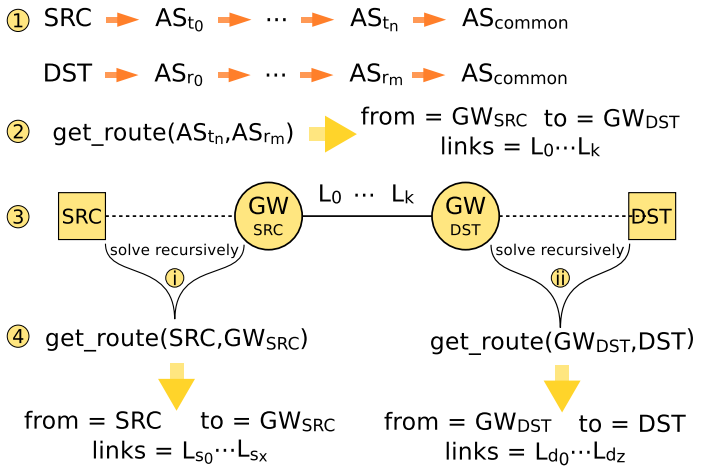


Figure 2.10: Main steps of the hierarchical routing mechanism.

account for delay heterogeneity and correlation, and from the accuracy of fluid models for contention. The following XML file describes a simple AS that comprises two peers. Each peer is described by its network coordinates, its processing power, and the characteristics of the network link that connects it the rest of the platform, *i.e.*, latency, upload and download bandwidths. The routing protocol used to exchange messages between these peers is Vivaldi [50].

```
<platform version="3">
  <AS id="AS0" routing="Vivaldi">
    <peer id="100030591" coordinates="25.5 9.4 1.4" power="1.5E9"
          lat="5E-4" bw_in="2.25E9" bw_out="2.25E9"/>

    <peer id="100036570" coordinates="-12.7 -9.9 2.1" power="7.3E8"
          lat="5E-4" bw_in="2.25E9" bw_out="2.25E9"/>
 </AS>
</platform>
```

A drawback of using descriptions of real-world platform configurations is that typically only a few such configurations are constructed. Then they do not necessarily cover a wide range of properties which, in turn, limits the representativeness of the described platforms. However, considering subsets of a large-scale multi-cluster platform multiplies the number of possible configurations. In [SC07], we applied this idea to the Grid'5000 platform to produce a compendium of configurations with specific characteristics, to be used for simulation experiments. Figure 2.11 shows the range of processor speeds in the Grid'5000 platform (on the x-axis) for the cores of the 18 clusters on which the HPLinPack benchmark was run in 2007. We see that the platform was fairly heterogeneous at this time, with the fastest processors computing about 45% faster than the slowest processors.

From this set of compute clusters we defined collections of subsets, in a view to span a sound spectrum of characteristics. One important characteristic is the degree of processor heterogeneity in the platform. We define the degree of heterogeneity, h, as $100 \times (s_{max}/s_{min} - 1)$, where s_{max} (resp. s_{min}) is the maximum (resp. minimum) processor speed (in billions of floating operations per second) in the platform. A zero value indicates a perfectly homogeneous system. Given the eighteen clusters in Grid'5000, no multi-cluster platform is fully homogeneous. We thus consider what we term "almost homogeneous" platforms, that is multi-cluster configurations with a low h value ($h < 10$). We also use this

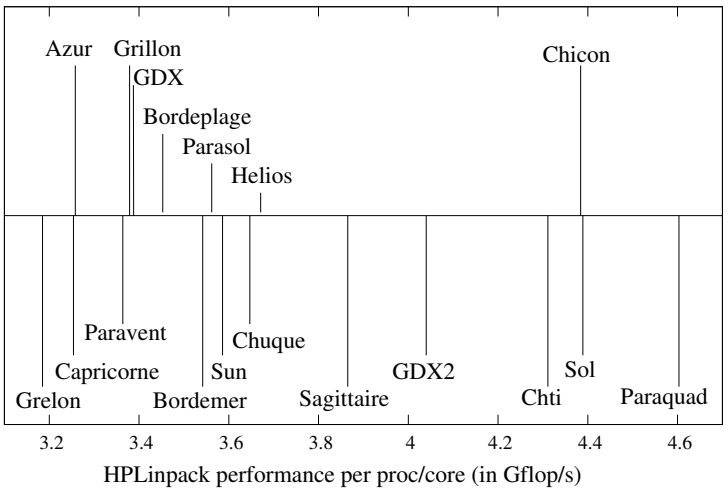


Figure 2.11: Grid'5000 Processor Speeds (in GFlop/s) in 2007.

definition when deriving heterogeneous two-cluster platforms. For configurations that comprise more than two clusters, or in other words platforms that consist of some "fast" clusters and of some "slow" clusters, we define h_{min} and h_{max} as follows. h_{min} is the heterogeneity factor between the fastest of the slow clusters and the slowest of the fast clusters. h_{max} is simply defined as the heterogeneity factor between the slowest cluster and the fastest cluster. Also, we limit the number of platform configurations whenever applicable. For instance, given the clusters in Grid'5000, when trying to generate configurations with one fast cluster and three slow clusters, we could end up choosing the three slow clusters among seven possibilities, for a total of 210 platform configurations, with many of these configurations virtually identical. Therefore, we choose to ignore many of these possibilities in order to keep the number of platform configurations reasonably low. From these definitions and the set of clusters listed in Figure 2.11, we list 356 individual multi-cluster platform configurations. These configurations contain one, two, four, or eight clusters, and can be categorized as homogeneous or heterogeneous, as summarized in Table 2.2.

Type	1 cluster	2 clusters	4 clusters	8 clusters	total
Homogeneous	18	10	10	10	48
Heterogeneous	-	82	133	93	308
Total	18	92	143	103	**356**

Table 2.2: Summary of platform configurations extracted from Grid'5000.

The compendium proposed in [SC07] was a first attempt to enlarge the set of realistic platform configurations available for simulation experiments, mainly in the scheduling field. It was also aiming at quantifying the diversity of the simulated platforms to avoid unforeseen biases in the selected set. Indeed a typical issue related to randomly generated platforms is that while the generated set can be large, it can also contain a lot of elements with redundant characteristics. However, the limitations of this first attempt are numerous. First it focuses on the Grid'5000 platform only, while it could be applied to any large scale multi-cluster infrastructure. Second, the subset selection has been done manually which is a very tedious process. Finally the selection criteria were totally subjective and driven by the targeted scheduling studies. Then the produced set of platform configurations may not reflect the

concerns of other users potentially coming from other research communities. For instance, the main selection criterion is the processing power heterogeneity. The network connectivity of the clusters is not considered. But some user may prefer to simulate a configuration in which all the clusters are located in a same site, while another may want to simulate a highly geographically distributed multi-cluster platform, both regardless of the computing heterogeneity.

In the direct continuation of these efforts to describe and generate sound and diverse platform configurations, we described in [QBS10] the principles of the Simulation pLAtform CReation and User-guided Modification) (SIMULACRUM) tool. SIMULACRUM is a generic synthesizer that aims at covering all the requirements expressed by several research communities given in Table 2.1. It also combines random generation and description of real-world platform configurations. Finally the main design goal of SIMULACRUM is to allow its users to define and control how the synthetic platforms have to be generated at every step of the flowchart depicted in Figure 2.12.

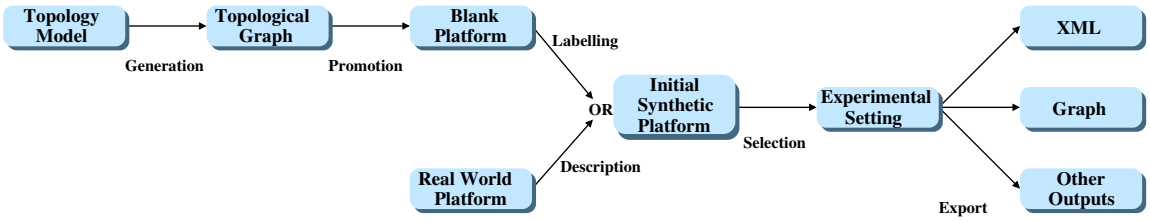


Figure 2.12: Generation flowchart of the SIMULACRUM tool.

This flowchart has two entry points. Descriptions, in the XML format proposed in [FQS08], of existing grid infrastructures, such as the Grid'5000 and DAS-3[8] platforms, are provided, as needed by cluster and grid computing communities. In what follows we detail the different steps of the longest path of the generation flowchart. This process mainly concerns the networking, large scale distributed systems, volunteer computing, and cloud computing research communities.

Topological Graph Generation To create a topological graph, SIMULACRUM relies on several models. Classical topologies such as ring, star, or clique are of course available. Moreover SIMULACRUM implements models that spread the nodes over a unit-square area and connect two nodes u and v with a probability $P(u, v)$ following different distributions such as uniform, exponential, Waxman [153], and Zegura [155]. The topologies produced by these models are flat in opposition to those produced by hierarchical generators such as Tiers. Finally, SIMULACRUM provides another class of interconnection generators encompassing degree-based models such as the one proposed by Baràbasi and Albert in [16]. This model, based on incremental growth of the platform and affinity connexion, is known to better follow power-laws [66].

At the end of this first step, SIMULACRUM manages a connected topological graph composed of abstract nodes without any particular type. The edges of this graph only represent the fact that two abstract nodes are connected or not. This graph just gives the structure of the experimental environment. The next two steps add qualitative information to these abstract entities.

Node Promotion The second step consists in converting the abstract nodes of the topological graph into computing (hosts and clusters) and networking (routers) resources. The difficulty here is to express complex decision-making processes such as "change one half of the graph leaves into low-cost desktop machines and the second half into small clusters; nodes with medium degree should be changed into powerful computational servers; nodes with high degree should remain routers".

[8] http://www.cs.vu.nl/das3/

Our approach is to define a chain of *promoters*, which describes the transformation of the topology graph into an interconnection of resources. A promoter is a decision-making rule encompassing a filter and a generator. Only the nodes of the topological graph are concerned by these promoters. At this stage, edges still only express if two resources are connected or not.

For each promotion rule, several filtering patterns are available. They can be combined in a logical AND manner to express several properties to respect. Some filters depend on the properties of the node, *e.g.,* its degree, while others depend on the targeted platform. For instance, a filter may be applied while the number of computing resources is under a certain threshold.

If a node gets caught in the filter of a promotion rule, it is promoted to the corresponding resource type. A node can be changed into a single host, *i.e.,* a desktop computer characterized by its compute speed, or a homogeneous cluster, *i.e.,* multiple hosts interconnected through a local area network. For both types of promotions, the characteristics of the target resource can be fixed by the user or picked uniformly within an interval. The nodes that are not selected by any filter become routers.

The promoters are considered in order for each node. The first promotion rule whose filter catches a node is applied and the subsequent promoters are skipped for this node. The decision-making process introduced above informally corresponds to the following chain of promoters.

Promoter 0: AND(is leaf, probability 0.5) $\Rightarrow$ small desktop
Promoter 1: node is leaf $\Rightarrow$ small cluster
Promoter 2: $degree \in [2, 5] \Rightarrow$ powerful server (other nodes remains routers)

It is also often useful to add arbitrary properties to the promoted nodes (represented as $key \times value$ couples). A list of *property adders* is then associated to each promoter. Each adder associates a given property to each node generated by its promoter. The value can be a string, or a numerical value picked uniformly in an interval. For instance, this allows for the description of services and data storage components that are typical in Cloud Computing.

Edge Labeling Once each node has been promoted into its final type, communication properties, *i.e.,* latency and bandwidth, still have to be associated to the edges of the topological graph. SIMULACRUM relies on the same promotion mechanism as for the nodes by using a chain of *edge labelers*. These rules are also applied in order, in an exclusive manner, and properties adders can be associated to labelers.

The available filters act on the length of the edges. The length of an edge is defined as the Euclidean distance between the nodes it interconnects on the unit-square area. When an edge is caught in a filter, it becomes a communication link that is labeled with a latency, a bandwidth and a sharing policy. The values of the first two labels can either be fixed by the user or uniformly picked within an interval. The sharing policy models whether or not a communication link will suffer from contention. At the end of this step, SIMULACRUM handles a generated synthetic platform described in the same way as real-world computing grids. From this point there may be a need for creating subsets of this original platform, depending on the research community.

Subset Selection SIMULACRUM provides two ways to select a subset from a given platform, be it generated or a description of an existing computing grid. First, a user can manually discard some of the hosts or clusters. For each cluster it is also possible to modify the number of hosts. Note that this modification can decrease or increase the number of computing resources and alter the properties of the generation model. During the selection of resources, the user is notified of the evolution of the characteristics of his/her experimental environment. This way the user can, for instance, continue to modify the platform until the desired resource heterogeneity is reached.

To obtain all the subsets of a platform that satisfy certain properties, SIMULACRUM also provides an automatic selection mechanism. This interactive process allows the user to express the different properties that a subset must meet as a chain of *filters*. At the end of the selection process, the list of all the subsets that passed through all filters is displayed.

Several filters are available. Some of them consider the structural properties of a subset, *i.e.,* the number of nodes in the topological graph, the number of hosts or clusters, or the diameter of the graph. Another class of filters allows the user to characterize the statistical distribution of the compute speed within a subset[9]. Note that the compute speed absolute value is less relevant than the ratio between the highest and lowest compute speeds to determine the heterogeneity of a given subset. Then we compute the statistical moments over the logarithm of the compute speed.

The *distribution support* filter fixes the interval within which the compute speed of each node of the subset must lie. The *average* filter ensures that the compute speed average remains within the given interval. The *variance* and *standard deviation* filters help to control whether the compute speeds are concentrated around the average or evenly distributed between the extrema. The *skewness* filter corresponds to the third standardized moment. It measures the asymmetry of the probability distribution. A negative value indicates that the mass of the distribution is concentrated on large values with few small values. The average is then bigger than the median. A positive value indicates the contrary. Finally the *kurtosis* filter corresponds to the fourth standardized moment, which measures the "peakedness" of the distribution. A high kurtosis means that most of the variance is due to infrequent extreme deviations, while in flatter distributions the variance comes from frequent but modestly-sized deviations.

After this last step, the user disposes of a completely defined platform configuration. its description is ready to be used as part of a simulation scenario. Table 2.1 summarized the requirements of each research community. Table 2.3 exemplifies some SIMULACRUM settings that correspond to these needs. For example, since Grid and Cloud researchers are mainly interested in interconnections of clusters, they should promote nodes into clusters (with either fixed or uniformly picked capacities.) Moreover the interconnection between clusters is of little interest in Grid community. A classical generator such as Waxman or Zegura is then sufficient. To mimic a Cloud in which the clusters are connected directly to the Internet, the Baràbasi-Albert topology generator will be preferred. In both cases, selecting subsets of existing platforms based on the computational power distribution is also an interesting approach. By contrast, large scale distributed platforms should probably be generated using any method and then filtered on graph metrics such as the diameter.

Community	Topology Model	Real world description	Node Promotion	Labelling	Properties
Networking	Baràbasi Albert	Internet-like	Fixed host	Latency from distance	WAN/LAN
Large Scale Distributed Computing	Waxman Zegura	Large scale	Fixed host	Fixed characteristics	None
Volunteer Computing	Baràbasi Albert	Internet-like	Uniform host	Lat & bw from distance	Disk
Cluster Computing	Only one cluster	Flat tree	Uniform cluster	None	Disk
Grid Computing	Waxman Zegura	Hierarchical	Uniform clusters	Lat & bw from distance	Disk, services
Cloud Computing	Baràbasi Albert	Internet-like	Uniform clusters	Lat & bw from distance	Disk, services

Table 2.3: Examples of SIMULACRUM settings fulfilling the needs of each research community.

Export The final step of the generation process allows users to make some final adjustments. SIMULACRUM exports an XML representation of the produced platform configuration or graphically dis-

[9] http://en.wikipedia.org/wiki/Descriptive_statistics

plays the platform using the classical `dot` tool. This representation is based on the description format of SIMGRID. It can be edited at will by the user within SIMULACRUM. Each modification of the XML is reflected in the graphical view. Once the user is satisfied with the produced XML description, s/he can save the corresponding file. This file can be directly used as input of any SIMGRID simulator.

To summarize, SIMULACRUM is a tool that produces, in interaction with the user, generic synthetic platform descriptions. It combines models and approaches found in existing tools, such as BRITE or GridG, to original features such as the definition of arbitrary properties. Contrary to other existing synthesizers, SIMULACRUM is not limited to a specific research community. Its modular generation process allows SIMULACRUM to be parametrized to fulfill the specific requirements of any community. A very interesting feature of SIMULACRUM is its ability to select subsets of existing platforms based on user-defined filters. It allows for a double-check of the characteristics of the selected platforms. For instance, it could help to understand the performance variations observed during an experiment in light of the inherent characteristics of the experimental settings.

To evaluate the benefit of such a tool over a compendium such as the one presented in [SC07], we investigated if it was possible to extract the same subsets of the Grid'5000 platform. To this end, we searched for subsets of the platform that comprise exactly 8 clusters and so that the heterogeneity degree is no larger than 1.1. It took SIMULACRUM 20 seconds to identify eleven such subsets out of the 4,194,201 possible configurations. For comparison purposes, in [SC07], only ten such subsets were listed. Moreover, it took much more time to determine all the interesting subsets manually.

2.4.2 Task Graphs

The second main input component of a simulation, with regard to Figure 2.1, on which I worked is the *application workload*. This component can have various meanings depending on the conducted simulation study. In the context of my work described in Chapter 1, application workload corresponds to the generation of application task graphs. These task graphs can be composed of sequential computation tasks to form DAGs or moldable tasks to represent PTGs. As for platforms, researchers are faced to the choice between a large set of synthetic application graphs or a smaller set of descriptions of actual parallel applications to assess the performance of their scheduling algorithms. Both have their pros and cons that are similar to those outlined for platforms.

On the one hand, task graphs describing actual parallel applications, such as scientific workflows, give some credit to the applicability of the studied scheduling algorithms. Indeed they are confronted to concrete execution scenarios and may lead to performance improvements in production systems. However, the good performance of an algorithm on a limited set of carefully chosen applications may hide more common or bad performance in more general settings. On the other hand, resorting to synthetic application task graphs allow to cover a broader range of application characteristics and solve this issue of evaluation the overall performance of algorithms. However, other potential drawbacks may come with the generation of large sets of synthetic task graphs.

The first and most obvious drawback of synthetic task graphs is their potential lack of realism. It is important to correlate the generation parameters with the analysis of the characteristics of existing workloads. This would prevent the production of a set of graphs that would never exist in real settings. To illustrate this issue, we can refer to the study conducted in [120]. The authors analyzed the workflow applications that run on the Austrian Grid platform. They found that most of these task graphs comprise around 30 tasks, 75% of them have fewer than 40 tasks, and only 5% of them have large numbers of tasks higher than 200. A sound generator should of course produce task graphs of any size to test the behavior of an algorithm in every conditions. This would allow the algorithm designer to tackle corner case issues and improve the algorithm. However, this study indicates that the distribution used to determine the number of tasks in a graph should not be uniform, but skewed around 30-40 nodes. Then it would better reflect the characteristics that raise from actual production workloads.

A second potential pitfall lies in the adequacy of the parameters of the task graph synthesizer to the general behavior of the studied algorithms. For instance, the performance of many DAG scheduling algorithms depends on the length of the critical path of the task graph. To validate such algorithms, a synthesizer should have to generate graphs with a large variety of the critical path length. A too narrow range of values would hinder the quality of the performance study. Moreover, synthesizers usually accept a lot of parameters as input. The Cartesian product of the number of values taken by each parameter usually leads to very large sets of randomly generated graphs. The algorithm designer (and most likely the reader of a validation study) using such large sets may confidently think that the more scenarios are tested, the sounder the validation is. However, it may often be only an illusion of validity. Some generation parameters may have no effect at all on the performance of the studied algorithms while other may be redundant. Many combinations may indeed lead to a very similar behavior. The consequence is that what should be the core of the performance analysis is diluted by these useless scenarios. It may lead to too flattering results for some algorithms by hiding very bad performance on a few set of important task graphs by good or even average performance on a lot of very similar graphs. Conversely, an algorithm may exhibit poor performance just because the set of graphs for which it performs well is largely outweighed by a large set of redundant unfavorable scenarios.

Detecting which properties of a task graph have an effect or not over the performance of the studied algorithms is a difficult problem. It adds up to the highly complex problem of characterizing and generating a truly representative synthetic workload, which is well introduced in [67]. Then, most validation studies in the field of DAG and PTG scheduling rely on a few graphs representing actual applications or larger sets of synthetic graphs produced by some home-made generators. Combinations of graphs from these two categories can be used to compensate some of their respective drawbacks.

Many projects are part of the effort to share input workloads for DAG scheduling studies. The Standard Task Graph Set [146][10] has been proposed as a benchmark suite for the evaluation of multiprocessor scheduling algorithms. It comprises 2,700 random graphs, whose size varies from 50 to 5,000 tasks, generated according to various methods. It also includes graphs representing actual applications, such as robot control or sparse matrix solver. The graphs in this set model either communications costs between compute tasks or not. They are described in a simple and comprehensive text format. An interesting thing about this project is that optimal schedule lengths are provided for all the graphs in the set. This allows developers to compare the performance of their algorithms to these optimal values and make fair and objective comparisons. The Task Graphs for Free (TGFF) tool [56] was also specifically designed for scheduling simulation. It produces Series-Parallel or Fan-in/Fan-out, *i.e.*, that mimic scatter/gather phases in a parallel application, task graphs. TGFF also comes with a built-in model to generate communication costs and deadlines. Unfortunately there is no way to control the random distribution of the attributes generated by TGFF. The GGen project [47] is not only a random graph generator, but also a graph analyzer. Its main aim is at providing users insights on the properties, *e.g.*, Minimum Spanning Tree, Max Independent Set, in/out degrees of the nodes, of the generated graphs and thus create proper and representative validation suites. GGen provides several generation methods that cover what is typically used in the scheduling literature. Graphs are exported in the Graphviz's DOT language that allows users to load the graphs with most of the available analysis tools. Finally, the developers of the Pegasus workflow management system [54] made available a generator of instances of the most popular data-driven applications managed by their system[11]. For each application, the general shape of the graph is fixed. The generator then acts on the size of the instance, that is the number of tasks. The tasks and data transfers that compose a graph are instantiated with costs gathered from actual executions on a computing grid. They proposed their own XML format, called DAX, to described these task graphs. A DAX file is decomposed in two parts. First the *jobs*, *i.e.*, compute tasks that compose the application, are described by a unique id, the name of the executable to run and the

[10] `http://www.kasahara.elec.waseda.ac.jp/schedule/`

[11] `https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator`

list of input and output files. Data-dependencies between tasks are derived from these lists of files. A task B depends on another task A if one of B's input file is an output of task A. The second part of a DAX file explicitly lists control dependencies that may exist between tasks.

Unfortunately, none of the aforementioned tools was designed to conduct PTG scheduling simulation studies. As it was mentioned in Section 1.2.1, a model of moldable task is required in addition to a model of DAG to obtain a PTG. Then I developed my own random PTG generator, called *daggen*, and made it publicly available[12]. I used this generator in my different publications on PTG scheduling [NS06, NSC07, Sut07, NS07, HRS08a, NS09, DNSC09, CDS10, CDS10, DS10]. It was also used in several publications by other authors with various application domains [2, 9, 22, 143, 152].

This generator produces PTGs described as a set of nodes in an intuitive text format. Each line lists information about a node, namely its index, list of successors, type, amount of work to execute (in flops) or data to transfer (in bytes), and, in our specific context, a parallelization overhead. As said in the previous chapter, moldable tasks are modeled according to the Amdahl's law. This last parameter then corresponds to the α parameter in Equation 1.1 and takes its value in $[0; 1]$. Note that setting α to 0 for every task allows a user of *daggen* to generate DAGs instead of PTGs. Nodes can have one of the four following types. `ROOT` is a unique artificial entry node with zero cost, with no predecessor, and with the entry node(s) of the PTG as successor(s). Similarly, `END` is a unique artificial exit node with zero cost, with no successor, and with the exit node(s) of the PTG as predecessor(s). Then the other nodes are either `COMPUTATION` or `TRANSFER` tasks. One restriction exists for `TRANSFER` nodes. As they represent point-to-point communications, such nodes can have only one successor. Figure 2.13 shows an example of PTG produced by *daggen* as a text file (left) and more graphically (right).

```
1  NODE_COUNT 17
2  NODE 0 1,2 ROOT 0.0 0.0
3  NODE 1 3,4,5 COMPUTATION 15000000000 0.20
4  NODE 2 6,7,8 COMPUTATION 24000000000 0.20
5  NODE 3 9 TRANSFER 100000000 0.0
6  NODE 4 10 TRANSFER 100000000 0.0
7  NODE 5 11 TRANSFER 100000000 0.0
8  NODE 6 9 TRANSFER 100000000 0.0
9  NODE 7 10 TRANSFER 100000000 0.0
10 NODE 8 11 TRANSFER 100000000 0.0
11 NODE 9 12 COMPUTATION 10000000000 0.20
12 NODE 10 13 COMPUTATION 6000000000 0.20
13 NODE 11 14 COMPUTATION 3000000000 0.20
14 NODE 12 15 TRANSFER 100000000 0.0
15 NODE 13 15 TRANSFER 100000000 0.0
16 NODE 14 15 TRANSFER 100000000 0.0
17 NODE 15 16 COMPUTATION 15000000000 0.20
18 NODE 16 - END 0.0 0.0
```

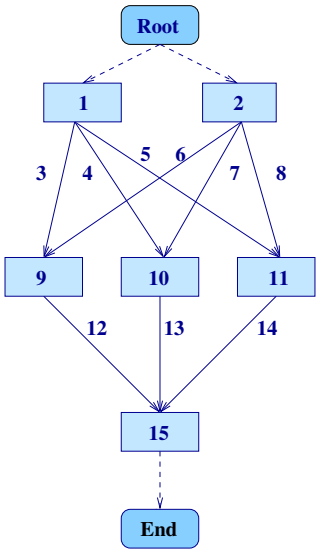


Figure 2.13: Example of a PTG produced by *daggen* as a text file (left) and grahically (right).

Here we recall what was introduced in Section 1.3 about the instantiation used in our studies. We generally assumed that a task operates on a dataset of d double precision elements (for instance a $\sqrt{d} \times \sqrt{d}$ square matrix). We arbitrarily assumed that processors have at most 1GByte of memory and thus $d \leq 121M$. We also assumed that d is above $4M$ (if d is too small, the moldable task should most likely be fused with its predecessor or successor). We modeled the *computational complexity* of a task, in

[12] `https://github.com/frs69wq/daggen`

number of operations, with one of the three following expressions, which are representative of common applications: $a \cdot d$ (*e.g.*, a stencil computation on a $\sqrt{d} \times \sqrt{d}$ domain), $a \cdot d \log d$ (*e.g.*, sorting an array of d elements), $d^{3/2}$ (*e.g.*, a multiplication of $\sqrt{d} \times \sqrt{d}$ matrices). For the first two types of complexity a is picked randomly between 2^6 and 2^9, to capture the fact that some of these tasks often perform multiple iterations. Four scenarios were considered: three in which all tasks have one of the three computational complexities above, and one in which task computational complexities are chosen randomly among the three. The volume of data to communicate by a `TRANSFER` node depends on the size of the data handled by its parent `COMPUTATION` node.

The structure of the generated PTGs is defined by four popular parameters in addition to the number of `COMPUTATION` nodes. The *width* determines the maximum parallelism in the PTG, that is the number of tasks per level. A small value leads to "chain" PTGs and a large value leads to "fork-join" PTGs. The *regularity* denotes the uniformity of the number of tasks in each level. A low value means that levels contain very dissimilar numbers of tasks, while a high value means that all levels contain similar numbers of tasks. The *density* denotes the number of edges between two levels of the PTG, with a low value leading to few edges and a large value leading to many edges. This parameter has a direct impact of the number of `TRANSFER` nodes that will be generated to form the PTG. These three parameters take values between 0 and 1. Furthermore we add random "jumps edges" that go from level l to level $l + jump$. Algorithm 13 details the generation procedure used by *daggen*. Uniform distributions are assumed when picking random values.

Algorithm 13 Generation procedure of *daggen*

Generation of n COMPUTATION tasks
  Determine the perfect number of tasks per level: $e^{width \times \log(n)}$
  Randomly assign a number of tasks per level
    by picking a number around the perfect value with $(100 * (1 - regularity))\%$ of latitude
  Randomly assign a cost to each task according to the *computational complexity*
  Pick a random value for α for each task
End
Generation of TRANSFER tasks
  Randomly assign a number of parents to each task:
    $\min(1 + random(0, density \times \#$**tasks in previous level**$)$, **#tasks in previous level**$)$
  if *jumps* are allowed then
    Select in which level to pick the parent
  end if
  Randomly select the parent of the `TRANSFER` node
  Add data volumes to transfer that derive from the size of the data handled by the parent task
End

In some studies, I had to adapt this generation method to produce *layered* PTGs with the particularity that all the tasks in a given precedence level have the same cost. Then all the transfers between the same two levels share the same communication cost. In some others, the costs of the `TRANSFER` nodes were all set to 0 to model applications without inter-task communications. In addition to this tool developed to generate a population of synthetic PTGs, I also developed some scripts to generate real PTGs from the Strassen matrix multiplication and FFT applications to obtain PTGs that are more regular than those produced by *daggen*. However, we usually considered in our studies four different computational complexity scenarios, as for the random PTGs.

A final contribution related to the use of task graphs in simulation studies was to help at the development of loaders of DOT and DAX files for SimDAG. This work, made with J.-N. Quintin and M. Quinson allows users to use graphs produced by GGen [47] or the Pegasus workflow generator as input of their SimDAG simulator. *daggen* has also been modified to export DAGs in the DOT format.

2.5 Result Acquisition and Analysis

Once a simulator is written and various inputs for this simulator, such as platform, workload, or simulation parameters, have been generated, a *simulation campaign* can be launched. Depending on the conducted study, such a campaign can imply the execution of several millions of independent simulation runs. For instance, in [NS06] we compared five scheduling heuristics over a range of 1,296 PTGs and 200 heterogeneous platforms for a total of 1,296,000 independent simulation runs. This raises several challenges related to the efficient execution of this large set of runs, retrieving and storing the obtained results so that pertinent performance information can be extracted later on.

As the size of the simulation campaigns I have conducted increased, I had to develop or use several tools to automate and ease their management. Indeed, a crude management approach based on simple shell scripts to launch simulation runs and large text files to store the results may work for small campaigns but becomes quickly limited and painful to use. In what follows I briefly detail the design and implementation choices I made to manage the simulation campaigns conducted for the evaluation of the scheduling algorithms in Chapter 1.

When developing a simulator, it is important to have in mind that a large simulation campaign is likely to follow. Then, the way input parameters are injected into the simulator and outputs are produced has to be carefully chosen. Most of the simulators I developed follow a simple rule: one set of input parameters corresponds to only one simulation run and produces only one result (maybe including several metrics). The advantages of this design choice are a greater flexibility in the execution of the campaign, an easier automation of the process, and some control on the output location. A potential drawback is that even though many simulation runs share a common subset of input parameters, typically platform and application descriptions, they have to be loaded each and every time. Then, the overall execution time of the campaign may be greater compared to a simulator that would load some combination of input parameters once and run all the tested algorithms on it, for instance. However, the improvements made around SIMGRID about the management of platforms and applications considerably reduced this overhead. The improved flexibility is thus worth the extra loading time. In this configuration, the total output set produced by the campaign is scattered in as many files as independent simulation runs. This prevents a straightforward filtering of results with basic tools such as `grep`. Other methods to manage results have to be implemented.

The following example shows a command line typical of the different simulators I developed. Depending of the studied problem, this command line can be more complex and have more parameters, but it comprises at least the set of input parameters described hereafter.

```
1 $./my_simulator -h heuristic -p platform.xml -d app1.dot [-d app2.dot ...]
```

A simulation run performed in a scheduling study always requires three input parameters: (i) the scheduling heuristic to study (plus some related configuration flags when needed); (ii) a description of the experimental environment, or platform, in the XML format supported by SIMGRID; and (iii) the description of one or several applications represented as DAGs or PTGs. The size of the campaign, *i.e.*, the total number of such independent simulation runs, is then determined by the cross-product of the numbers of values used for each parameter.

The second condition to ensure the most flexible management of a simulation is to have independent and easy to parse outputs. Most of my simulators produce results formatted as in the example below.

```
1 biCPA:chti.xml:FFT16-0.0:434.724:8694.483
```

Information is separated by colons to ease further parsing. This output line starts with the parameters given as input, *i.e.*, heuristic, platform, and application, that describe a unique simulation run. Then, it stores values for the different metrics of interest, makespan and total work for instance.

To launch hundreds of thousands independent simulation runs, that require thousands of distinct input files, and retrieve a similar amount of outputs, I relied on a simple but powerful tool. The AppLeS Parameter Sweep Template (APST) framework [40] was developed by J. Hayes at UCSD, in the team where I spent my post-doctoral year. As its name says, APST has been designed to handle *parameter sweep* applications, a category of applications that totally corresponds to a simulation campaign. Indeed, the same code, *i.e.,* the simulator, is executed for a broad range of input parameters, *i.e.,* the simulation scenarios. The main advantage of APST is its simplicity. It can run in user mode without any software dependency. The user simply launch a daemon, declare the resources to use and the tasks to execute in an XML file, and then submit this file to the daemon. The mapping of tasks on resources is fully automatized. Moreover, the status of the campaign, *e.g.,* how many tasks have completed or failed, can be easily monitored. For my scheduling studies, I generally limited the resources to use to my own laptop. In some cases, APST was combined to a script that acquires as much available resources as possible on Grid'5000. Those extra processors were accessed in a *best effort* mode, *i.e.,* they could be preempted at any time by a regular job submitted by another user.

To store all the produced results, draw graphs, and fill tables that were used in articles to support the performance analysis of some scheduling heuristics, I found convenient to rely on a relational database. Indeed, the output format introduced earlier can be directly mapped to a database entry. Resorting to a database greatly simplifies the comparison of heuristics and reduces the time needed to extract processed data. For large campaigns, storing results in flat text files would have made the post-processing too cumbersome. Then, I developed many scripts to populate mySQL tables with the simulation results, and to extract, aggregate, compare, or combine these results. These scripts have evolved as I improved my analysis methodology as explained in Section 1.3. These methods ranged from extracting only average values to analyzing the full distribution for the considered performance metrics.

Another useful analysis tool I used for many of my scheduling studies is visualization. Scheduling algorithm usually aim at optimizing an objective function, *e.g.,* minimizing the makespan. Then, the evaluation of these algorithms is often limited to the analysis of the final achieved value. What led to this makespan, that is the schedule itself, is hardly ever analyzed. It is because it is hardly possible for humans to get a rough idea of an entire schedule by looking only at log files. Visualization helps the analysis of schedules in different cases. First, a new algorithm might perform better than all its competitors in most cases. However, there might be corner cases which could be easily spotted with a graphical representation of the schedules. Second, the debugging of scheduling algorithms is greatly eased by visualization. But, while scheduling is a broadly covered field in Computer Science, only a few tools exist that help researchers to develop scheduling algorithms.

In [HHS10], we presented, with S. Hunold and R. Hoffmann, the software tool Jedule[13] that can visualize arbitrary schedules as Gantt charts. A schedule is displayed in two dimensions, one that corresponds to the resources of the system, *e.g.,* processors, cores, or hosts, while the other corresponds to time. Scheduled tasks are depicted by rectangles in such a representation. Jedule has been designed with PTG scheduling in mind. Then, it handles compute tasks that span on several (potentially non contiguous) resources and complex data redistributions. Jedule relies on its own custom XML structure to represent schedules that can be automatically dumped from any SimDAG simulator.

Jedule was a great help for the evaluations conducted in [CDS10] and [CDS10]. As detailed in Chapter 1, these articles were about the scheduling of multiple PTGs on a single cluster. Visualization helped us to check the validity of one of the proposed approaches which consists in distributing the resources among the applications. Each application then has to build its own schedule according to this *constrained resource allocation*.

A critical issue for such an algorithm is to ensure that each schedule respects its resource constraint. Figure 2.14 shows a schedule produced by the considered algorithm for four PTGs. We can see that the tasks of each application are mapped on distinct processors, hence confirming that the algorithm

[13] `http://sourceforge.net/projects/jedule/`

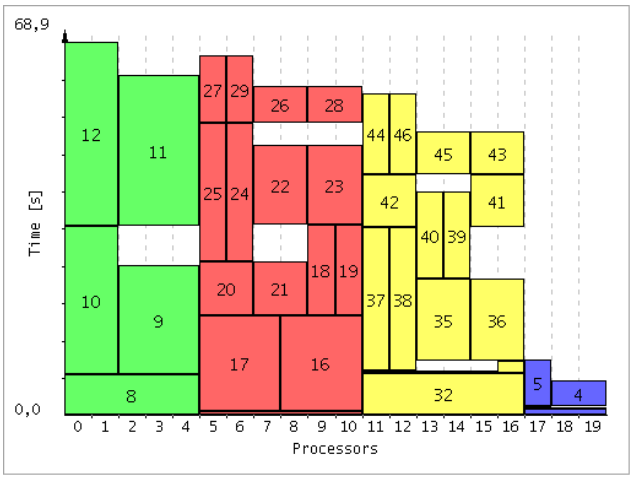


Figure 2.14: Jedule output for the schedule produced by one of the algorithms proposed in [CDS10]. Four PTGs, each having its own color, are scheduled on a cluster of 20 processors. The resource constraints imposed by the algorithm are respected.

does what it was designed for. It also points out that the initial distribution of the processors among the applications is too restrictive. For instance, processors 17 to 19 are clearly underused. Such information which could be extracted from text logs, but with more efforts, immediately highlights the need for more complex algorithms such as the one that was later proposed in [CDS10]. In this context, Jedule was also used to see the impact of a conservative backfilling step applied at the end of the scheduling process. A comparison of the Jedule outputs with and without backfilling allows for a check that no task is delayed by this step. The reduction of the total idle time can also be easily quantified.

Jedule was also useful to spot strange phenomena more easily. As an illustration, Figure 2.15 shows the Jedule output of the schedule produced by HEFT [147] for a DAG of 50 tasks on a heterogeneous platform made of four small homogeneous clusters that respectively comprise two or four processors.

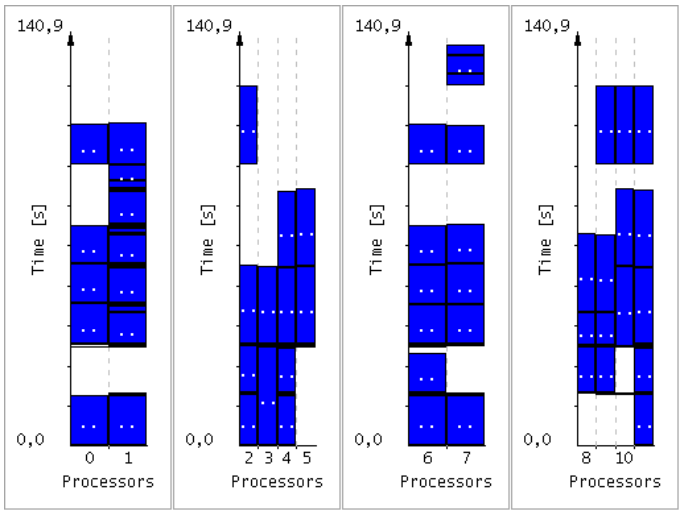


Figure 2.15: Jedule output of the schedule by HEFT of a DAG of 50 task on a heterogeneous platform.

From this output, we suspected a bad scheduling decision for the the last task executed on processor 2. Indeed, there are three other tasks with exactly the same characteristics, *i.e.*, execution time, input data size, and dependencies to satisfy. These three tasks are respectively executed on processors 9, 10, and 11. The logic indicates that the fourth task should have been scheduled on processor 8 instead of processor 2. This tends to indicate a flaw in the scheduling algorithm. We thus checked the logs of the scheduling process. It appeared that processor 2 actually leads to the earliest finish time for this task. The scheduling decision was then correct despite the glaring issue. In presence of inter-task communications, moving a task from one cluster to another should lead to a greater finish time, but this scheduling decision shows the opposite. Sending data to another cluster is as costly as executing the task locally. The source of the strange behavior pointed out by Jedule was in fact the description of the execution platform. The latency of the backbone link connecting the clusters was the same as that of the links connecting the processors in a cluster. In a real setting, this inter-cluster latency is likely to be much higher. Then we corrected our description to reflect a more realistic setting.

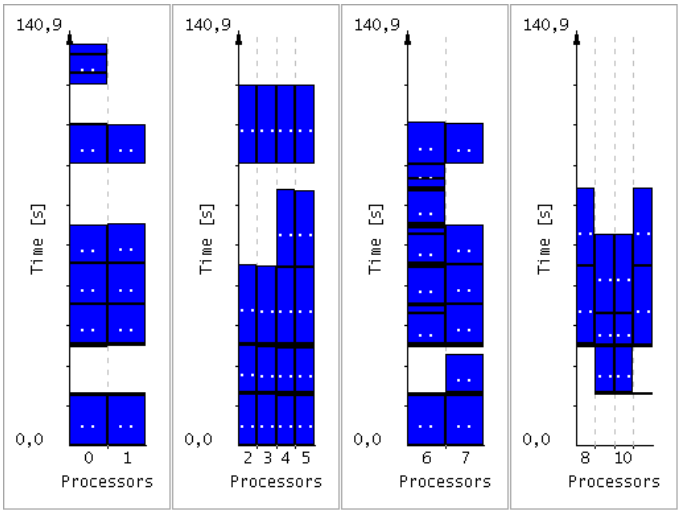


Figure 2.16: Jedule output of the schedule by HEFT of a DAG of 50 task on a heterogeneous platform with a greater latency on the backbone link.

The schedule obtained on this modified platform is showed by Figure 2.16. It does not exhibit odd scheduling decisions anymore. The two fast clusters (processors 0-1 and 6-7) are chosen first and then the larger but slower clusters are used. We can also see that one of these slow clusters is more heavily used. This reflects the impact of the higher backbone latency on the scheduling decisions. Finally, it is worth noting that the overall makespan is the same for both schedules (140.9 s). If we had only relied on this metric to detect suspect behaviors, we would have missed this particular issue. This demonstrates the benefits of visualization on the analysis of simulation results, at least in a scheduling context.

Management of simulation campaigns and result analysis are very active fields within the SIMGRID project. These topics are even considered as simulation pillars. Advance visualization techniques, such as those offered by Triva [134], *e.g.*, treemaps or time and space aggregations, have proved their importance to better understand various phenomena occurring during a simulation [135]. To optimize a simulation campaign, and thus minimize the numbers of independent runs to execute, techniques related to the Design of Experiments are investigated. Such methods are widely used in other sciences but scarcely in Computer Science. Being part of the SIMGRID team and following closely the conducted investigations will help me to improve my own experimental methodology. In the near future, I plan to apply such methods to run smaller campaigns that however lead to more backed-up and sound results.

2.6 Dimensioning Through Simulation

Large-scale distributed computing systems, such as grids or clouds, are victims of their success. Driven by the High Energy Physics community since the European DataGrid project, computing and data grids such as EGEE, and now EGI, have allowed physicists to store, share, and analyze the tremendous amount of data produced by the Large Hadron Collider at CERN. Other communities, such as biology, earth science, or even humanities, are also evolving to data-driven sciences. The direct consequence of this data deluge is the constant and mandatory upscale of data and computing centers. For instance, the IN2P3 Computing Center has recently doubled the floor area of its computing room to be able to host and power the resources required to fulfill its engagements up to 2019.

In this context, dimensioning becomes a main concern to optimize the utilization of these computing and storage infrastructures. Indeed, it is crucial to determine precisely the investment to grant for each type of resources and how to physically organize them to get the "best bang for the buck". Unfortunately it is hard to estimate the relevance of a solution without implementing it. Such decisions are thus taken based on years of experience shared by system administrators and users. The former knows how complex systems work while the latter have expertise on the behavior of their applications. Nevertheless this process lacks of objective data about the performance of a given candidate infrastructure. Benchmarks suites exist, such as the LINPACK benchmark used to establish the Top500[14] list, but they can only be executed once the resources have been bought and deployed. Any unforeseen behavior can then lead to large but vain expenses.

An alternative would be to resort to simulation to obtain the expected objective indicators and compare each possible evolution of the infrastructure. In 2010, I initiated a CNRS *Projet International de Coopération Scientifique* (PICS) between the IN2P3 Computing Center and the University of Hawai'i at Manoa to work with H. Casanova on this topic of dimensioning through simulation. In this project, we aim at evaluate, compare, strengthen, and integrate within the SIMGRID toolkit two complementary approaches: *on-line* simulation, also called simulation via direct execution, and *off-line* simulation, also called *post-mortem* simulation, of parallel applications. We focus on applications relying on the MPI [83] standard as they represent the vast majority of currently deployed parallel applications. In on-line simulation the application is executed but part of the execution takes place within a simulation component. In off-line simulation a trace of a previous execution of the application is "replayed" on a simulated platform. In the next sections I detail the contributions made for each approach respectively.

2.6.1 Single Node On-Line Simulation of MPI Applications with SMPI

One option for simulating the execution of an MPI application is *on-line* simulation. In this approach, the actual code of the application, with no or marginal modification, is executed on a *host platform* that attempts to mimic the behavior of the *target platform*. Part of the instruction stream is then intercepted and passed to a simulator. LAPSE is a well-known on-line simulator developed in the early 90's [57]. In LAPSE, the parallel application executes normally but when a communication operation is performed a corresponding communication delay is simulated on the target platform using a simple network model (affine point-to-point communication delay based on link latency and bandwidth). MPI-SIM [13] builds on the same general principles, with the inclusion of I/O subsystem simulation in addition to network simulation. A difference with LAPSE is that MPI processes run as threads, which is enabled by a source code preprocessor (*e.g.*, to privatize global variables). Another project similar in intent and approach is the simulator described in [130]. The BigSim project [158] also builds on similar ideas. However, unlike MPI-SIM, BigSim allows for the simulation of computational delays on the target platform. This makes it possible to simulate "what if?" scenarios not only for the network but also for the compute nodes of the target platform. Simulation of computation delays in BigSim is done based either on user-supplied

[14] http://www.top500.org

projections for the execution time of each block of code (as done also in [84]), on scaling execution times measured on the host platform by a factor that accounts for the performance differential between the host and the target platforms, or on sophisticated execution time prediction techniques such as those developed in [139]. The weakness of such approaches is that since the computational application code is not executed, the computed application data is erroneous. Consequently, application behavior that is data-dependent is lost. This is acceptable for many regular parallel applications, but can make the simulation of irregular applications (*e.g.,* branch-and-bound) questionable at best. Aiming for high accuracy, the work in [106] uses a cycle-accurate hardware simulator of the target platform to simulate computation delays, which leads to a high ratio of simulation time to simulated time.

The complexity of the network simulation model has a high impact on speed and scalability, thus compelling many authors to adopt simplistic network models. One simplification, for instance, is to use monolithic performance models of collective communications rather than simulating them as sets of point-to-point communications [12, 145]. Another simplification used in most aforementioned simulators, whether off-line or on-line, is to ignore network contention because simulating it is known to be costly [159]. The work in [145] proposes the use of simple analytical models of network contention for off-line simulation. An exception is the MPI-NetSim on-line simulator [121], which provides full-fledge contention simulation via a packet-level discrete-event network simulator. As a result, the simulator may run more slowly than the application, which poses time coherence problems for on-line simulation. The solution in [121] is to slow down the entire system (*i.e.,* inserting sleep calls during the application execution) so that the simulator has the time to simulate all network traffic without inducing timing skew. This approach has also been used in the general-purpose simulation environment MicroGrid [141]. Another exception is the PEVPM on-line simulator [84]. PEVPM relies on extensive benchmarks of the target platform that provide probability distributions of communication times, which can in turn be used to model network contention phenomena. Finally, note that two options for general-purpose on-line simulation are to reconfigure the cluster interconnect of the host platform to mimic that of the target platform [148], or to load the host platform with a judiciously chosen synthetic user-level workload [33].

One difficulty faced by most MPI-specific on-line simulators is that the simulation, because done via direct execution of the MPI application, is inherently distributed. Parallel discrete event simulation raises difficult correctness issues pertaining to process synchronization. For the simulation of parallel applications, techniques have been developed to speed up the simulation while preserving correctness (*e.g.,* the asynchronous conservative simulation algorithms in [122], the optimistic simulation protocol in [158]). A way to side-step this difficulty is to run the simulation on a single node which also challenging as it requires large amounts of computing and memory resources. For most aforementioned on-line approaches, the resources required to run a simulation of an MPI application are commensurate to those of that application. In some cases, those needs can even be higher (*e.g.,* an extra node to run the network simulation component [121], or costly cycle-accurate simulation of the application's code [106]). One way to reduce the computing needs of the simulation is to avoid executing computational portions of the application and simulate only expected delays on the target platform [84, 158]. Reducing the need for memory resources is more difficult. For instance, simulations in [84], which run on a single-node, are for applications with small memory footprints. In general, if the target platform is a large cluster, then the host platform must also be a large cluster. However, a solution proposed in [1] consists in removing large data arrays from the simulation with the help of the compiler. When doing so, the modified application produces erroneous results. But, for non-data-dependent applications memory usage reductions up to 4 orders of magnitudes are reported.

In [CSG+11], we detailed, in collaboration with P.-N. Clauss, M. Stillwell, S. Genaud, H. Casanova, and M. Quinson, the aims and internals of another on-line simulator, called Simulated MPI (SMPI). As mentioned in Section 2.2.1, SMPI is implemented as an API of the SIMGRID toolkit. In its current implementation SMPI implements the following subset of the MPI standard:

- error codes, predefined datatypes, and predefined and user-defined operators;
- process groups, communicators, and their operations (except Comm_split);
- these point-to-point communication primitives: Send_Init, Recv_Init, Start, Startall, Isend, Irecv, Send, Recv, Sendrecv, Test, Testany, Wait, Waitany, Waitall, and Waitsome;
- these collective communication primitives: Broadcast, Barrier, Gather, Gatherv, Allgather, Allgatherv, Scatter, Scatterv, Reduce, Allreduce, Scan, Reduce_scatter, Alltoall, and Alltoallv.

An SMPI simulation runs in a single process, with each MPI process running in its own thread. However, these threads run sequentially, under the control of the fully sequential simulation kernel of SIMGRID. The potential drawback of a sequential kernel is that simulation time may increase drastically with the scale of the simulation. However, SIMGRID relies on the analytical simulation models implemented in SURF that can be computed quickly, leading to scalable simulation capabilities.

SMPI was designed to be used as seamlessly as possible. Figure 2.17 shows the compilation chains provided by SMPI for C and FORTRAN codes. Two wrappers, `smpiff` and `smpic99`, take source code as input and produce an object file after source-code modifications. As MPI processes are replaced by threads within a single process, global variables need to be privatized so that each thread has its own copy. For C codes, we use the Coccinelle semantic patching tool [116] to locate global variables automatically, and rely on a Perl script to privatize them. For FORTRAN codes, the provided wrapper internally resorts to the `f2c` tool for translating a FORTRAN code into an equivalent C code. Thanks to the clean and regular output from `f2c`, the global variables in the translated FORTRAN code are made private by a straightforward Perl. The resulting C codes are then passed to the `gcc` compiler through the `smpicc` wrapper.

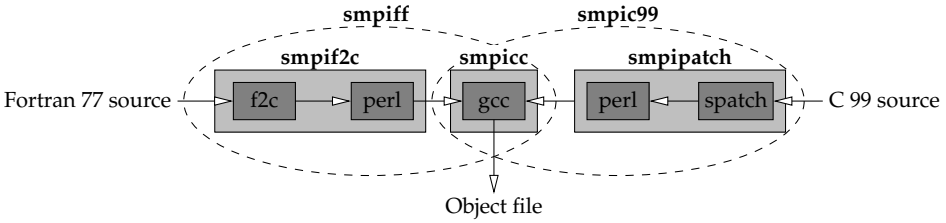


Figure 2.17: Application compilation with SMPI.

Once compiled with `smpicc`, the produced executable file can be executed thanks to a command line very similar to that MPI users are accustomed to.

```
bob:~$ smpirun -np 8 \
  -platform target_platform.xml -hostfile process_mapping \
  my_MPI_app <application parameters>
```

Users have to call `smpirun` instead of the classical `mpirun`. They still give the number of processes used to run the instance (`-np`), as well as the way these processes are mapped onto physical resources (`-hostfile`), and the executable itself with its own parameters. The only extra information given to SMPI is a description of the *target platform* (`-platform`) that gives the characteristics of the simulated environment in which the application will execute.

One of the main aims for SMPI is to simulate an application on a large target platform while using a single node as the host platform, which we call the "host node." While the motivation for single-node simulation is clear, it is a challenging proposition for on-line simulation due to the computing and memory requirements of the simulated applications.

In general, the amount of time needed to execute the computational portions of the application's code, or computing bursts, on a single node is proportional to the number of nodes of the target platform that are used by the application. As in [84, 158], we opt to replace each burst in the simulation by the corresponding expected delay on a target platform node. Thus, unless the application's execution is data-dependent, the computational time of the application when it runs in simulation could be negligible. The main question, however, is how to determine the delay of each burst on the target platform. We allow for the execution of each burst only the first n times the burst occurs, and then using the average delay computed over these n samples as the delay in the simulation for future occurrences of this burst. Using $n > 1$ is useful for bursts that exhibit execution time variations, *e.g.*, due to application data. Since the bursts are measured on the host node, their durations are used directly for simulating a target platform comprised of nodes identical to that of the host node. Otherwise, we simply allow the user to specify a factor by which burst durations can be scaled to account for a performance differential between the host node and the nodes of the target platform. We allow for $n = 0$, in which case the user must supply a number of flops (which is then transformed into a delay using the aforementioned factor) for simulating the corresponding burst on the target platform.

The time to execute each burst $n > 0$ times on the node platform is proportional to the number of nodes in the target platform, since each MPI process executes each CPU burst n times. The scalability of this approach may thus not be acceptable because simulation time increases linearly with the number of simulated nodes. In many parallel applications, computations are regular, meaning that the MPI processes execute identical or similar bursts. This is the case, for instance, for most applications using the Single Program Multiple Data (SPMD) paradigm. Therefore, SMPI allows the measurement of the execution times of the first n bursts on any MPI process. The simulation time of application computation is then independent on the number of nodes in the target platform, and thus scalable. This can be enabled easily by a code preprocessor that takes n as input and inserts global counters and if-then-else statements around the code for each burst. In the current version of SMPI, macros were developed for the standard C preprocessor. They have to be manually inserted by the user in his/her code.

```
1  ...
2  MPI_Init(...);
3  SMPI_SAMPLE_LOCAL(10)
4  { < Some computation (A) > }
5  SMPI_SAMPLE_GLOBAL(10)
6  { < Some computation (B) > }
7  SMPI_SAMPLE_DELAY(1048576)
8  { < Some computation (C) > }
9  < Some computation (D) >
10 MPI_Finalize();
11 ...
```

Figure 2.18: SMPI macros to reduce computing requirements.

Figure 2.18 shows a code sketch that uses these macros. At line 3, the `SMPI_SAMPLE_LOCAL` macro is used to indicate that the following burst (in between curly braces) should be executed and timed 10 times by *each* MPI process, and subsequently bypassed and replaced by a simulation of a delay equal to the average of the 10 measured execution times. At line 7, the `SMPI_SAMPLE_GLOBAL` macro is similar but the burst is measured only 10 times in total (possibly when executed by 10 different MPI processes), before its execution is bypassed and replaced by an average delay. At line 11, the block of code following the `SMPI_SAMPLE_DELAY` is never executed but instead replaced in the simulation by the given amount of flops. These macros are expanded into one or more calls to various functions that look up and update

hash tables where each entry contains a unique identifier (based on source file name and line number), execution counters, reference counters, and/or pointers to user arrays.

For applications that are irregular or data-dependent, replaying previously measured burst durations may not lead to accurate results. In the worst case, all bursts would need to be executed. In this case, single-node simulation would suffer from severe scalability issues. Should these issues endanger the applicability of the approach, one would have to face the challenges of developing a parallel discrete event simulator that can be executed on a cluster, as done for instance in MPI-SIM or MPI-NetSim.

A problem with single-node on-line simulation is that the memory footprint of the application cannot be accommodated on the host node unless the number of nodes in the target platform is small and/or the application's footprint is small. In an SMPI simulation, all MPI processes run as threads that share the same address space. In this case, two techniques are proposed in [1] for removing large array references:

Technique #1: Because MPI processes run as threads, references to local arrays can be replaced by references to a single shared array. If the MPI application has m processes that each use an array of size s, then the memory requirement is reduced from $m \times s$ to s.

Technique #2: Because a burst is simulated by replaying a delay rather than by executing its code, memory references in that code can be removed, which can lead to the removal of potentially large, now unreferenced, arrays.

Both techniques are implemented in SMPI, but the second one can only be used if n, the number of measurements of each burst duration, is 0. In this case, as in [1], burst durations are user-provided and the burst code is effectively removed. Here also we rely on macros designed for the standard C preprocessor as shown in Figure 2.19.

```
1 ...
2 double *data = (double*)SMPI_SHARED_MALLOC(...);
3 MPI_Init(...);
4   < Some computation (A) >
5 MPI_Finalize();
6 SMPI_FREE(a);
7 ...
```

Figure 2.19: SMPI macros to reduce memory requirements.

At line 3 array `data` is allocated using the `SMPI_SHARED_MALLOC` macro. This macro allows the array to be allocated only once and to be shared by all simulated MPI processes. Similarly, at line 7, the `SMPI_FREE` macro is used so that the array is freed only once.

Another goal of SMPI is to perform accurate simulation of both point-to-point and collective communications. One option is to use a packet-level discrete event network simulator, as in MPI-NetSim [121]. The drawback is that packet-level simulation is neither fast nor scalable. For instance, simulation time grows roughly linearly with message size. Simulation time can then be longer than simulated time, which poses difficulties for preserving the coherence of an on-line simulation. Not surprisingly, most of the simulators have opted for the not (fully) realistic, but simple, standard affine model defined by a latency and bandwidth parameter. In this model the time to transfer a message of size s in bytes from one node to another is $\alpha + s/\beta$ where α is the network latency in seconds and β the bandwidth in bytes/sec. Unfortunately, this model fails to capture the behavior of real-world cluster interconnects using TCP and popular MPI implementations (*e.g.*, OpenMPI [74] over a Gigabit Ethernet switch). For instance, a message under 1 KiB fits within an IP frame, in which case the achieved data

transfer rate is higher than for larger messages. Also, MPI implementations for `MPI_Send()` typically switch from buffered to synchronous mode above a certain message size. Consequently, instead of being an affine function of message size, communication time is likely to be *piece-wise linear*.

SMPI also uses an analytical network model that can be computed quickly and in scalable way. However, we contend that this model is more accurate than the analytical models used in previously developed MPI simulators. Indeed, SMPI models point-to-point communication times with a piece-wise linear model with an arbitrary number of linear segments. Each segment is obtained using linear regression on a set of real measurements. The number of segments and the segments boundaries are chosen such that the product of the correlation coefficients is maximized. In practice, we find that the model should be instantiated for three segments, leading to eight parameters defining the model (two for defining the boundaries of the three segments, and one latency and bandwidth parameter for each segment). To illustrate the necessity to use a piece-wise linear model, we compared the results achieved by SKaMPI (using OpenMPI) and by SMPI for a simple ping-pong test between two machines

Figure 2.20 shows communication time versus message size, using a logarithmic scale for both axes. We display three sets of SMPI results. The results "Default Affine" are obtained for an affine model calibrated on the cluster using the time to send a 1-byte message for the latency and the maximum achievable bandwidth using the TCP/IP protocol (*i.e.*, approximately 92% of the peak bandwidth). This is the standard method for instantiating the affine model, and corresponds to the approach taken by many MPI simulators. The "Best-Fit Affine" results are for an affine model instantiated using the latency and bandwidth values that minimize the average logarithmic error with respect to the SKaMPI results. We include these results to see whether a linear model could be inherently inaccurate. Finally, the "Piece-Wise Linear" results are for the piece-wise linear model instantiated by SMPI.

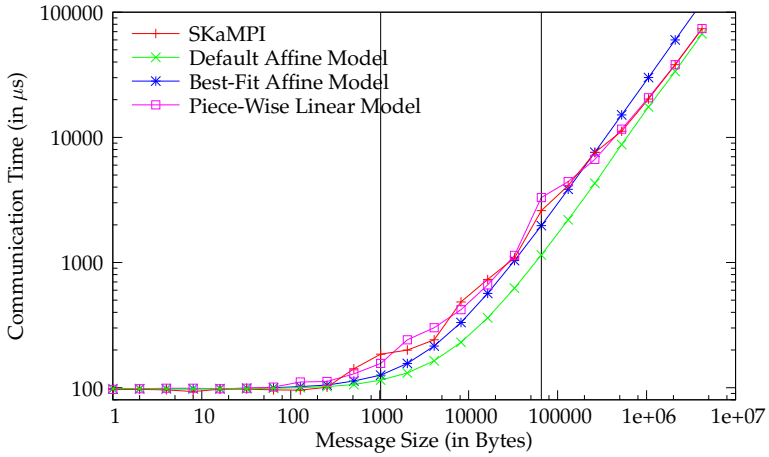


Figure 2.20: Comparison between a SKaMPI run and SMPI (default affine, best-fit affine, and piece-wise linear models) for a ping-pong operation between two machines.

We see that the piece-wise linear model matches the real-world results very well. By contrast, both affine models fail to capture the entire real-world behavior. The Default Affine model is accurate for small and big messages, but inaccurate in between while the Best-Fit Affine model performs better for medium-sized messages, but overestimates communication time for big messages.

Furthermore SMPI takes into account effects induced by packet-based communication over a single link, a very common interconnection pattern. Multiple concurrent communications that appear to occur simultaneously from the application point of view are actually sequenced on the output link. The effective starting time of each of the these communications is thus delayed by a constant *gap* for each communication that is already pending.

As mentioned before, an SMPI simulation takes as input the number of MPI processes, their command-line arguments, and a specification of the target platform. This specification is written in XML using SIMGRID's DTD described in Section 2.4.1. In the context of SMPI, the specification contains descriptions of the cluster nodes, including a performance indicator measured in number of floating point operations performed in one second. Then, the performance of the host node is given as argument to the simulation program, allowing to scale the timings obtained on the host node to what would be experienced on the nodes of the target platforms. The specification also lists network elements, which are on paths between cluster nodes. The performance of point-to-point communications on these links is described by eight parameters. While the values of these parameters can be chosen arbitrarily by the SMPI user, it is likely difficult to simply pick reasonable values. This is why we *calibrate* the SMPI simulation by automatically instantiating these parameters based on point-to-point experiments executed on two nodes of one or more real-world clusters. It is then possible to modify this instantiation to run simulations for "what if?" scenarios (*e.g.*, simulate a network that achieves 30% higher data transfer rate for large messages).

We use the freely available SKaMPI [129] benchmarking framework to perform simulation calibration. Using the simple ping-pong MPI benchmark provided by SKaMPI, we obtain data transfer times achieved for a wide range of message sizes. We can then automatically fit the experimental data to a piece-wise linear model, thereby obtaining an instantiation of the required parameters. A user can easily perform such instantiation when wanting to simulate a particular cluster deployment. Alternatively, this instantiation can be conducted by a third party, for a range of typical cluster deployments, and made publicly available. SMPI users can then reuse these instantiations, or modify them to explore reasonable "what if?" scenarios.

In [CSG+11], we demonstrated the accuracy, scalability and speed of SMPI simulations for scenarios ranging from simple point-to-point communication to more complex communication benchmarks. We showed that a piece-wise linear model is necessary for accurate simulation of MPI communications on a cluster. We also showed that a network model without contention, such as the one used by most of the on-line MPI simulators, always underestimates the completion time of a scatter operation. Conversely our piece-wise linear model with contention leads to simulated execution times that are very close to the performance of MPI implementations. On average, the difference between SMPI and MPICH2 was almost the same as the difference between OpenMPI and MPICH2. Finally, the proposed techniques aiming at reducing the computation and memory requirements of SMPI-based simulations, allowed us to push some scalability limits while keeping a reasonable simulation time.

2.6.2 Off-Line Simulation with Time-Independent Trace Replay

An alternate approach for simulating the execution of an MPI application is *off-line* simulation in which a log, or trace, of MPI communication events (time-stamp, source, destination, data size) is first obtained by running the application on a real-world platform. A simulator then replays the execution of the application as if it were running on a *target platform*. This approach has been used extensively, as shown by the number of trace-based simulators described in the literature since 2009 [89, 145, 118, 156, 87]. The typical approach is to decompose the trace in time intervals delimited by the MPI communication operations. The application is thus seen as a succession of computation bursts and communication operations. An off-line simulator then simply computes simulated delays in a way that accounts for the performance differential between the platform used to obtain the trace and the platform to be simulated. Computation delays are typically computed by scaling the durations of the CPU bursts in the trace [145, 118, 87]. Network delays are computed based on a simulation model of the network.

One limitation of these off-line simulators is that trace acquisition is not scalable. Indeed, to simulate the execution of an application on a platform of a given scale, the trace must be acquired on a homogeneous platform of that same scale, so that time-stamps are meaningful. In some cases, extrapolating a smaller trace to larger numbers of compute nodes is feasible [89], but not generally applicable.

Furthermore, the use of time-stamps requires that each trace be accompanied with a description of the platform on which it was obtained, so as to allow meaningful scaling of computation delays. In this work we address the trace acquisition scalability limitation by using time-independent traces.

Another limitation is that these simulators typically use simplistic network models, because they are straightforward to implement and scalable. Possible simplifications include: not using a network model but simply replay original communication delays [87]; ignoring network contention because it is known to be difficult and costly to simulate [89, 156, 159]; using monolithic performance models of collective communications rather than simulating them as sets of point-to-point communications [145, 87, 12]. Other simulators opt for accurate packet-level simulation, which is not scalable and leads to high simulation times [118]. In this work we address this limitation by leveraging the validated and scalable network models in the SIMGRID framework.

One well-known challenge for off-line simulation is the large size of the traces, which limits the scalability of trace acquisition and can prevent running the simulation on a single node. Mechanisms have been proposed to improve scalability, including compact trace representations [145] and replay of a judiciously selected subset of the traces [156]. In this work, we discuss how the traces used by our framework can be compacted and we demonstrate the scalabilty of our trace acquisition procedure.

Time-stamped traces for use in off-line simulation cause several problems, in particular the need to acquire traces on large-scale, homogeneous platforms to conduct large-scale simulations. To obviate these problems we proposed in [DMQS11], in collaboration with F. Desprez, G. Markomanolis and M. Quinson, to conduct off-line simulation using *time-independent* traces. For each event occurring during the execution of the traced application, *e.g.,* a CPU burst or a communication operation, we log its volume (in number of executed instructions or in number of transferred bytes) instead of the time when it begins and ends. The main advantage is that our logged information does not depend on the hardware characteristics of the platform on which the trace is collected, with the exception of the processor family. The size of the messages sent by an application is not likely to change according to the specifics of the network interconnect, and the number of instructions performed within a `for` loop does not increase with the processing speed of the CPU. This claim does not hold for adaptive MPI applications that modify their execution path according to the execution platform. Such applications, which represent a small fraction of production MPI applications, are outside the scope of this work.

A time-independent trace is a list of *actions* performed by each process of an MPI application. An action is described by the *rank* of the process that performs it, a *type* (computation or communication), a *volume* (number of instructions or number of bytes), and some action-specific parameters (*e.g.,* the rank of the receiving process for a one-way communication).

```
1  for (i=0; i<4; i++)
2   if (myId == 0){
3    /* Compute 1M instructions */
4    MPI_Send(1MB,...,(myId+1));
5    MPI_Recv(...);
6   } else {
7    MPI_Recv(...);
8    /* Compute 1M instructions */
9    MPI_Send(1MB,...,(myId+1)%4);
10  }
```

```
1 0 compute 1e6
2 0 send 1 1e6
3 0 recv 3 1e6
```

```
1 1 recv 0 1e6
2 1 compute 1e6
3 1 send 2 1e6
```

```
1 2 recv 1 1e6
2 2 compute 1e6
3 2 send 3 1e6
```

```
1 3 recv 2 1e6
2 3 compute 1e6
3 3 send 0 1e6
```

Figure 2.21: Simple MPI application on a ring (left) and corresponding time-independent trace (right).

The left-hand side of Figure 2.21 shows a simple MPI application executed on a ring with four processes. Each process computes one million instructions and sends one million bytes to its neighbor.

The right-hand side of the figure displays the corresponding time-independent trace. For large numbers of processes and/or numbers of actions, it may be preferable to split the time-independent trace so as to obtain one trace file per process.

MPI actions	Trace entry
CPU burst	`<rank> compute <volume>`
MPI_Send	`<rank> send <dst_rank> <volume>`
MPI_Isend	`<rank> Isend <dst_rank> <volume>`
MPI_Recv	`<rank> recv <src_rank> <volume`
MPI_Irecv	`<rank> Irecv <src_rank> <volume`
MPI_Broadcast	`<rank> bcast <volume>`
MPI_Reduce	`<rank> reduce <vcomm> <vcomp>`
MPI_Allreduce	`<rank> allReduce <vcomm> <vcomp>`
MPI_Alltoall	`<id> allToAll <send_volume> <recv_volume>`
MPI_Alltoallv	`<id> allToAllv <send_buffer> <send_counts> <recv_buffer> <recv_counts>`
MPI_Barrier	`<rank> barrier`
MPI_Wait	`<rank> wait`
MPI_Waitall	`<id> waitAll`

Table 2.4: Time-independent actions corresponding to supported MPI communication operations.

Table 2.4 lists the MPI functions that can be replayed by our Time-Independent Trace Replay Framework. Our framework replays traces using an MPI application simulator provided as part of SIMGRID. This simulator allows us to simulate the MPI collective communication operations in Table 2.4 in a way that corresponds to popular MPI implementations (MPICH2 [82] and OpenMPI [74]).

The acquisition procedure of a time-independent execution trace, depicted in Figure 2.22, comprises three steps: (i) the instrumentation of the target application; (ii) the execution of this instrumented version; and (iii) the gathering of the different traces to a single location.

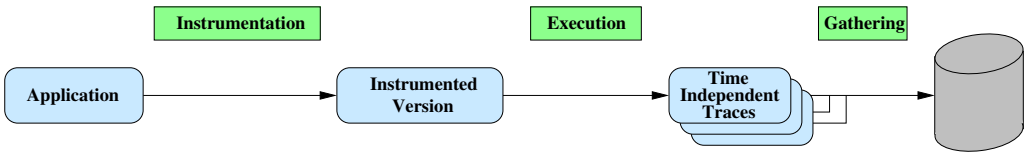


Figure 2.22: Time-independent trace acquisition process.

The first step of the acquisition procedure is to instrument the MPI application to be simulated, so that an execution of the instrumented application generates application event traces that can be used to replay the application in simulation. The only information required by our Time-Independent Trace Replay Framework are: (i) the volume of computation in between two MPI calls at each MPI process, measured in number of instructions; (ii) the name of and parameter values passed to each MPI call; (iii) the volume of data transferred by each communication operation; and (iv) the exact sequence of computation bursts and communication operations executed by each process.

Several tools are available that instrument MPI applications for various purposes including debugging, profiling, performance debugging, and execution visualization [136, 76, 102, 27, 104, 101, 142]. After a thorough evaluation of the major existing tools to profile or trace parallel applications conducted in [DMS13], we decided to base our first prototype on the Tuning and Analysis Utilities (TAU) Performance System [136], a popular tool for post-morterm analysis of MPI applications. TAU can generate

application event traces and does report on hardware performance counters through the Performance Application Programming Interface (PAPI) [27] interface, and can thus be used for our purpose. In fact, like most of these other tools, TAU provides capabilities well beyond our needs. Two undesirable side-effects of instrumentation are *overhead*, *i.e.,* extra execution time for obtaining the traces, and *skew*, *i.e.,* increase of the application's instruction count due to the insertion of instrumentation code. Unnecessary instrumentation, in our case instrumentation that generates trace data beyond the information strictly needed for our replay framework, would then unnecessarily increase overhead and skew. It turns out that TAU enables *selective instrumentation*, by which parts of the application's source code can be ignored for the purpose of instrumentation. This feature can be used in an attempt to avoid unnecessary instrumentation, namely enabling only the tracing of MPI calls and the gathering of numbers of executed instructions in between these calls. This is achieved by telling TAU to exclude all application source files from instrumentation. We have also implemented our own instrumentation method. The MPI standard exposes two interfaces for each MPI function, one prefixed with `MPI_` and the other prefixed with `PMPI_`, the former calling the latter directly. This provides developers with the opportunity to insert their own code, *e.g.,* for profiling purposes, in the implementation of all `MPI_` functions. This mechanism is used by several of the aforementioned tools, and we ourselves use it to insert code that is executed upon entry and exit for all MPI calls. This code retrieves hardware counters through PAPI, and generates event traces like the example trace above. This approach is guaranteed to perform the minimal amount of instrumentation needed for our purpose.

To measure overhead and skew we perform experiments using one of the NAS Parallel Benchmarks (NPB) [14]. The NPB are a suite of programs commonly used to assess the performance of parallel platforms. Each benchmark can be executed for 7 different classes, denoting different problem sizes: S (the smallest), W, A, B, C, D, and E (the largest). For example, a class D instance corresponds to approximately 20 times as much work and a data set almost 16 times as large as a class C problem.

We execute four different versions of the LU factorization from the NPB suite, all compiled with the highest level of compiler optimization. The first version is the original benchmark augmented with two calls to PAPI inserted at the beginning and the end of the LU computation to measure the total number of executed instructions. Since the overhead and skew due to these two calls are negligible, we call this version "original". The second version is called "TAU-full" and corresponds to the benchmark instrumented using TAU with default configuration settings. The third version is called "TAU-reduced" and corresponds to the benchmark instrumented using TAU but enabling instrumentation exclusion to reduce overhead and skew. The fourth version is called "minimal" and corresponds to the benchmark instrumented using our own method with wrappers on the PMPI interface. We execute all versions on the same cluster, called *graphene*, that comprises 144 2.53GHz Quad-Core Intel Xeon x3440 nodes. Each core has a L2 cache of 2 MB. The nodes are spread across four cabinets, and interconnected by a hierarchy of 10 Gigabit Ethernet switches.

We compute the instrumentation overhead as the percentage difference in execution time between an instrumented version and the original version. For each MPI process, we compute the instrumentation skew as the percentage difference in total number of instructions between an instrumented and the original application. For the instrumented application this number is computed as the average of the numbers of instructions for all the CPU bursts.

Figure 2.23 shows the skew for various instances of the LU benchmark. Each data point is obtained as an average over ten executions and aggregates the skews measured on all the processes. As expected, the TAU-full instrumentation method leads the highest skew as it is the most intrusive methods. The induced skew ranges from 3.66% to 21.62%. As the number of instructions is a fundamental component in our replay framework, the higher the skew, the less accurate the simulations will be. In other words, our framework will simulate the instrumented application rather than the original application. The TAU-reduced and Minimal instrumentation methods greatly reduce the skew (on average by a factor of 3.69 and 8.97, respectively). The lowest skew is achieved by the Minimal instrumentation method. It is always under 5% and in most of the cases under 2%. The highest skew caused by the Minimal

instrumentation method (4.76%) is obtained for the B-128 instance. In this particular case a relatively small input data is distributed among 128 processes, so that each process holds only less than a hundred kilobytes of data. As a result each process performs a small volume of computation, meaning that the instrumentation instructions represent a non-negligible fraction of the executed instructions.

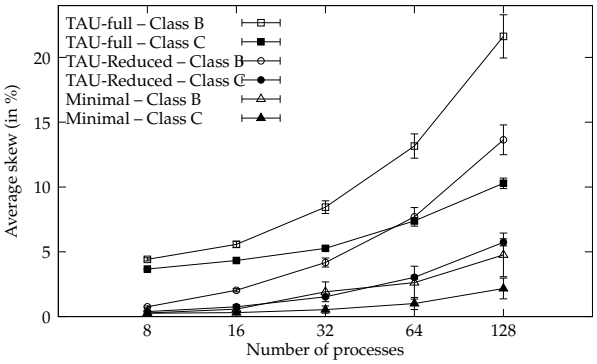


Figure 2.23: Instrumentation skew for the three instrumented LU benchmarks.

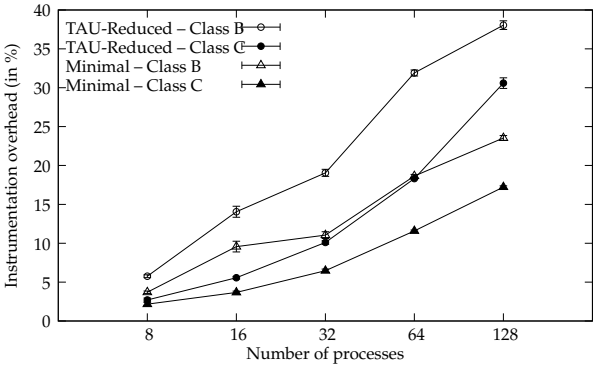


Figure 2.24: Overhead for the reduced and minimal instrumentation methods.

Figure 2.24 shows the overhead in terms of execution time induced by the TAU-reduced and Minimal instrumentation methods for various instances of the LU benchmark. The TAU-Full instrumentation method is not displayed here, as its high skew discards it as a valid choice for our off-line simulation framework. As for the skew, we see that the Minimal instrumentation we implemented reduces the overhead (on average by a factor of 1.6) with regard to the TAU-reduced method. While this figure indicates that the overhead can become large (up to 23.5% for the B-128 instance), in general it remains low. Indeed, the overhead induced by the Minimal instrumentation method ranges from 1.55 to 2.01 seconds for class B instances and from 0.8 to 2.6 seconds for class C instances. But as the original execution time of the application greatly decreases as more processes are used, the relative overhead increases. Since large number of processes are generally used to solve only large problem instances, unlike the class B LU benchmark, we conclude that the instrumentation overhead is well within acceptable limits.

Once the application has been instrumented, it has to be executed to obtain the desired trace. There are only two requirements for obtaining a valid time-independent trace in a view to simulating an MPI application with n processes on an arbitrary platform: (i) the instrumented application must be executed with n processes; and (ii) each process must fit in main memory. This is in sharp contrast with time-dependent traces, which must be obtained on homogeneous platforms with as many compute nodes as that in the simulated platform. In particular, our method makes it possible to execute the instrumented application in four ways:

Regular mode: execution on a single homogeneous cluster with one MPI process per processor. This is the way in which off-line simulators obtain time-stamped traces. As discussed earlier, this mode requires as many processors as that in the platform to be simulated, which limits its scalability. All three modes hereafter are only applicable to time-independent traces.

Folded mode: execution on a single homogeneous cluster with more than one MPI process per processor. This allows for the acquisition of traces for larger instances of the application than can be executed in regular mode on the same cluster. The folding factor is only limited by the available amount of memory on the processors.

Composite mode: execution on heterogeneous or multiple non-identical clusters. In this mode, the user can aggregate disparate processors together, such as those in multiple homogeneous clusters,

so as to augment the scale of the trace without requiring a single (large) homogeneous cluster. The only constraint is to select processors of a same family to prevent inconsistencies in the execution.

Composite and folded mode: a combination of folded and composite mode. This mode further increases the scalability of trace acquisition by executing multiple MPI processes per processor in a non-homogeneous platform.

Experiments conducted in [DMQS11] showed that the time needed to execute the instrumented application increases with the folding factor, which is expected as several processes compete for a single CPU. However, we saw that the execution time is increased by a factor smaller than the folding factor. Moreover composite execution on two sites, separated by more than 400 miles, requires roughly twice as much time as that in the regular execution for a highly communicating application. The increase in execution time is thus not surprising and commensurate to the number of sites. Finally we observed that combining the composite and folded acquisition modes (using two sites) does not lead to a simple multiplication of their respective overhead. Indeed, folding processes reduces the amount of data transfers to be done across a Wide Area Network. We conclude that folded, composite, or a combination of these two modes can be used to obtain large-scale traces with reasonable trace acquisition times.

An interesting property of time-independent traces is exemplified by these experiments. A tracing tool such as TAU will produce traces with some erroneous timestamps for most scenarios, due to external load on the system and or transient operating systems behaviors. A simulator using these traces would then predict an execution time close to that of the corresponding acquisition scenario instead of the targeted Regular mode execution time. Preventing such a behavior would require an accurate description of the acquisition platform along with the trace. With time-independent traces, the simulated time is totally independent of the acquisition scenario. Only slight variations (under 1%) are observed caused by hardware counter accuracy issue, and in fact a dedicated platform is not even required.

Depending on the chosen instrumentation method, an extra step may be added to the acquisition procedure right after the execution. The Minimal instrumentation that we proposed produces traces directly in the expected format, which is the main advantage of developing an *ad-hoc* instrumentation method. However, our first implementation based on TAU produced many files once the instrumented application completes. These files fall in two categories: *trace* files and *event* files. The generated trace files are named:

`tautrace.<node>.<context>.<thread>.trc`,

where `<node>` is the rank of the MPI process whose execution is logged in the file. The two other fields, *i.e.*, `<context>` and `<thread>`, are only used for multi-threaded applications. In this case, TAU distinguishes each thread and groups the threads according to the virtual address space they share.

A *trace file* is a binary file that includes all the events that occur during the execution of the application for a given process. For each event, this file indicates when this event (*e.g.*, a function call or an instrumented block) starts and finishes. The time spent and the number of computed instructions between these begin/end tags are also stored. For MPI events all the parameters of the MPI call, including source, destination, and message size, are stored.

To reduce the size of the trace files, TAU stores a unique id for each distinct traced event instead of its complete signature. The matching between the ids and the functions descriptions can be found in the *event files*. These files are named:

`events.<node>.edf`.

There is only one event file per MPI process. Each event file contains information about each traced function. For any function, an event file stores its numerical id, the group it belongs to, *e.g.*, `MPI` for all MPI functions, a tag to distinguish TAU events from those defined by the user, and the *name type* which is the actual name of the traced function. Some extra parameters required by TAU can also be stored into an event file. For instance, the keyword `EntryExit` is used to declare a function that occurs between two separate events. Conversely the `TriggerValue` keyword typically corresponds to a counter that

increases monotonically from the beginning of the execution. Such a trigger has to be activated twice to determine the evolution of the counter value during the corresponding period of time.

The following example shows two entries of an event file generated by TAU that corresponds respectively to the `MPI_Send` function and to the access to a hardware counter that measures the number of instructions.

```
1 49 MPI 0 "MPI_Send()  " EntryExit
2 1 TAUEVENT 1 "PAPI_TOT_INS" TriggerValue
```

Before replaying the target application in a simulation context, two more steps are mandatory. First we have to *extract* a time-independent trace from the trace and event files produced by TAU. Second we have to *gather*, and sometimes *merge*, the extracted traces on a single node where the replay takes place.

As the trace files generated by TAU are binary files, there is a need for an interface to extract information. Such an API is provided by the TAU library[15]. This tool provides the necessary functions to handle a trace file, including a function to read events. It also defines a set of eleven callback methods, that correspond to the different types of events that appear in a TAU trace file. For instance there are callbacks for entering or exiting a function and triggering a counter. The implementation of these callback methods is let to the charge of the user.

We developed a C/MPI parallel application that implements the different callback methods of the Trace Format Reader (TFR) library. This program basically opens, in parallel, all the TAU trace files and read them line by line. For each event, the corresponding callback function is called. To illustrate how the necessary data to produce a time-independent trace are extracted, we detail the case of a call to the `MPI_Send` function. Figure 2.25 presents the parameters of the different callbacks related to this function call on process 1 in a readable format. Each line starts by the process id, the thread id, the time at which the event occurred and the name of the event. The remaining fields are event dependent.

```
1 1 0 1.42947e+06 EnterState     49
2 1 0 1.42947e+06 EventTrigger    1  164035532
3 1 0 1.4295e+06  EventTrigger   46  163840
4 1 0 1.4295e+06  SendMessage   0 0  163840     1 0
5 1 0 1.4299e+06  EventTrigger    1  164035624
6 1 0 1.4299e+06  LeaveState     49
```

Figure 2.25: List of callbacks related to a call to the `MPI_Send` function.

As mentioned earlier, the event that corresponds to a `MPI_Send` is tagged as `EntryExit` in the event file with the event id 49. The first occurring callback will then be the `EnterState` event (line 1). The matching `LeaveState` event (line 6) defines the scope of events related to the function call. Four events are enclosed between these boundaries. Two of them (lines 2 and 5) correspond to the hardware counter measuring the number of instructions, as identified in the event file. These two events are used to respectively ends the computing burst preceding the MPI call and starts the next one. The number of instructions computed within a MPI call, mainly due to buffer allocation costs, are ignored as they are accounted for by the network model. The last two events are related to the sent message. The `EventTrigger` on line 3 only provides the size of the message (163,840 bytes), which is not enough to build an entry in the time-independent trace. The `SendMessage` event (line 4) gives more information, namely the process and thread ids of the receiver, the size of the message, and the MPI tag and communicator for this communication.

[15] `http://www.cs.uoregon.edu/research/tau/docs/newguide/ch06s02.html`

Thanks to all the information extracted from both TAU trace and event files, we can generate the following entry of a time-independent trace:

```
1 p1 send p0 163840
```

Note that for asynchronous and collective communications, the extraction process is more complex. For instance, the mandatory information to write the entry corresponding to a `MPI_Irecv`, *e.g.*, the receiver id, are given by a `RecvMessage` event which generally occurs within the `MPI_wait` function. This implies to implement some lookup techniques to retrieve all the necessary parameters.

As mentioned earlier, one issue with off-line simulation is the large size of the traces. This size directly depends on the number of actions executed by the processes. For applications that mix computations and communications, it usually grows linearly with the number of processes, since each process performs roughly the same amount of computation regardless of the number of processes (*i.e.,* the problem size is scaled with the number of processes). The size of the traces is also impacted by the data size handled by the application, which is also directly related to the number of actions performed. We measured an average number of 15 characters per action in the conducted experiments.

One challenge for trace acquisition is the time needed to aggregate (potentially large) trace files generated at a large number of compute nodes. This corresponds to a standard "gather" collective communication operation, and it is known that using a K-nomial reduction tree allows for efficient gathering of the files in $\log_{(K+1)} N$ steps, where N is the number of files and K is the arity of the tree. We developed a simple script to perform this operation, which can be made efficient by picking an appropriate K value given to the number of trace files and the number of compute nodes.

Design choices make that our trace format is not optimized for space. Authors have proposed compact trace representations [145]. In our case, one could devise a binary trace format that would remove most of the redundant characters. Alternately, one can simply use compression algorithms for our text trace files, which reduces trace file size significantly but not as much as if a custom compact trace format were used. When showing the scalability of our trace acquisition procedure in the next section we first present results without any compaction/compression, and then some results using compression.

Our off-line MPI application simulator is tightly connected to SIMGRID. Since version 3.3.3, released in August 2009, SIMGRID allows users to describe an applicative workload as a time-independent trace. Scenarios for the off-line simulation of MPI applications are more specific than the one described by Figure 2.1. As shown in the upper part of Figure 2.26, three input files are needed to replay such traces with SIMGRID. Apart from the *time-independent trace(s)*, descriptions of the *simulated platform*, in the format described in Section 2.4.1, and of the *deployment* of the application, *i.e.,* how simulated processes are mapped onto simulated processors, are also needed. The top left of Figure 2.26 describes a compute cluster that comprises four homogeneous machines interconnected through a switch, while the top right indicates on which node of the cluster each process will run. For instance, the MPI process of rank 0 will be executed on the node named `c-0.me`. These input are passed to the *trace replay tool* which, in turn, is built on top of the *simulation kernel* in SIMGRID. Decoupling the simulation kernel, and then the simulator, from the simulation scenario offers flexibility. A wide range of "what if?" scenarios can be explored without modifying of the simulator and instead simply changing the input files.

A necessary step for obtaining accurate performance predictions through simulation is the calibration of the simulation tool. In our context, calibration is used to determine the rate at which a CPU processes instructions and the latency and bandwidth of communication links. These values are then used to instantiate the platform description file. To determine the CPU's instruction processing rate we execute small application instances with only 4 processes on compute nodes as similar as possible or identical to those in the platform to be simulated. So although the simulation can run on any computer, we still require a small platform that's representative of the platform to be simulated, which may

be large-scale. Since the instruction rate can be impacted by memory hierarchy effects we run several application instances with different problem sizes to determine several rates. The instantiation of the network parameters is done by running the `Pingpong_Send_Recv` experiment of the SKaMPI [129] benchmark suite, executed between two nodes interconnected by network technology representative of the interconnect in the platform to be simulated. The latency corresponds to the time to send a zero-byte message, while the bandwidth corresponds to the peak bandwidth achieved when exchanging large messages. Once these values are obtained, it is of course possible to scale them so as to simulate various scenarios that deviate from the system used to conduct the calibration.

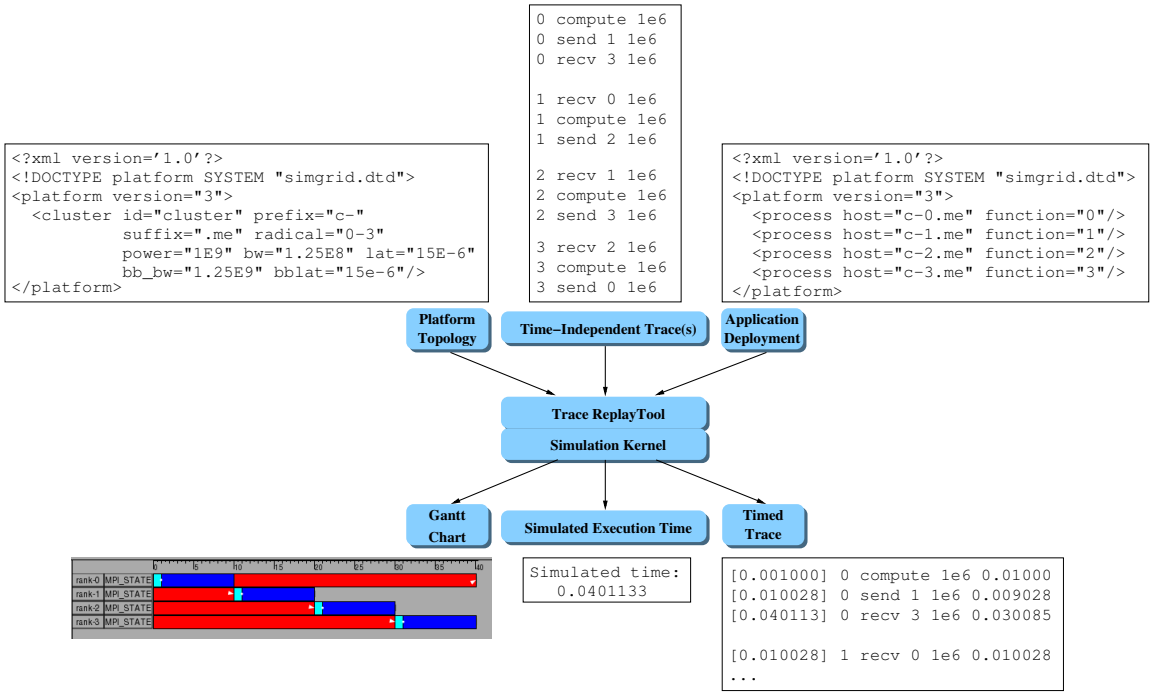


Figure 2.26: Inputs and outputs of the SIMGRID trace replay framework.

The *simulation kernel* is that of SIMGRID that was described in Section 2.2.2. For network resources, we rely on the model developed for SMPI which was specialized for cluster interconnects and takes into account the specifics of MPI implementations. This model was introduced in Section 2.6.1.

The mechanism needed to replay time-independent traces was first implemented using the MSG API provided by SIMGRID. This simulator has to:

1. Include a function that corresponds to the expected behavior of a given action. This has to be done for each action that occurs in the trace. Generally such function just calls one or several MSG functions. For instance, the code for the `compute` action is

```
1 static void compute(xbt_dynar_t action){
2   char *amount = xbt_dynar_get_as(action,2,char *);
3   m_task_t task = MSG_task_create(NULL,parse_double(amount),0,NULL);
4   MSG_task_execute(task);
5   MSG_task_destroy(task);
6 }
```

An entry of the trace file is passed to this function as a dynamic array (`xdt_dynar_t`) of strings,

one for each field of the entry. Once the amount of instructions to compute is extracted (line 2), it is possible to create (line 3) the corresponding SIMGRID task and execute it (line 4). This SIMGRID task is destroyed (line 5) as soon as the computation of `amount` instructions has been simulated.

2. Register this function with `MSG_action_register`. This call made in the `main` function of the simulator links the action keyword (as defined in Table 2.4) to the function defined in the previous step. For instance to link the `compute` keyword in the trace to the above function, the `main` function of the simulator includes the following call

```
1 MSG_action_register("compute", compute);
```

3. Call the function `MSG_action_trace_run` that takes either a trace file name or `NULL` as input. When no file name is given, this mean that there exists one trace file per process. In this case, the names of these trace files are given in the platform file, as shown below.

```
1 <process host="cluster-1.site.fr" function="p1">
2   <argument value="SG_process1.trace"/>
3 </process>
```

The first results presented in [DMQS11] were obtained using this first implementation with MSG. However, this design choice forced us to mimic the behavior of MPI primitives, *e.g.,* collectives operations or protocols depending on the message size, with crude simplifications. Moreover it decoupled this effort on the off-line simulation of MPI applications to that on on-line simulation also taking place within the SIMGRID project with SMPI. Then we have reimplemented in [DMS12] the replay mechanism directly within SMPI. Each call to a MPI function is associated to a small function that parse the parameters of the action in the trace and calls an SMPI function. Figure 2.27 illustrates this mechanism with the implementation of the *send* action.

```
1 static void action_send (const char *const *action){
2  int to = atoi(action[2]);
3  double size = parse_double(action[3]);
4  smpi_mpi_send (NULL, size, MPI_BYTE, to, 0, MPI_COMM_WORLD);
5 }
```

Figure 2.27: Implementation of the *send* action using the SMPI internal API.

From a user point of view, replaying a time-independent trace simply amounts to running the following program called `smpi_replay`:

```
1 int main(int argc, char *argv[]){
2  smpi_replay_init(&argc, &argv);
3  smpi_action_trace_run();
4  smpi_replay_finalize();
5  return 0;
6 }
```

This program, which initializes some data structures, loads a trace, and destroys the data structures, is a regular SMPI program. As such, it is compiled with the `smpicc` wrapper and launched thanks to the `smpirun` command as follows:

```
1 smpirun -np 8 -hostfile hostfile -platform platform.xml \
2         ./smpi_replay trace_description
```

where `np` and `hostfile` are classical command-line switches of the standard `mpirun`. The escription of the simulated platform is also provided in a file named `platform.xml` file. Finally, `smpi_replay` takes a single file as argument, called `trace_description` in the example command-line above. This file contains a list of all the names of the trace files associated to the MPI processes. If this file contains a single entry, all the processes perform the same (simulated) actions in a single trace file. Otherwise, each process performs the actions in its own trace file.

An off-line simulation can produce various types of output as indicated in the bottom part of Figure 2.26. In this work, we focus on obtaining a *simulated execution time*, which serves as a prediction of the execution time of the target application in the particular experimental scenario described by the platform and deployment files. It is also possible to generate a time-stamped trace that corresponds to this particular scenario by adding timers (measuring simulated time) in the trace replay tool. In the context of the ANR USS SIMGRID project, a complete and customizable built-in tracing mechanism has been added to SIMGRID by L. Schnorr. Thanks to it, on-line and off-line simulated executions of MPI applications can be traced either at the application level or at the kernel level. This tracing mechanism produces timed traces in the Pajé format[16] [142] and allows users to draw Gantt charts of the executions. More interestingly, relying on similar tracing mechanisms and the same output format may allow us to graphically compare both simulation approaches and actual executions of the considered applications. For instance, Figure 2.28 shows Gantt charts of the same application obtained by using SMPI, actually running the application with OpenMPI, and by replaying its associate time-independent trace.

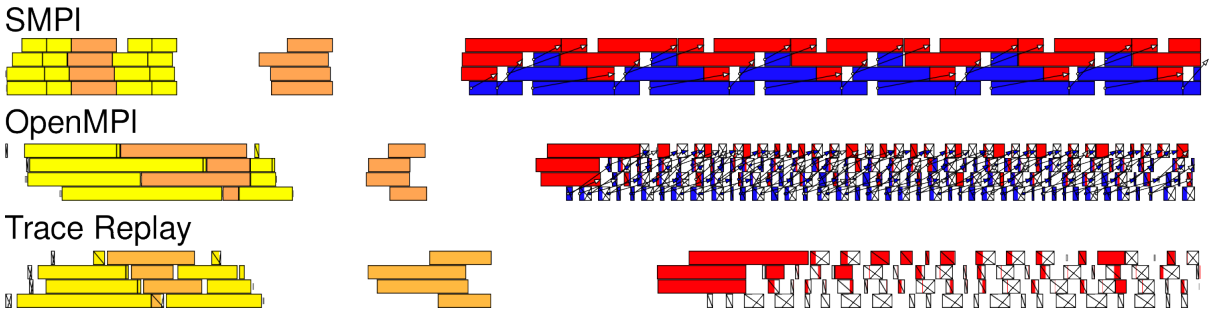


Figure 2.28: Comparison of timed traces of the execution of an MPI application obtained using SMPI, running OpenMPI, or replaying a time-independent trace.

We can see how different these Gantt charts are. It reminds us how difficult the accurate simulation of MPI applications is. Nevertheless, such a visualization tool helps us to identify what are the sources of difference between the executions. This cannot be done by only considering the final simulated time. Analysis of such graphs allowed the SIMGRID development team to better understand some of the subtleties of MPI implementations and to modify our implementations of both SMPI and trace replay tool accordingly. As Figure 2.28 shows, it is an on-going challenging work on which will be detailed in the conclusion of this chapter.

Finally it would also be interesting to derive a profile of the application from this timed trace, *i.e.*, a summary of the time spent by each function with regard to the overall execution time. This last kind of output requires complex analysis tools such as those develop in the TAU and Scalasca [76] projects.

We demonstrated in [DMQS11] and [DMS12] that time-independent trace and totally decoupling the acquisition and replay processes allow us to solve the typical scalability problem of tools for the off-line simulation of MPI applications. Having a homogeneous compute clusters at scale available is no more mandatory to study the behavior of large application instances. Heterogeneous and distributed

[16] `http://paje.sourceforge.net`

platforms to acquire large traces without impacting the quality of the simulation. Experiments showed a reasonable trace size, simulation time, and accuracy. However, improvements can be made in this trace replay framework which will be listed in the next sections as part of the outlook of this chapter.

To favor the use of this original off-line simulation framework, we wrote a manual [MS11] that describes step-by-step how to create an adapted Grid'5000 appliance. This full fledged system image comprises all the tools needed to acquire and replay time-independent traces from an MPI application.

2.7 Conclusion and Outlook

My different works related to the simulation of distributed systems and applications can be summarized by Figure 2.29. They all took place within the SIMGRID software stack or are related to what composes a simulation, *i.e.*, inputs and outcomes. The blue blocks represent components of a simulation in which I am/was a major contributor, while those in red indicate a secondary role in the proposed solutions. The gray blocks are either components for which little can be done, *e.g.*, logs or application deployment, or the lowest layers of the SIMGRID internal stack that I did not have to deal with, *i.e.*, SimIX and SURF.

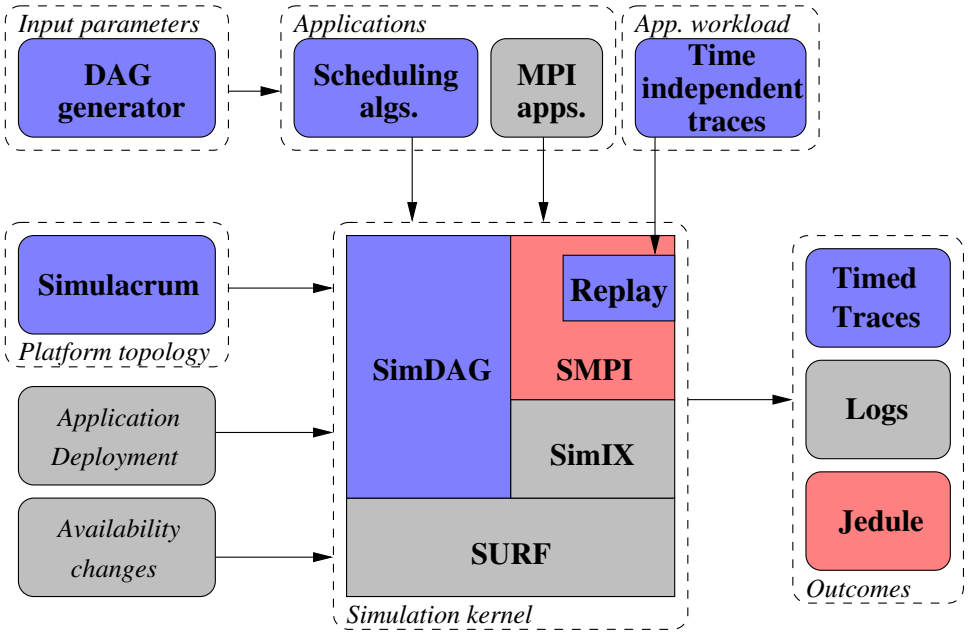


Figure 2.29: Summary of works in or around SIMGRID.

This figure has the same structure as Figure 2.1 that presented the components of a classical simulation environment. It shows that I contributed to the three main components that are the simulator itself, the simulation scenario, and the management of the results of its execution. More interestingly this figure shows how my relation to SIMGRID has evolved from the end of my Ph.D. The left-hand side of the figure corresponds to my first usage of SIMGRID for the evaluation of all the scheduling algorithms that were presented in Chapter 1. Improving my methodology of evaluation in this context brought me to propose sound generators of input parameters and help to propose tools for results analysis. Inevitably, I ended up by contributing directly on SimDAG to in turn help users to use it for their own research. The right-hand side of the figure corresponds to my most recent work on using simulation to help at the dimensioning of computing infrastructures. It covers the on-line simulation of MPI applications with SMPI, and the complementary off-line approach with the time-independent replay framework.

Each of these contributions to the SIMGRID ecosystem brings its own share of future work. About SimDAG and its usage, my efforts will first be focused on extending the API to offer more opportunities to users to develop their own simulators of scheduling algorithms. Among the envisioned additions to the current core functionalities are the management of computing resources according to the *unrelated* resource model, the exposition of the models of storage resources, *i.e.*, disks, that can only be accessed through the MSG API at the moment, and the usage of descriptions of IaaS cloud platforms for the scheduling of application workflows. In its current version, SimDAG, and actually MSG too, assumes a *uniform* model for computing resources. This means that for two tasks t_i and t_j and two machines p_u and p_v, if the execution time of t_i on p_u, or $t(t_i, p_u)$, is greater than that of t_j on p_u, then $t(t_i, p_v) > t(t_j, p_v)$. Moreover if $t(t_i, p_u) > t(t_i, p_v)$ then $t(t_j, p_u) > t(t_j, p_v)$. In other words, the execution time of a compute task is only defined by its amount of work to perform and the processing power of the machine it is executed on. However, an *unrelated* computing resource model is assumed in many scheduling studies that consider heterogeneous computing platforms as a target. In this model, the execution time of a task also depends on other characteristics of the machines that compose the execution environment. It is then possible to have $t(t_i, p_u) > t(t_j, p_u)$ **and** $t(t_i, p_v) < t(t_j, p_v)$. From a practical standpoint, this model may reflect phenomena such as the memory affinity of a compute kernel, or the use of different versions of a code that are optimized either for a CPU or a GPU. It is usually implemented by passing a matrix of precomputed execution times (whose size is the product of the numbers of tasks and processors) as an input of a simulator. A simple way to add an unrelated resource model to SimDAG would be to offer the capacity to the user to provide such a matrix as well as some functions to use its contents. Simple scaling techniques would then be applied on the amount of work of each task and the existing CPU model would be used. While this simple approach would meet the requirements expressed by most users, it is limited as it does not capture the interactions between tasks of different profiles when they have to share resources. Defining a proper resource sharing for the unrelated resource model, typically in the SURF layer of SIMGRID, is a challenging task. It implies to better understand how a given compute kernel behaves on a particular resource and how this behavior evolves from a machine to another. Furthermore more complex descriptions of both resources and tasks would be required. Improving (by making it more complex) the way computations are simulated is part of the roadmap of the development team of SIMGRID for the next few years. I will then contribute to it as a member of this team. The second extension of the SimDAG API is related to storage resources which were not modeled at all in SIMGRID before the 3.7 version released in May 2012. A basic model of disk and a subset of a POSIX API are now available. It allows a user to open, read, write, close, stat, and list files. But these functions are only available through the MSG API. This prevents SimDAG users to benefit from it, while such functions could be useful to study some scheduling algorithms. For instance, most scientific workflow management systems, such as Pegasus [54] or Taverna [90], generally use files and a shared file system to transfer date from one task to another. Without a storage model and the associate access functions, it is impossible to simulate and study the behavior of such tools in a realistic way. Users are forced to replace I/O operations by communications on the network, or consider that the I/O cost is included in the execution time of a task. This is what has been done in [DS10] for instance. But these workarounds do not allow users to measure the impact of the storage subsystem on the performance of their simulated applications. As models and API are already there, only little development is required to fill this gap and extend the range of scenarios that can be handled by SimDAG. This kind of transfer of features that already exists somewhere in the SIMGRID stack to SimDAG can also be applied to the simulation of IaaS cloud platforms. Ongoing development efforts concern these increasingly popular execution environments. Indeed a simulator of a cloud broker that mimics the usage of the Amazon Elastic Compute Cloud (EC2) has been proposed recently while a model of virtual machine is under integration within SURF. Unfortunately, they are currently limited to the Java bindings of the MSG API. Disposing of these abstractions in SimDAG would allow for the development of simulators, that would leverage all the strengths of SIMGRID, to study issues such as those raised in [111] on provisioning and scheduling for scientific workflow ensembles in IaaS clouds.

A complementary effort to these developments is to abate the learning curve of new users. This effort has already begun with the writing of a step-by-step tutorial[17] that describes all the concepts underlying SimDAG and details how to write a (simple) scheduling algorithm. Improving the currently available documentation by providing more commented examples is also a sometimes tedious but important task. Finally giving access to the source code of simulators that were used in my publications is another way to help new users to become familiar with SimDAG and develop what they need for their own research. First moves in this direction have already been done by making available the source code of the simulator used in [DS10][18] or developing a set of classical DAG scheduling algorithms[19].

I also plan to continue the efforts on the generation and diffusion of the different input parameters that compose a simulation scenario. As mentioned in Section 2.4.2, the generator of DAG that I have developed is publicly available[20]. To go further it would be interesting to collaborate with the developers of GGen [47] to make this tool fully usable with SIMGRID. The current focus of GGen is on the shape of the produced DAGs to ensure the generation of a representative set of application task graphs. But this generator lacks of a sound method to associate realistic values to the nodes and edges of a graph that would respectively correspond to amounts of work to compute and sizes of data to transfer. Considering both existing scientific workflow profiles [97] and information gathered from various MPI applications should allow for the proposition of a generator for such values. The resulting tool would then produce outputs similar to that of daggen but using thorougher generation models. In terms of platform topologies that are injected into simulators, I would like to revive an initiative I started in 2008. The objective of the Platform Description Archive (PDA)[21] was to provide a place where researchers can exchange descriptions of various large scale platform configurations used in their publications. This effort was inspired by archives of workloads, namely the Parallel Workloads Archive[22] and the Grid Workloads Archive[23], that were submitted to batch systems and computing grids respectively. In addition to the archive itself, the description format detailed in [FQS08] and the SIMULACRUM tool [QBS10] tool were proposed. Unfortunately, by lack of time, this archive was never truly populated. With the recent proposition of a new description format that is more scalable [BLM+12], a close relation with the technical committee of the Grid'5000 testbed, and my position in a large production computing center that is part of an even larger distributed computing infrastructure, there are great opportunities to obtain faithful descriptions of different testbeds and make them available to the community. Such an archive can also been as a step towards a better reproducibility of simulation experiments by allowing researchers to conduct studies based on execution environments used in previous publications.

The last future work related to SIMGRID addresses more challenging issues and will drive my activities for the next few years. The following threefold objective is to enable a production-grade usage of SIMGRID. First, I aim at strengthening the SMPI ecosystem to allow developers of MPI applications running in production to explore what-if scenarios and obtain performance assessments or find unexpected behaviors in a seamless way. Second, I plan to extend the simulation capacity of SIMGRID to hierarchical storage infrastructure, such as the one deployed at the CC IN2P3. Third, I aim at providing a realistic representation of a production infrastructure, from the interconnection topology to the dynamic behavior of compute nodes while including storage. The objective is to allow users of the modeled production infrastructure to study scheduling or deployment strategies in simulation and transferring them into production without experiencing completely different behaviors. These expected contributions, detailed in Section 3.3, will be the occasion to connect my research activities and a production usage of a distributed infrastructure. This topic is the core of the next chapter of this document.

[17] http://simgrid.gforge.inria.fr/tutorials/simdag-101.pdf
[18] https://github.com/frs69wq/biCPA
[19] svn://scm.gforge.inria.fr/svn/simgrid/contrib/trunk/DAGSched
[20] https://github.com/frs69wq/daggen
[21] http://pda.gforge.inria.fr/
[22] http://www.cs.huji.ac.il/labs/parallel/workload/
[23] http://gwa.ewi.tudelft.nl

Chapter 3

Bridging the Gap Between Research and Production

3.1 Introduction

In the introduction of the first Chapter of this document, I mentioned how different scientific fields, such as physics, biology, earth science, medical research, or even humanities, structure their applications as scientific workflows. This fact is only a consequence of how these disciplines changed their way to produce scientific results over the last few years. As stated by Jim Gray in the introduction of [88], sciences now resort on a *fourth paradigm*. Thousands years ago, science was *empirical*. Observing and describing natural phenomena were the first steps to understand them. Then, in the last few hundred years, science became *theoretical*. Models were proposed to describe these phenomena in a more general context. During the last century, the complexity of some of these theoretical models grew so much that they became too complicated to be solved analytically. Then, science became *computational* and people started to run simulations to get new scientific results. Nowadays, many sciences rely on the analysis and mining of huge amounts of data. Thus we talk about *data-intensive sciences*. As an illustration, we can cite biology and its DNA sequencing tools, physics with the LHC or large telescopes, medical imaging in which the picture resolution is always higher and higher, Computer Science with the recent spreading of sensor networks and the raising interest for interactions in social networks, or even humanities with the creation of digital archives. One of the main interest of this fourth paradigm is to unify the previous three paradigms to some extent. Instruments, or simulations, produce data that are treated by various software packages before being stored. This justifies the spreading of workflows even in disciplines that were not used to rely on numerical simulation or data analysis. Legacy codes and efficient software toolkits are now ordinarily combined to build a complete chain of digital processing from the acquisition of data to the production of a scientific result. Moreover, the produced data can be further analyzed and faced to theoretical models. The biggest challenge is now to determine which data in this deluge of information are the most interesting and worth of being kept and mined. Most of the research and development aiming at addressing this challenge are grouped together under the generic term of *Big Data*. As many generic terms, Big Data may have different meanings depending on the user community, the applications, or the kind of processing that are considered. In Section 3.2.2, I present some of the popular understanding of this term and position my research activities with regards to these definitions.

Regardless of the chosen definition, handling Big Data, or simply said large volumes of data, raises two common challenges: storing a lot of data requires a lot of disk and tape devices; and doing any processing on these data requires a lot of computing power. In both cases, the needs may exceed what

a single cluster or even the resources of a department can provide. The main consequence of the emergence, or generalization, of *data-intensive* sciences, is thus the ever more crucial need for large scale Distributed Computing Infrastructures (DCIs) to store and analyze the produced data. The generic denomination of DCI covers a broad spectrum of infrastructures that can be categorized according to the type of application they run and their main user communities.

For loosely coupled applications, *e.g.,* parameter sweep applications or (ensembles of) scientific workflows, the main objective is often to run as many instances as possible over a reasonable and/or constrained time period. For instance, part of the computing related to the data produced by the LHC relies on Markov Chain Monte Carlo (MCMC) methods, for which more runs lead to more accurate results. Another example is the Cybershake scientific workflow [81] designed to forecast ground motion in Southern California thanks to 3D wave propagation simulations. Executing a prioritized ensemble of instances of such a workflow allows physicists to cover a larger geographic area by investigating the most populated areas first. The amount of available resources then determines which part of the ensemble can be executed before a given deadline. Finally, a third category of loosely coupled applications derives from the Divisible Load Theory. The global amount of data to analyze can be divided in any number of independent chunks of arbitrary size. A famous example of such applications is the SETI@Home project that searches for extraterrestrial intelligence by analyzing radio telescope data. The common properties of these types of applications are that they usually deal with large amounts of data and the relative independence of the jobs or tasks they are made of. To execute such applications, a DCI thus has to favor the number of point of executions, *i.e.,* CPU or (virtual) cores, and storage capacity over the intrinsic performance of each computing or storage component to maximize *throughput*. Computing grids, *e.g.,* the Worldwide LHC Computing Grid (WLCG), and desktop grids, *i.e.,* composed of unused resources provided by personal computers connected to the Internet, are DCIs that correspond to this definition. This kind of application-DCI couple is well suited to benefit of the concepts of resource virtualization and leasing made popular by the emerging Cloud technologies [11]. Indeed, deploying the application along with its customized software environment thanks to virtual machines eases the exploitation of distributed heterogeneous computing resources. The capacity to offload a part of the applicative workload on public commercial cloud infrastructures is also an interesting feature that can be exploited by such applications. Then, there are more and more initiatives to shift from classical computing grids to (federation of) public or private Clouds. More details on cloud technologies and how they can be leveraged by applications running in production will be given in Section 3.2.1.

For tightly coupled applications, *i.e.,* parallel applications in which processes periodically exchange data during the execution, the main challenges are to minimize the impact of the parallelization overhead due to communications and to get "the best bang for the buck" from the computing resources, *i.e.,* being as close as possible of their peak performance. Among these high demanding applications, we can cite all the numerical analysis codes that rely on linear algebra kernels solved by one of the incarnations of the LAPACK library[8], be it using message passing [23], optimized for GPUs [140], or multi-cores[24]. Another example is the Dark Energy Universe Simulation (DEUS) project [4], that performed the largest N-Body simulation of the full observable universe. This application followed the gravitational infall of 550 billions particles in a 95 billion light-years box, *i.e.,* from the original Big Bang to the present day, to follow the evolution of Dark Matter particles in an expanding universe dominated by Dark Energy. Such a huge simulation required 10 millions hours of computing time and generated more than 50PB of data throughout the run. The key to performance for these tightly coupled applications is to use optimized computing nodes, *e.g.,* multi-core CPUs of the latest generation, GPUs, or many-core accelerators, and a low latency and high bandwidth network organized in a topology that minimizes the distance between pairs of such computing nodes. A supercomputer is thus the appropriate kind of DCI as it favors *performance* over throughput. The most powerful supercomputer in the world at the time of writing is the Tianhe-2 cluster[1] that comprises more than three millions cores de-

[1] http://www.top500.org/lists/2013/11/

veloping a performance of 33.86 Petaflop/s (10^{15} floating point operations per second) on the reference HPLinPack benchmark and a theoretical peak performance of 54.9 Petaflop/s (at the cost of an energy consumption of 17.8MW). Due to their unprecedented scale and the size of the application that can exploit them, these DCIs also face Big Data and storage issues. For instance, filling the Petabyte of memory of the Tianhe-2 cluster implies the existence of a very large disk storage capacity and huge amounts of manipulated data.

In the HPC field, *i.e.*, the domain of tightly coupled applications and supercomputers, the ever increasing requirements of the applications coming from various scientific domains, both in terms of computing and network capacities, drive many research activities in Computer Science. The resulting improvements of hardware and low-level software libraries usually induce a direct, and often seamless, benefit for production applications. However, the situation is different in the High Throughput Computing (HTC) world of computing grids and loosely coupled applications. Since the early reflections made in the European DataGrid project[2] (2001-2003) about how to set up a DCI shared across widely distributed scientific communities to handle the huge amount of data to be produced by the LHC, a clear cut occurred, in France, between the community of scientists that were going to use this infrastructure in production and that of researchers in Computer Science. The main difference of opinion was about the choice of middleware to operate the Grid and deal with data and jobs. The former was constrained to have something up and running as soon as possible to prepare the future exploitation of data. Then they advocate the use of the Globus toolkit [71, 72] that was the reference tool at that time. The latter was pressing to be patient and perform a clear requirement study and survey of the literature to avoid reinventing the wheel. The irony is that, at that time, I was hired to form the French user community of the Grid to the installation and usage of the Globus Toolkit, while, as a young researcher in Computer Science, I was not convinced by it to be the best option. The consequences were that, on the one hand, the French users and contributors of the production grid infrastructure were forced to follow design choices imposed by CERN. Their leadership and capacity to promote interesting developments that might benefited to the whole collaboration was then hindered. On the other hand, the research community lost a wonderful occasion to transfer its work to another discipline that strived for outstanding discoveries. While some success stories exist, such as Distributed Interactive Engineering Toolbox (DIET) [35] used in production for the Décrypthon project [17], the two communities that worked on Grids, *i.e.*, research and production, evolved in parallel and in a mutual disregard for almost ten years. However, for the last few years, and thanks to the efforts of some key people in both communities, bridges started to be established. My hiring as a researcher in Computer Science in a service unit such as the CC IN2P3, which is the main contributor to the French production grid, was part of this process. Part of my activities is then dedicated to ease the communication between research and production by making connections between actors on both sides, initiating collaborations, or promoting activities coming from one community among the other when I foresee a potential benefit or an opportunity for cross-fertilization.

In this chapter, I first define some key concepts that underlie the highly popular topics (a.k.a. buzzwords) that are Cloud Computing, Big Data, and High-(Performance/Throughput/...)-Computing in Section 3.2. Then, I detail two complementary approaches to bridge the gap that exists between research and production in the context of computing grids and clouds. The first approach, described in Section 3.3, consists in bringing SIMGRID up to a production-grade level so that it can be trustfully used not only by researchers in Computer Science that study computing grids, but also by scientists for other domains that use these computing grids in production. The second approach, presented in Section 3.4, will be to apply the results, or the acquired knowledge and expertise, coming from my research on scheduling and resource management to applications and environments used in production. Finally I conclude this chapter in Section 3.5.

[2] http://eu-datagrid.web.cern.ch/

3.2 Buzzwords and Key Concepts

3.2.1 Cloud Computing

Many researchers currently work on Cloud Computing, but what does Cloud Computing really mean? Miron Livny, the father of the HTCondor distributed computing system [144], once said "I've been doing research in *Clouds* before it was called *Grids*." This provocative quote shows how one concept of DCI has outplaced the other in most research activities while based on similar infrastructures and raising similar research issues. Beyond the provocation, it would be unfair to say that IaaS clouds are just grids with a commercial model (the famous pay-as-you-go model introduced by Amazon [11]). The main difference between grid and IaaS cloud infrastructures comes from the *virtualization* of computing and storage resources. In grids, at least on EGI, (almost) all the available resources run the same Operating System (Scientific Linux) and similar working environments, *e.g.,* libraries, are installed. Such a uniform configuration of the resources might be seen as a good way to simplify both the administration and usage of large scale distributed infrastructures. But, in fact, this creates an extra burden for both administrators and users. The former have to maintain an Operating System that may not be the most convenient from their system administration point of view. They also have to deal with several versions of the same Operating System from the moment when upgrades are decided, by an agreement at the European level, until all users have migrated their applicative stacks. This usually leads to long and cumbersome migration periods. The latter have to install and manage in user space all the software that is not available by default. This extra burden can distract the users from their main interest, which is to run production jobs and find new scientific results thanks to the computing infrastructure. The interesting feature of virtualization that underlies IaaS clouds, consists in deploying virtual machines that run fully customized system images, *i.e.,* Operating system, libraries, and working environment, on physical machines. Virtualization addresses the aforementioned issues related to the uniform setting of the grid. Users can configure, or be helped to, their working environment once for all, and system administrators can decouple the management of the infrastructure from the software stack(s) needed for production usage. However, cloud computing cannot be reduced to virtualization only. At the Infrastructure as a Service level, virtual machines can be instantiated, and stopped, in a dynamic way to adapt the infrastructure to the applicative workload. Then we talk about the *elastic execution* of applications, that has to be exploited by new scheduling algorithms to improve performance. Moreover, several virtual machines can be allocated to the same physical machine to fully use its resources thanks to *consolidation*. Virtual machines can also migrate from one physical machine to another, or be suspended and resumed, to adapt to dynamic changes in application requirements or react to hardware failures. These techniques aim at optimizing the energy consumption of applications and improving fault tolerance, and raise interesting research questions. Besides IaaS, cloud computing can also be declined at the Platform as a Service (PaaS) and Software as a Service (SaaS) levels. In a PaaS cloud, the underlying infrastructure, *i.e.,* virtual and physical machines, network, storage, and system images, is handled by a framework, *e.g.,* Microsoft Azure, or Google App Engine, and its management is hidden to the user. However, the user keeps control over the application and the customization of the execution environment. A PaaS framework also provides a set of tools to design, deploy, and manage applications. It can thus be seen as a development environment that specifically targets the underlying IaaS cloud. At the top of a cloud stack lies the SaaS level, that is the most seamless, and common, usage of a cloud computing infrastructure that a user can find. Access to applications of various complexity, ranging from email service or office suite to complex scientific codes, that are hosted somewhere in "the cloud", is made through a convenient web-based interface. Then the end user does not have to care about, partly because s/he does not even know, where the underlying applications are running, or where the processed data is stored. When compared to the grid ecosystem, the SaaS level corresponds to scientific portals that hide everything but a comprehensive interface to the non-expert user. But this ease of use comes along with non negligible trust, security, and confidentiality issues.

The main issue with the current activities on IaaS, PaaS, or SaaS cloud, be they in research, production, or even industry, is that despite the diversity of meanings and usages, everything often boils down to the single buzzword "Cloud". Sometimes this vocable is even used to cover techniques, such as virtualization, that underly the lowest level of the cloud stack, but cannot be designed as "Cloud". Such a reduction of a broad and diverse range of possibilities to a single term might create a certain confusion. The comic strip in Figure 3.1 illustrates this confusion. A non-negligible fraction of those people who praised Cloud Computing as the ideal solution to solve all the issues they currently encounter with Grid are alas often unable to develop beyond the generic (and somehow mysteriously magical) idea of "The Cloud" to specify which flavor would be the most beneficial to their usage. As solutions become more mature at each level of the cloud stack, such an aggregated and meaningless vision of the cloud tends to disappear. However, identifying matches between user expectations and the right incarnation of "The Cloud" is essential to a broader and effective adoption of this technological evolution.

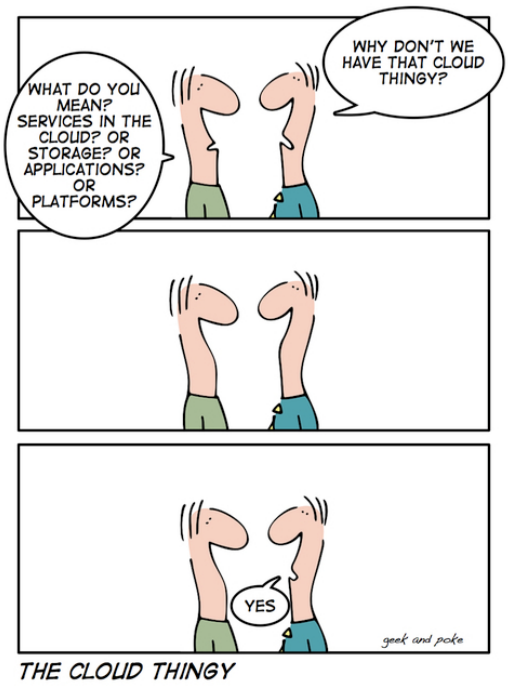


Figure 3.1: The awful truth about cloud computing understanding.

Another potential issue related to Cloud computing, when we consider both research and production points of view, is a certain risk of mismatch between what would be useful to users of production infrastructures and what are the hot topics in research. For instance, many users in various scientific disciplines would be very interested in high-level SaaS solutions, *i.e.*, domain-specific scientific gateways, whose development is not necessarily seen as a valuable research activity in Computer Science. Conversely, numbers of papers on scheduling are targeting public clouds, such as the one provided by Amazon. Such IaaS clouds allow for elastic executions that offer more freedom in the scheduling process, and add new metrics, such as the monetary cost, to optimize. While this leads to interesting research topics, it might, however, be unrelated to the practices of production users that are used to execute their workloads on sufficiently provisioned DCIs without any cost of operation (at least no monetary cost), and are thus less concerned by such research activities. Keeping production needs and research interests related is going to be crucial if we do not want to reproduce the same errors made at the beginning of the Grid computing era.

3.2.2 Big Data

Many researchers currently work on Big Data, but what does Big Data really mean? Dan Ariely, a psychology and economy professor at Duke University, once posted on his Facebook page "Big data is like teenage sex: everyone talks about it, nobody really knows how to do it, everyone thinks everyone else is doing it, so everyone claims they are doing it ...". As for Cloud, this other provocative statement somehow reflects a confuse situation created by the use of a single generic buzzword to cover a broad spectrum of usages and activities. Depending on the production method, the type of data, the processing, and the user community, the definition of what is covered by the generic term of Big Data greatly varies. For instance, handling and processing the huge amount of data produced by the LHC does not raise the same issues, and does not call for the same solutions as indexing web pages based on their contents, processing DNA sequences in near real time, or querying on Petabyte-scale databases.

A key element that is common to all these scientific challenges is the *volume* of data they have to deal with. However, volume is just one aspect in the definition of Big Data, and not necessarily the most important one. It is certainly not the aspect that allows one to capture the specifics of a given Big Data problem or solution. Then, two other aspects are commonly considered to refine the definition of Big Data in a given context: *variety* and *velocity*. Variety refers to data types that become more and more prevalent in modern science, such as images, text strings, entire documents, web logs, or even sound and movie files, but for which traditional relational databases are poorly suited. Velocity is about how fast data becomes obsolete, hence how fast it has to be analyzed to produce meaningful information. Taken together, these three "Vs" of Big Data were first introduced by Douglas Laney, an analyst from Gartner, on his blog on Big Data in 2001. A fourth "V", for Veracity, has then been added to emphasize on the necessity of the quality of initial data and the trust one can have on these data. This four "Vs" definition matches the most common understanding of Big Data that is implicitly associated to a third term: *Analytics*. Indeed, uncovering patterns and problems, or said differently mining information, in a data set that is created and aggregated very quickly and must be used swiftly, usually implies Volume, Variety, Velocity, and Veracity. This perfectly corresponds to the activities of the early pioneers of Big Data that have been the largest, web-based, social media companies, *i.e.,* Google, Yahoo!, and Facebook. Techniques and software designed for Big Data Analytics, such as MapReduce and Hadoop or NoSQL are well suited and gained a lot of interest in the last couple of years.

However, a broader evolution is the rise of data-intensive and data-driven applications. Not all the applications that deal with large amounts of data fit in the four "Vs" definition. For instance, the physics experiments associated to the LHC certainly produce data sets that imply both Volume and Veracity, but Variety is less important in this context. Moreover, the key to the discovery of new particles is not to replace data as they are produced after a new collision in the accelerator, *i.e.,* implying Velocity, but to accumulate them until the statistical interval of confidence becomes sufficiently small. Nevertheless, such applications lead to Big Data challenges, but they are different of those expressed by data mining applications. The size of the scientific collaborations accessing the produced data, the long life cycle of these data that exceed the already long period of their production, and the cost of end-to-end processing leading to a scientific breakthrough raise issues such as data distribution across dozens of large data-centers, data preservation, and data provenance. Furthermore the generalization of GPUs caused a dramatic increase of the number of cores in recent energy-efficient HPC supercomputers. This trend is not going to change on the road to Exascale. Then, the amount of data needed to keep all these processing units busy dramatically increases too. Here we have Volume and Velocity, *i.e.,* data evolves during the computation, but not Variety as HPC usually amounts to crushing numbers. Veracity can also be discussed as the probability of bug or data corruption is likely to increase with the size of both application and platform. This kind of applications also comes with its share of Big Data issues, such as optimizing network, storage, and memory I/Os, finding tradeoffs between transfers and replication, or ensuring correctness throughout an execution. These examples show that Big Data does not reduce to Analytics and that the meaning of this generic term strongly depends on the context it is used in.

3.2.3 HPC

Many researchers currently work on HPC, but what does HPC really means? The most commonly accepted, and most widely spread, meaning for HPC refers to the execution of large scale tightly coupled parallel applications on the most powerful supercomputers in the world, while getting "the best bang for the buck" of the most recent hardware, *e.g.*, multi-core CPUs, GPUs, many-core accelerators, or low latency/high bandwidth network interconnects, these machines are made of. But is not running hundreds of thousands of independent sequential jobs per day on a worldwide computing and storage grid to analyze tens of Petabytes of data, and eventually finding a new particle that really looks like a Higgs Boson, also a High Performance (in terms of) Computing?

The main differences between these two visions of high-yield computing are related to the structure of the executed applications and the objective functions to optimize. In HPC, one job is usually executed on N computing units, and I/O, *i.e.*, communications over the network, memory transfers, or accesses to the storage devices, have to be minimized to maximize the efficiency. Conversely, in HTC, N jobs are usually executed on one core each, while aiming at optimizing the overall throughput, *i.e.*, completing as much as possible over a given time period. However, some similarities can be found between typical HPC using supercomputers and HTC exploiting the distributed and aggregated power of the grid. In both cases an ever increasing number of computing units are involved to compute millions of billions of instructions per second, while consuming and/or producing huge amounts of data, that usually have to be routed through complex network topologies. Despite their distinct technical aspects, the HPC and HTC communities in Europe have adopted similar schemes for the management and distribution of the computing resources. In both cases, a hierarchical tiered organization exists that ranges from small to medium clusters located in laboratories to European federations, *i.e.*, the Partnership for Advanced Computing in Europe (PRACE) and EGI, through national computing centers, *i.e.*, the Centre Informatique National de l'Enseignement Supérieur (CINES), Institut du Développement et des Ressources en Informatique Scientifique (IDRIS), and Très Grand Centre de Calcul (TGCC) for HPC and the CC IN2P3 for HTC. Moreover, small scale resources located in laboratories are federated and mutualized thanks to national initiatives, *i.e.*, the Equip@meso project for HPC and France Grilles for HTC.

As explained in the introduction these two worlds have evolved in parallel, with no or only limited interaction, for more than a decade. This parallel evolution is less glaring in other countries, where HPC and HTC activities were more intricate than in France. In Germany for instance, some computing centers provides computing and storage resources to serve the needs of both communities in a single place. However, several factors speak in favor of more interactions. First, the perpetual race for performance in HPC, illustrated by the current Exascale challenge, drives the evolution of hardware and software in the computing field. In a matter of a few years, what is cutting edge material in HPC becomes mainstream and is found in the commodity clusters used in HTC. Then, it is not a complete surprise to see some major users of HTC resources facing the same limitations encountered by HPC users some years ago. Indeed, while the density of servers increases, with more and more cores per node, the amount of memory per core often remains the same, but with extra sharing-related issues. Then, some users have to request more cores than needed just to gain access to more memory space. Other users consider to parallelize their sequential codes to fully benefit of multi-core processors or GPUs, which are now common hardware components. For these reasons a transfer of expertise from HPC to HTC users would be highly beneficial for the latter. Second, the largest supercomputers now comprise more than a million of individual cores. A direct consequence is that the amount of data loaded, computed on, transferred, or stored on such machines also increased dramatically. As an example we already mentioned the DEUS simulation that generates more than 50PB of data throughout a run. HPC users are thus facing issues related to the management of huge amounts of data, which has become a daily routine in HTC with the success of the LHC experiments. There, the beneficial transfer of expertise has to be from HTC to HPC. We can see that mutual benefit and cross-fertilization are possible, but subjected to more interactions between the HPC and HTC communities.

3.3 Towards Production-Grade Simulation

The previous Chapter, which details the work I did on the simulation of distributed systems and applications since the end of my Ph.D., represents only a fraction of the impressive body of work that has been done on the SIMGRID project, by the core development team, over the last fifteen years. During this time period, SIMGRID evolved from a project developed in-house at a single laboratory into an international collaboration that involves more than 20 active contributors. It also evolved from a very domain-specific simulator into a *versatile scientific instrument*, whose performance and accuracy are continuously and thoroughly assessed, for the study of large scale distributed systems. This sustained evolution of SIMGRID was made possible by different factors: obtaining *recurring funding*, which provides manpower, *improving the code base*, which is accomplished via both custom and standard tools, and retaining and growing the *user base*. This quest for sustainability was detailed in [CGL+13].

The important financial support to the development of SIMGRID that is provided by the ANR through the SONGS project implied some commitments on the future of the tool. Indeed, the SONGS project has been selected as a *platform* project, which implies several expectations from the ANR in a view to foster the structuring of scientific communities on top of a technical infrastructure of common interest. More precisely, the proposed infrastructure, *i.e.*, SIMGRID, has to be generic, open to actors outside its community of origin, and sustained after the end of the funded project. The versatility of SIMGRID ensures its usability in various domains of application. However, most of the current users of SIMGRID are researchers in Computer Science, and more specifically researchers in the field of parallel and distributed computing, which is the community of origin of the tool. Efforts thus have to be made to make SIMGRID evolve towards a production-grade quality that is a necessary condition to be used by researchers at the interface between Computer Science and others sciences. The leads that have to be followed to ensure the sustainability of the development of SIMGRID have been mentioned earlier. Another option, which also calls for a production-grade quality, would be to start a service company to help industrial users to assess the performance of their parallel applications thanks to the accurate, validated, and scalable simulation tool that SIMGRID is. In this section, I detail three complementary actions to achieve the aforementioned objectives. First, in Section 3.3.1, I explain how to improve the realism of the input parts of a simulation scenario related to the description of the static and dynamic characteristics of production DCIs. Then, Section 3.3.2 details the required efforts on the simulation of MPI applications to improve usability and target developers of applications running in production, be they academic or from industry. Finally, Section 3.3.3 describes why extending SIMGRID with storage abstractions would be a valuable asset for production data-centers such as the CC IN2P3.

3.3.1 Realistic Simulation of DCIs for Scientific Experiments

As explained in the introduction of this Chapter, scientific applications coming from various domains require an ever increasing amount of resources to deliver results for ever growing problem sizes in a reasonable time frame. While large projects in numerical analysis, modeling, and simulation, that rely on tightly coupled parallel applications, were able to afford and leverage expensive supercomputers, other communities whose applications are more loosely coupled opted for commodity clusters. Following the example of the pioneering EGEE initiative, these individual and geographically scattered computing and storage resources are now commonly mutualized across European countries to form complex DCIs and address the challenges raised by large scale applications. Among the efforts, EGI is an established collaborative effort which involves more than 50 institutions in over 40 countries and is based on off-the-shelf components. Supercomputer centers followed a similar federated approach to share their users and workloads across Europe. The largest HPC machines are then grouped within the PRACE initiative[3].

[3] http://http://www.prace-ri.eu

Designing, optimizing, scheduling, executing, and understanding the performance behavior of scientific applications on such heterogeneous and geographically distributed DCIs are very challenging tasks. It usually implies to perform multiple optimization cycles that include code development, testing and debugging, real executions at scale, monitoring of these executions, data collection, performance analysis, and finally the definition of new optimization and tuning strategies. This complex process becomes especially cumbersome and time-consuming if not supported by appropriate tools. The most difficult part in this process certainly is the actual (monitored) execution of the application to optimize. What is already complex on a controlled, customized, and dedicated DCI such as the Grid'5000 experimental testbed, becomes even more complex on heterogeneous, dynamically evolving, and shared DCIs such as EGI. The external load generated by other users induces an important performance variability. Moreover, executions are usually non-deterministic and the infrastructure itself suffers from non-negligible reliability issues due to its scale. All these factors combined enforce the application developers to conduct repeated experiments to produce statistically relevant results and determine the right optimization strategies. However, repeatedly running the same application for optimization purposes is time and resource consuming (in both manpower and hardware) and in contradiction with the regular production usage of the infrastructure. It also limits the scope of investigation, *i.e.,* the experimental scenarios an application developer could explore, as time, resources, and also the energy to power them cannot be wasted for the pure sake of scientific curiosity.

To overcome these difficulties, computer scientists or domain experts, usually rely on mathematical models [78, 79], experimental platforms, or simulators [19, 29, 39], to reproduce the behavior of production DCIs under controlled conditions, which remains, however, challenging [85]. The SimgLite project, funded in 2010 by the Institut des Grilles du CNRS (IDG) and Aladdin/Grid'5000, was a first attempt to reproduce the behavior of a production DCI under controlled conditions. More precisely, the objective was to deploy a controlled execution environment on the Grid'5000 experimental testbed that relies on the gLite middleware used on the EGEE/EGI production DCI. Load and failures measured during executions on EGEE would be *modeled* and injected into SIMGRID to be *simulated* while some components of the gLite middleware would be *emulated* on Grid'5000. This ambitious project thus covered all the aforementioned techniques allowing researchers to reproduce the behavior of production DCIs. Unfortunately, there were no follow-up to this initial collaborative effort. However, it had the merit to bootstrap other initiatives on this important topic. Hereafter, I detail one of them that focuses on the modeling and simulation part.

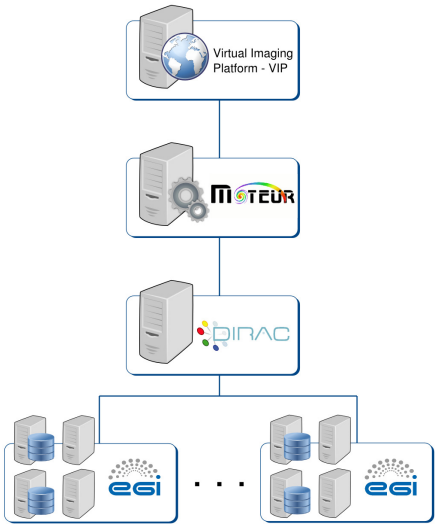
As illustrated in the previous Chapter, simulation is an interesting approach to explore thousands of experimental scenarios in a reasonable time. It also frees application developers and domain experts of the complexity and variability of the underlying infrastructures. Moreover, it allows simulation users to evaluate the behavior of applications and optimization strategies in situations that seldom occur on production platforms and are thus difficult to reproduce. However, a major drawback of most existing simulation tools is that they are not tuned to properly render of the specifics of production DCIs. Such a tuning of the simulation tool is of utmost importance to ensure that the performance indicators obtained in simulation, *e.g.,* the gain offered by a new scheduling algorithm, remain valid once the optimization is used in production on the real infrastructure. Then there exists a need for a framework enabling the lightweight exploration and evaluation of optimization techniques for scientific applications on real DCIs through extensive, reproducible, and, more importantly, realistic simulations before deploying them in production. Such a framework has to cover (slightly adapted versions of) the different components of a typical simulation environment that were listed in Figure 2.1. The production DCI is represented by the description of its topology and the characteristics of the computing and storage resources it is made of, as well as the dynamic execution conditions, *e.g.,* network background, startup times, or failures, it is submitted to. In a production context, the application part of a simulator has to distinguish the end-user application itself, *e.g.,* a parallel MPI application, or an embarrassingly parallel Monte-Carlo simulation, from the intermediate services offered by the middleware, *e.g.,* scheduling or

data management. These two components may each accept a different set of input parameters, which broadens the range of "what-if" scenarios that can be explore, hence increasing the number of generated simulation runs required to conduct a production-grade simulation study.

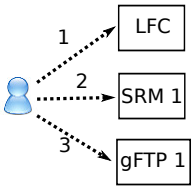
The SIMGRID toolkit constitutes the ideal starting point to build such a production-grade simulation framework. First, the components of a simulation listed above, *i.e.,* description of the platform, expression of the dynamic execution conditions, simulator of middleware services, and end user application, are all separated by design in SIMGRID. Then it is possible to bring each of these components up to a production level without compromising the usability of the whole framework. Moreover this eases the validation of the design choices, implementation, and impact on both performance and realism for each component. Second, earlier works in and around SIMGRID lay the foundations for the simulation of large scale production DCIs. In terms of platform representation, the multi-purpose format introduced in [BLM+12] and described in Section 2.4.1 is scalable enough to describe such platforms. For instance, the hierarchical description based on the concept of autonomous systems combined to the exploitation of regular patterns, *e.g.,* clusters, allowed us to describe the full Grid'5000 platform (10 sites, 40 clusters, 1,500 nodes) in a 61 KiB XML file. Moreover the SIMULACRUM [QBS10] and PDA initiatives ensure that descriptions of large scale production DCIs can be reused and made available beyond the initial study they have been proposed. The injection of dynamic changes in the availability of resources, including possible downtimes, by the means of external traces is a feature that exists in SIMGRID since its first releases. Thanks to the trace integration mechanism implemented at the SURF level [59], such traces can be used to mimic the impact of dynamic execution conditions without compromising the scalability of the simulation. Then, various kinds of grid and cloud services have been implemented on top of SIMGRID. Many scheduling algorithms were obviously proposed, as SIMGRID has been originally developed for this kind of simulation studies. But more complex services were also developed, such as a simulator of the Distributed Infrastructure with Remote Agent Control (DIRAC) pilot job system [36], resource management system [32], MapReduce tools [103], or data management services [132]. More recently, and in the context of the SONGS project, a cloud broker that simulates the usage of the Amazon public cloud has also been proposed [55]. Finally, the three API offered by SIMGRID to its users allows for the simulation of a broad range of applications.

For all these reasons, SIMGRID was used to design a first prototype of a simulator of a production DCI to study and improve the behavior of the scheduling algorithms hidden behind a scientific gateway dedicated to medical imaging [31]. This first study, despite its success, motivates the improvement of the simulation environment up to a production-grade level of quality. It highlighted a lack of realism caused by crude but forced abstractions. The objective was to simulate the execution of a Monte-Carlo application, called GATE, on EGI. This application is submitted through the Virtual Imaging Platform (VIP), which is one of most used scientific gateways to access the EGI resources. VIP currently accounts for more than 500 users from more than 50 different countries and serves around ten applications for medical simulation and neuroimage analysis. VIP describes applications as workflows executed with the MOTEUR [80] engine through the DIRAC pilot job scheduler. MOTEUR interprets the workflow representation and converts it into a set of tasks. These tasks corresponds to one or several jobs that are put into the DIRAC scheduling queue. The pilot jobs, launched by DIRAC on the EGI resources, then pull applications jobs from this queue. In addition to this stack of services that handle computations, other services related to data management also have to be simulated. Storage resources are accessed through a three-layer stack. Files are transferred using the gridFTP protocol, while a Storage Resource Manager (SRM) schedules file transfers on the available gridFTP backends. Finally, a central Logical File Catalog (LFC) stores information about file replicas and provides a single logical addressing space for distributed storage. Figure 3.2 summarizes the different components to simulate in this study.

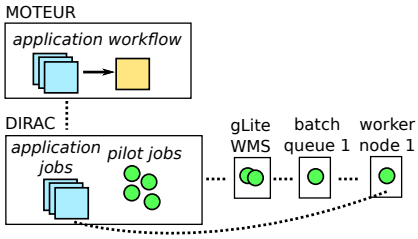
Figure 3.2(a) gives an overview of the VIP architecture. Figure 3.2(b) depicts the three-layer stack implemented to manage files and storage resources. Finally, Figure 3.2(c) shows how the applicative workload is handled by MOTEUR and then DIRAC once it has been submitted through the VIP portal.



(a) Architecture of the Virtual Imaging Platform.



(b) File transfer.



(c) Workload management.

Figure 3.2: Overview of the VIP architecture and associate services (© S. Camarasu-Pop).

Starting from the study in [31] and aiming at developing a production-grade and validated simulator, we propose to follow an incremental approach. A first level consists in relying on direct trace injection to evaluate models and compensate unexpected modeling issues. A second level extends the range of simulation scenarios that one can study and offers a better integration of the proposed developments within the SIMGRID ecosystem. Indeed, trace injection maximizes accuracy as exact values measured in the real experiment are passed to the simulator. To some extent, it corresponds to a perfect replay of a real experiment. However, this provides only limited insight as it prevents the study of alternative scenarios. Instead, the design of sound models leads to more flexibility and allows for extensive input testing, provided that models are properly validated to ensure simulation accuracy. Hereafter we detail the proposed methodology to build simulated versions of each targeted component, *i.e.*, platform description, dynamic execution conditions, services, and end-user application.

Platform description. Despite various attempts, there is currently no suitable description of production DCIs such as EGI for simulations. This is mainly due to the lack of detailed information on the configuration of its more than 300 distributed sites and the inter-site network connections. Gathering such information is particularly challenging because of its high variability. Then in [31], a slightly modified version of the description of the Grid'5000 platform was used as placeholder. A first step towards a more realistic description of a production DCI consists in extracting information about hosts, clusters, and network links and topology from execution traces. This will result in a set of descriptions of subsets of the target DCI. Then these sub-descriptions can be combined spatially and along time to obtain the description of a larger subset of the real platform along with its evolution over time. This requires to define how partial, and potentially conflicting, views have to be combined. For instance, host performance is likely to be defined as the most recent observed value, while network bandwidth, which is impacted by background load, should be defined as the maximum of all observed values. A comparison with information made available by the resource providers will be done to ensure that such a description built from execution traces remains sound and representative of the entire production DCI.

Dynamic execution conditions. It mainly consist in job queuing times and execution failures. In [31], such latencies, extracted from execution traces, were simulated using the availability and state file mechanism provided by SIMGRID. But resorting to traces presents some drawbacks. First it breaks the causal relation between the background load created by other applications that submit jobs to the platform, and the observed latency durations. Second, it conceals the relation between failure causes and failure times. To address these issues, a solution consists in collecting statistically significant sets of job latency and failure traces to draw probabilistic distributions. Then a simulator will randomly select values in these distributions. The impact of background load then becomes less predictable, hence more realistic. A more ambitious option would be to simulate not only the studied application but also the background load. For instance sets of jobs obeying to some workload models [109] could be injected in the simulator, and indifferently handled by the job schedulers.

Simulated services. The services that impact application performance are file management, workflow management, job scheduling, and resource provisioning. As shown in Figure 3.2(b), file management is simulated as a three-layer architecture with a logical catalog that maintains relations between logical file identifiers and their physical locations, a scheduling layer that decides when transfers start, and a transport layer that performs the actual transfers. As for job latencies, file transfers can be easily simulated by injecting durations extracted from execution traces. But this method is limited as it ignores network topology and file location. To address these limitations, file transfers can be simulated by the network operations they induce at the file catalog, scheduling, and transport layers. However, this approach ignores important components of the duration of a given file transfer too, *e.g.*, I/O transfers to the storage system, and hierarchical organisation of a data center combining disk and tape storage. A second step will be to include these components into the simulation framework for an increased realism. The workflow management layer can either be implemented with job submission timestamps extracted from traces, leading to an exact replay of a given workload, or by simulating the algorithms used by the workflow engine to orchestrate job submissions according to precedence constraints. This second approach has the advantage to mimic the behavior of the engine used in production and to allow for the resubmission of jobs in case of failures. Similarly, we could simulate the job scheduling process by enforcing the same mapping to resources as in a real execution or by mimicking the behavior of the scheduling component, *e.g.*, the DIRAC scheduler. Finally, the resource provisioning on a production DCI is often done by submitting pilot jobs to the batch queues of the different sites. Here we can reuse the latency models designed to describe the dynamic execution conditions. Then we can simulate the submission and scheduling of the pilot jobs that are treated as regular jobs by resource management systems. For this a tool such as SimBatch [32] would be a great asset from the SIMGRID ecosystem.

Applications. The applications, such as those submitted through the VIP portal are characterized by their workflow description as well as the execution time and volume of input/output data of workflow activities. Execution and data transfer times can be extracted from traces and injected in the simulator. This prevents the introduction of errors caused by an inaccurate platform description. Again, while this option is likely to be the most realistic, it is also the most rigid. To have more flexibility in the simulation scenarios, representative data volumes and computing activities have to be derived from the analysis of typical workloads and injected in the simulator.

Following this methodology will lead to a validated end-to-end simulator allowing to reproduce the behavior of executions on production DCIs and support the validation of new experimental methods before production deployment. Furthermore, all the simulated services, *i.e.*, file and workflow management, job scheduling, and resource provisioning, can be reused in other studies involving production infrastructures. Then, the developed generic methods for building DCI platform descriptions out of application traces have to be made available to researchers, thanks to the PDA initiative, for instance.

3.3.2 MPI as a Simulation

The work done in the context of the Ph.D. thesis of G. Markomanolis [DMQS11, DMS12] laid the foundations of an original framework for the off-line simulation of MPI applications. The interest of the approach based on time-independent traces has been demonstrated. The acquisition process is now mature with the proposition of a minimal custom instrumentation method and the documentation of the procedure to acquire a trace on a distributed infrastructure such as Grid'5000 [MS11]. It offers great perspectives in terms of scalability and might even be used beyond the original objective of helping at the dimensioning of computing infrastructures to predict the performance of parallel applications.

The effort on the on-line simulation of MPI applications allowed by SMPI, mainly supported by colleagues in Grenoble, led to great improvements. The piece-wise linear model introduced in Section 2.6.1 has been refined to obtain a *hybrid* model [BDG+13] that combines the strengths of both fluid and LogGPS [93] models. Fluid models, commonly used in the simulation kernel of SIMGRID, represent communications by *flows* that are simulated as single entities rather than as sets of individual packets. They belong to the broader family of delay-based network models which are faster and easier-to-instantiate than packet-level models. The LogGPS model is a extension of the seminal LogP model [49]. Its main characteristics are: the expression of overhead and transmission times as *continuous piece-wise linear* functions of message size; accounting for *partial asynchrony* for small messages, *i.e.,* sender and receiver are busy only during the overhead cycle and can overlap communications with computations the rest of the time; a *single-port model*, *i.e.,* a processor can only be involved in at most one communication at a time; and *no topology* support, *i.e.,* contention on the core of the network is ignored. The proposed model thus addresses the main drawbacks of the LogGPS model by accounting for factors such as network topology and contention. Another improvement concerns the automatic selection of optimized algorithms for collective operations. Popular implementations of the MPI standard, *e.g.,* OpenMPI and MPICH2, adapt the communication scheme to both data size and number of participating processes. Then most of the available algorithms for various collective operations have been implemented within SMPI to allow SIMGRID to mimic the behavior of actual MPI runtime systems.

The combination of these improvements on both off-line and on-line approaches raised the confidence in the simulation results up to a level that allows us to confirm or infirm performance measurements obtained with real experiments. This can be illustrated by the analysis of simulated and actual execution times of the Conjugate Gradient (CG) benchmark from the NPB suite. Figure 3.3 shows that for instances of the CG benchmark up to 64 processes, SIMGRID correctly predicts the execution time with an absolute error of at most 9%. However, when the benchmark is executed with 128 processes, the error dramatically increases to more than 25%.

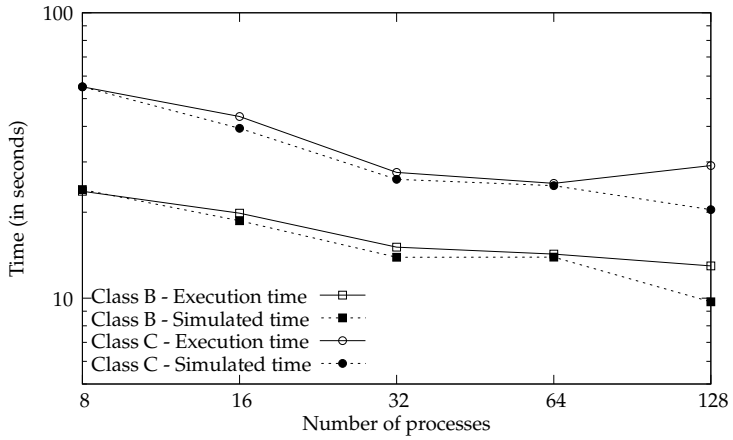


Figure 3.3: Comparison between simulated and actual execution times for the CG benchmark.

We investigated the cause of this large simulation error. As the 128 processes are scattered over the four cabinets of the cluster we used, we suspected a poor mapping of the processes to the compute nodes as the source of the inaccuracy. To confirm this suspicion we instrumented the benchmark with TAU to visualize the communication patterns. The actual execution time of the instrumented version is 14.4 seconds and the simulated execution time is only 9.9 seconds. The Gantt chart visualization for two seconds of a class B benchmark execution is showed in Figure 3.4.

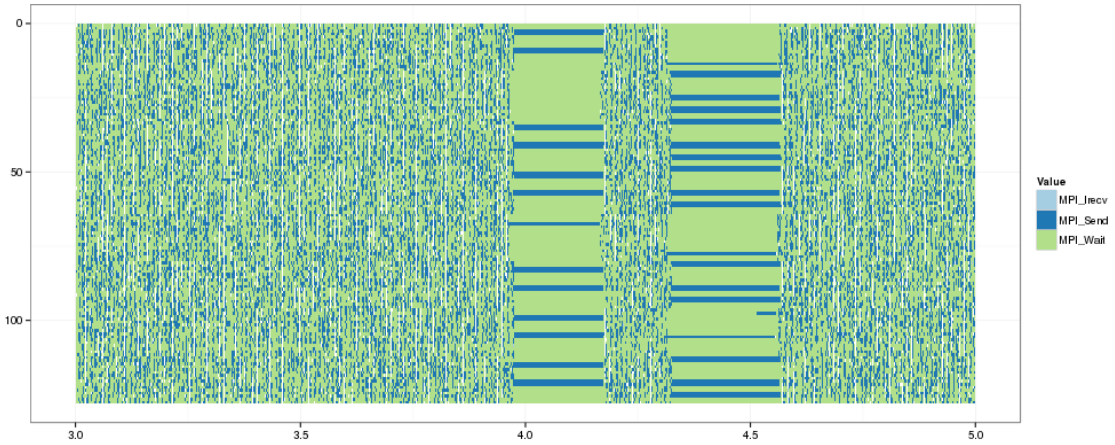


Figure 3.4: Two seconds Gantt-chart of the actual execution of a class B instance of CG for 128 processes.

Our suspicion was in fact incorrect, but the source of the inaccuracy is glaring. We see two outstanding zones of `MPI_Send` and `MPI_Wait` operations. These operations typically take from a few microseconds from up to less than a millisecond. In these zones, however, they take exactly 200 milliseconds. Our guess is that, due to high congestion, the switch drops packets and slows down at least one process to the point where it stops sending until a timeout of 200 milliseconds is reached. Because of the particular communication pattern of the CG benchmark, blocking one process impacts all the other processes. This phenomenon occurs 24 times leading to a overall delay of 4.86 seconds. Without this behavior, the real execution would then take approximately 9.55 seconds, which is much closer to the 9.9 second actual execution time (that is an absolute error of 3.54%). The same phenomenon was observed for class C executions of the benchmark. Such timeout issues could be similar to what is known as the *TCP incast problem* as observed in cloud environments [44]. These delays are linked to the default TCP re-transmission timeout, which is equal to 200 milliseconds by default in Linux. Although such value has recently been decreased from one second to 200 milliseconds to adapt with recent evolution of Internet characteristics, it remains inadequate for a compute cluster. Thanks to the quality of the off-line simulation of this benchmark allowed by SIMGRID, we were able to highlight such a protocol collapse, which would clearly have to be fixed.

Despite these encouraging results about the accuracy and predictive value of off- and on-line simulation of MPI applications with SIMGRID, some research and development efforts are still required to allow for a production usage of the proposed framework. By production usage, we mean the capacity to assess both the performance of any MPI application under various experimental conditions and the behavior of a given computing infrastructure when executing a set of well-defined benchmarks. The required efforts concern three distinct parts of the framework: the calibration of the hybrid network model; the precise description of the target platform; and the calibration of the instruction rate of the compute nodes. The former two are tightly linked and primarily related to SMPI, while the latter only concerns the off-line approach.

The hybrid network model introduced in [BDG+13] that underlies SMPI is piece-wise linear and

partially derives from the LogGPS model. This means that it includes thresholds related to message size and essential parameters, *i.e.,* latency, bandwidth, and sender and receiver overheads. A realistic instantiation of this model requires to run a set of basic point-to-point communication experiments on a sample of the targeted infrastructure. Concretely, we have to run two "ping" and one "ping-pong" experiments. The ping experiments measure the time spent in the `MPI_Send` (resp. `MPI_Recv`) function by ensuring that the receiver (resp. sender) is always ready to communicate. The ping-pong experiment measures the transmission delay. To span a broad range of message sizes while avoiding sequencing measurement bias, the message size is exponentially sampled from 1 byte to 100MiB. Once results have been obtained for all these runs, a complex analysis can be performed to determine the values of the different components of the model. This essential analysis step has been documented to ensure its reproducibility[4]. However, it assumes that input data, *i.e.,* the results of the ping and ping-pong experiments, are already available. To enable a production usage of SMPI, we thus have to automate and document not only the analysis but also the experiments that are required to feed the model for a given cluster. This is a necessary condition for the adoption of SMPI by actual developers of MPI applications. Moreover, creating an archive of results obtained by the SIMGRID development team, along with documentation on the acquisition method, and the corresponding instantiation of the model would be a valuable asset. Not only it would allow the development team to ease the reproducibility of experiments, but giving access to such an archive to users would allow them to test their applications again a stock of configurations.

The experiments above only allow the model to determine the time to send a message between two machines regardless of network topology and contention. However, within a cluster environment the mutual interactions between send and receive operations cannot safely be ignored. The proposed hybrid model takes these phenomena into account by following a *multi-port* modeling approach. The communication capacity of a node is thus limited by the network bandwidth it can exploit. To quantify the impact of network contention on a point-to-point communication between two processors in a same cabinet, we artificially created contention and measured the bandwidth as respectively perceived by the sender and the receiver. We increased the number of concurrent bidirectional transfers up to half the size of switch interconnecting the nodes and measured the bandwidth on the sender (B_s) and receiver (B_r) sides. A single-port model would lead to $B_s + B_r = B$ on average since both directions strictly alternate. A classical multi-port delay-based model would estimate that $B_s + B_r = 2 \times B$ since communications would not interfere with each other. However, we observed that $B_s + B_r \approx 1.5 \times B$, which means that both models fail to capture what actually occurs. This bandwidth sharing effect is modeled enriching the simulated cluster description. Each node is provided with three links: an uplink and a downlink, so that send and receive operations share the available bandwidth separately in each direction; and a specific *limiter* link, whose bandwidth is $1.5 \times B$, shared by all the flows to and from this processor. This modification is not enough to model contention at the level of a large cluster that comprises several cabinets interconnected through 10Gb links. Experiments showed that these links also become limiting when shared between several concurrent pair-wise communications between cabinets. This effect corresponds to the switch backplane and to the protocol overhead and is captured following the same approach by describing the interconnection of two cabinets as three distinct links (uplink, downlink, and *limiter* link). The resulting topology for the *graphene* cluster is depicted on Figure 3.5 and can easily be described in a compact way within SIMGRID thanks to the hierarchical description format proposed in [BLM+12]. Again, documentation, automation, and archiving of experiments are also required here to reach a production-grade quality level.

The computation part of our off-line simulation framework also has to be modified to improve the accuracy of the simulated execution times. Because of the limited expressiveness of the platform description format and, more importantly, of the simplistic CPU model implemented within the simulation kernel, a compute node is currently assumed to process instructions (or floating point operations

[4] `http://mescal.imag.fr/membres/arnaud.legrand/research/smpi/smpi_loggps.php`

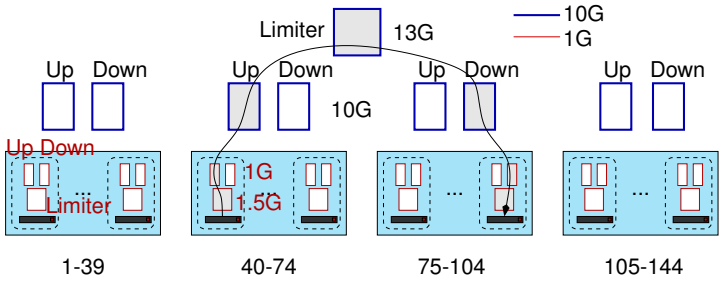


Figure 3.5: Modeling the *graphene* cluster: rectangles represent capacity constraints. Grayed rectangles represent constraints involved in a communication from node to node 40 to node 104.

depending on the chosen semantic) at a unique and fixed rate for the *whole execution* of an application. However, the type of executed instructions, their number, or how they access memory, have an impact on the rate at which they are processed. For instance, the first iteration of a loop may load data into cache while subsequent iterations experience high cache hit rate. Then a compute node should not be described by a single processing rate to reflect this observed behavior.

To illustrate this issue Figure 3.6 shows compute rates vs. number of instructions for all computation bursts as measured during a small 4-process execution of the LU benchmark from the NPB suite. The darkness of each data point is based on the number of compute bursts. We observe that compute rates vary by orders of magnitudes, and that the average value (depicted by a horizontal line) corresponds to few actual compute rates.

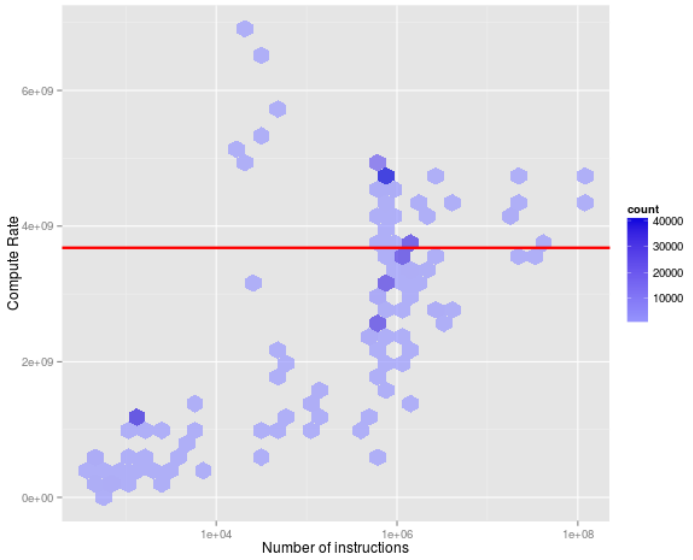


Figure 3.6: Distribution of compute rate (instruction/sec) vs. number of instructions for all compute bursts during a 4-process execution of the LU benchmark. Average rate shown as a horizontal line.

In general, we do not know the compute rate distribution of the compute bursts for an application execution on a hypothetical platform to be simulated. Ideally a trace should include information about the characteristics of each compute action to allow the simulation kernel to select an appropriate rate for its execution. But this implies to have a very precise performance profile of the simulated applica-

tion. Determining such a profile is a very complex and challenging task. It becomes even more complex in our context as we can obtain performance information only during the calibration step, *i.e.,* by executing a small instance of the target application executed on a subset of the target platform. A lot of investigations and experiments will be required to propose a great calibration and profiling method.

However, we already started to study a promising, but not perfect, approach. It consists in grouping compute actions according to their number of instructions within predefined intervals and then to affect a specific processing rate to each of these intervals. This approach does not reflect the fact that two compute kernels with very different profiles may process the same number of instructions at very different rates. However, it seems to be a interesting refinement to test with regard to the current solution based on a single average processing rate. A complementary approach would be to leverage some of the techniques used by ScalaTrace [117, 127]. To reduce the size of the traces it produces, ScalaTrace identifies patterns in the application code, *e.g.,* a `MPI_Send` occurring within a loop, and replace all the instances of a given pattern, *e.g.,* all the calls to `MPI_Send` made while iterating the loop, by a single entry in the trace during the execution of the instrumented application. Different methods are proposed to combine the execution delay of each instance into a single average value or a distribution of the measured delays. These techniques, pattern identification and delay combination, could be easily transposed to our framework to go further than a simple average processing rate. A interesting side effect would be to reduce the size of our time independent traces.

A last possible amelioration of our time-independent trace replay framework would be to increase the rate at which the traces are read and processed. Experiments made on execution traces of a LU factorization showed that our framework is currently able to replay between 90,000 an 125,000 actions per second. For very large traces, such as that of a class E instance of the LU benchmark executed with 16,384 processes, such a rate leads to prohibitive simulation time. Indeed, this trace contains around 70 billions of actions. Replaying it would require more than 8 days of simulation on a single desktop machine. However, neither the trace format nor the trace loading mechanism have been optimized for performance, then there is room for improvement. Moreover, current efforts in the SIMGRID project aim at enabling the distribution of a simulation of several compute nodes. The trace replay framework would be a good candidate to benefit of this capacity. Traces could be distributed on several disk to allow for concurrent accesses and the time to process the traces should be greatly reduced.

In addition to these developments, I aim at proposing a complete benchmark suite, based on off-line simulation. It consists in a collection of traces for various MPI applications augmented with a description of their specificities, a calibration procedure to accurately instantiate descriptions of simulated platforms, and an efficient execution and analysis framework to study of multiple "what-if?" scenarios.

3.3.3 Storage Dimensioning Through Simulation

Besides the need for computing resources, the tremendous increase in scientific data production and the ever-growing need for data analysis and preservation coming from data-intensive sciences create a great emphasis on storage components. Understanding the performance of a storage subsystem or dimensioning it properly are concerns independent of the scale and type of the associate computing infrastructure. Data centers, Supercomputers, Computing Grids, and Clouds all comprise storage components whose specifics may differ. Moreover these components may be combined in a hierarchical way to build a complex storage system. For instance, the CC IN2P3 IN2P3 Computing Center is one of the eleven Tier-1 data centers that store and process data produced by the Large Hadron Collider at CERN. Then it has to host a huge amount of data, *i.e.,* several Petabytes, that are accessed by physicists on a daily basis. The storage infrastructure is then composed of: (i) a tape library that provides large capacity at a low cost; (ii) a mix of Storage Area Network (SAN) and Network-attached Storage (NAS) on disks that acts as a cache with better performance of the mass storage subsystem; and (iii) a computing farm of around 1,000 servers that has their own disks. A fourth level made of Solid State Drives

(SSDs) could also have been added to benefit of the high throughput and low latency of such devices to further decrease the access time to certain frequently used data.

Devising the appropriate storage infrastructure for a given workload and defining sharing policies among users are complex tasks. Decisions are usually taken based on years of experience shared by system administrators and users. The former know how complex systems work while the latter have expertise on the behavior of their applications and express resource pledges. Nevertheless this process lacks of objective data about the performance of a given candidate infrastructure. Expertise is subjective and perceptions might be contradictory. An alternative would be to resort to simulation to obtain the expected objective indicators and compare candidate infrastructures on a fair basis without having to deploy them. Moreover, when a storage infrastructure is used in production, it becomes impossible to test optimization methods, *e.g.,* replication strategies or data placement related to popularity, without causing a disruption of the service. Here again, simulation is the alternative, provided that the target infrastructure can be modeled in a realistic way.

The initial motivation for adding storage simulation capacities to SIMGRID has been brought by colleagues at CERN and formalized in 2010 by the SimData project funded by the IDG and Aladdin through a joint call on *interfaces between research on grids and production grids*. The team, led by V. Garonne at CERN, develops and maintains the distributed data management system for the ATLAS experiment. This system handles petabytes of data made available to thousands of physicists. They aim at optimizing data placement and replication strategies to improve the overall throughput of ATLAS compute jobs, hence minimizing the user angriness metric. However, conducting any experiment on the infrastructure itself to test an optimization strategy would disrupt the production usage and almost inevitably results in unacceptable service degradations for the users. Resorting to simulation is then mandatory. In this context, storage components are modeled and simulated at a very coarse grain. A typical simulation scenario consists in moving or replicating data from a grid site to another. Each grid site, *i.e.,* a data or computing center with storage capacity, is seen as a black box, whose internal technical infrastructure is assumed as unknown. The only parameters that describe such a site thus are a total capacity and the input/output bandwidths that are experienced by the global data management system. These parameters are instantiated thanks to a huge amount of data that have been collected and analyzed at CERN. While the SimData project allowed us to initiate an interesting collaboration with promising outcomes for both teams, it was not enough to fully enable storage simulation within SIMGRID. Indeed, the lack of the fundamental concepts and models in the lower layers of the tool and the absence of a proper API to manage storage abstractions forced our colleagues to develop everything they needed in user space. In other words, a high level storage management simulator was produced but not properly integrated into SIMGRID. Since November 2012, with P. Veyre and other members of the SONGS project, we then started to add storage abstractions to the SIMGRID toolkit.

From a user point of view, storage corresponds to two basic concepts: a *storage space* on which *files* are stored. A file can be abstracted by its complete name, *i.e.,* that includes the absolute path in the file system tree, its size, and the storage space it is stored on. Indeed, other file-related information, *e.g.,* access rights, creation or last modification dates, are not to be handled by the simulation kernel. Then, if a user requires such information, s/he has to manage them in the code of his/her simulator. SIMGRID provides a mechanism of *properties* based on key/value pairs that can be associated to any resource and an API to retrieve these properties during the simulation. This mechanism can be extended to include files. For similar reasons, the proposed extension does not consider the contents of files, nor does it allow users to navigate in the file tree. The operations on files that are exposed to the users are then: *opening* and *closing* a file, *seeking* into a file up to a certain offset, *reading* or *writing* a certain amount of data, regardless of its meaning, from or to a file, and *moving*, *copying*, *renaming*, or *deleting* a file.

For a user, a storage component mainly corresponds to a space of a given *capacity* on which files can be stored in a persistent way. A storage space contains a *set of files* on which the aforementioned operations can be performed. It has a *name* and a *type*, ranging from a single disk to a share file system or a tape library, and is accessed from a *mount point*. The same storage space can be mounted by several

workstations, allowing for the sharing of the data stored on it. A compute node can also mount several storage elements, to simulate the use of different partitions for instance. The main operation related to a storage space in user space is to list its contents. The operations that impact the contents of a storage space, *i.e.*, creation, modification, or deletion of files, dynamically modify its *available* and *used capacities*.

Files are associated to storage components at the beginning of a simulation when the description of the whole platform is loaded. Typical data centers usually host several millions of files on their storage infrastructure. Then creating a simulated entity for every single file stored on a large scale distributed platform would lead to a prohibitive memory footprint and is thus hardly possible. The contents of a storage space is then considered as an inert list of files described by their full name and size, as already mentioned. A simulated entity is only created when a given file is opened and destroyed upon closing. We consider at least two description and access methods depending on the number of files that are stored in a given storage space. For small to medium amounts of files, using a *text description* is a simple and reasonable approach. For larger contents, a *database* whose entries would contain the same information as in a text file would constitute an easier and more scalable approach to manage the contents of storage systems.

From the simulation toolkit standpoint, a storage space is a *resource* as CPUs or network links are. Its usage is then defined by a *model* that describes the evolution of an action, *e.g.*, reading or writing a file, on this resource under certain constraints, *e.g.*, access throughput or replication algorithms. Such a model also defines how the resources have to be shared by concurrent actions to reflect the actual behavior of the simulated storage component. We envision several models in our implementation. Most of the concurrent simulation tools model the time to perform an IO operation by an affine linear function of the file size. A storage is then described by a set of latency and bandwidth couples of values respectively for read and write operations. A third value might be used to model the maximal connection bandwidth of the resource that is typically less than the sum of read and write bandwidth values. A simpler variant of this model consists in ignoring latencies. The validity range of this variant is reduced to large files, for which the IO time is important enough to hide the initial setup time. We also consider implementing a *piece-wise* linear model, similar to that proposed for TCP over Ethernet communications [CSG+11]. Indeed, the bandwidth experienced by an IO operation is likely to depend on the size of the accessed file. Such a model is harder to instantiate but would lead to better accuracy, without compromising simulation speed. Following the same lead, another way to describe the performance of storage components would be to provide *bandwidth matrices* as done by the authors of [105]. Values would depend on the number of concurrent accesses. Tape libraries are often managed by systems that hide the location of the files on tape to users (or to higher level software packages). Moreover several factors specific to such libraries may greatly impact the time to access a given file, *e.g.*, whether the tape is already mounted or not, the position of the tape in the library, the position of the file on the tape, or the load of the tape reader. Devising a deterministic and yet realistic model for this kind of storage would then be a too complex task, pleading for stochastic models that describe the time to complete an operation on a file by a distribution law.

Each of the aforementioned model description requires to be instantiated with sound values that are representative of the usage of the storage devices we aim at simulating. Instantiation is a complex and time consuming procedure. For single disks and disk bays managed by a file system, benchmarking tools such as IOzone [94] can be used to obtain the requested information. For tape libraries or more complex aggregations of basic storage components, we plan to conduct a statistical analysis of usage logs available on servers at the CC IN2P3 to feed the proposed models with realistic values. To ease off this burden on the users, we also aim at proposing a set of stock definitions of common storage components. They will be available as part of the set of examples distributed with the SIMGRID toolkit. We plan to provide instantiated models of various types of single devices, *e.g.*, hard disk, SSD, or tape, and aggregated devices, *e.g.*, shared file system, disk bay, or tape library.

We added the necessary support in the SURF and SIMIX layers to handle files and storage components and extended the MSG API with functions corresponding to the aforementioned features. We

defined two new structures, `msg_file_t` and `msg_storage_t`, that represent the user view of files and storage spaces respectively. Sizes and capacities are expressed as `sg_size_t` which corresponds to 64-bit unsigned integers. Figure 3.7 gives an example of the declaration of storage components in the XML description format provided by SIMGRID. A *storage type* that corresponds to a single hard drive disk, whose capacity is of 500GB, is first declared (lines 3-8). Storage elements of this type will be simulated according to the `linear_no_lat` model that ignores access latencies and determines the time to read (resp. write) a file by dividing its size by the read (resp. write) bandwidth `Bread` (resp. `Bwrite`). If read and write operations occur simultaneously, the bandwidth they are allocated will be limited to the value of `BConnection`. Contents are attached to this type of storage, meaning that it will be inherited by all the declared instances (line 4). The list of files is given in a the `contents.txt` text file and follows a UNIX syntax for paths. Then two instances, named `Disk1` and `Disk2`, are declared (lines 10-12) and respectively mounted on `/home` by the `alice` and `bob` hosts (lines 14-20). Note that the contents can also declared in the `<storage>` tag, overwriting that of the `<storage_type>` tag.

```
1  <platform version="3">
2    <AS id="AS0" routing="Full">
3      <storage_type id="single_HDD" model="linear_no_lat" size="500GB"
4                    content_type="txt_unix" content="contents.txt" >
5         <model_prop id="Bwrite" value="50Mbps" />
6         <model_prop id="Bread" value="100Mbps " />
7         <model_prop id="Bconnection" value="125Mbps" />
8      </storage_type>
9
10     <storage id="Disk1" typeId="single_HDD"/>
11     <storage id="Disk2" typeId="single_HDD"
12              content_type="txt_unix" content="contents2.txt" />
13
14     <host id="alice" power="1Gf">
15       <mount id="Disk1" name="/home"/>
16     </host>
17
18     <host id="bob" power="1Gf">
19       <mount id="Disk2" name="/home"/>
20     </host>
21   </AS>
22 </platform>
```

Figure 3.7: Example of storage description in the SIMGRID format.

Our development roadmap is driven by a concrete use case: developing a simulated version of the data storage infrastructure currently deployed at CC IN2P3. This infrastructure is hierarchical and complex both in terms of hardware and software components as shown in Figure 3.8. The huge amount of data from physics experiments to store, *i.e.*, tens of Petabytes of data, imposes to resort to high capacity mass storage. Data are then kept on magnetic tapes stored in robotic libraries managed by the High Performance Storage System (HPSS). As tapes induce prohibitive access latencies, an additional disk-based cache level of smaller capacity (around 10PB) lays between the tape libraries and the local disks on compute nodes. The data stored on this disk storage layer are accessed and analyzed by multiple scientific collaborations. Each of them usually has its own preferred file format, access method, and usage pattern. This leads to various cache management systems at the disk level: dCache [73], XRootD [61], and iRODS [114]. These tools are all interfaced with HPSS through the Tape Request Scheduler (TReqS) [133] that, in turn, make calls to HPSS through the Remote File IO (RFIO) API.

While such a complex and hierarchical infrastructure raises many challenges, it also allows for an incremental bottom-up development. First, we aim at designing a simulator of HPSS and its RFIO access API. Second, we will develop simulators of the available disk management systems on top of the HPSS simulator. Third, we will plug a simulator of TReqS between disk managers and HPSS to assess its benefits and study novel scheduling heuristics. Achieving this goal goes through several milestones. First, we have to write an accurate description of the physical organization of the storage

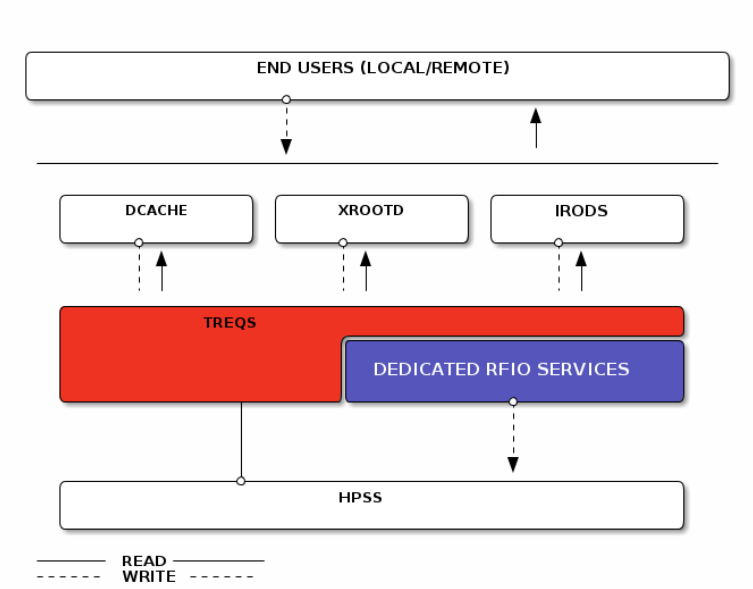


Figure 3.8: Overview of the hierarchical storage infrastructure of the IN2P3 Computing Center.

infrastructure, *e.g.*, robotic tape libraries, disk storage space, network interconnection. Second, we have to analyze usage logs to understand and model the behavior of storage elements at each level of the hierarchy, and then instantiate the described infrastructure. Third, we have to add these new models to the simulation kernel. Finally we have to design, develop, and validate the different simulators. We aim at following a thorough methodology for the whole process.

This is the performance, or access bandwidth, of the upper layer of that storage software stack that is seen from outside the computing center and simulated in [105]. Eventually, we would like to compare such performance information that hide all the internal details, to those obtained through a model of the complete stack.

In parallel of this work that extends the lower layers of SIMGRID and makes a heavy use of the MSG API, we plan to expose storage and file abstractions in the SimDAG API to allow for the simulation of data-driven scientific workflows.

3.4 Applying Research in Scheduling to Production Applications

The second approach to bridge the gap between research and production is to take advantage of the experience and expertise I acquired since the end of my Ph.D. to apply results of my own research, either directly or by adapting them, to problems raised by actual production applications. My unique position as a researcher in Computer Science in a computing center devoted to the service of many challenging scientific experiments is a great opportunity for achieving this objective. Since 2008, I aimed at initiating connections between researchers in Computer Science and users of the CC IN2P3 that lead to formal collaborations. But most of the applications currently served by the computing center comes from the LHC experiments. They are well established and typical representatives of loosely coupled, embarrassingly parallel applications. Then there was almost no opportunity to transfer advanced scheduling techniques, generally addressing parallel applications or workflows, to production in this context. However, the next generation of physics experiments, that will effectively start to produce data by the

end of the decade, are of a different kind. Most of them are astroparticle physics applications that are associated to ground telescopes, such as the Cherenkov Telescope Array (CTA) or Large Synoptic Survey Telescope (LSST) experiments, or spatial telescopes, such as the EUCLID project. The common characteristic of these experiments is to rely on a computing model which is very different from that of the LHC experiments. Among the foreseen methods to analyze the data produced by these instruments are executions on GPUs and many-core accelerators, large scale databases, and the use of the MapReduce programming paradigm. Consequently, the CC IN2P3 should see its operation evolves from a predominant HTC mode, with the execution of a large number of independent sequential jobs, to dealing with a more HPC workload made of parallel or orchestrated computations. These applications thus have specifics that are closer to my domain of expertise. This constitutes a great opportunity to reconcile my research interests and activities that the Institut National de Physique Nucléaire et Physique des Particules (IN2P3) can benefit of. Acting more upstream within large physics collaborations, at a time when computing models and technological choices are still under discussion, has two advantages. First, participating to the expression of the requirements, in terms of computing and data management, made by the applications makes it possible to analyze these needs in light of what is, or can be, done in research on related topics. Second, at this level it is possible to influence the decisions in a way such that the problems that arise can be matched to solutions that are not necessarily ready to be applied in production. More concretely, the CTA and EUCLID experiments are considering using Hadoop[5] and more generally MapReduce to handle part of their productions. This programming paradigm and the related scheduling issues are part of my research activities as mentioned in the conclusion of Chapter 1. Then, the LSST experiment will not only have to manage Petabyte-scale databases but also to take advantage of GPUs to process images of the sky. There again, research in scheduling can be applied to optimize the performance of data processing. However, to achieve this goal it is essential to interact more actively with key people and committees within the IN2P3. Success is not guaranteed, but the challenge is nevertheless exciting.

Alternate, and more secure, opportunities exist to apply the results of my research to production applications and environments. I propose to start this effort with two concrete use cases that are detailed in the next two sections. The first option is to consider an application supported by a smaller, and thus more accessible, established collaboration. In section 3.4.1, I describe how to apply workflow optimization techniques to an astroparticle physics application. The second approach consists in transferring results from research to production by considering not an application, but a core tool of almost any production infrastructure. In Section 3.4.2, I explain how to adapt the resource management system to a specific workload such as that of the CC IN2P3 in a way to optimize performance metrics beyond the classical throughput.

3.4.1 Workflow Optimization for the Detection of Supernovae

The Nearby Supernova Factory (SNfactory) is an astroparticle physics experiment that aims at measuring the expansion history of the Universe and explore the nature of Dark Energy thanks to the detection of *Type Ia supernovae* [3]. SNFactory is an international collaboration between several groups in the United States and France, whose major partners are the team of S. Perlmutter at the Lawrence Berkeley National Laboratory (LBNL) and the team of G. Smadja at the Institut de Physique Nucléaire de Lyon (IPNL). The experiment is designed to address a wide range of supernova issues using detailed observations of low-redshift supernovae. The underlying scientific workflow looks for supernova objects in pictures of the sky taken by a telescope in Hawai'i. It is composed by a set of legacy astronomy codes orchestrated by in-house scripts and run on the resources of the CC IN2P3. Such a crafted way to execute a workflow might be sufficient to obtain the expected scientific results provided that the execution is performed on the computing resources of a large and very reliable data center, and that performance,

[5] `http://hadoop.apache.org`

i.e., getting the results as fast as possible, is not the main objective. However, this orchestration method fails to provide comprehensive feedback to physicists when some of the jobs that compose the workflow do not complete. Moreover, the aggregation of subtasks has been empirically determined based on the experience of users more interested in feasibility than performance. Then, it lacks of flexibility and has not been optimized to get the best performance from the computing resources.

The main objective of this collaboration is to redesign the execution of the SNFactory production using a more classical scientific workflow description and relying on a proper workflow engine. The first step consists in deconstructing the existing coarse-grain workflow to express the entire execution as a chain of fine-grain computations and data transfers. Then all the empirical aggregation disappears, which leads to more opportunities for scheduling decisions. We decided to describe this new workflow in the format of an existing workflow engine to prevent unnecessary development efforts. We selected the Pegasus workflow engine, developed at the Information Sciences Institute of the University of Southern California [54]. The rationale, beyond existing contacts with the Pegasus team, was that one of flagship application of this tool, named Montage [21], has similar characteristics in terms of input data and type of processing to the SNFactory experiment. Indeed, both applications take pictures of the sky as input and then apply several filters and composition operations to produce the desired result. Another motivation comes from the workflow description format itself, the DAX format that can be parsed by SIMGRID, as explained in Section 2.4.2. Then, it will be easier to conduct a thorough performance study to test different optimization strategies in simulation before executing the SNFactory workflow with Pegasus on an actual execution platform.

Such a preliminary study has been conducted by J. Rouzaud-Cornabas to evaluate the opportunity to run SNFactory workflow on a multi-Cloud environment. A 3-step scheduling mechanism was investigated. The first step partitions the workflow graph into sub-workflows according to the pricing model of the target infrastructure. The second provisions virtual machine instances from one or multiple IaaS cloud providers, while the third and last step, actually schedules the tasks of each sub-workflow on the provisioned virtual machines. While the second step of this scheduling mechanism is specific to IaaS clouds, the other two correspond to the initial objectives. The first step automates the process of grouping rather small compute tasks together into larger clusters. This was previously done empirically to adapt the execution to operation constraints as best as (humanly) possible. It is more robust and sound to perform this partitioning based on predefined and objective constraints. The third step that schedules sub-workflows can reduce the overall execution time of a workflow. Such a performance optimization was not a primary concern within the SNFactory collaboration as the main objective is (as often in production) to ensure that results are obtained in a reasonable time. However, in a period of constrained budgets and where a great emphasis is put on the reduction of energy consumption, it becomes worth considering any available opportunity of performance optimization as a primary objective. Finally, we can note that the second step could be translated to the currently used batch system to a certain extent. Indeed, provisioning virtual machines on an IaaS cloud is not so different to submitting jobs of a certain shape and length to a batch system. Then some results obtained while aiming at clouds could be useful in a batch/grid environment and be rapidly put in production.

The next step is to conduct a series of experiments on the Grid'5000 experimental testbed to assess the performance of the SNFactory application as a whole and the behavior of each component of its workflow. In these experiments, the workflow will be handled by the Pegasus engine, and its default scheduler. Different execution environments, *i.e.,* single cluster, multi-cluster, or even IaaS cloud, will be considered. The analysis of the obtained results will help us to determine where optimization strategies could be applied, be they for performance or better fault tolerance.

Besides these scientific aspects, an important milestone towards the application of research results in scheduling to production applications, is to formalize this beginning collaboration. The preliminary work has been done by members of the LIP, IPNL, and CC IN2P3, but it would be beneficial to involve the American part of the SNFactory collaboration and the Pegasus team to go further.

3.4.2 Multi-objective scheduling for large computing platforms

Over the last decades specialized software has been developed to efficiently deliver the required computing power to many users at the same time. One of such software component is the JRMS, or *batch scheduler*. The main goal of a JRMS is to satisfy users' demands for computing resources and achieve a good *performance* in overall system utilization by efficiently assigning jobs to resources. This assignment involves three main abstraction layers: (i) the declaration of a job, where the demand of resources and job characteristics are expressed; (ii) the scheduling of the jobs on the resources; and (iii) the launching and placement of job instances on the available computing resources along with the control of the execution of the jobs. Some years ago, when data centers were consisted of simple and medium size homogeneous architectures, the functioning of a JRMS was rather straightforward. Its main intelligence and complexity were centered around the efficient scheduling of jobs. However, the continuously increasing demand for computing power by applications made users more demanding for a certain quality of service. Then the efficient assignment of large number of resources to an evenly large number of users raised new issues such as the scalability of both job launcher and scheduler.

Since the first proposals of JRMSs in the eighties, different software packages have been proposed as evolutions of some older software or with new designs. Commercial systems, *e.g.*, LSF, LoadLeveler, PBSPro, or Moab generally support a large number of architectures and operating systems, provide highly developed graphic interfaces for visualization, monitoring and transparency of usage. Their open-source alternatives, *e.g.*, Condor [144], OAR [34], SLURM [154], GridEngine [77], or Torque, provide more innovative features and a certain flexibility when compared to commercial solutions. However, all these JRMSs are based on similar ideas and techniques where the scheduler is the corner stone of the software. The selection of a given tool is then driven by other concerns, such as the typical workload, old habits, or political or financial reasons.

To illustrate the importance of the JRMS in a production computing center, and motivate the need for research on scheduling that can be transferred to production, I detail hereafter the concrete use case of the CC IN2P3. The very specific workload submitted to this computing center had a huge influence on batch scheduling. Indeed, typical jobs in High Energy Physics are sequential, have long execution times, and access large volumes of data. Moreover, resources are shared by several (sometimes competing) physics collaborations whose cumulated pledges exceed the offered computing capacity. As a consequence the CC IN2P3's computing farm is always full, with almost as much jobs waiting in the batch queue as jobs running, and this, whatever the periodic growth of the resource pool.

To handle such a particular workload, the decision was taken to develop an in-house batch scheduler, called the Batch Queuing System (BQS), back in 1992. This tool has been enriched over the years with a large set of specific scheduling rules that answer encountered issues. It is interesting to note that the capacity to submit parallel jobs to BQS was introduced only in 2005! Moreover, and until 2009, the pool of computing resources was physically split to handle either sequential or parallel jobs, with more than 20 times as much nodes allocated to sequential jobs as to parallel ones. The obvious reason for such design choices is that parallel jobs only represent a small fraction of the daily workload executed at the CC IN2P3. BQS has then been constantly tuned and adapted according to the production constraints and expectations, but such a fine management allowed by a tool developed in-house comes with the burden of developing and maintaining a complex code. In 2009, the benefits offered by BQS stopped to be worth the invested time and human resources, hence a new JRMS has been selected to replace BQS and be put in production. A thorough comparative study led to the selection of the Grid Engine batch scheduler, originally developed by Sun Microsystems [77] and then successively acquired by Oracle and Univa Corporation. The comparison of the candidate JRMS was based on more than 60 criteria grouped in 15 categories. Some of these categories were related to the quality of the candidate software solutions, *i.e.*, scalability, robustness, operation, support, or cost. Others were specific to the context of CC IN2P3, *e.g.*, support of the Andrew File System (AFS), support of heterogeneous hardware and software, monitoring and accounting, resource capping, or interface with grid middleware.

The internal scheduler itself, and the capacity to configure and tune it, was obviously the subject of one category, but surprisingly not the most important one. Finally, other concerns outside this evaluation scheme were considered and were sufficient to discard some tools before their evaluation. Condor [144] and SLURM [154] suffered of an insufficient adoption in the High Energy Physics community and an assumed lack of interface with grid middleware, while OAR [34] was seen as a research project despite its use in production on Grid'5000. In the end, commercial solutions, *i.e.*, Grid Engine, LSF, PBS-Pro, were clearly favored to open-source solutions. The exception is Torque/Maui which is an open-source tool, but the Maui cluster scheduler can be replaced by the commercial MOAB solution.

This example highlights the difficulty to transfer innovation coming from research to production, where the minimization of risks remains the most important factor in decision making. However, this does not have to prevent researchers to look for optimization techniques that could be applied, and be beneficial, to a production environment. Two conditions have to be met to ensure that the proposed solutions are going to be adopted. First, all the specifics of the production workload and environment have to be taken into account as early as possible in the design of original scheduling heuristics. Simple, but unrealistic abstractions, which are commonplace in theoretical research on scheduling, would lead to solutions that are inapplicable in a real, and complex, system. Similarly a theoretically guaranteed, but extremely long to execute, algorithm is likely to be intractable with a JRMS that manages ten of thousands nodes and hundreds of thousands jobs. Second, the value of integrating a new scheduling mechanism into the production JRMS and its robustness to production constraints have to be proven. Again, a theoretical study, even with formal proofs, is not enough to convince operation teams to switch and risk a degradation of performance or failures. Writing a simple simulator to produce a set of graphs for some standard traces such as those provided by the Parallel Workload Archive (PWA) [70] is not sufficient enough. A more convincing solution is to demonstrate that the proposed solution is beneficial *in the same conditions* as those experienced by the computing center. This means to: (i) extend the performance study up to the proper integration of the scheduling algorithm into the code of the batch scheduler; (ii) let this modified JRMS manage a set of resources with the same, or close enough, specifics as those of the computing center, *e.g.*, number of nodes, heterogeneity, or configuration rules; and (iii) submit to this system a workload that corresponds to what is handled by the computing center on a daily basis. Furthermore, such a production-grade performance study has to show that the new scheduling mechanism offers a concrete and significant added value, *e.g.*, by optimizing a new metric such as energy consumption, while it does not degrade the existing performance metrics.

The aim of the MOEBUS ANR project[6], to which I participate, is at conducting such end-to-end performance studies going from the theoretical analysis up to the integration into a batch scheduler. However, the scientific context is different of that of the CC IN2P3. The focus of this project is on HPC systems and thus has a broader scope. GPUs and other hardware accelerators are included in the target platform and the placement of processes within a parallel job is also considered. Such materials and types of jobs are part of the CC IN2P3's landscape but represent a non-significant share of it. Nevertheless, this share might increase over the next few years to accommodate the needs of the forthcoming astroparticle physics applications. The targeted batch schedulers are also different as OAR and SLURM will be considered.

Despite these differences, I plan to take advantage of my position at CC IN2P3 and my participation to this project to reach some sort of mutual benefit. For instance, the MOEBUS project aims at developing performance models to describe applicative workloads submitted to batch schedulers. This requires to obtain and analyze logs of job submissions and executions. Obtaining such information for the particular workload of the CC IN2P3, which is very different of a HPC workload, would be a great added value to the project while helping at the potential integration of proposed algorithms into the CC IN2P3's JRMS. Moreover, the main objective of the MOEBUS project is to simultaneously optimize

[6] `http://moebus.gforge.inria.fr`

unconventional performance metrics such as fairness and energy consumption. Optimizing the former could help to automate the fair attribution of resources to concurrent physics experiments according to their pledges and the initial arbitrage made by the direction. Minimizing the latter is a major concern is every computing center, including the CC IN2P3 as the share of energy in the overall operation cost is ever increasing. For these reasons, this ANR project is a good candidate to bridge the gap between research and to apply results of my own research to a production context.

3.5 Conclusion

The data deluge observed in several scientific fields coincides with a simpler usage of large scale production DCIs such as EGI thanks to scientific portals and virtualization techniques. In this context it seems possible to get some control back over this computing grid whose access to new users was difficult and where strategic technical orientations were more imposed than chosen. There is thus an opportunity to fully take advantage of all the expertise acquired on grid computing in the last decade and combine it with the excellent research work in Computer Science made by French teams to develop and transfer innovative solutions. However, this raises many challenges in terms of research to use the resources of production DCIs in an efficient and robust way.

In this chapter, I developed two complementary approaches to bridge an existing gap between research and production that are in the direct continuation of my earlier activities and should address some of the raised challenges. The first approach extends the activities in the domain of simulation of distributed systems and applications detailed in Chapter 2 to bring SIMGRID up to a production-grade level of quality. The ambition is to propose a complete and trustworthy simulation ecosystem that would allow users of, and researchers on, large scale distributed infrastructures to design, evaluate, and transfer innovative solutions more easily. Thanks to the versatility of the SIMGRID toolkit, it is possible to propose advances for both HTC (Section 3.3.1) and HPC (Section 3.3.2) infrastructures and applications. Moreover, important aspects underlying the Cloud and Big Data topics also fall in the domain of application of SIMGRID. As detailed in Section 3.3.3, I propose to contribute to the simulation of storage infrastructures more than on that of virtualization or IaaS Cloud which is already well covered by other SIMGRID contributors. The second approach builds up on all the expertise on scheduling acquired with the work on parallel task graphs, which was presented in Chapter 1. I have developed and studied the performance of scheduling algorithms without going to using them on concrete use cases for almost ten years, but, since I have been hired at the CC IN2P3, my major concerns is to be able to propose practical solutions to users coming from other scientific fields, starting with the physics collaborations from the IN2P3. As detailed in Section 3.4, the needs expressed by the new experiments that will be served by the CC IN2P3 in a few years better match my expertise than the current workload. Moreover the time before these applications enter their production phase can be used to participate and hopefully influence the design choices to favor interactions with computer scientists. Sections 3.4.1 and 3.4.2 showed that in the meantime, interesting collaborations and cross-fertilization opportunities exists than I plan to develop. Being a researcher in Computer Science hosted in a computing center dedicated to physics, but also a member of a research team in a major laboratory is maybe a peculiar situation, but more importantly it is a real chance.

General Conclusion

The last part of this document is an occasion to cast another light on the contributions detailed in the previous chapters. I decided to structure this document around the two main topics I worked on for the last twelve years: *Scheduling* in Chapter 1 and *Simulation* in Chapter 2. It may let think that these two complementary activities have been developed in parallel in a relative independence, but the reality is different. My activities on scheduling fueled those on simulation, and *vice versa*, which is a good illustration of what I deeply like in research. As many colleagues have, I have been repeatedly submitted by friends and family to the same fundamental question since I embraced my research career: *"So . . . you are a researcher . . . Did you find something, hu?"* It is very difficult to answer such a provocative (but usually friendly) question asked by people that have a certain vision (that I am not going to judge here) of research. A literally answer would be to list all or some gratifying achievements, such as designing a clever heuristic to tackle a NP-Hard scheduling problem or developing a simulator that mimics the actual behavior of something with less than 5% of error in broad range of scenarios. However, such an answer is likely to make eyebrows raise, puzzle the asker, and reinforce the idea that researchers are strange people. Moreover, it does not explain that the most interesting finding for a researcher is a new problem to solve, which might be even harder to understand.

For about ten years, I went back and forth from scheduling to simulation. There were several occasions where I could not use the simulation tool to evaluate what I wanted to do in scheduling. For instance, when the original API of SIMGRID, that was specifically designed for DAG scheduling was removed, I had to cope with the remaining MSG, but was soon limited in my investigation of new heuristics. This led to the rebirth of the original API through the development of SimDAG, as explained in Section 2.3. Another example is when I became unsatisfied of using randomly generated platform description whose realism and representativeness could be discussed. As it hindered the demonstration of the applicability of the proposed scheduling heuristic, I started to consider descriptions of actual compute clusters, such as those of the Grid'5000 testbed, but was soon faced to scalability and diversity issues. This led me to propose a modification of the description format of SIMGRID and the SIMULACRUM tool, both detailed in Section 2.4.1. At that point, I was more deeply involved in the development of SIMGRID. Then I started to add what I needed for my research on scheduling into the code base myself. Then the new features and capacities offered by the simulation toolkit allowed me to design and evaluate more scheduling heuristics, such as those presented in Section 1.5. This co-evolution of my scheduling and simulation activities somehow came to a natural end when my first Ph.D. student graduated and I was hired at CC IN2P3. This was the occasion to start a new cycle of research in which my activities in scheduling are less important. However, the conclusion of Chapter 1, my recent implication in the MapReduce and MOEBUS ANR projects, and the contents of Section 3.4 show that this topic remains important to me and that I still plan to contribute to this domain.

This second cycle of research is more focused on simulation activities and driven by issues coming from my working environment, *i.e.*, a computing center dedicated to the service of physics applications. These ongoing activities have been described in Section 2.6 and all through Chapter 3. Here again, a certain co-evolution exists between my work on simulation and my aim at bridging a gap between research and production. A good illustration is the origin of the topic of dimensioning through simulation, which is at the core of the thesis of my second Ph.D. student. It came from the conjunction of two events. A preliminary framework for the simulation of MPI application was added to SIMGRID in

2009. It offered great perspectives by broadening the domain of application of SIMGRID to HPC. At the same time, he CC IN2P3 has upgraded its computing farm dedicated to parallel, hence MPI, jobs. I was surprised by the chosen method to decide of this upgrade. It consisted in sending a simple two-choice question to users to know whether they would prefer to see a doubling of the number of cores or the addition of a High Performance Network, *i.e.*, an InfiniBand interconnect, to the existing platform. The subjectivity of this method and the lack of objective performance indicators to help the decision taking brought me to investigate whether and how SIMGRID and SMPI could be used in such a context. The developments presented in Section 2.6 and the good results we achieved let see great perspectives that have been detailed in Section 3.3.2. Moreover, the excellent dynamics around SIMGRID and all the efforts to deliver a production-grade simulation toolkit encourage me to extend this work that combines simulation and production concerns to other topics such as the realistic simulation of production DCIs developed in Section 3.3.1 or that of hierarchical mass storage infrastructures as detailed in Section 3.3.3.

Besides the main topics of scheduling, simulation, and connections between research and production that have been addressed in this document, there exists another potential and personally interesting perspective. The recent addition of storage simulation capacities to SIMGRID offers me the opportunity to come back to my very first research activities, but better armed than fifteen years ago. Indeed, my baby steps in research were on Out-of-Core computing when I studied how to manage large amounts of data that exceed the memory capacity of a compute platfom from the Scilab mathematical software. I continued to work on this subject for some time by applying Out-of-Core techniques to pipeline the execution of wavefront algorithms [CDS05, CGS08], but the lack of an appropriate simulation tool forced me to evaluate the proposed solutions through real executions, that were complex, time-consuming, and of limited scale. This prevented me to publish some results that were quite impossible to reproduce because of evolution of the hardware and required a lot of time and efforts. Then this activity has been put on hold for many years, but revisiting these old results thanks the new features of SIMGRID, and somehow stepping back in time, would be a thrilling and amusing perspective.

To conclude this document, I would like to emphasize on something that I tried to improve over the years and aim at further developing in the future. It concerns the methodology that underlies the production of scientific results. My different activities, not only designing scheduling heuristics or developing a simulation tool, but also reviewing papers or evaluating grant proposals, showed me how Computer Science was still a young science and sometimes lacked of rigor in the way scientific results are obtained, presented, evaluated, and analyzed. There are too many published research papers, including some of my own alas, whose results are not reproducible or only applicable to a limited (and not always well bounded) scope. We, Computer Scientists, certainly have a lot to learn from older sciences, such as physics or biology, that usually disregard results that are not backed up by a thorough and reproducible experimentation methodology. Striving for better practices, and aiming at more Open Science is a great and exciting challenge. There is a long way to go to be on par with other sciences, and I will certainly not pretend to achieve this alone. However, working on a regular basis with physicists from the IN2P3 thanks to my particular position at CC IN2P3 and being part of this wonderful team that develops and sustains SIMGRID would certainly help me to go into the right direction and bring my own stone to this virtuous endeavor.

List of Acronyms

Δ-CTS Δ-Critical Tasks Scheduling. 38, 39

SIMULACRUM Simulation pLAtform CReation and User-guided Modification). 78–81, 108, 118, 135

AFS Andrew File System. 132

ANR Agence Nationale de la Recherche. 55, 58, 59, 105, 116, 133–135

API Application Programming Interface. 15, 58–60, 62–66, 69, 90, 101, 103, 107, 118, 126–129, 135

APST AppLeS Parameter Sweep Template. 86

ARC Action de Recherche Collaborative. 66

AS Autonomous System. 75, 76

ATER Attaché Temporaire d'Enseignement et de Recherche. 1

ATLAS A Toroidal LHC ApparatuS. 59, 126

biCPA Bi-Criteria Critical Path Area-Based scheduling. 27–29, 48, 52, 53, 56

BLAS Basic Linear Algebra Subroutines. 6

BQS Batch Queuing System. 132

BRITE Boston university Representative Internet Topology gEnerator. 73, 81

BSP Bulk Synchronous Parallel. 64

CAFM Coarse-grain Allocation Fine-grain Mapping. 48, 49

CC IN2P3 IN2P3 Computing Center. 1, 2, 108, 111, 115, 116, 125, 127–136

CERN Organisation Européenne pour la Recherche Nucléaire. 59, 111, 126

CG Conjugate Gradient. 121, 122

CINES Centre Informatique National de l'Enseignement Supérieur. 115

CNRS Centre National de la Recherche Scientifique. 1

CPA Critical Path Area-Based scheduling. 17–21, 23, 25–28, 31, 43, 56

CPR Critical Path Reduction. 17, 18

CRA Constrained Resource Allocation. 46, 49, 54

CSP Concurrent Sequential Processes. 59, 60, 62, 65

CTA Cherenkov Telescope Array. 130

DAG Directed Acyclic Graph. 7, 9, 11, 31, 36, 42, 54, 60, 62–66, 69, 70, 81–85, 87, 108, 135

DAGue Directed Acyclic Graph Unified Environment. 54

DCI Distributed Computing Infrastructure. 110–113, 116–120, 134, 136

DEUS Dark Energy Universe Simulation. 110, 115

DHT Distributed Hash Table. 71

DIET Distributed Interactive Engineering Toolbox. 111

DIRAC Distributed Infrastructure with Remote Agent Control. 118, 120

DMHEFT Dynamic MHEFT. 39, 40, 42

DTD Document Type Definition. 74, 95

EC2 Elastic Compute Cloud. 107

EFT Earliest Finish Time. 38, 41

EGEE Enabling Grids for E-sciencE. 1, 116, 117

EGI European Grid Infrastructure. 72, 112, 115–119, 134

FFT Fast Fourier Transform. 16, 17, 38, 84

GENIE Grid ENabled Integrated Earth. 55

GPU Graphical Processing Unit. 5, 6, 107, 110, 114, 115, 130, 133

HCPA Heterogeneous Critical Path Area-Based scheduling. 31, 32, 36, 37, 45

HEFT Heterogeneous Earliest Finish Time. 36, 37

HPC High Performance Computing. 59, 111, 114–116, 130, 133, 134, 136

HPN High Performance Network. 59

HPSS High Performance Storage System. 128

HTC High Throughput Computing. 111, 115, 130, 134

IaaS Infrastructure as a Service. 56, 59, 72, 107, 112, 113, 131, 134

iCASLB iterative Coupled processor Allocation and Scheduling algorithm with Lookahead and Backfill. 26, 53

ID-IMAG Informatique et Distribution, Informatique, Mathématiques et Applications de Grenoble. 1

IDG Institut des Grilles du CNRS. 117, 126

IDRIS Institut du Développement et des Ressources en Informatique Scientifique. 115

IN2P3 Institut National de Physique Nucléaire et Physique des Particules. 130, 134, 136

IPC Inter-Process Communication. 60

IPNL Institut de Physique Nucléaire de Lyon. 130, 131

IQR Inter-Quartile Range. 15

JRMS Job and Resource Management System. 10, 13, 54, 132, 133

LBNL Lawrence Berkeley National Laboratory. 130

LFC Logical File Catalog. 118

LHC Large Hadron Collider. 1, 59, 109–111, 114, 115, 129, 130

LIP Laboratoire de l'Informatique du Parallélisme. 1, 131

LoC-MPS Locality Conscious Mixed Parallel processor allocation and Scheduling algorithm. 26

LORIA Laboratoire lorrain d'informatique et de ses applications. 1

LPTF Longest Processing Time First. 48, 49

LSDC Large Scale Distributed Computing. 59

LSST Large Synoptic Survey Telescope. 130

MAGMA Matrix Algebra on GPU and Multicore Architectures. 6

MAGS Malleable Allocations with Guaranteed Stretch. 49, 50, 53

MCGAS Multi-Cluster Guaranteed Allocation Scheduling. 33, 34, 36

MCMC Markov Chain Monte Carlo. 110

MCPA Modified Critical Path Area-Based scheduling. 21

MHEFT Mixed-parallel HEFT. 36–39

MPI Message Passing Interface. 60, 89–106, 108, 116, 117, 121–123, 125, 135, 136

NAS Network-attached Storage. 125

NPB NAS Parallel Benchmarks. 98, 121, 124

NREN National Research and Education Network. 72

OSPF Open Shortest Path First. 75

OTaPHe Ordonnancement de Tâches Parallèles en milieu Hétérogène. 66

P2P Peer-to-Peer. 59, 71, 75

PaaS Platform as a Service. 112, 113

PAPI Performance Application Programming Interface. 98

PDA Platform Description Archive. 108, 118, 120

PICS *Projet International de Coopération Scientifique. 89*

PLASMA *Parallel Linear Algebra Software for Multicore Architectures. 6*

PRACE *Partnership for Advanced Computing in Europe. 115, 116*

PTG *Parallel Task Graph. 3, 7, 9–21, 26–29, 31–33, 36–40, 42–56, 64, 65, 69, 81–86*

PWA *Parallel Workload Archive. 133*

RATS *Redistribution-Aware Two-Steps. 23, 25*

ReP *Reuse Processors. 39–41*

RFIO *Remote File IO. 128*

RIP *Routing Information Protocol. 75*

SaaS *Software as a Service. 112, 113*

SAF *Smallest Area First. 48, 49*

SAN *Storage Area Network. 125*

SCRAP *Self-Constrained Resource Allocation Procedure. 43–45, 54*

SMP *Symmetric Multi-Processor. 32, 33*

SMPI *Simulated MPI. 90–95, 103–106, 108, 121–123, 136*

SONGS *Simulation Of Next Generation Systems. 59, 116, 118, 126*

SPMD *Single Program Multiple Data. 92*

SPTF *Shortest Processing Time First. 49*

SRM *Storage Resource Manager. 118*

SSD *Solid State Drive. 125, 127*

TAU *Tuning and Analysis Utilities. 97–102, 105, 122*

TFR *Trace Format Reader. 101*

TGCC *Très Grand Centre de Calcul. 115*

TGFF *Task Graphs for Free. 82*

TReqS *Tape Request Scheduler. 128*

UCSD *University of California, San Diego. 1, 86*

USS SIMGRID *Ultra Scalable Simulation with* SIMGRID. *58, 105*

VIP *Virtual Imaging Platform. 118, 120*

WLCG *Worldwide LHC Computing Grid. 110*

XBT *eXtended Bundle of Tools. 60, 69*

XML *eXtended Markup Language. 66, 73, 74, 76, 78, 80–82, 85, 86, 95, 118, 128*

List of Figures

1.1 Example 5-task PTG, with two possible configurations. Each task is labeled as $X : x, y$, where X is the task's name, x is the task's allocation, and y is the task's execution time. . 7
1.2 Illustration of a heterogeneous multi-cluster platform made of three homogeneous clusters. 8
1.3 Example of the two-step schedule of a PTG. 11
1.4 A taxonomy of Parallel Task Graph scheduling problems. 12
1.5 Evolution of T_{CP} and T_A, throughout the allocation procedure of CPA for a random PTG of 6 tasks on clusters of 10 (a) and 30 (b) processors. 19
1.6 Schedule of a Strassen matrix multiplication produced by the CPA algorithm on a cluster of 20 processors. 20
1.7 Evolution of T_{CP}, T_A, and T_A^{geo} throughout the allocation procedure for a random PTG of 6 tasks on a cluster of 30 processors. 21
1.8 Gantt chart of the schedule of a six-task PTG on a cluster of 30 processors using the original T_A (a) and T_A^{geo} (b). 22
1.9 Illustration of the *allocation packing* performed during the mapping step. 23
1.10 Motivating example to stretch or pack task allocations to save some data redistributions. 23
1.11 Illustration of the benefits of a *backfilling* post-processing step. 26
1.12 Evolution of T_{CP}, T_A, and T'_A throughout the allocation procedure of CPA for a random PTG of 50 tasks on a cluster of 20 processors. 27
1.13 Evolution of $C_{P'}/C_P$ and $W_{P'}/W_P$ when P' varies for a random PTG of 20 tasks on a cluster of 20 processors. 29
1.14 2D projections of all $(\mu, b, 1/\beta)$ triplets with performance ratio $1/\beta$ lower than 10. 35
1.15 Domain of b and μ values for which MCGAS's performance ratio is lower than 10, for a particular platform configuration. 36
1.16 Illustration of the allocation reduction strategies added to MHEFT for a 1 Gflops task to be scheduled on a cluster of 30 processors. 38
1.17 Common problem for dynamic schedulers: placing and executing a task v_i at time $t_r(v_i)$ may lead to an unfavorable schedule if the number of available processors is small. 40
1.18 Postponing strategy of the DMHEFT algorithm for task v_i that becomes ready at time $t_r(v_i)$ on cluster c^k. 41
1.19 Impact of an ordering limited to the list of ready tasks (bottom right) on the schedule length with regard to a global ordering (top right). 45
1.20 Illustration of the impact of task ordering in the mapping step of the `SELFISH` heuristic on the resulting schedule. 47
1.21 Example of period and malleable allocations for the concurrent scheduling of 6 PTGs on a cluster of 47 processors. 51
1.22 A taxonomy of Parallel Task Graph scheduling problems with the proposed heuristics. . . 53
1.23 Typical workflow a MapReduce application. 54
1.24 Non-deterministic workflow control nodes and constructs. 55
1.25 Extracting sub-workflows from OR-splits, OR-joins and Cycles. 56
2.1 Components of a classical simulation environment. 58
2.2 SIMGRID components. 59
2.3 Design and internals of SIMGRID. 61
2.4 Scheduling literature and SIMGRID representations of an application structured as a DAG. 63

2.5 Communication matrix of a data redistribution between a source allocation on five hosts to a destination allocation on three hosts. . . . 65
2.6 SimGrid v1 code sample [37]. . . . 67
2.7 SimDAG code sample. . . . 68
2.8 SimDAG code sample using typed tasks and auto-scheduling features. . . . 70
2.9 Illustration of hierarchical network representation. Circles represent processing units and squares represent network routers. Bold lines represent communication links. AS2 models the core of a national network interconnecting a small flat cluster (AS4) and a larger hierarchical cluster (AS5), a subset of a LAN (AS6), and a set of peers scattered around the world (AS7). . . . 75
2.10 Main steps of the hierarchical routing mechanism. . . . 76
2.11 Grid'5000 Processor Speeds (in GFlop/s) in 2007. . . . 77
2.12 Generation flowchart of the SIMULACRUM tool. . . . 78
2.13 Example of a PTG produced by *daggen* as a text file (left) and grahically (right). . . . 83
2.14 Jedule output for the schedule produced by one of the algorithms proposed in [CDS10]. Four PTGs, each having its own color, are scheduled on a cluster of 20 processors. The resource constraints imposed by the algorithm are respected. . . . 87
2.15 Jedule output of the schedule by HEFT of a DAG of 50 task on a heterogeneous platform. 87
2.16 Jedule output of the schedule by HEFT of a DAG of 50 task on a heterogeneous platform with a greater latency on the backbone link. . . . 88
2.17 Application compilation with SMPI. . . . 91
2.18 SMPI macros to reduce computing requirements. . . . 92
2.19 SMPI macros to reduce memory requirements. . . . 93
2.20 Comparison between a SKaMPI run and SMPI (default affine, best-fit affine, and piece-wise linear models) for a ping-pong operation between two machines. . . . 94
2.21 Simple MPI application on a ring (left) and corresponding time-independent trace (right). 96
2.22 Time-independent trace acquisition process. . . . 97
2.23 Instrumentation skew for the three instrumented LU benchmarks. . . . 99
2.24 Overhead for the reduced and minimal instrumentation methods. . . . 99
2.25 List of callbacks related to a call to the `MPI_Send` function. . . . 101
2.26 Inputs and outputs of the SIMGRID trace replay framework. . . . 103
2.27 Implementation of the *send* action using the SMPI internal API. . . . 104
2.28 Comparison of timed traces of the execution of an MPI application obtained using SMPI, running OpenMPI, or replaying a time-independent trace. . . . 105
2.29 Summary of works in or around SIMGRID. . . . 106
3.1 The awful truth about cloud computing understanding. . . . 113
3.2 Overview of the VIP architecture and associate services (© S. Camarasu-Pop). . . . 119
3.3 Comparison between simulated and actual execution times for the CG benchmark. . . . 121
3.4 Two seconds Gantt-chart of the actual execution of a class B instance of CG for 128 processes. . . . 122
3.5 Modeling the *graphene* cluster: rectangles represent capacity constraints. Grayed rectangles represent constraints involved in a communication from node to node 40 to node 104. . . . 124
3.6 Distribution of compute rate (instruction/sec) vs. number of instructions for all compute bursts during a 4-process execution of the LU benchmark. Average rate shown as a horizontal line. . . . 124
3.7 Example of storage description in the SIMGRID format. . . . 128
3.8 Overview of the hierarchical storage infrastructure of the IN2P3 Computing Center. . . . 129

List of Tables

1.1 Classification of job types based on specifying number of processors used. 6
1.2 Summary of notations related to computing platforms. 9
1.3 Summary of notations related to Parallel Task Graphs. 10
1.4 Summary of the contributions to the problem of scheduling a single Parallel Task Graph on a single cluster or a multi-cluster. 42
2.1 Requirements for synthetic experimental environments per research community. 72
2.2 Summary of platform configurations extracted from Grid'5000. 77
2.3 Examples of SIMULACRUM settings fulfilling the needs of each research community. . . 80
2.4 Time-independent actions corresponding to supported MPI communication operations. . 97

List of External Links

1 http://mapreduce.inria.fr 55
2 http://uss-simgrid.gforge.inria.fr 58
3 http://infra-songs.gforge.inria.fr 59
4 http://www.loria.fr/~suter/OTAPHE/ 66
5 http://simgrid.gforge.inria.fr/tutorials/simdag-101.pdf 70
6 svn://scm.gforge.inria.fr/svn/simgrid/contrib/trunk/DAGSched 70
7 http://www.egi.eu/ 72
8 http://www.grid5000.fr/ 72
9 http://www.cs.vu.nl/das3/ 78
10 http://en.wikipedia.org/wiki/Descriptive_statistics 80
11 http://www.kasahara.elec.waseda.ac.jp/schedule/ 82
12 https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator . 82
13 https://github.com/frs69wq/daggen 83
14 http://sourceforge.net/projects/jedule/ 86
15 http://www.top500.org 89
16 http://www.cs.uoregon.edu/research/tau/docs/newguide/ch06s02.html . . . 101
17 http://paje.sourceforge.net 105
18 https://github.com/frs69wq/biCPA 108
19 http://pda.gforge.inria.fr/ 108
20 http://www.cs.huji.ac.il/labs/parallel/workload/ 108
21 http://gwa.ewi.tudelft.nl 108
22 http://www.top500.org/lists/2013/11/ 110
23 http://eu-datagrid.web.cern.ch/ 111
24 http://http://www.prace-ri.eu 116
25 http://mescal.imag.fr/membres/arnaud.legrand/research/smpi/smpi_loggps.php 123
26 http://hadoop.apache.org 130
27 http://moebus.gforge.inria.fr 133

Publications Since the End of the Ph.D.

Book Chapters

[ABS14] Hamid Arabnejad, Jorge Barbosa, and Frédéric Suter. *High-Performance Computing on Complex Environments*, chapter Fair Resource Sharing for Dynamic Scheduling of Workflows on Heterogeneous Systems. Parallel and Distributed Computing Series. John Wiley & Sons, June 2014. In press.

International Journals

[ACB+13] Gabriel Antoniu, Alexandru Costan, Julien Bigot, Frédéric Desprez, Gilles Fedak, Sylvain Gault, Christian Pérez, Anthony Simonet, Bing Tang, Christophe Blanchet, Raphael Terreux, Luc Bougé, François Briant, Franck Cappello, Kate Keahey, Bogdan Nicolae, and Frédéric Suter. Scalable Data Management for Map–Reduce–Based Data–Intensive Applications: a View for Cloud and Hybrid Infrastructures. *International Journal of Cloud Computing*, 2(2-3):150–170, 2013.

[CDS10] Henri Casanova, Frédéric Desprez, and Frédéric Suter. On Cluster Resource Allocation for Multiple Parallel Task Graphs. *Journal of Parallel and Distributed Computing*, 70(12):1193–1203, December 2010.

[DNSC09] Pierre-François Dutot, Tchimou N'Takpé, Frédéric Suter, and Henri Casanova. Scheduling Parallel Task Graphs on (Almost) Homogeneous Multi-cluster Platforms. *IEEE Transactions on Parallel and Distributed Systems*, 20(7):940–952, July 2009.

International Conferences

[ABB+12] Gabriel Antoniu, Julien Bigot, Christophe Blanchet, Luc Bougé, François Briant, Franck Cappello, Alexandru Costan, Frédéric Desprez, Gilles Fedak, Sylvain Gault, Kate Keahey, Bogdan Nicolae, Christian Pérez, Anthony Simonet, Frédéric Suter, Bing Tang, and Raphael Terreux. Towards Scalable Data Management for Map-Reduce-based Data-Intensive Applications on Cloud and Hybrid Infrastructures. In *Proceedings of the First International IBM Cloud Academy Conference (ICA CON)*, pages 272–290, Research Triangle Park, NC, April 2012.

[BDG+13] Paul Bédaride, Augustin Degomme, Stéphane Genaud, Arnaud Legrand, George S. Markomanolis, Martin Quinson, Mark Stillwell, Frédéric Suter, and Brice Videau. Toward Better Simulation of MPI Applications on Ethernet/TCP Networks. In *Proceedings of the 4th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, Denver, CO, November 2013.

[BLM+12] Laurent Bobelin, Arnaud Legrand, David Alejandro González Márquez, Pierre Navarro, Martin Quinson, Frédéric Suter, and Christophe Thiery. Scalable Multi-Purpose Network Representation for Large Scale Distributed System Simulation. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 220–227, Ottawa, Canada, May 2012.

[CDMS12] Eddy Caron, Frédéric Desprez, Adrian Muresan, and Frédéric Suter. Budget Constrained Resource Allocation for Non-Deterministic Worflows on an IaaS Cloud. In *Proceedings of the 12th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, volume 7439 of *Lecture Notes in Computer Science*, pages 186–201, Fukuoka, Japan, September 2012. Springer.

[CDS04] Henri Casanova, Frédéric Desprez, and Frédéric Suter. From Heterogeneous Task Scheduling to Heterogeneous Mixed Parallel Scheduling. In Marco Danelutto, Domenico Laforenza, and Marco Vanneschi, editors, *Proceedings of the 10th International Euro-Par Conference (Euro-Par)*, volume 3149 of *Lecture Notes in Computer Science*, pages 230–237, Pisa, Italy, August/September 2004. Springer.

[CDS05] Eddy Caron, Frédéric Desprez, and Frédéric Suter. Out-of-Core and Pipeline Techniques for Wavefront Algorithms. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS)*, Denver, CO, April 2005.

[CDS10] Henri Casanova, Frédéric Desprez, and Frédéric Suter. Minimizing Stretch and Makespan of Multiple Parallel Task Graphs via Malleable Allocations. In *Proceedings of the 39th International Conference on Parallel Processing (ICPP)*, pages 71–80, San Diego, CA, September 2010.

[CGL+13] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. SimGrid: a Sustained Effort for the Versatile Simulation of Large Scale Distributed Systems. In *Proceedings of the 1st Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE)*, Denver, CO, November 2013.

[CGS08] Pierre-Nicolas Clauss, Jens Gustedt, and Frédéric Suter. Out-of-Core Wavefront Computations with Reduced Synchronization. In Julien Bourgeois, Fran cois Spies, and Didier El Baz, editors, *Proceedings of the 16th Euromicro International Conference on Parallel, Distributed and network-based Processing (PDP)*, pages 293–300, Toulouse, France, February 2008. IEEE.

[CSG+11] Pierre-Nicolas Clauss, Mark Stillwell, Stéphane Genaud, Frédéric Suter, Henri Casanova, and Martin Quinson. Single Node On-Line Simulation of MPI Applications with SMPI. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Anchorage, AK, May 2011.

[DMQS11] Frédéric Desprez, George S. Markomanolis, Martin Quinson, and Frédéric Suter. Assessing the Performance of MPI Applications Through Time-Independent Trace Replay. In *Proceedings of the 2nd International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI)*, pages 467–476, Taipei, Taiwan, September 2011.

[DMS12] Frédéric Desprez, George S. Markomanolis, and Frédéric Suter. Improving the Accuracy and Efficiency of Time-Independent Trace Replay. In *Proceedings of the 3rd International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, Salt Lake City, UT, November 2012.

[DS10] Frédéric Desprez and Frédéric Suter. A Bi-Criteria Algorithm for Scheduling Parallel Task Graphs on Clusters. In *Proceedings of the 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 243–252, Melbourne, Australia, May 2010.

[HCS11] Sascha Hunold, Henri Casanova, and Frédéric Suter. From Simulation to Experiment: A Case Study on Multiprocessor Task Scheduling. In *Proceedings of the 13th Workshop on Advances in Parallel and Distributed Computational Models (APDCM)*, pages 660–667, Anchorage, AK, May 2011.

[HHS10] Sascha Hunold, Ralf Hoffmann, and Frédéric Suter. Jedule: A Tool for Visualizing Schedules of Parallel Applications. In *Proceedings of the 1st International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI)*, pages 169–178, San Diego, CA, September 2010.

[HRS08a] Sascha Hunold, Thomas Rauber, and Frédéric Suter. Redistribution Aware Two-Step Scheduling for Mixed-Parallel Applications. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster)*, pages 50–58, Tsukuba, Japan, September 2008.

[HRS08b] Sascha Hunold, Thomas Rauber, and Frédéric Suter. Scheduling Dynamic Workflows onto Clusters of Clusters using Postponing. In *Proceedings of the 3rd International Workshop on Workflow Systems in e-Science (WSES)*, pages 669–674, Lyon, France, May 2008.

[NS06] Tchimou N'Takpé and Frédéric Suter. Critical path and area based scheduling of parallel task graphs on heterogeneous platforms. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 3–10, Minneapolis, MN, July 2006.

[NS07] Tchimou N'Takpé and Frédéric Suter. Self-Constrained Resource Allocation for Parallel Task Graph Scheduling on Shared Computing Grids. In *Proceedings of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pages 36–41, Cambridge, MA, November 2007.

[NS09] Tchimou N'Takpé and Frédéric Suter. Concurrent Scheduling of Parallel Task Graphs on Multi-Clusters Using Constrained Resource Allocations. In *Proceedings of the 10th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC)*, Rome, Italy, May 2009.

[NSC07] Tchimou N'Takpé, Frédéric Suter, and Henri Casanova. A Comparison of Scheduling Approaches for Mixed-Parallel Applications on Heterogeneous Platforms. In *Proceedings of the 6th International Symposium on Parallel and Distributed Computing (ISPDC)*, Hagenberg, Austria, July 2007. IEEE Computer Press.

[QBS10] Martin Quinson, Laurent Bobelin, and Frédéric Suter. Synthesizing Generic Experimental Environments for Simulation. In *Proceedings of the 5th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, pages 222–229, Fukuoka, Japan, November 2010.

[Sut07] Frédéric Suter. Scheduling Δ-Critical Tasks in Mixed-Parallel Applications on a National Grid. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing (Grid)*, pages 2–9, Austin, TX, September 2007.

Technical and Research Reports

[DMS13] Frédéric Desprez, George S. Markomanolis, and Frédéric Suter. Evaluation of Profiling Tools for the Acquisition of Time-Independent Traces. Technical Report RT-0437, Institut National de Recherche en Informatique et en Automatique (INRIA), July 2013. Available at `http://hal.inria.fr/hal-00842396`.

[FQS08] Marc-Eduard Frincu, Martin Quinson, and Frédéric Suter. Handling Very Large Platforms with the New SimGrid Platform Description Formalism. Technical Report RT-0348, Institut National de Recherche en Informatique et en Automatique (INRIA), February 2008. Available at `http://hal.inria.fr/inria-00256883`.

[MS11] George S. Markomanolis and Frédéric Suter. Time-Independent Trace Acquisition Framework – A Grid'5000 How-to. Technical Report RT-0407, Institut National de Recherche en Informatique et en Automatique (INRIA), April 2011. Available at `http://hal.inria.fr/inria-00586052`.

[SC07] Frédéric Suter and Henri Casanova. Extracting Synthetic Multi-Cluster Platform Configurations from Grid'5000 for Driving Simulation Experiments. Technical Report RT-0341, Institut National de Recherche en Informatique et en Automatique (INRIA), August 2007. Available at `http://hal.inria.fr/inria-00166181`.

Publications Related to the Ph.D.

International Journals

[CCCV+01] Eddy Caron, Serge Chaumette, Sylvain Contassot-Vivier, Frédéric Desprez, Eric Fleury, Claude Gomez, Maurice Goursat, Emmanuel Jeannot, Dominique Lazure, Fréderic Lombard, Jean-Marc Nicod, Laurent Philippe, Martin Quinson, Pierre Ramet, Jean Roman, Franck Rubi, Serge Steer, Frédéric Suter, and Gil Utard. Scilab to Scilab//, the OURAGAN Project. *Parallel Computing*, 27(11):1497–1519, October 2001.

[CDQS04] Eddy Caron, Frédéric Desprez, Martin Quinson, and Frédéric Suter. Performance Evaluation of Linear Algebra Routines. *International Journal of High Performance Computing Applications*, 18(3):373–390, 2004. Special issue on Clusters and Computational Grids for Scientific Computing (CCGSC'02).

[CDS05] Eddy Caron, Frédéric Desprez, and Frédéric Suter. Parallel Extension of a Dynamic Performance Forecasting Tool. *Scalable Computing: Practice and Experience*, 6(1):57–69, 2005. Special issue on selected papers of ISPDC'02.

[DS04] Frédéric Desprez and Frédéric Suter. Impact of Mixed-Parallelism on Parallel Implementations of Strassen and Winograd Matrix Multiplication Algorithms. *Concurrency and Computation:Practice and Experience*, 16(8):771–797, July 2004.

International Conferences

[BDS03] Vincent Boudet, Frédéric Desprez, and Frédéric Suter. One-Step Algorithm for Mixed Data and Task Parallel Scheduling Without Data Replication. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03)*, Nice, France, April 2003.

[CDL+02] Eddy Caron, Frédéric Desprez, Frédéric Lombard, Jean-Marc Nicod, Martin Quinson, and Frédéric Suter. A Scalable Approach to Network Enabled Servers. In B. Monien and R. Feldmann, editors, *Proceedings of the 8th International EuroPar Conference (Research Note)*, volume 2400 of *Lecture Notes in Computer Science*, pages 907–910, Paderborn, Germany, August 2002. Springer-Verlag.

[CLQS02] Philippe Combes, Frédéric Lombard, Martin Quinson, and Frédéric Suter. A Scalable Approach to Network Enabled Servers. In A. Jean-Marie, editor, *Advances in Computing Science - ASIAN 2002. Internet Computing and Modeling, Grid Computing, Peer-to-Peer Computing, and Cluster Computing. Seventh Asian Computing Science Conference*, volume 2550 of *Lecture Notes in Computer Science*, pages 110–124, Hanoï, Vietnam, December 2002. Springer-Verlag.

[CS02] Eddy Caron and Frédéric Suter. Parallel Extension of a Dynamic Performance Forecasting Tool. In *Proceedings of the International Symposium on Parallel and Distributed Computing (ISPDC'02)*, pages 80–93, Iasi, Romania, July 2002.

[DQS01] Frédéric Desprez, Martin Quinson, and Frédéric Suter. Dynamic Performance Forecasting for Network-Enabled Servers in a Heterogeneous Environment. In H.R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, volume III, pages 1421–1427, Las Vegas, June 2001. CSREA Press. ISBN: 1-892512-69-6.

[DS01] Frédéric Desprez and Frédéric Suter. Mixed Parallel Implementations of the Top Level of Strassen and Winograd Matrix Multiplication Algorithms. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01)*, San Francisco, April 2001.

Bibliography

[1] Vikram Adve, Rajive Bagrodia, Ewa Deelman, and Rizos Sakellariou. Compiler-Optimized Simulation of Large-Scale Applications on High Performance Architectures. *Journal of Parallel and Distributed Computing*, 62(3):393–426, 2002.

[2] Rushi Agrawal and Vaishali Sadaphal. Batch Systems: Optimal Scheduling and Processor Optimization. In *Proceedings of 18th International Conference on High Performance Computing (HiPC)*, Bangalore, India, December 2011.

[3] Greg Aldering, Gilles Adam, Pierre Antilogus, Pierre Astier, Roland Bacon, Sébastien Bongard, Christophe Bonnaud, Yannick Copin, Delphine Hardin, François Hénault, David Howell, Jean-Pierre Lemonnier, Jean-Michel Levy, Stewart Loken, Peter Nugent, Reynald Pain, Arlette Pécontal, Emmanuel Pécontal, Saul Perlmutter, Robert Quimby, Kyan Schahmaneche, Gérard Smadja, and Michael Wood-Vasey. Overview of the Nearby Supernova Factory. In J. A. Tyson and S. Wolff, editors, *Survey and Other Telescope Technologies and Discoveries*, volume 4836 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, pages 61–72, December 2002.

[4] Jean-Michel Alimi, Vincent Bouillot, Yann Rasera, Vincent Reverdy, Pier-Stefano Corasaniti, Irène Balmès, Stéphane Requena, Xavier Delaruelle, and Jean-Noel Richet. First-Ever Full Observable Universe Simulation. In *Proceedings of the 2012 IEEE/ACM Conference on High Performance Computing Networking, Storage and Analysis (SC '12)*, Salt Lake City, UT, November 2012.

[5] Ilkay Altintas, Sangeeta Bhagwanani, David Buttler, Sandeep Chandra, Zhengang Cheng, Matthew Coleman, Terence Critchlow, Amarnath Gupta, Wei Han, Ling Liu, Bertram Ludäscher, Calton Pu, Reagan Moore, Arie Shoshani, and Mladen Vouk. A Modeling and Execution Environment for Distributed Scientific Workflows. In *Proceedings of the 15th International Conference on Scientific and Statistical Database Management (SDDBM)*, pages 247–250, Cambridge, MA, July 2003.

[6] Gene Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS 1967 Spring Joint Computer Conference*, volume 30, pages 483–485, April 1967.

[7] David P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Proceedings of the 5th International Workshop on Grid Computing*, pages 4–10, Pittsburgh, PA, November 2004.

[8] Ed Anderson, Zhaojun Bai, Christian Bischof, L. Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, A. Greenbaum, Sven Hammarling, A. McKenney, and Danny Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[9] Hamid Arabnejad and Jorge Barbosa. Performance Evaluation of List Based Scheduling on Heterogeneous Systems. In Michael Alexander, Pasqua D'Ambra, Adam Belloum, George Bosilca, Mario Cannataro, Marco Danelutto, Beniamino Di Martino, Michael Gerndt, Emmanuel Jeannot, Raymond Namyst, Jean Roman, Stephen Scott, Jesper Larsson Träff, Geoffroy Vallée, and Josef Weidendorfer, editors, *Proceedings of the Euro-Par 2011: Parallel Processing Workshops - CCPI, CGWS, HeteroPar, HiBB, HPCVirt, HPPC, HPSS, MDGS, ProPer, Resilience, UCHPC, VHPC*, volume 7155 of *Lecture Notes in Computer Science*, pages 440–449, Bordeaux, France, 2011. Springer.

[10] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a Unified platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[11] Amazon Web Services. `http://aws.amazon.com/`, January 2014.

[12] Rosa Badia, Jesús Labarta, Judit Giménez, and Fransesc Escalé. Dimemas: Predicting MPI applications behavior in Grid environments. In *Proceedings of the Workshop on Grid Applications and Programming Tools*, 2003.

[13] Rajive Bagrodia, Ewa Deelman, and Thomas Phan. Parallel Simulation of Large-Scale Parallel Applications. *International Journal of High Performance Computing Applications*, 15(1):3–12, 2001.

[14] David Bailey, E. Barszcz, John Barton, D. Browning, Robert Carter, Leonardo Dagum, Rod Fatoohi, Paul Frederickson, T. Lasinski, Robert Schreiber, Horst Simon, Venkat Venkatakrishnan, and Sisira Weeratunga. The nas parallel benchmarks - summary and preliminary results. In *Proceedings of Supercomputing '91*, pages 158–165, Albuquerque, NM, November 1991.

[15] Savina Bansal, Padam Kumar, and Kuldip Singh. An Improved Two-Step Algorithm for Task and Data Parallel Scheduling in Distributed Memory Machines. *Parallel Computing*, 32(10):759–774, 2006.

[16] Albert-Lázló Barabási and Réka Albert. Emergence of Scaling in Random Networks. *Science*, 286:509–512, October 1999.

[17] Nicolas Bard, Raphaël Bolze, Eddy Caron, Frédéric Desprez, Michaël Heymann, Anne Friedrich, Luc Moulinier, Ngoc-Hoan Nguyen, Olivier Poch, and Thierry Toursel. Décrypthon Grid - Grid Resources Dedicated to Neuromuscular Disorders. In *Proceedings of the 8th HealthGrid conference*, Paris, France, June 2010.

[18] Olivier Beaumont, Lionel Eyraud-Dubois, and Young Joon Won. Using the Last-mile Model as a Distributed Scheme for Available Bandwidth Prediction. In *Proceedings of the 17th Int. European Conference on Parallel and Distributed Computing (EuroPar)*, volume 6852 of *Lecture Notes in Computer Science*, pages 103–116, Bordeaux, France, September 2011. Springer-Verlag.

[19] William Bell, David Cameron, Luigi Capozza, Paul Millar, Kurt Stockinger, and Floriano Zini. OptorSim - A Grid Simulator for Studying Dynamic Data Replication Strategies. *International Journal of High Performance Computing and Applications*, 17(4):403–416, 2003.

[20] Michael A. Bender, Soumen Chakrabarti, and S. Muthukrishnan. Flow and Stretch Metrics for Scheduling Continuous Job Streams. In *Proceedings of the Ninth Annual ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 270–279, San Francisco, CA, January 1998.

[21] G. Bruce Berriman, John Good, Anastasia Laity, Attila Bergou, Joseph Jacob, Dan Katz, Ewa Deelman, Carl Kesselman, Gurmeet Singh, Mei-Hu Su, and Roy Williams. Montage: a Grid Enabled Image Mosaic Service for the National Virtual Observatory. In *Astronomical Data Analysis Software and Systems (ADASS) XIII*, volume 314, page 593, 2004.

[22] Debmalya Biswas and Blaise Genest. Minimal Observability for Transactional Hierarchical Services. In *Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering (SEKE)*, pages 531–536, San Francisco, CA, July 2008.

[23] L. Susan. Blackford, Jaeyoung Choi, Andrew Cleary, Eduardo D'Azevedo, Jammes Demmel, Inderjit Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, Ken Stanley, David Walker, and R. Clinton Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[24] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Hérault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, YarKhan Asim, and Jack Dongarra. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with PLASMA. In *Proceedings of the 12th Workshop on Parallel and Distributed Scientific and Engineering Computing*, pages 1432–1441, Anchorage, AK, May 2011.

[25] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Hérault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A Generic Distributed DAG Engine for High Performance Computing. *Parallel Computing*, 38(1-2):37–51, 2012.

[26] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Ladislau Bölöni, Muthucumaru Maheswaran, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, Bin Yao, Debra A. Hensgen, and Richard F. Freund. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *Journal of Parallel Distributed Computing*, 61(6):810–837, 2001.

[27] Shirley Browne, Jack Dongarra, Nathan Garner, George Ho, and Philip Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[28] Kevin Butler, Patrick McDaniel, and William Aiello. Optimizing BGP security by exploiting path stability. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 298–310, Alexandria, VA, November 2006.

[29] Rajkumar Buyya and Manzur Murshed. GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. *Concurrency and Computation: Practice and Experience*, 14(13–15):1175–1220, December 2002.

[30] Kenneth Calvert, Matthew Doar, and Ellen Zegura. Modeling Internet Topology. *IEEE Communications Magazine*, 35(6):160–168, June 1997.

[31] Sorina Camarasu-Pop, Tristan Glatard, and Hugues Benoit-Cattin. Simulating Application Workflows and Services Deployed on the European Grid Infrastructure. In *Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 18–25, Delft, Netherlands, May 2013. IEEE Computer Society.

[32] Yves Caniou and Jean-Sébastien Gay. Simbatch: an API for Simulating and Predicting the Performance of Parallel Resources Managed by Batch Systems. In *Proceedings of the Workshop on Secure, Trusted, Manageable and Controllable Grid Services (SGS) in conjonction with EuroPar'08*, pages 223–234, Las Palmas de Gran Canaria, Spain, August 2008.

[33] Louis-Claude Canon, Olivier Dubuisson, Jens Gustedt, and Emmanuel Jeannot. Defining and Controlling the Heterogeneity of a Cluster: the Wrekavoc Tool. *Journal of Systems and Software*, 83(5):786–802, 2010.

[34] Nicolas Capit, Georges Da Costa, Yannis Georgiou, Guillaume Huard, Cyrille Martin, Grégory Mounié, Pierre Neyron, and Olivier Richard. A Batch Scheduler with High Level Components. In *Proceedings of the 5th International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 776–783, Cardiff, UK, May 2005.

[35] Eddy Caron and Frédéric Desprez. Diet: A Scalable Toolbox to Build Network Enabled Servers on the Grid. *International Journal of High Performance Computing Applications*, 20(3):335–352, 2006.

[36] Eddy Caron, Vincent Garonne, and Andrei Tsaregorodtsev. Definition, Modelling and Simulation of a Grid Computing Scheduling System for High Throughput Computing. *Future Generation Computing Systems*, 23(8):968–976, 2007.

[37] Henri Casanova. Simgrid: A Toolkit for the Simulation of Application Scheduling. In *Proceedings of the first IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 430–437. IEEE Computer Society, May 2001.

[38] Henri Casanova, Arnaud Legrand, and Loris Marchal. Scheduling Distributed Applications: the SimGrid Simulation Framework. In *Proceedings of the third IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 138–145. IEEE Computer Society, May 2003.

[39] Henri Casanova, Arnaud Legrand, and Martin Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In *Proceedings of the 10th IEEE International Conference on Computer Modeling and Simulation (UKSim)*, Cambridge, UK, March 2008.

[40] Henri Casanova, Graziano Obertelli, Francine Berman, and Richard Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. In *Proceedings of the High Performance Networking and Computing Conference (SC)*, Dallas, TX, November 2000.

[41] Soumen Chakrabarti, James Demmel, and Katherine Yelick. Models and Scheduling Algorithms for Mixed Data and Task Parallel Programs. *Journal of Parallel and Distributed Computing*, 47(2):168–184, 1997.

[42] Graham Chapman, John Cleese, Terry Gilliam, and Eric Idle. *Monty Python and the Holy Grail*. Methuen Publishing Ltd., April 2002.

[43] Hongtu Chen and Muthucumaru Maheswaran. Distributed Dynamic Scheduling of Composite Tasks on Grid Systems. In *Proceedings of the 12th Heterogeneous Computing Workshop (HCW)*, Fort Lauderdale, FL, April 2002.

[44] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. Understanding tcp incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM SIGCOMM Workshop on Research on Enterprise Networking (WREN)*, pages 73–82, Barcelona, Spain, August 2009. ACM.

[45] Walfredo Cirne and Francine Berman. A Model for Moldable Supercomputer Jobs. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, CA, April 2001.

[46] Jamison Collins, Dean Tullsen, Hang Wong, and John Shen. Dynamic Speculative Precomputation. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 306–317, Austin, TX, December 2001. IEEE/ACM.

[47] Daniel Cordeiro, Grégory Mounié, Swann Perarnau, Denis Trystram, Jean-Marc Vincent, and Frédéric Wagner. Random Graph Generation for Scheduling Simulations. In *Proceedings of 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools)*, Malaga, Spain, March 2010.

[48] Peter Couvares, Tevik Kosar, Alain Roy, Jeff Weber, and Kent Wenger. Workflow Management in Condor. In Ian Taylor, Ewa Deelman, Dennis Gannon, and Matthew Shields, editors, *Workflows for e-Science*, pages 357–375. Springer, January 2007.

[49] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 1–12, San Diego, CA, 1993.

[50] Frank Dabek, Russ Cox, M. Frans Kaashoek, and Robert Morris. Vivaldi: A Decentralized Network Coordinate System. In *Proceedings of the ACM SIGCOMM 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 15–26, Portland, OR, 2004.

[51] Silas De Munck, Kurt Vanmechelen, and Jan Broeckhove. Improving The Scalability of SimGrid Using Dynamic Routing. In *Proceedings of the 9th International Conference on Computational Science (ICCS)*, volume 5544 of *Lecture Notes in Computer Science*, pages 406–415, Baton Rouge, LA, May 2009. Springer.

[52] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commununications of the ACM*, 51(1):107–113, 2008.

[53] Dieter Debels and Mario Vanhoucke. The Discrete Time/Cost Trade-off Problem: Extensions and Heuristic Procedures. *Journal of Scheduling*, 10(4-5), October 2007.

[54] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph Jacob, and Daniel Katz. Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming Journal*, 13(3):219–237, 2005.

[55] Frédéric Desprez and Jonathan Rouzaud-Cornabas. SimGrid Cloud Broker: Simulating the Amazon AWS Cloud. Research Report RR-8380, INRIA, November 2013. Available at `http://hal.inria.fr/hal-00909120`.

[56] Robert Dick, David Rhodes, and Wayne Wolf. TGFF: Task Graphs for Free. In Gaetano Borriello, Ahmed Jerraya, and Luciano Lavagno, editors, *Proceedings of the Sixth International Workshop on Hardware/Software Codesign (CODES)*, pages 97–101, Seattle, WA, March 1998.

[57] Phillip Dickens, Philip Heidelberger, and David Nicol. Parallelized Direct Execution Simulation of Message-Passing Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1090–1105, 1996.

[58] Romain Dolbeau, Stéphane Bihan, and François Bodin. HMPP^{TM}: A Hybrid Multi-core Parallel Programming Environment. In *Proceedings of the First Workshop on General Purpose Processing on Graphics Processing Units*, Boston, MA, October 2007.

[59] Bruno Donassolo, Henri Casanova, Arnaud Legrand, and Pedro Velho. Fast and Scalable Simulation of Volunteer Computing Systems Using SimGrid. In *Proceedings of the Second Workshop on Large-Scale System and Application Performance (LSAP)*, pages 605–612, Chicago, IL, June 2010.

[60] Bruno Donassolo, Arnaud Legrand, and Claudio Geyer. Non-Cooperative Scheduling Considered Harmful in Collaborative Volunteer Computing Environments. In *Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, Newport Beach, CA, May 2011.

[61] Alvise Dorigo, Peter Elmer, Fabrizio Furano, and Andrew Hanushevsky. XROOTD – A Highly Scalable Architecture for Data Access. *WSEAS Transactions on Computers*, 1(4.3), 2005.

[62] Allen B. Downey. A Model For Speedup of Parallel Programs. Technical Report UCB/CSD-97-933, EECS Department, University of California, Berkeley, January 1997.

[63] Pierre-François Dutot. Hierarchical Scheduling for Moldable Tasks. In *Proceedings of the 11th International Euro-Par Conference*, volume 3648 of *Lecture Notes in Computer Science*, pages 302–311. Springer-Verlag, 2005.

[64] Pierre-François Dutot, Lionel Eyraud-Dubois, Grégory Mounié, and Denis Trystram. Bi-criteria Algorithm for Scheduling Jobs on Cluster Platforms. In *Proceedings of the 16th ACM symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 125–132, Barcelona, Spain, June 2004.

[65] Pierre-François Dutot, Grégory Mounié, and Denis Trystram. Scheduling Parallel Tasks – Approximation Algorithms. In J. Y.-T. Leung, editor, *Handbook of Scheduling*, chapter 26. CRC Press, 2004.

[66] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On Power-Law Relationships of the Internet Topology. In *ACM Annual Conference of the Special Interest Group on Data Communication (SIGCOMM)*, pages 251–262, Cambridge, MA, September 1999.

[67] Dror Feitelson. *Workload Modeling for Computer Systems Performance Evaluation*. [Online], 2012. Available at: `http://www.cs.huji.ac.il/~feit/wlmod`.

[68] Dror Feitelson and Larry Rudolph. Toward Convergence in Job Schedulers for Parallel Supercomputers. In *Proceedings of the Second Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, volume 1162 of *Lecture Notes in Computer Science*, pages 1–26, Honolulu, HI, April 1996.

[69] Dror Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth Sevcik, and Parkson Wong. Theory and Practice in Parallel Job Scheduling. In *Proceedings of the Third Workshop on Job Scheduling Strategies for Parallel Processing (JSSSP)*, volume 1291 of *Lecture Notes in Computer Science*, pages 1–34, Geneva, Switerland, April 1997. Springer.

[70] Dror Feitelson, Dan Tsafrir, and David Krakov. Experience with the Parallel Workloads Archive. Technical Report 2012-6, School of Computer Science and Engineering, The Hebrew University of Jerusalem, April 2012.

[71] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of High Performance Computing Applications*, 11(2):115–128, 1997.

[72] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[73] Patrick Fuhrmann and Volker Gülzow. dCache, Storage System for the Future. In Wolfgang E. Nagel, Wolfgang V. Walter, and Wolfgang Lehner, editors, *Proceedings of the 12th International Euro-Par Conference*, volume 4128 of *Lecture Notes in Computer Science*, Dresden, Germany, August 2006. Springer.

[74] Edgar Gabriel, Graham Fagg, Georges Bosilca, Thara Angskun, Jack Dongarra, Jeffrey Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph Castain, David Daniel, Richard Graham, and Timothy Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 97–104, Budapest, Hungary, September 2004. Springer.

[75] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[76] Markus Geimer, Felix Wolf, Brian Wylie, and Bernd Mohr. A Scalable Tool Architecture for Diagnosing Wait States in Massively Parallel Applications. *Parallel Computing*, 35(7):375–388, 2009.

[77] Wolfgang Gentzsch. Sun Grid Engine: Towards Creating a Compute Power Grid. In *Proceedings of the First IEEE International Symposium on Cluster Computing and the Grid*, pages 35–39, Brisbane, Australia, May 2001. IEEE Computer Society (Los Alamitos, CA).

[78] Tristan Glatard and Sorina Camarasu-Pop. Modelling Pilot-Job Applications on Production Grids. In *Proceedings of the 7th International workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (Heteropar 09)*, pages 140–149, Delft, The Netherlands, August 2009.

[79] Tristan Glatard and Sorina Camarasu-Pop. A Model of Pilot-Job Resource Provisioning on Production Grids. *Parallel Computing*, 37(10-11):684–692, 2011.

[80] Tristan Glatard, Johan Montagnat, Diane Lingrand, and Xavier Pennec. Flexible and Efficient Workflow Deployement of Data-Intensive Applications on Grids With MOTEUR. *International Journal of High Performance Computing Applications*, 22(3):347–360, August 2008.

[81] Robert Graves, Thomas Jordan, Scott Callaghan, Ewa Deelman, Edward Field, Gideon Juve, Carl Kesselman, Philip Maechling, Gaurang Mehta, Kevin Milner, David Okaya, Patrick Small, and Karan Vahi. CyberShake: A Physics-Based Seismic Hazard Model for Southern California. *Pure and Applied Geophysics*, 168(3-4):367–381, 2011.

[82] William Gropp. MPICH2: A new start for MPI implementations. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting*, volume 2474 of *Lecture Notes in Computer Science*, page 7, Linz, Austria, Oct. 2002. Springer.

[83] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. Scientific And Engineering Computation Series. MIT Press, 2nd edition, 1999.

[84] Duncan Grove and Paul Coddington. Communication Benchmarking and Performance Modelling of MPI Programs on Cluster Computers. *Journal of Supercomputing*, 34(2):201–217, 2005.

[85] Jens Gustedt, Emmanuel Jeannot, and Martin Quinson. Experimental Validation in Large-Scale Systems: a Survey of Methodologies. *Parallel Processing Letters*, 19(3):399–418, 2009.

[86] Tarek Hagras and Jan Janecek. A Simple Scheduling Heuristic for Heterogeneous Computing Environments. In *Proceedings of the 2nd International Symposium on Parallel and Distributed Computing (ISPDC'03)*, pages 104–110, Ljubljana, Slovenia, October 2003.

[87] Marc-André Hermanns, Markus Geimer, Felix Wolf, and Brian Wylie. Verifying Causality between Distant Performance Phenomena in Large-Scale MPI Applications. In *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 78–84, Weimar, Germany, February 2009.

[88] Tony Hey, Stewart Tansley, and Kristin Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, October 2009.

[89] Torsten Hoefler, Christian Siebert, and Andrew Lumsdaine. LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model. In *Proceedings of the ACM Workshop on Large-Scale System and Application Performance*, pages 597–604, Chicago, IL, June 2010.

[90] Duncan Hull, Katherine Wolstencroft, Robert Stevens, Carole Goble, Matthew Pocock, Peter Li, and Thomas Oinn. Taverna: a Tool for Building and Running Workflows of Services. *Nucleic Acids Research*, 34(Web Server issue):729–732, July 2006.

[91] Sascha Hunold. Low-Cost Tuning of Two-Step Algorithms for Scheduling Mixed-Parallel Applications onto Homogeneous Clusters. In *Proceedings of the 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 253–262, Melbourne, Australia, May 2010.

[92] Sascha Hunold, Thomas Rauber, and Gudula Rünger. Dynamic Scheduling of Multi-Processor Tasks on Clusters of Clusters. In *Proceedings of the 6th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (HeteroPar)*, pages 507–514, Austin, TX, September 2007.

[93] Fumihiko Ino, Noriyuki Fujimoto, and Kenichi Hagihara. LogGPS: a Parallel Computational Model for Synchronization Analysis. In *Proceedings of the eighth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 133–142, Snowbird, UT, 2001.

[94] The IOzone Filesystem Benchmark. `http://www.iozone.org/`, October 2006.

[95] Michael Iverson and Füsun Özgüner. Hierarchical, Competitive Scheduling of Multiple DAGs in a Dynamic Heterogeneous Environment. *Distributed System Engineering*, 6(3):112–120, 1999.

[96] Klaus Jansen and Hu Zhang. An Approximation Algorithm for Scheduling Malleable Tasks under General Precedence Constraints. *ACM Transactions on Algorithms*, 2(3):416–434, 2006.

[97] Gideon Juve, Ann L. Chervenak, Ewa Deelman, Shishir Bharathi, Gaurang Mehta, and Karan Vahi. Characterizing and Profiling Scientific Workflows. *Future Generation Computer Systems*, 29(3):682–692, 2013.

[98] Md Kamruzzaman, Steven Swanson, and Dean Tullsen. Inter-core Prefetching for Multicore Processors Using Migrating Helper Threads. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 393–404, Newport Beach, CA, March 2011.

[99] Yang-Suk Kee, Henri Casanova, and Andrew Chien. Realistic Modeling and Synthesis of Resources for Computational Grids. In *Proceedings of ACM/IEEE SuperComputing 2004 (SC)*, Pittsburgh, PA, November 2004.

[100] Dalibor Klusácek, Ludek Matyska, and Hana Rudová. Alea - Grid Scheduling Simulation Environment. In *Proceedings of the 7th International Conference on Parallel Processing and Applied Mathematics (PPAM'07)*, volume 4967 of *Lecture Notes in Computer Science*, pages 1029–1038, Gdansk, Poland, September 2007.

[101] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias Müller, and Wolfgang Nagel. The Vampir Performance Analysis Tool-Set. In *Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing (HLRS)*, pages 139–155, Stuttgart, Germany, July 2008.

[102] Andreas Knüpfer, Christian Rössel, Dieteran Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, WolfgangE. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch, editors, *Tools for High Performance Computing 2011*, pages 79–91. Springer Berlin Heidelberg, 2012.

[103] Wagner Kolberg, Pedro de Botelho Marcos, Julio Anjos, Alexandre Miyazaki, Claudio Geyer, and Luciana and Arantes. MRSG – A MapReduce Simulator over SimGrid. *Parallel Computing*, 39(4-5):233–244, April-May 2013.

[104] Rick Kufrin. Perfsuite: An Accessible, Open Source Performance Analysis Environment for Linux. In *Proceedings of the 6th International Conference on Linux Clusters: The HPC Revolution 2005 (LCI-05)*, Chapel Hill, NC, April 2005.

[105] Mario Lassnig, Thomas Fahringer, Vincent Garonne, Angelos Molfetas, and Martin Barisits. A Similarity Measure for Time, Frequency, and Dependencies in Large-Scale Workloads. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011.

[106] Edgar León, Rolf Riesen, Arthur Maccabe, and Patrick Bridges. Instruction-Level Simulation of a Cluster at Scale. In *Proceedings of the International Conference for High Performance Computing and Communications (SC)*, Portland, OR, November 2009.

[107] Renaud Lepère, Denis Trystram, and Woeginger Gerhard. Approximation Algorithms for Scheduling Malleable Tasks under Precedence Constraints. In *Proceedings of the 9th Annual European Symposium on Algorithms (ESA)*, number 2161 in Lecture Notes in Computer Science, pages 146–157, Aarhus, Denmark, August 2001. Springer-Verlag.

[108] Renaud Lepère, Denis Trystram, and Gerhard Woeginger. Approximation Algorithms for Scheduling Malleable Tasks Under Precedence Constraints. *International Journal on Foundations of Computer Science*, 13(4):613–627, 2002.

[109] Hui Li, David Groep, and Lex Wolters. Workload Characteristics of a Multi-cluster Supercomputer. In Dror. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Proceedings of the 10th workshop on Job Scheduling Strategies for Parallel Processing*, volume 3277 of *Lecture Notes in Computer Science*, pages 176–193. Springer, 2005.

[110] Dong Lu and Peter Dinda. GridG: Generating Realistic Computational Grids. In *ACM SIGMETRICS Performance Evaluation Review*, volume 30, pages 33–40, March 2003.

[111] Maciej Malawski, Gideon Juve, Ewa Deelman, and Jarek Nabrzyski. Cost- and Deadline-Constrained Provisioning for Scientific Workflow Ensembles in IaaS Clouds. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, November 2012.

[112] Anthony Mayer, Steve McGough, Nathalie Furmento, William Lee, Steven Newhouse, and John Darlington. ICENI Dataflow and Workflow: Composition and Scheduling in Space and Time. In *UK e-Science All Hands Meeting*, pages 627–634. IOP Publishing Ltd, 2003.

[113] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. BRITE: An Approach to Universal Topology Generation. In *Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS)*, Cincinnati, OH, August 2001.

[114] Reagan W Moore and Arcot Rajsekar. iRODS: Data Sharing Technology Integrating Communities of Practice. In *Proceedings of the IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, pages 1984–1987, 2010.

[115] Ahuva Mu'alem and Dror Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. *IEEE Transactions on Parallel and Distributed Computing*, 12(6):529–543, 2001.

[116] Gilles Muller, Yoann Padioleau, Julia Lawall, and René Rydhof Hansen. Semantic Patches Considered Helpful. *SIGOPS Operating Systems Review*, 40:90–92, July 2006.

[117] Michael Noeth, Franck Mueller, Martin Schulz, and Bronis de Supinski. Scalable Compression and Replay of Communication Traces in Massively Parallel Environments. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*, pages 1–11, Long Beach, CA, March 2007.

[118] Alberto Núñez, Javier Fernández, José-Daniel Garcia, Felix Garcia, and Jesús Carretero. New Techniques for Simulating High Performance MPI Applications on Large Storage Networks. *Journal of Supercomputing*, 51(1):40–57, 2010.

[119] Hyunok Oh and Soonhoi Ha. A Static Scheduling Heuristic for Heterogeneous Processors. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Proceedings of the Second International Euro-Par Conference on Parallel Processing (Euro-Par'96)*, volume 1124 of *Lecture Notes in Computer Science*, pages 573–577, Lyon, France, August 1996.

[120] Simon Ostermann, Radu Prodan, Thomas Fahringer, Alexandru Iosup, and Dick Epema. Trace-Based Characteristics of Grid Workflows. In Thierry Priol and Marco Vanneschi, editors, *From Grids to Service and Pervasive Computing*, volume 10 of *CoreGRID*, pages 191–204. Springer-Verlag, 2008.

[121] Brad Penoff, Alan Wagner, Michael Tüxen, and Irene Rüngeler. MPI-NetSim: A network simulation module for MPI. In *Proceedings of the 15th International Conference on Parallel and Distributed Systems (ICPADS)*, Shenzen, China, December 2009.

[122] Sundeep Prakash, Ewa Deelman, and Rajive Bagrodia. Asynchronous Parallel Simulation of Parallel Programs. *IEEE Transactions on Software Engineering*, 26(5):385–400, 2000.

[123] Martin Quinson. GRAS: a Research and Development Framework for Grid and P2P Infrastructures. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*. Acta Press, 2006.

[124] Andrei Radulescu, Cristina Nicolescu, Arjan van Gemund, and Pieter Jonker. CPR: Mixed Task and Data Parallel Scheduling for Distributed Systems. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, CA, April 2001. Best Paper Award.

[125] Andrei Radulescu and Arjan van Gemund. A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling. In *Proceedings of the 15th International Conference on Parallel Processing (ICPP)*, pages 69–76, Valencia, Spain, September 2001.

[126] Shankar Ramaswamy, Eugene Hodges IV, and Prithviraj Banerjee. Compiling MATLAB Programs to ScaLAPACK: Exploiting Task and Data Parallelism. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS)*, pages 613–619, Honolulu, HI, April 1996.

[127] Prasun Ratn, Franck Mueller, Bronis de Supinski, and Martin Schulz. Preserving Time in Large-scale Communication Traces. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, pages 46–55, Island of Kos, Greece, June 2008.

[128] Thomas Rauber and Gudula Rünger. Compiler Support for Task Scheduling in Hierarchical Execution Models. *Journal of Systems Architecture*, 45(6-7):483–503, 1999.

[129] Ralf Reussner, Peter Sanders, and Jesper Larsson Träff. SKaMPI: a Comprehensive Benchmark for Public Benchmarking of MPI. *Scientific Programming*, 10(1):55–65, 2002.

[130] Rolf Riesen. A Hybrid MPI Simulator. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 1–9, Barcelona, Spain, September 2006.

[131] Rizos Sakellariou and Henan Zhao. A Hybrid Heuristic for DAG Scheduling on Heterogeneous Systems. In *Proceedings of the 13th Heterogeneous Computing Workshop (HCW'04)*, Santa Fe, NM, April 2004.

[132] Elizeu Santos-Neto, Walfredo Cirne, Francisco Vilar Brasileiro, and Aliandro Lima. Exploiting Replication and Data Reuse to Efficiently Schedule Data-Intensive Applications on Grids. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Proceedings of the 10th International Workshop on Job Scheduling Strategies for Parallel Processing*, volume 3277 of *Lecture Notes in Computer Science*, pages 210–232. Springer, 2005.

[133] Jonathan Schaeffer and Andrés Gómez Casanova. TReqS: The Tape REQuest Scheduler. *Journal of Physics: Conference Series*, 331, 2011.

[134] Lucas Mello Schnorr, Guillaume Huard, and Philippe Navaux. Triva: Interactive 3D Visualization for Performance Analysis of Parallel Applications. *Future Generation Computer Systems*, 26(3):348–358, 2010.

[135] Lucas Mello Schnorr, Arnaud Legrand, and Jean-Marc Vincent. Detection and Analysis of Resource Usage Anomalies in Large Distributed Systems Through Multi-scale Visualization. *Concurrency and Computation: Practice and Experience*, 2012.

[136] Sameer Shende and Allen Malony. The Tau Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[137] Gilbert Sih and Edward Lee. A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–187, 1993.

[138] Martin Skutella. Approximation Algorithms for the Discrete Time-Cost Tradeoff Problem. *Mathematics of Operations Research*, 23(4):909–929, 1998.

[139] Allan Snavely, Laura Carrington, Nicole Wolter, Jesús Labarta, Rosa Badia, and Avi Purkayastha. A Framework for Application Performance Modeling and Prediction. In *Proceedings of the International Conference for High Performance Computing and Communications (SC)*, Baltimore, MD, November 2002.

[140] Fengguang Song, Stanimire Tomov, and Jack Dongarra. Enabling and Scaling Matrix Computations on Heterogeneous Multi-Core and Multi-GPU Systems. In *Proceeedings of the International Conference on Supercomputing*, pages 365–376, Venice, Italy, June 2012.

[141] Hyo Jung Song, Xainan Liu, Dennis Jakobsen, Ranjita Bhagwan, Xingbing Zhang, Kenjiro Taura, and Andrew Chien. The MicroGrid: a scientific tool for modeling computational grids. In *Proceedings of the ACM/IEEE conference on Supercomputing*, Dallas, TX, November 2000.

[142] Benhur Stein, Jacques Chassin de Kergommeaux, and Pierre-Eric Bernard. Pajé, an Interactive Visualization Tool for Tuning Multi-Threaded Parallel Applications. *Parallel Computing*, 26:1253–1274, 2000.

[143] Tomasz Szepieniec and Marian Bubak. Investigation of the DAG Eligible Jobs Maximization Algorithm in a Grid. In *Proceedings of the 9th IEEE/ACM International Conference on Grid Computing (Grid)*, pages 340–345, Tsukuba, Japan, September 2008.

[144] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[145] Mustafa Tikir, Michael Laurenzano, Laura Carrington, and Allan Snavely. PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications. In *Proceedings of the 15th International EuroPar Conference*, volume 5704 of *Lecture Notes in Computer Science*, pages 135–148, Delft, The Netherlands, August 2009.

[146] Takao Tobita and Hironori Kasahara. A Standard Task Graph Set for Fair Evaluation of Multiprocessor Scheduling Algorithms. *Journal of Scheduling*, 5(5):379–394, 2002.

[147] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[148] Amin Vahdat, Ken Yocum, Kevin Walsh, Pryia Mahadevan, Dejan Kostic, Jeffrey Chase, and David Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[149] Wil van der Aalst, Alistair Barros, Arthur ter Hofstede, and Bartek Kiepuszewski. Advanced Workflow Patterns. In *Proceedings of the 7th International. Conference on Cooperative Information Systems (CoopIS)*, pages 18–29, Eilat, Israël, September 2000.

[150] Nagavijayalakshmi Vydyanathan, Sriram Krishnamoorthy, Gerald M. Sabin, Ümit V. Çatalyürek, Tahsin M. Kurç, Ponnuswamy Sadayappan, and Joel H. Saltz. An Integrated Approach for Processor Allocation and Scheduling of Mixed-Parallel Applications. In *Proceedings of the 35th International Conference on Parallel Processing (ICPP)*, pages 443–450, Colombus, OH, August 2006.

[151] Nagavijayalakshmi Vydyanathan, Sriram Krishnamoorthy, Gerald M. Sabin, Ümit V. Çatalyürek, Tahsin M. Kurç, Ponnuswamy Sadayappan, and Joel H. Saltz. An Integrated Approach to Locality-Conscious Processor Allocation and Scheduling of Mixed-Parallel Applications. *IEEE Transactions on Parallel and Distributed Systems*, 20(8):1158–1172, 2009.

[152] Lizhe Wang, Gregor von Laszewski, Jai Dayal, and Fugang Wang. Towards Energy Aware Scheduling for Precedence Constrained Parallel Tasks in a Cluster with DVFS. In *Proceedings of the 0th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, pages 368–377, Melbourne, Australia, May 2010.

[153] Bernard Waxman. Routing of Multipoint Connections. *IEEE Journal on Selected Areas in Communications*, 6(9):1617–1622, December 1988.

[154] Andy Yoo, Morris Jette, and Mark Grondona. SLURM: Simple Linux Utility for Resource Management. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Proceedings of the 9th International Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 44–60. Springer, Seattle, WA, 2003.

[155] Ellen Zegura, Kenneth Calvert, and Michael Donahoo. A Quantitative Comparison of Graph-based Models for Internet Topology. *IEEE/ACM Transactions on Networking*, 5(6), December 1997.

[156] Jidong Zhai, Wenguang Chen, and Weiming Zheng. PHANTOM: Predicting Performance of Parallel Applications on Large-Scale Parallel Machines Using a Single Node. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 305–314, January 2010.

[157] Henan Zhao and Rizos Sakellariou. Scheduling Multiple DAGs onto Heterogeneous Systems. In *Proceedings of the 15th Heterogeneous Computing Workshop (HCW)*, Rhodes Island, Greece, April 2006.

[158] Gengbin Zheng, Gunavardhan Kakulapati, and Laxmikant Kale. BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, Santa Fe, NM, April 2004.

[159] Gengbin Zheng, Terry Wilmarth, Praveen Jagadishprasad, and Laxmikant Kalé. Simulation-Based Performance Prediction for Large Parallel Machines. *International Journal of Parallel Programming*, 33(2-3):183–207, 2005.