Workflow-based applications performance and execution cost optimization on cloud infrastructures

Tram Truong Huu

Université Nice Sophia Antipolis

ECOLE DOCTORALE STIC

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THESE

pour obtenir le titre de

Docteur en Sciences

de l'Université Nice Sophia Antipolis

mention

INFORMATIQUE

présentée et soutenue par

Tram TRUONG HUU

le 13 décembre 2010

Workflow-based applications performance and execution cost optimization on cloud infrastructures

Rapporteurs:

M	**Emmanuel**	**JEANNOT**
Mme	**Pascale**	**VICAT-BLANC PRIMET**

Jury:

M	**Tristan**	**GLATARD**	
M	**Emmanuel**	**JEANNOT**	
M	**Johan**	**MONTAGNAT**	Directeur
M	**Michel**	**RIVEILL**	Président

A mes parents et Johan Montagnat.

Remerciements

J'exprime mes profonds remerciements à toutes les personnes qui, de près ou de loin, ont contribué à la concrétisation de ce travail de thèse de doctorat. Ces remerciements sont rédigés dans un moment émouvant que j'aurais peut-être oublié un des noms qui suivent. Mais ces remerciements auraient certainement été tout autres.

Tout d'abord, mes remerciements s'addressent à Johan Montagnat, mon directeur de thèse. Sans lui, cette thèse n'aurait jamais vu le jour. Tout au long de ces trois années d'études, il a su me laisser la liberté nécessaire à l'accomplissement de mes travaux, tout en y gardant un œil critique et avisé. Sa grande disponibilité, sa capacité à suggéger des pistes de recherche si riches et sa patience pour parvenir à m'expliquer mille fois une chose simple malgré mon faible niveau de français, ont sûrement été la clé de réussite de ce travail. A côté de son encadrement scientifique, son encouragement ainsi que son enthousiasme ont sans cesse renouvelé mon ardeur au travail et ont ainsi constitué un précieux soutien. Pour toutes ces raisons, cette thèse lui est dédiée.

Ensuite, je suis très reconnaissant à Pascale Vicat-Blanc Primet et Emmanuel Jeannot d'avoir accepté le rôle de rapporteur, d'autant que je dois reconnaître que je ne leur ai pas facilité la tâche. Leurs remarques et suggestions lors de la lecture de mon manuscrit m'ont permis d'apporter des améliorations à la qualité de ce dernier. Un grand merci aussi à Pascale pour la collaboration intéressante que nous avons eu pendant ces trois années dans le projet HIPCAL. Grâce à cette collaboration, j'ai eu l'occasion de travailler avec les ingénieurs experts de l'équipe RESO à Lyon, Olivier Mornard, Fabienne Anhalt, Guilherme Koslovski et Romaric Guillier, qui m'ont aidé à compléter des expériences de la partie de validation expérimentale de cette thèse.

Je tiens à remercier les membres du jury. Michel Riveill pour avoir accepté la charge de président du jury. Emmanuel Jeannot, Tristan Glatard et Johan Montagnat pour avoir participé dans le jury malgré leur emploi du temps très chargé.

Je remercie aussi très vivement les membres de deux équipes, MODALIS et RAINBOW pour leur gentillesse et leur accueil chaleureux. Diane Lingrand pour sa correction quotidienne des fautes de français qui m'a permis d'améliorer pas à pas. Clémentine, Cédric, Javier, Ketan et les autres avec qui j'ai partagé ces années inoubliables.

Enfin, remercier mes parents, mes frères et les proches de ma famille pour m'avoir permis d'être ce que je suis et m'avoir donné la liberté nécessaire à mon épanouissement. Que cette thèse témoigne de mon respect et de mon amour. Thanks a lot to Chuyen for her patience to wait for me for the last six years. Her support and attention gave me the motivation to complete a milestone in my life.

Abstract

Cloud computing is increasingly exploited to tackle the computing challenges raised in both science and industry. Clouds provide computing, network and storage resources on demand to satisfy the needs of large-scale distributed applications. To adapt to the diversity of cloud infrastructures and usage, new tools and models are needed. Estimating the amount of resources consumed by each application in particular is a difficult problem, both for end users who aim at minimizing their cost and infrastructure providers who aim at controlling their resources allocation. Although a quasi-unlimited amount of resources may be allocated, a trade-off has to be found between (i) the allocated infrastructure cost, (ii) the expected performance and (iii) the optimal performance achievable that depends on the level of parallelization of the application. Focusing on medical image analysis, a scientific domain representative of the large class of data intensive distributed applications, this thesis proposes a fine-grained cost function model relying on the expertise captured from the application. Based on this cost function model, four resources allocation strategies are proposed. Taking into account both computing and network resources, these strategies help users to determine the amount of resources to reserve and compose their execution environment. In addition, the data transfer overhead and the low reliability level, which are well-known problems of large-scale distributed systems impacting application performance and infrastructure usage cost, are also considered.

The experiments reported in this thesis were carried out on the Aladdin/Grid'5000 infrastructure, using the HIPerNet virtualization middleware. This virtual platform manager enables the joint virtualization of computing and network resources. A real medical image analysis application was considered for all experimental validations. The experimental results assess the validity of the approach in terms of infrastructure cost and application performance control. Our contributions both facilitate the exploitation of cloud infrastructures, delivering a higher quality of services to end users, and help the planning of cloud resources delivery.

Résumé

Les infrastructures virtuelles de cloud sont de plus en plus exploitées pour relever les défis de calcul intensif en sciences comme dans l'industrie. Elles fournissent des ressources de calcul, de communication et de stockage à la demande pour satisfaire les besoins des applications à grande échelle. Pour s'adapter à la diversité de ces infrastructures, de nouveaux outils et modèles sont nécessaires. L'estimation de la quantité de ressources consommées par chaque application est un problème particulièrement difficile, tant pour les utilisateurs qui visent à minimiser leurs coûts que pour les fournisseurs d'infrastructure qui visent à contrôler l'allocation des ressources. Même si une quantité quasi illimitée de ressources peut être allouée, un compromis doit être trouvé entre (i) le coût de l'infrastructure allouée, (ii) la performance attendue et (iii) la performance optimale atteignable qui dépend du niveau de parallélisme inhérent à l'application. Partant du cas d'utilisation de l'analyse d'images médicales, un domaine scientifique représentatif d'un grand nombre d'applications à grande échelle, cette thèse propose un modèle de coût à grain fin qui s'appuie sur l'expertise extraite de l'application formalisée comme un flot. Quatre stratégies d'allocation des ressources basées sur ce modèle de coût sont introduites. En tenant compte à la fois des ressources de calcul et de communication, ces stratégies permettent aux utilisateurs de déterminer la quantité de ressources de calcul et de bande passante à réserver afin de composer leur environnement d'exécution. De plus, l'optimisation du transfert de données et la faible fiabilité des systèmes à grande échelle, qui sont des problèmes bien connus ayant un impact sur la performance de l'application et donc sur le coût d'utilisation des infrastructures, sont également prises en considération.

Les expériences exposées dans cette thèse ont été effectuées sur la plateforme Aladdin/Grid'5000, en utilisant l'intergiciel HIPerNet. Ce gestionnaire de plateforme virtuelle permet la virtualisation de ressources de calcul et de communication. Une application réelle d'analyse d'images médicales a été utilisée pour toutes les validations expérimentales. Les résultats expérimentaux montrent la validité de l'approche en termes de contrôle du coût de l'infrastructure et de la performance des applications. Nos contributions facilitent à la fois l'exploitation des infrastructures de cloud, offrant une meilleure qualité de services aux utilisateurs, et la planification de la mise à disposition des ressources virtualisées.

Contents

Chapter 1

Introduction

Over the last years, distributed computing infrastructures have been increasingly exploited for tackling the computation challenges raised in both science and industry. They provide computing, network and storage resources to deal with the needs of large-scale applications. From a single site *cluster*, where the amount of resources is limited, distributed computing infrastructures have rapidly evolved to larger scale *Grids* where users have the illusion that they are working on an unlimited-sized infrastructure. Depending on the parallelism level supported in their applications, users can recruit a large amount of resources to speed up their applications execution. Grids are well suited to efficiently process massively data parallel application through the delivery of High Throughput Computing. *Clouds* are a natural next step of distributed computing infrastructures towards the allocation of resources on demand. Users on Clouds reserve resources to satisfy their performance requirements and pay the execution cost "per use". A challenging problem, both for business and research communities, is how to execute applications in a cost-efficient manner to obtain the desired level of performance. Focusing on medical image analysis, a scientific domain representative of the large class of data intensive distributed applications, this thesis proposes several optimizations in terms of data management, resource allocation and infrastructure reliability to find out an acceptable trade-off between the execution cost and achievable performance. In this thesis, we particularly focus on cloud infrastructures which enable better control of resources allocation and therefore execution cost. However, the results of this thesis are applicable to a broader set distributed computing infrastructures.

1.1 Distributed multi-user systems

Since the late 1960s, the term *distributed system* has appeared in the computer science dictionary to depict an aggregation of autonomous computers that communicate through a computer network. Computers interact with each other to achieve a common goal [Enslow, 1978]. Scientific and business organizations build their own *cluster* infrastructures to fulfill their internal computation requirements. With the rise of the Internet, the pressure for decentralization and distribution of software and hardware resources has increased tremendously. Not only service decomposition in Information Technology is occurring inside an enterprise but also software and hardware resources can be assembled from cross enterprises and service provider systems. This leads to the need for new abstractions and architectures that can

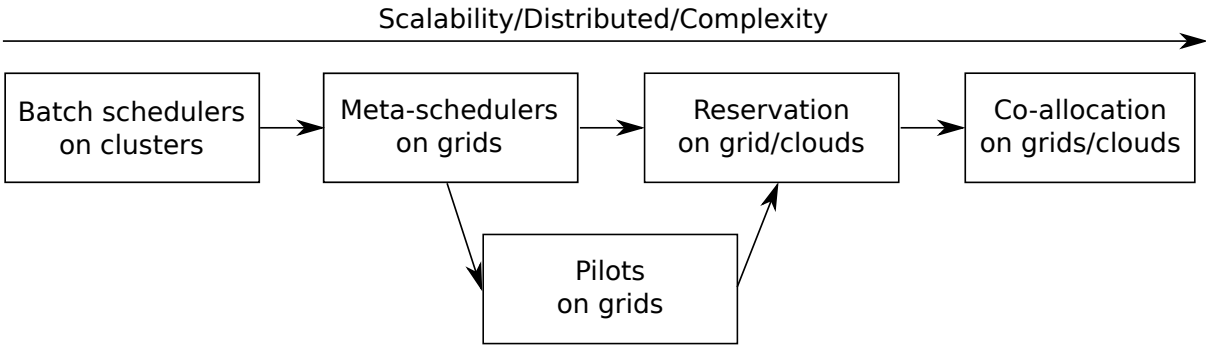


Figure 1.1: Evolution of distributed systems and their resources allocation mechanisms

enable dynamic resources sharing in an environment which spans over multiple administrative domains distributed across the world.

Grids with a flexible, secure and coordinated sharing of heterogeneous resources among dynamic collections of individuals and institutions, are emerging as a pinnacle of the evolution of distributed computing infrastructures [Foster et al., 2001]. Since their foundation, Grids have become increasingly important as they are employed in various critical areas such as weather forecasting, flight control system, simulation, bio-informatics or biomedical applications. Distributed systems become *multi-users* due to the rapid growth of their user communities and the push of institutions towards better sharing of computing resources. As the goals and the way of using systems differ between users, user communities on distributed computing systems are divided into *Virtual Organizations* (VOs). Each VO refers to a dynamic group of users defining a set of resources-sharing rules and conditions. VO members share resources and collaborate with each others for the purpose of one or more identified goals.

Clouds, the new generation of distributed systems, provide resources on demand. By using virtualization of both operating system and network, clouds give users more flexibility to design customized execution environments. In addition, the amount of cloud resources reserved for an execution can vary elastically depending on the applications workload.

Despite the evolution of distributed computing infrastructures and strong efforts put forward to harness systems complexity and satisfy users requirements, many challenging problems still remain. Among them, resources reservation and scheduling are NP-complete problems as proved in the literature [Ullman, 1975]. How to best allocate resources to a given application with a determined input data set is still a difficult question to answer. Many allocation approaches have been implemented to adapt to the evolution of distributed systems. Figure 1.1 depicts the evolution of resources allocation approaches observed over the last years to adapt to the emergence of large-scale distributed systems. On small scale systems centrally managed, tasks schedulers are used. The move to open large-scale grid infrastructures introduce the need for meta-schedulers in charge of coarse-grained load balancing. Finally, the emergence of dedicated virtualized environments push towards new methods of resources allocation.

1.1.1 Resources allocation

Efficient utilization of resources is not only needed from the financial perspective in a commercial distributed system. It is also one of the main requirements in non-business type sharing of resources, such as in scientific Grids. High efficiency results in cost-effectiveness and therefore justifies the use of dis-

tributed systems. However, it is challenging to meet the quality of service objectives of an application running on a distributed computing infrastructure while maintaining high system performance. The reason comes from the fact that optimizing the quality of services delivered to users on an infrastructure usually conflicts with efficiency goals of resources providers. Furthermore, distributed infrastructures are shared by multiple applications that belong to different administrative domains. There are many ways to reserve the resources to execute the application on distributed infrastructures through the middleware and resource managers based on a scheduling policy. Sections 1.1.1.1 to 1.1.1.6 review the main approaches depicted in figure 1.1 and give some insight on their adaptation to distributed infrastructures evolution.

1.1.1.1 Batch scheduler

The most popular resource manager used on a traditional system is the *batch* scheduler. As presented in figure 1.2a on page 21, a batch scheduler receives tasks submissions which are independent sequential or parallel computations requested by users. It schedules the execution of these tasks on the resources it manages, determining when and where each task is to be executed. Consequently, resources are allocated along with the arrival of tasks and they are accessed through tasks submission. Compared to static scheduling problems which presume perfect knowledge of the current execution system state and fully predictable changes, real batch systems have to face the periodic arrivals of new tasks in the queue and uncertainties such as exact tasks duration. Therefore, they have to update their schedule dynamically accordingly.

The simplest batch scheduling policy is *First Come First Served* (FCFS) which is supported in many batch schedulers such as OAR [Capit et al., 2005], Condor [Tannenbaum et al., 2001, Litzkow et al., 1988], PBS/OpenPBS[1], NQS [Albing, 1993] and gLite[2]. With this naive strategy, there is a possibility that a task has to wait for a long time in the queue before being executed. A very short task located at the end of the batch queue has to wait until all tasks located before it in the queue finish. Smarter policies have been proposed in the literature to help users to access their resources as soon as possible. One of theses policies is to calculate the priority of the task based on some properties of the task and policies of the batch queue as implemented in the *Maui batch scheduler*[3] [Bode et al., 2000]. The task priority is determined by task properties such as the requested resource requirements and the time it has waited in the queue. These properties are combined in a formula with weights specified by the system administrator. After the priority of all tasks in the queue is calculated, resources are allocated to the task according to its priority order in the batch queue. Priorities have to be updated periodically to take into account new tasks and real waiting time.

1.1.1.2 Meta-scheduler

Batch systems are often used at the local level where resources are usually homogeneous and the number of resources is limited. A single resource manager can then control all available resources. Tasks are

[1] Portable Batch System (PBS), http://www.pbsworks.com/
[2] The gLite middleware, http://glite.web.cern.ch/glite/
[3] http://www.clusterresources.com/products/maui-cluster-scheduler.php

allocated directly from the batch queue to computing resources. Such scheduling frameworks do not provide load balancing across several clusters and they have limited scalability when the size of infrastructure increases. A *meta-scheduler* (figure 1.2b) provides a multi-level scheduling mechanism to adapt to the heterogeneity of resources on multi-site infrastructures (GrADS [Vadhiyar and Dongarra, 2002], GridWay [Huedo et al., 2004]). It considers more global view of the execution environment, and distributes jobs between different local schedulers (*e.g.* batch schedulers). This hierarchical scheduling mechanism is therefore more scalable and implements site-wise load balancing. By enabling the heterogeneity of low-level schedulers, meta schedulers satisfy better the user request for a large variety of resources.

Batches and meta-schedulers implement the *push* model which composes of three phases. The scheduler:

1. collects resources status for the entire infrastructure;
2. decides job allocation to resources; and
3. submits jobs to resources.

In the first phase, all information concerning the system has to be published. In a large scale infrastructure, this is often impractical, and information will often be unavailable, incorrect, or out of date. In the second phase, the decision of the best match between jobs and resources is an NP-complete problem. The size of this problem increases along with the number of jobs in the queue and available resources. The scheduler can become overloaded and cause some delays in the scheduling process [Cancio et al., 2004].

Additionally, there is a considerable overhead associated with a job submission as the process relies on using a meta-scheduler, which itself communicates with other schedulers. At least, two queueing and scheduling tasks are going to be involved. For larger experiments that involve rather long-running jobs this does not represent intractable scheduling computations. On the other hand, for some other types of work, especially the ones involving large number of short-running jobs, the overhead is large and meta-schedulers become inefficient.

1.1.1.3 Advance reservation

Batches and meta-schedulers schedule user resource requests on the fly which sometimes does not satisfy users requirements. For distributed applications which need to guarantee the Quality of Service (QoS), resources reservations are effective technologies. Resources reservation techniques include advance reservation and immediate reservation. An advance reservation is a restricted delegation of a particular resource capability over a user-defined time interval. Through a reservation interface, users specify resource characteristics, start time and duration. Immediate reservation can be considered as advance reservation which requires to start immediately. We focus on the advance reservation to analyze the features it provides to users.

Resources reservation for a specific time in the future ensures that all resources would be simultaneously available at the execution time of the application. Reserving resources in advance, users can provide an upper bound on the response time. For an application with sequential tasks, for example,

the response time of the first resource in sequence can be the start time of the reservation for the second resource and so on; thus guaranteeing the end-to-end response time.

Advance reservation has received significant attention from the research community. Especially applied to grid infrastructures, advance reservation has been considered as an important requirement for future grid resource management systems. For computing and storage resources, advance reservation was introduced as a part of the *Globus Architecture for Reservation and Allocation* (GARA) [Foster et al., 1999]. Numerous batch systems have been integrated with an advance reservation service such as Maui [Bode et al., 2000], PBS or OAR [Capit et al., 2005].

Despite their attractive features, advance reservations increase the complexity of the scheduling problem and can cause infrastructure performance degradation. They leave idle time fragments in resource schedules where no reservation can be made. As studied in [Smith et al., 2000, Sulistio and Buyya, 2004], with only 20% of users requests arriving as advance reservation, the utilization of the grid infrastructure can go down as low as 66% in the case where no task reserved in advance. To overcome this drawback, several scheduling algorithms have been proposed, such as one presented in [Farooq et al., 2006].

1.1.1.4 Pilot jobs

Using a regular batch submission interface but enabling resources reservation at the user level, the *pilot jobs* mechanism can be considered as a bridge between batch systems and systems supporting resources reservation. As described in figure 1.2c, the pilots are composed of two main components: the pilot agent, which is responsible for executing the computing task and the pilot master which provides the computing task to the pilot agents. To initiate a pilot pool, a master process is started locally and a desired number of agents is submitted to the infrastructure using a plain job submission process via meta-scheduler. Once a pilot is executed on a resource, it reports back to its master and pulls a computing task for execution on its resource. This approach is therefore implementing the *pull* model. As soon as there is no computing task to process, the worker terminates itself, freeing the resource for other users. If, for any reason the agent fails or looses the connection with the master, the master can reassign the computing task to another agent as soon as one becomes available. Compared to the push model, the scheduling decision in the pull model is more local and less computationally complex. The pull model only involves finding one task to match the pulling resource when the pilot request for a task to execute. The master only deals with the tasks of a single user and therefore it is less subject to overhead. Scalability is ensured by deploying multiple dedicated masters. Several pilot jobs frameworks have been developed. For example, systems interfaced with EGEE[4] include DIANE [Mościcki, 2003] WISDOM-II [Ahn et al., 2008, Jacq et al., 2008], ToPoS[5], BOINC tasks [Kacsuk et al., 2008], gPTM3D in radiology [Germain et al., 2008], DIRAC [Casajus et al., 2010] and CONDOR glideIns [Sfiligoi, 2008].

The pilot mechanism allows users to create a virtual private set of computing resources reserved for executing their computing tasks. By installing or uninstalling the pilot agent on computing resources, the size of the resources pool is flexibly controlled. The available underlying resources with the infrastructure also becomes more reliable for the end user when using piloting. Pilots hide broken resources

[4] http://eu-egee.org/
[5] https://wiki.nbic.nl/index.php/ToPoS

because only the successfully started pilot jobs get an opportunity to process computing tasks. They can provide accurate information about available resources which makes matchmaking much more reliable. They also allow users to perform basic sanity check of running environments before they start the real work. Any problem that occurs during the execution can be immediately reported back to the master that can react upon it.

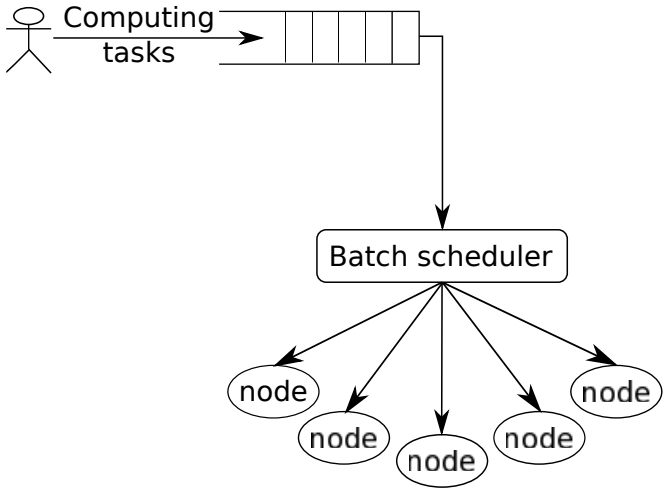
However, since pilot jobs are submitted through a scheduling agent (*e.g.* batch system), it is not possible to synchronize resources with pilot jobs mechanism. Users can submit n pilot jobs to acquire n resources but they acquire a resource only after a pilot agent successfully starts and no guaranty is assured that all n pilot agents are started at the same time. Communications between computing nodes are therefore difficult to implement. Furthermore, the pilot mechanism introduce new levels of complexity in the resources management. Resource reservation and optimization is still manually performed by users in each application.

1.1.1.5 Summary

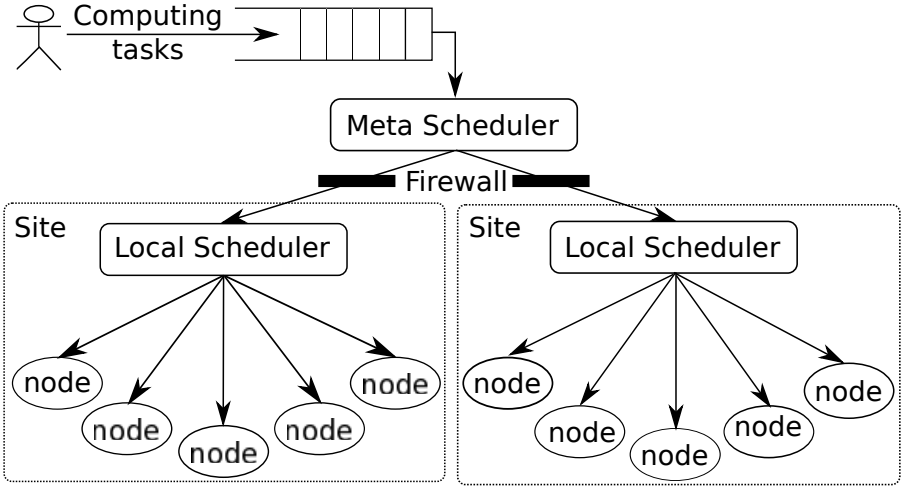
To summarize, figure 1.2 presents the shift of scheduling approaches towards the support of large-scale systems. The coarse grain implementation in traditional systems (figure 1.2a) is simple but it is not scalable and users do not have much choices in the computing resources location and start time of their tasks. Moving down, the grain level is finner, scheduling systems are more complex but they are more scalable and bring increased reliability and quality of services. Meta-schedulers (figure 1.2b) accommodate with heterogeneous local site policies to adapt to users requirements. Advance reservation allows users to reserve a set of resources for a future experiment. While this mechanism guarantees QoS constraints, it can cause infrastructure performance degradation. Furthermore, users have to estimate the amount of resources to reserve, although such an estimation is not straightforward, especially when considering distributed applications. Pilot jobs (figure 1.2c) implement the pull model which supports the decoupling of workload submission from resources assignment. This results in a flexible execution model, which in turn enables the distributed scale-out of applications on multiple and possibly heterogeneous resources. Although some limitations remain, pilot jobs is a new reservation mechanism towards the creation of a dedicated set of computing resources with increased reliability and flexibility for each VO. Shielding users from the heterogeneity of underlying resources, the pilot mechanism brings to users a unique interface to reserve computing resources and submit computing tasks. However, many problems need more investigation. Resource co-allocation is a major problem, many applications need multiple resources which could be located on different infrastructure sites. These resources have to be available simultaneously before the application execution. Schedulers therefore need to be able to co-ordinate resources scheduling.

1.1.1.6 Resources co-allocation

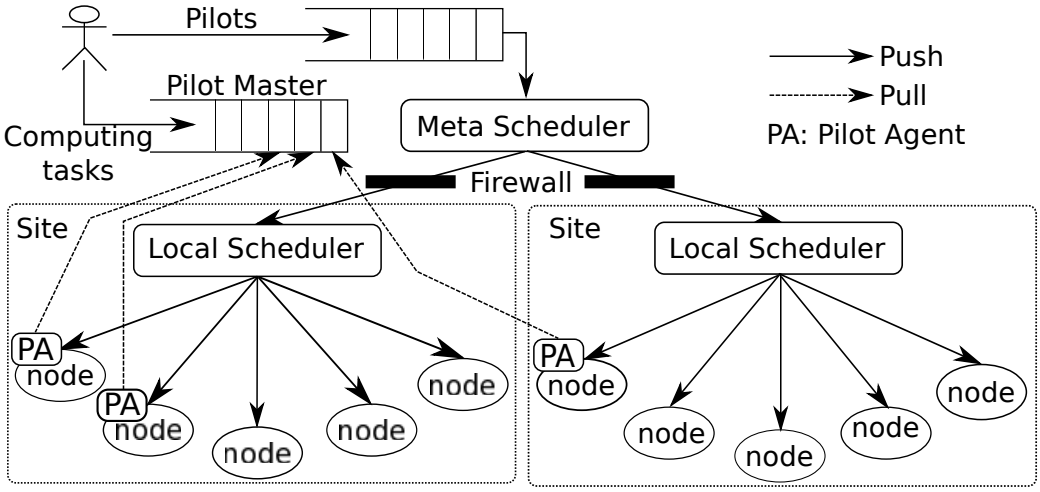
Distributed systems may be used to execute parallel applications which can efficiently use several processors, taking advantage of different resources on the infrastructure. In order to achieve this, a parallel programming model has to be proposed, and the resource management systems must be able to co-allocate resources, possibly from different administrative domains.



(a) Batch schedulers (push model)



(b) Meta schedulers (push model)



(c) Pilot jobs mechanism (pull model)

Figure 1.2: Granularity of resources allocation approaches

The *Message Passing Interface* (MPI) [MPI Forum, 1994] has been broadly used for developing parallel applications with inter-process communication running on multiple resources in a single site environment. However, executing these applications on multi-site environments imposes different challenges due to the heterogeneity of both computing and network resources. Intra-site communications have a much lower latency than inter-site communications. Complying with the MPI standard, libraries have been developed that take into account these challenges, for instance PACX-MPI [Gabriel et al., 1998], MPICH [Gropp et al., 1996] or MPICH-G [Foster and Karonis, 1998]. The main drawback of the MPI approach is that the number of resources requested by the application is "static". It cannot vary during the submission and execution of the application according to the infrastructure status. Resources on multi-site infrastructures have to be available at the same time and all processes must share the same life cycle in the execution. Consequently, an application might declare how many resources it needs, but it has no way to perform this choice taking into account the infrastructure status: the resource manager locks this application until all requested resources are available. Additionally, the fault-tolerance is not guaranteed in this kind of applications. A failure occurring on an individual resource in the set of requested resources might cause the collapse of the whole application.

Executing parallel applications on multi-site infrastructures requires a more complex and fine-grained resource allocation mechanism. This resource allocation mechanism usually refers to *resource co-allocation* [Czajkowski et al., 1999]. Resources reservation and management on a multi-site infrastructure is obviously more difficult than on a single site due to several problems such as *site autonomy*, *heterogeneous substrate*, *online control*, etc. Resulting challenges imposing on the resource co-allocator are the distributed transaction, fault tolerance, inter-site network overhead and schedule optimization. Most of existing work on resource co-allocation focus on at least two of these challenges. On the scheduling problem, in [Snell et al., 2000], authors presented three scheduling strategies for co-allocation focussing on the location of resources: (i) Specified co-allocation in which users specify resources and their location; (ii) general allocation in which users do not specify the resource location; and (iii) optimal scheduling in which the scheduler tries to determine the best location for every requested resource to optimize the cost, performance, response time or any other criteria specified by users. Other research work handling the difference between the actual and estimated computation and communication cost has been presented in [Alhusaini et al., 2000]. This approach tries to minimize the completion time (*i.e.* makespan) of a set of applications rather than a single application. It composes of two steps: The first step is an off-line planning in which the scheduler assigns tasks to resources while assuming that all applications hold all required resources for their whole execution. The second step is a run-time adaptation in which the scheduler maps tasks on resources according to the actual computation and communication costs which may differ from the estimated cost used in the first step.

All solutions dealing with challenges of resources co-allocation are developed in system middlewares. The Globus Architecture for Reservation and Allocation (GARA) [Foster et al., 1999] is considered the first project taking into account the Quality of Service for the resource co-allocation request. It enables applications to co-allocate resources including computing, network and storage resources. It uses advance reservations to support the co-allocation with QoS. OAR [Capit et al., 2005] is a batch scheduler that has been used in Grid'5000 [Cappello et al., 2005][6]. It also supports the

[6] https://www.grid5000.fr

co-allocation by using advance reservations based on *all-or-none* policy. Job Submission Service (JSS) [Elmroth and Tordsson, 2009] is a tool for resource brokering focusing on the software component interoperability. This tool has been used in NoduGrid[7] and Swegrid[8]. It relies on advance reservations for resources co-allocation. These advance reservations are flexible in the sense that users can provide a start time interval for the allocation. Other systems also support resource co-allocation such as KOALA [Mohamed and Epema, 2008], HARC [MacLaren et al., 2006], GridARS [Takefusa et al., 2007].

Each of these middlewares includes some form of resource brokers or meta-schedulers which ensure that users requirements are met across sites or clusters. Meta-schedulers services range from resource discovery, application staging, data management, and resource management. However, they are often tied to a particular unified software stack such as gLite or Globus. Due to the heterogeneity of resources distributed in multiple VOs which may have different local policies, users and application developers face many different choices of local schedulers, system protocols and resource brokers. Different approaches have been adapted to allow users to access any systems through a standardized protocol. Nimrod/G provides an economy based scheduling algorithm when choosing computing resources [Buyya et al., 2000]. The Globus Grid Resource Allocation and Management (Globus GRAM) provides a standardized interface to access different local schedulers [Czajkowski et al., 1998]. Condor-G offers users of condor pools to also take advantage of Globus-based grids [Frey et al., 2002].

1.1.1.7 Trade-off between resources and reservation cost

Resources reservation is crucial in dynamic environments such as grids. The reservation phase provides some confidence that a subsequent allocation request will succeed. Moreover, resources reservation prevents resources from becoming overloaded due to simultaneous requests from users. This could make it difficult or even impossible for an application to satisfy its temporal constraints or even carry on. Resources reservation is obviously a difficult problem which has attracted many investigations from the research community. Optimization of resources usage is another major problem when executing application on distributed infrastructures. From an infrastructure provider point of view, the major challenge is to account (financially or not) for resources usage according to specific criteria (*e.g.* fair share among users, degressive price, etc.). On a batch system, the infrastructure has no choice but to wait for the termination of the application to compute the resource usage cost. We then refer to the *post-measurement* model. On systems implementing advance reservation requests, the system provider can base the estimate of the execution on the resources specification of the applications. We refer to the *pre-measurement* model. Clouds, in particular, use a simple cost computation model that let users take care of precisely estimating the amount of resources to reserve. This practice is less suitable for dedicated infrastructures, such as scientific Grid infrastructures, for which providers are not only interested in billing but also aim at improving quality of services and optimizing resources sharing. This strong limitation has motivated us to propose a finer grain model to (i) decide on the amount of resources to allocate to each application, and (ii) compute the resources usage cost (chapter 3).

[7] http://www.nordugrid.org/
[8] http://www.snic.vr.se/projects/swegrid

From a user point of view, the problem of determining the size of the infrastructure to deploy for supporting a given application run is often a difficult one. Although a quasi-unlimited amount of computing resources may be allocated, a trade-off has to be found between (i) the allocated infrastructure cost, (ii) the performance expected and (iii) the optimal performance achievable, that depends on the level of parallelization of the application. Without assistance, users have to resort to a qualitative appreciation of the optimal infrastructure to allocate, based on their previous experience with the application and the distributed system used. Determining the amount of computational and storage resources needed for each application run is often not sufficient when considering distributed applications. Communication network bandwidth is also a critical resource, especially for data-intensive applications. Shared among the infrastructure users, network bandwidth may impact application performance significantly. It is possible to reserve several computing resources among a huge number of resources of the infrastructure but reserving the whole physical link is not possible and unrealistic.

1.1.2 The Infrastructure as a Service concept on Clouds

Cloud computing infrastructures are providing resources on demand for tackling the needs of large-scale distributed applications. They provide the computing, network and storage resources matching the user requirements to compose a dedicated execution environment. This execution environment becomes a *service* at the infrastructure level. Virtualization technologies applied to both computing and network resources allow cloud providers to better satisfy the user needs and control their resources. Thanks to virtualization technologies, users on clouds have the illusion that they are using their own, confined infrastructure while they are sharing with other users the same physical infrastructure. In this section, a review of virtualization technologies and cloud infrastructures is given.

1.1.2.1 Virtualization

Modern computers are powerful and usually underutilized; even an inexpensive computer running a typical Web client is about 5% utilized [Virtualization, 2009]. Virtualization has become mandatory to get the most from today's typically underutilized computing resources. Not only, is it a mean of presenting the illusion of many smaller *virtual machines* (VMs), each of them running a separate operating system instance and hosting multiple services, but also does it provide a confined and specific environment.

Numerous *virtual machine monitor* (VMM) systems using virtualization have been designed to subdivide these machines. A VMM is a software layer that virtualizes all resources of a physical machine, thereby defining and supporting the execution of multiple virtual machines. The interface exported by a VMM is a virtualized hardware/software interface, including a CPU, physical memory, and I/O devices. A VMM typically executes directly on physical hardware, and more specifically, below the level of operating systems. Within each VM, a "guest" operating system provides the customary set of high-level abstractions such as files or network sockets. VMware [Devine et al., 1998] and Connectix[9] both virtualize the commodity of computer hardware and implement a full virtualization of underlying hardware. IBM supports a paravirtualized version of Linux for their ZSeries mainframes allowing a large number of Linux instances to run simultaneously. The VMM has also been used by Disco to allow commod-

[9] Connectix. Product Overview: Connectix Virtual Server http://www.connectix.com/products/vs.html

ity operating systems to run efficiently on ccNUMA machines [Bugnion et al., 1997, Govil et al., 1999]. The vMatrix [Awadallah and Rosenblum, 2002] project is based on VMware and aims at building a platform for moving code between different machines. XEN [Barham et al., 2003] is an x86 virtual machine monitor which allows multiple operating systems to share the conventional hardware in a safe and resource managed fashion. It can host up to 100 virtual machines instances simultaneously on a modern computer with a low performance overhead. Denali [Whitaker et al., 2002] scarifies the strong security and performance isolation.

In summary, virtualization technologies allow cloud providers to flexibly deploy and configure their infrastructure. Users requirements on the virtual machine performance are easy to satisfy by adjusting the number of virtual instances hosted on the same physical resource. Although users share together the same set of physical resources, thanks to the virtualization technologies, they have the illusion that they are executing their applications on an isolated, secured and dedicated environment.

1.1.2.2 Cloud infrastructures

Cloud infrastructures are increasingly explored to provide transparency to users when co-allocating multiple resources hosted on single or multi-site infrastructures. By using virtualization technologies, virtual execution environments can be dynamically and easily formed to deploy and execute applications with various requirements. Existing Grid infrastructures have tried to combine the virtualization on top in order to create Cloud infrastructures and provide resources on demand. Numerous commercial Cloud infrastructures have successfully promoted their infrastructures and allowed users to reserve the resources and execute their applications in the "pay as you go" manner. Some allow users to choose the OS and customize it (*e.g.* Amazon EC2[10], Enomaly's Elastic Computing Platform (ECP)[11], GOGRID[12]), others are less configurable such as 3tera's AppLogic[13] which has no OS choice and b-hive[14] which is data-center oriented.

Both scientific and business organizations are finding Cloud infrastructures valuable as a way to improve:

- IT management since the virtualization on Cloud infrastructures allows more flexible management of resources;
- Reliability since it is easy in a virtual environment to failover to another virtual machine when necessary and quickly restart virtual machines and applications;
- Flexibility by being able to readily build up and tear down virtual data or computing centers based on needs; and
- Security by being able to deploy customized security system.

[10] http://aws.amazon.com/ec2/
[11] http://www.enomaly.com
[12] http://gogrid.com
[13] http://www.3tera.com
[14] http://www.bhive.net

From an infrastructure provider perspective, *Infrastructure as a Service* (IaaS) *cloud computing* represents a fundamental change from precedent distributed infrastructures. Pilots mechanism built on top of grids give users the illusion that they are working on an infrastructure with reservation support, although this reservation mechanism is manually managed by the application. Furthermore, on Grid infrastructures, the control over how resources are used is based on local software and policy choices. On the contrary, thanks to the virtualization that isolates the resource leased to the user, the infrastructure provider turns control of that resource over to the user in a secured manner. The control mechanism and security policies are left to the user responsibility. Moreover, by using virtualization, Cloud infrastructures can co-allocate several execution environments on the same set of physical resources. This increases the efficiency of resource utilization.

It is well known that the underlying Cloud infrastructure environment is inherently large-scale, complex, heterogeneous and dynamic. It aggregates large number of independent computing and network resources and data stores. Managing these infrastructures while guaranteeing the QoS is thus a challenging technical problem. However, most of cloud infrastructures in operation are proprietary and rely upon infrastructures that are invisible to the research community, or explicitly designed not to be instrumented and modified by systems researchers interested in cloud infrastructures. It is difficult to understand the behavior and the management mechanism of these infrastructures. Looking at the Research Cloud infrastructures, we can indicate several projects such as the ones listed below:

The Nimbus toolkit (formerly known as the "virtual workspace service") [Keahey et al., 2005] was developed with the goal of providing an open source implementation of a service that allows a client to lease remote resources by mapping environments, or "workspaces" (*e.g.* implemented by VMs), onto those resources. Its primary objectives are to provide infrastructure semantics addressing the needs of the scientific community through resource leases. A single virtual workspace called *atomic workspace* is defined through *workspace metadata* which contains all information needed for deployment such as the virtual image to deploy, CPU, memory, etc. Several atomic workspaces can be combined to form a virtual cluster whose networking configuration can be different between its atomic workspaces.

The HIPerNet software is developed in the context of the HIPCAL[15] project. HIPCAL studies a new grid substrate paradigm based on confined virtual cluster concept for resource control in grids. In addition to virtual host allocation, it implements new approaches for bandwidth sharing and end-to-end network quality of service guarantees. The resources of global infrastructure including computing resources, storage and network resources are partitioned in virtual infrastructures (aggregation of virtual machines coupled with virtual channels) and dynamically composed. These virtual clusters are multiplexed in time and space, isolated and protected. Users use the VXDL language [Koslovski et al., 2008] to describe the needed resources (*e.g.* number of computing resources, network bandwidth, network topology, etc.). This language is also extended for the infrastructure reliability.

Collaborating in the HIPCAL project, the work presented in this thesis is based on the HIPerNet framework. Our proposals presented in chapter 3 have been adopted to do experiments on the Aladdin/-Grid'5000 testbed through this framework. The HIPerNet framework has been used to deploy virtual machines, control the bandwidth and manage the users reservations. The performance assessment of this infrastructure is presented in [Vicat-Blanc Primet et al., 2009b, Koslovski et al., 2009].

[15] http://hipcal.lri.fr/

1.1.3 Summary

In this section, we addressed the resource reservation and allocation problem on distributed systems. From a seminal distributed system like local-area network, distributed systems have rapidly evolved to clouds to better satisfy the users computation needs. Along with this evolution, several parallel programming paradigms have been developed to help users harnessing the computation performance of distributed systems. With virtualization technologies, cloud infrastructures offer the flexibility to co-allocate the resources including the computing, networking resources and other non-functional requirements according to users specifications. Despite this strong utility, distributed systems still need more investigation on resource optimization, especially on cloud infrastructures where users may be accounted for their resource utilization.

Many domains require high performance computation. We selected medical image analysis applications as a case study to do the research presented in this thesis since they represent a broad class of distributed data intensive applications. In the next section, we detail this application class and the needs of cloud infrastructures for these applications.

1.2 Cloud infrastructures for medical image analysis applications

1.2.1 Medical image analysis applications

Imaging has become an essential component in many fields of biomedical research and clinical practice. Biologists study cells and generate 3D confocal microscopy data sets, virologists generate 3D reconstructions of viruses from micrographs, radiologists identify and quantify tumors from Magnetic Resonance Imaging (MRI) and Computerized Tomography (CT) scans, and neuroscientists detect regional metabolic brain activity from Positron Emission Tomography (PET) and functional MRI scans. These digital medical images represent tremendous amounts of data, in the order of tens to hundreds of MB for each MR image, and of hundreds for a CT-scan. Consequently, the annual production of a single radiology center is estimated to tens of TB per year. To face the growing requirements of image analysis, sophisticated computerized quantification and visualization tools have been developed over the three past decades.

The main commonality of these applications is their data-intensive characteristic since they have to manipulate large volumes of data and for many patients. Each image processing procedure can last up to several hours. Moreover, the management of these applications is complex due to the non-trivial semantics and the data privacy. A medical image itself is often of low interest if it is not related to a context (patient medical files, other similar cases...). A medical image is therefore often processed as a correlated data set. Distributed systems are needed to exploit potential parallelism and speed up the execution of the application.

1.2.1.1 Medical image analysis workflow

Medical image analysis applications are complex not only in terms of data semantic and privacy but also in terms of the structure of the application. Each application is usually composed of several algorithms which interact with each other to transfer intermediate data. This leads to the needs for a

high-level language which enables the description of the structure of the application with precedence constraints in the platform independent manner. We describe medical image analysis applications using a *workflow* language which proved to be a suitable abstraction for distributing such procedures. A workflow is often represented as a *Directed Acyclic Graph* (DAG) that consists of nodes which represents tasks and edges which represent data and control dependencies between them. Workflow-based applications are interpreted by workflow management systems such as Pegasus [Deelman et al., 2005], MOTEUR [Glatard et al., 2008b], P-GRADE [Kacsuk et al., 2003], DIET MA-DAG [Amar et al., 2006], etc. which automatically transform the specifications into executable workflows that can be executed on distributed resources. Using workflow description offers several advantages, such as (i) ability to build dynamic applications which orchestrate distributed resources, (ii) utilizing resources that are located in a particular domain to increase throughput or reduce data transfer costs, and (iii) execution spanning multiple administrative domains to obtain specific processing capabilities.

1.2.1.2 Resources allocation for workflow execution

Algorithms involved in a workflow are usually heterogeneous in computation (*i.e.* execution time). On current systems, each invocation of workflow algorithms processing a data item is realized as batch job that can be defined as non-interactive computational task submitted to the infrastructure independently. The application performance therefore depends on the availability of resources on the infrastructure. If any task is pending for a long time in the queue, the application performance will be impacted. Although pilots help users reserving quickly resources, current systems may not satisfy the expected performance (*e.g.* meet the deadline of a patient) and hardly optimize resources reserved for workflow-based applications. Resource co-allocation with advance reservation is a solution which allows users to achieve the desired performance and optimize resources usage. Based on the information extracted from the application logic, users can optimize and designe their confined execution environment.

1.2.1.3 Security concerns

The security requirements of medical applications cannot be tempered with, at the risk of discarding distributed computing usage in this area. All data belongs to patients whose confidentiality needs to be preserved in order to fulfill strict hospital privacy rules, in particular when data is transported from acquisition sources to processing sites and for storage of intermediate computing results. Additionally, it is not acceptable for any clinical institution that the access to its data resources be managed externally by a centralized organization. The access control policy should ensure that each health organization solely controls its own data. The access control technique should allow for reactive access control rules to be set-up in the context of medical studies, whose life time is short (typically weeks) and the group composition highly dynamic (small specialist groups are involved in each study, possibly evolving along time to embark larger consortiums as needed by the experiments).

1.2.1.4 Summary

To summarize, medical applications require:

- **High-level descriptions** to ensure that applications are described in a platform independent way. This description must allow users to exploit parallel computing on distributed systems to achieve desired performance;
- **Complex resource allocation mechanisms** to co-allocate resources to applications that guarantee the performance requirements. The allocation mechanism should support resources reservation and the optimization of resources usage;
- **High security levels** to ensure that data privacy is protected during the execution on distributed systems as well as on the communication channels.

1.2.2 Cloud infrastructure for medical image analysis applications

As mentioned in section 1.2.1, medical image analysis applications are data and computation intensive. They need a parallel implementation to get executed in a reasonable amount of time compatible with the clinical practice constraints. It is even more critical with applications that require interactivity for which users can only remain a reasonably short amount of time in front of their computer screen, waiting for the algorithm to process the data and return output. Local area parallelism is widely available today through MPI and batch-oriented applications. However, due to the limited number of local resources available in medical sites, these applications sometimes need to execute on wide area infrastructure. Additionally, it is difficult to set up multi-medical center studies if the application is executed on local area.

Grid infrastructures have been used for such applications since their foundation [Germain et al., 2005]. Grids provide an infrastructure allowing the medical community to access and manipulate medical data. Grids offer the computing power needed to validate algorithms on large datasets and to process complete databases for applications requiring statistics such as epidemiology and image registration. The transparent access to medium to high-end computing systems through grid middleware broadens the applicability of augmented reality as a medical tool. For instance on the EGEE infrastructure, many medical image analysis applications have been ported in, such as [Montagnat et al., 2004, Glatard et al., 2005, Blanquer Espert et al., 2005].

Clouds are a natural next step of Grids towards the provisioning of Infrastructure as a Service. We can see many benefits of using Clouds over Grid or Cluster resources for this kind of applications. Cloud infrastructure can at least ensure two important requirements of medical image analysis applications summarized in section 1.2.1. The elasticity of Cloud infrastructures allows users to increase or decrease the size of the execution environment to run their application, considering the number of input data to achieve the desired performance (*e.g.* meet the deadline for a patient). Resources co-allocation is better satisfied on cloud infrastructures, including network resources thanks to virtualization technology that enables the sharing of link bandwidth among users without interfering with each other. The isolation of the infrastructure help users to enhance the security to protect the privacy of patient data without disturbing other users. With the distributed storage services provided by cloud infrastructures, the large amount of data of these applications can be retrieved from or sent to closest possible location to the computing resources or client. However, cloud infrastructures do not support enough mechanisms to optimize the resources usage. In particular, for complex distributed data intensive applications, such as medical image analysis, which is described in workflow format, the amount of resources needed for each

algorithm varies during the execution. We mainly focus on this limitation in this thesis. The objective is to propose a model to help both users and infrastructure providers to optimize resources usage.

1.3 Manuscript contributions and organization

The work presented in this manuscript studies the problem of porting medical images analysis applications in cloud infrastructures to run these applications while jointly optimizing the cost and desired performance. On the one hand, we study the portability challenges to manage the transparency of the porting process or to minimize the modifications on the client and application side. We do experiments to assess the performance and evaluate the overhead when porting such applications in cloud infrastructures. On the other hand, after successfully porting applications in cloud infrastructures, we optimize their execution based on several performance criteria. The first purpose of the cloud infrastructure is to provide the resources on demand and users pay for resource reservation in the "pay as you go" manner. Taking advantage of this feature, we propose strategies to determine the amount of resources needed to run the application while minimizing the resource reservation cost. Thanks to HIPerNet middleware which is applying virtualization to both computing and network resources, these strategies can take into account the network bandwidth between the database and computing nodes or between computing clusters if needed. The cloud infrastructure provides many features that benefit the application such as high security level and infrastructure reliability.

The study made in this thesis is however limited to workflow-based applications since the proposed strategies are based on an estimation which is only produced when sufficient information about applications is available. This information includes the volume of the input data processed and the output data produced by each algorithm, as well as the average execution time of the algorithm. Thanks to workflow formalism, this information can be extracted and the application logic can be interpreted to produce such an estimation.

Chapter 2. Chapter 2 presents a taxonomy on the workflow languages. We present the features and drawback of each workflow language. We explain why we need the workflow languages for medical image analysis applications and how to use the workflow language to exploit parallelism. Two main problems when executing workflow-based applications on distributed systems are considered in this chapter. Data management is the first problem that we analyze. Existing approaches on data management are then reviewed. The second problem is workflow scheduling and resources allocation. A classification of workflow scheduling approaches is given. Based on this classification, existing algorithms are detailed and categorized. Through the analysis of the state of the art, we present our motivation for the study made in this thesis.

Chapter 3. In this chapter, we present our contribution to optimize the execution of workflow-based applications on a cloud infrastructure. We introduce the cost function model which helps users on the one hand to determine the size of infrastructure to deploy for supporting a given application run and infrastructure providers on the other hand to account for resources usage according to specific criteria [Truong Huu and Montagnat, 2010]. Four allocation strategies are described based on the estimation

made by workflow scheduler enriched in the MOTEUR workflow engine [Glatard et al., 2008b]. The estimation can differ from the real execution due to the variation of computational process execution time with input data or stochastic processes causing unforeseeable execution time. A technique is proposed in this chapter to address this problem. We also address in this chapter other concerns that impact application performance and cost on cloud infrastructures such as low reliability and data transfers overhead.

Chapter 4. In this chapter, we present all experiments to validate the proposals described in chapter 3. We first introduce the experimental testbed, the Aladdin/Grid'5000 experimental infrastructure. We then detail the HIPerNet virtualization middleware that we used to manipulate virtual machines. The workflow-based medical application use case is next described. The first experiment is conducted to evaluate the infrastructure virtualization impact. Its results assess the performance impact within 10% which is acceptable when considering other advantages of system virtualization for most non time-critical applications [Koslovski et al., 2009]. We then present experiments conducted for each proposal presented in chapter 3. These experiments assess the performance of our approaches on resource usage optimization, reliability support, and prove that cloud infrastructures are suitable for medical image analysis applications [Truong Huu et al., 2011, Koslovski et al., 2010].

Chapter 2

Workflows for medical image analysis applications

This chapter deals with the process of the description and execution of medical applications on distributed systems. We start the chapter by underlining the need for a workflow language as a high-level description abstraction for medical applications. An overview on existing workflow data-driven languages is then given. Executing workflow applications on distributed systems involves several critical problems. We analyze in this chapter three main problems: data management, workflow scheduling and resource allocation. Existing approaches to address these problems are then reviewed to motivate the forthcoming study of this thesis.

Ce chapitre traite du processus de description et d'exécution des flots applicatifs d'analyse d'image médicale sur des infrastructures à grande échelle. Nous commençons le chapitre en soulignant la nécessité d'un langage de flot applicatif comme une abstraction de description de haut niveau pour des applications d'analyse d'image médicale. Une taxonomie sur les langages existant dirigés par les données est ensuite présentée. L'exécution des flots applicatifs sur les infrastructures à grande échelle implique plusieurs problèmes critiques. Nous analysons dans ce chapitre trois problèmes principaux: l'allocation de ressources, la gestion de données et l'ordonnancement. Les approches existantes traitant ces problèmes sont ensuite étudiées pour motiver les prochaines études de cette thèse.

Both medical research and clinical practice are nowadays involving large quantities of digital data and require large-scale computation, as a result of the use of ever more digital probes in medicine. For instance, in medical images analysis, a single research study may require many complex image processing algorithms which can communicate with each other to exchange data. These algorithms are repeatedly executed on a data set of up to thousands of image files. Execution efficiency demands the use of parallel or distributed systems, but few medical researchers have time or expertise to write the necessary parallel applications.

From a computer science point of view, such applications can be described as *workflows*. A workflow is a graph whose nodes represent data analysis processes and arcs represent inter-dependencies. Inter-dependencies may be data dependencies (data exchange needed between subsequent processes) or pure control dependencies (the dependency only expresses a synchronization of processes execution in time). The representation and execution of medical applications as workflows enable a generic processing of many similar image analysis tasks. As mentioned above, medical applications require heavy computation, dominated by their data parallel nature. The workflow-based approach eases the description and deployment of such computation over distributed systems. It decouples the application from the execution infrastructure, thus releasing the application developers from the most complex computational problems, especially parallelization. It exposes to non-expert medical users a simple, accessible formalism to describe medical applications.

Executing workflow-based applications on a distributed system involves many problems such as security, infrastructure reliability, data management and workflow scheduling.

- The security problem concerns the data privacy when transferring and executing on the infrastructure;

- The infrastructure reliability ensures applications are successfully executed and return to users accurate results;

- The data management is important due to the data exchange between workflow tasks. One extreme condition, is to use a central server to store all input data and intermediate results, thus decreasing the complexity of the data management problem. Another extreme, is to transfer the data directly from a computing resource to another one which needs this data. The problem is even more complicated when executing applications on multi-user distributed systems. Network resources are shared among users, the delay of data transfer on shared link can significantly impact applications performance.

- The critical problem addressed in this thesis is workflow scheduling and resources allocation. Workflow scheduling is a process that maps the execution of workflow tasks on distributed resources. It allocates suitable resources to workflow tasks so that the execution can be completed to satisfy objective functions imposed by users. As it was proven an NP-complete problem, many heuristics which solve a particular problem and meta-heuristics which solve a class of problems have been proposed. Depending on characteristics of the distributed system, a scheduling approach could be chosen.

We begin this chapter by analyzing the needs of workflow systems for medical applications. We then make an overview on existing workflow languages which meet users requirements on the abstraction and the parallelism exploitation. We then discuss existing data management approaches before presenting the workflow scheduling and resource allocation problem.

2.1 Towards the workflow description for medical image analysis applications

Medical image analysis applications are often composed of several algorithms dealing with large data sets. Some are dependent on each other which means there are precedence constraints (control or data) between them. An algorithm cannot be executed before the termination of its precedences. Others are independent which means they can be executed in parallel on several computing resources. Moreover, each algorithm can be used in many applications, for instance, image pre-processing algorithms. These characteristics impose on both medical community and computer science two major challenges: the reusability of application code and parallelism exploitation.

2.1.1 On the reusability of application code

Code reuse has been practiced from the earliest days of programming. Programmers have always reused sections of code, templates, functions, and procedures. Code reuse is the idea that a partial or complete computer program written at one time can be, should be, or is being used in another program written at a later time. The reuse of programming code is a common technique which attempts to save time and energy by reducing redundant work.

The *software library* is a good example of code reuse. Programmers may decide to create internal abstractions so that certain parts of their program can be reused, or may create custom libraries for their own use. Some characteristics that make software more easily reusable are modularity, loose coupling, high cohesion, information hiding and separation of concerns.

Object Oriented Programming (OOP) allows users to define classes which could be reusable in a number of different applications. In OOP, code is reused in the form of objects. These objects are often contained in vast libraries of reusable code. Frameworks take the process even further, providing more robust and disciplined systems of reuse. By obtaining and reusing parts of systems which have already been "tried and tested", we can exploit the principal advantages of object-oriented programming techniques over procedural programming techniques. OOP enables programmers to create modules that do not need to be changed when a new type of object is added. A programmer can simply create a new object that inherits many of its features from existing objects. This makes object-oriented programs easier to modify. Despite efforts of many OOP languages such as C++ and Java, OOP did not fulfill the promise of complete reusability as noticed by Gannon in [Gannon, 2007].

> "Object-oriented programming was thought to be the solution to reusability but it only got us part of the way. Object-oriented concepts are powerful but they do not guarantee that a class built for one application can be easily reused in another. To build truly reusable

> software, one must design the software as part of a component architecture that defines rules and contracts for deployment and reuse."

The code reuse took a new step in 1968, when McIlroy envisioned an industry of reusable software *components* [McIlroy, 1968]. He emphasized the idea of subassemblies and interchangeable parts which could both be applied to industrial products and software. He wrote in his paper that:

> "My thesis is that the software industry is weakly founded, in part because of the absence of a software components subindustry. ... A components industry could be immensely successful."

However, at that moment, the software community did not yet agree on what a software component exactly was. Many debates have been conducted to find the answer and to determine whether objects in OOP could be considered as components or not. In 1997, Sametinger gave in [Sametinger, 1997] a precise definition:

> "Reusable software components are self-contained, clearly identifiable artefacts that describe and/or perform specific functions and have clear interfaces, appropriate documentation and a defined reuse status."

Since then, reuse of software components has become more and more important in various aspects of software engineering. Structuring a complex application into largely independent components has several advantages. It is easy to distribute components among various engineers to allow parallel development. Maintenance is easier when clean interfaces have been designed for components since changes can be made locally without having unknown effects on the whole application. And, if components interrelations are clearly documented and kept to a minimum, it becomes easier to exchange components and incorporate new ones into an application.

Web services are nowadays emerging technologies to reuse software as a service that can deliver its functions over the Internet without being installed locally. Thanks to XML and other Web standards (*e.g.* Simple Object Access Protocol (SOAP), Web-Service Description Language (WSDL), Universal Description Discovery and Integration (UDDI)), applications can run in different environments and at different locations to exchange information, interoperate, and be combined more readily than ever before. Based on HyperText Transfer Protocol (HTTP), a ubiquitous transport mechanism in today's computing environments, it is possible for application functions to interact within and across enterprises. Each application function is responsible for defining its inputs and outputs using the standards for Web services, so that other applications can interact with it.

Web services have been well exploited in the domain of medical image analysis applications. The MammoGrid project has delivered its SOA-based Grid application to enable distributed computing spanning national borders. This application harnesses the use of huge amounts of medical image data to perform epidemiological studies, advanced image processing, radiographic education and ultimately, tele-diagnosis over communities of medical virtual organisations [Amendolia et al., 2004]. The *Bronze Standard* application has also been gridified based on web services. It tackles the problem of qualifying registration algorithms accuracy [Glatard et al., 2006a]. In the MediGRID project [Kottha et al., 2007],

web services have been used to compose medical applications which are executed on grid infrastructures through the MediGRID portal.

Finally, workflows are a new paradigm which facilitates code reuse by dictating a component-based model for workflow activities. These components are executed as part of a workflow. Indeed, as more data, analysis tools and other resources are nowadays delivered as web services, users can describe and enact their applications by orchestrating distributed and local services into a workflow. This process often involves several steps: (i) choosing a set of appropriate services based on functional and non-functional properties of services, (ii) ordering them in sequence according to the application logic by solving the connectivities between services, and (iii) converting the complex process into a target workflow language which can be executed on a platform. Defining workflow-based applications in such a way allows them to be processed as a component. Workflow activities are therefore plugged in and out easily.

2.1.2 Parallelism exploitation

We especially underline in this thesis the need of parallelism for medical image analysis applications. Facing the growth of the volume of datasets, medical image analysis algorithms need a parallel implementation to get executed in a reasonable amount of time compatible with clinical practice constraints. Efficiency is even more critical for applications that require interactivity. In this case, users can only remain a reasonably short amount of time in front of their computer screen, waiting for the algorithm to process data and return an output. Support for parallel computations is therefore mandatory for these applications. Local area parallelism is widely available today through message passing interfaces but does not satisfy the needs of the medical community due to the lack of resources at local sites. Grids and Clouds satisfy better the parallelism needs. Workflow languages are emerging as the most suitable manner to describe the structure of medical image analysis applications. Interpreted by workflow managers with the support of distributed middleware, workflow-based applications can exploit following parallelism levels:

Asynchronous service calls. The first important parallelism level can be exploited is referred to asynchronous service calls. Multiple applications services can be invoked simultaneously. These invocations need to be non-blocking to exploit the parallelism. GridRPC and Web services are two standards supporting the asynchronous invocations [Nakada et al., 2005, Perera and Ranabahu, 2006]. Given that asynchronous invocations are possible, three other parallelism levels described below can be supported to harness the power of the infrastructure and satisfy the performance requirements.

Workflow parallelism. Depending on the structure of the application, several services can be executed in parallel on many data items. For instance, if we consider the simple example shown in figure 2.1, services P_2 and P_3 may be executed in parallel. This parallelism level is implemented in all existing service-oriented workflow managers.

Data parallelism. Clinical practice leads to continuous patient data acquisition daily and consequently produces many data sets. Services can be instantiated as several computing tasks running on different

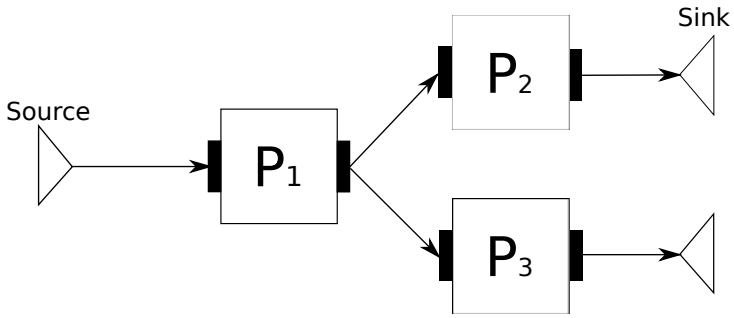


Figure 2.1: Example of a workflow exploiting data, workflow and service parallelism.

P_3	X	D_0 D_1 D_2
P_2	X	D_0 D_1 D_2
P_1	D_0 D_1 D_2	X

Table 2.1: Data parallel execution of the workflow presented in figure 2.1

computing resources and processing different input data in parallel. The major challenge of data parallelism is the ability of the service to process many parallel connections. The application should be able to create multiple threads to independently process each input data item. These threads should not conflict with each other to avoid data access conflicts. Considering the workflow given in figure 2.1 and supposing that we want to execute this workflow on 3 independent input data items D_0, D_1 and D_2, the data parallel execution of this workflow is presented in table 2.1. P_1 begins the execution by processing all input data items in parallel. P_2 and P_3 are idle (represented by a cross) since they have to wait for P_1 whose result is their input data. When P_1 finishes its execution, P_2 and P_3 will start in parallel according to workflow parallelism and they also process in parallel all input data items resulted from P_1.

Service parallelism. *Service parallelism* or pipelining represents the concurrent execution of two independent input data items by two different services which are sequentially linked. Considering again the workflow in figure 2.1 and the input data set composes of 3 independent items D_0, D_1 and D_2. Supposing that the data parallelism is eliminated, table 2.2 presents the execution of this workflow. The abscissa axis represents time. P_i executes all input data items sequentially. Service parallelism occurs when different data items appear on different cells of the same column.

Exploiting those three types of parallelism in workflow-based applications does not require specific parallel programming skills since they can be directly determined from the graph of services. Their exploitation is mandatory to speed up the application execution, obtain the desired performance and harness the computing power of distributed systems.

P_3	X	D_0	D_1	D_2
P_2	X	D_0	D_1	D_2
P_1	D_0	D_1	D_2	X

Table 2.2: Service parallel execution of the workflow presented in figure 2.1

2.1.3 Conclusions

In this section we underlined the need of workflows for medical image analysis applications. From a computer science point of view, medical applications can be described as a workflow as it requires many mostly independent services. Additionally, medical image analysis applications usually involve large data sets for different needs such as statistical studies, performance evaluation, epidemiology, etc. They require heavy computations, dominated by workflow, data and service parallelism. The workflow approach brings to users the ability to develop and distribute such applications over remote resources on distributed systems. It also decouples applications from the execution infrastructures, shielding the application developers from the complexity of computational problems, in particular parallelism.

Many workflow approaches have been developed in the literature. Each system often includes a workflow language for describing applications and a workflow manager for interpreting the workflow description and managing the application execution. However, they are often designed for a specific need. It is difficult to find an approach that covers a broad set of features and meets medical image analysis requirements. In the next section, we highlight several workflow approaches and motivate our choice.

2.2 Data-driven workflow languages

Workflow languages play an important role in the workflow design process given that they constrain the kind of computational pattern that can be represented. As a matter of fact, a large number of workflow languages exist today, exhibiting the variety of needs for workflow-based applications. From raw Directed Acyclic Graphs (DAGs) such as DIET MA-DAG [Caron and Desprez, 2006] or CONDOR DAGMan[1], workflow languages have rapidly evolved to the abstraction for parallel computations such as Petri Nets [Alt and Hoheisel, 2005], meta-model [Nemo et al., 2007], data-driven such as Scufl [Turi et al., 2007] and scripting languages such as SwiftScript [Zhao et al., 2007]. Each of these approaches can be defended through some aspects well covered in their design: DAGs are convenient for scheduling [Hall et al., 2007, Malewicz et al., 2006], Petri Nets can be used to detect properties such as potential deadlocks, data-driven languages ease the description of applications logic for non-expert users and scripting is extensively used by programmers for prototyping, etc.

In this thesis, we focus on the data-driven languages as they are particularly appealing for designing scientific data analysis applications and executing them on a distributed system. Particularly, data-driven languages meet requirements of medical image analysis applications on the coarse-grain abstraction description. Data-driven languages also satisfy medical analysis applications on the parallelism exploitation. They implicitly express the parallelism. They separate the definition of data to process from the

[1] http://www.cs.wisc.edu/condor/dagman

graph of activities to be applied to the data. This separation of the scientific data to analyze and processing logic is convenient in many experimental situations: an application is designed and implemented independently of data sets to consider, and the same workflow can be reused for analyzing different data sets without any change.

2.2.1 The Scufl language

Scufl was introduced within the myGrid project[2] to represent data flows enacted through the Taverna workflow engine [Oinn et al., 2007]. Scufl is a simple graph-oriented language which is defined through several XML tags described in the following paragraphs.

Processors. In the Scufl language, computing activities are called `processors`. For convenience there exist different processor kinds. Without loss of generality we will focus on three types of processors: (i) *java internal processors* which are executing predefined java-coded operations at the level of the workflow engine, (ii) *beanshell processors*[3] which are interpreting java user code locally and (iii) *web service processors* which are invoking standard web services. Beyond the convenience of separating data and application logic, processors may fire multiple times depending on data items which they receive to process. The processor XML tag can contain many different tags which specify the processor behavior. For example, beanshell processors have specific tags to hold the java code to be interpreted, web service processors have another tag to define the service WSDL description document endpoint, etc. A commonality to all processors is that they define named input and output *ports*. Ports are buffers which hold either data items sent to the processor for computation or data items produced by the processor.

Data links. An output port of a processor P_0 and an input port of a subsequent processor P_1 are connected through links. *Data links* are by far the most widely used in Taverna data flows: a data link expresses a data dependency between P_0 and P_1. P_1 can only be enacted once it received one data item or more into all its input ports through data links. When exactly P_1 fires is determined by the processor iteration strategy defined below.

Coordination constraints. Coordination constraints are a specific kind of processor links which do not require any data to be exchanged between connected processors. The target processor of a coordination constraint can only fire once the source processor has completely executed (*i.e.* once it has fired for all data sets to process and it is certain that no further firing will be needed in the execution of this workflow). It is to be noted that cycles of linked processors may exist in Scufl. However, the behavior of the Taverna enactor in presence of cycles is ill-defined. A clear semantic for data link cycles can be defined. Control links cannot appear within a cycle given that the completion execution of a processor within a cycle cannot be determined.

Iteration strategies. Despite its apparent simplicity, the Scufl language provides a rich data flow semantic through *iteration strategies*. They define how many times a processor fires when it receives input

[2] myGrid UK e-Science project: www.mygrid.org
[3] http://www.beanshell.org

data on two or more input ports. There are two basic iteration strategies when considering a pair of input ports p_0 and p_1 of processor P: *dot* and *cross*. In the case of a dot product, the processor will fire once for each pair of input data items received on p_0 and p_1. For example, if p_0 receives data items a and b, and p_1 receives c and d, then the processor will fire two times $P(a, c)$ and $P(b, d)$. The dot product corresponds to a traditional *one-to-one* execution semantic. In case of parallel execution, the order of data items processed may be shuffled and care has to be taken by the workflow manager to respect the user expected semantics of computations. The cross product corresponds to an *all-to-all* execution semantic. In the former example, 4 data items would be produced: $P(a, c)$, $P(a, d)$, $P(b, c)$ and $P(b, d)$. Combining dot and cross produces in an arithmetic expression, complex iteration strategies can be defined for processors with more than two input ports.

List data sequences. An important aspect of the Taverna workflow engine is to support lists of consecutive data items and lists of embedded list semantics. However, lists are not clearly part of the Scufl language but rather a consequence of the data flow strategy implemented in Taverna: only the beanshell processor type properly handles lists although there is no reason why lists should be related to a specific processor kind. It is a flaw in Taverna design which is fixed in the GWENDIA language presented in section 2.2.4. With lists, several data items can be logically considered as a single group of data. Some processors may process a complete list in a single invocation while others may process list items individually depending on the semantic of the processor. For instance, an arithmetic "square" operation may be invoked on individual integers while a statistical "mean" operation will be invoked on a list of integers. Embedded lists enable multiple level data sets management and provide support for synchronization of data items before processor invocation.

2.2.2 Abstract Grid Workflow Language (AGWL)

The Abstract Grid Workflow Language (AGWL) is the workflow language used by the ASKALON workflow manager [Fahringer et al., 2007] which offers two interfaces for generating large-scale scientific workflows in a compact and intuitive representation: graphical modeling using UML standard and a programmatic XML based language. The AGWL workflow can be either generated from a graphical UML description or directly written by the end-user. The AGWL workflow description is definitely independent from the execution infrastructure. Workflow applications are designed without dealing with either the complexity of the execution infrastructure or any specific implementation technology. A dedicated scheduler is responsible for resources allocation and a resource manager handles resources reservation. The detail specification of AGWL is given in [Fahringer et al., 2005].

Activities. An AGWL workflow consists of *activities*. An activity can be either an *atomic activity*, which refers to a single computational task, or a *compound activity*, which encloses some atomic activities that are connected by control and data flow. Control flows are defined through explicit links among activities by using different control-flow constructs (*e.g.* sequences, loops...). Users specify data flows by connecting input and output ports among activities which may not necessarily be directly connected with control flow links. A workflow is a compound activity. The compositions of an AGWL workflow

application or a compound activity is done by specifying all its enclosed activities as well as their control and data flow connections.

Control flows. AGWL supports basic control flow among activities as in conventional programming languages. Basic control flows are: the standard sequential flow (*e.g.* `sequence`), exclusive choice (*e.g.* `switch`, `if-then-else`) and sequential loops (*e.g.* `for`, `forEach`, `while` and `do-until`). AGWL can be used to specify multiple exits from loops. The `break` construct can be used to exit the execution of a loop construct. AGWL also supports an `exit` construct to terminate the execution of the entire workflow.

In addition to basic control flow constructs, AGWL supports advanced control flow elements. These constructs provide the abstraction to compose advanced and complex workflow applications and to specify the concurrency in a natural manner: `parallelFor` and `parallelForEach`.

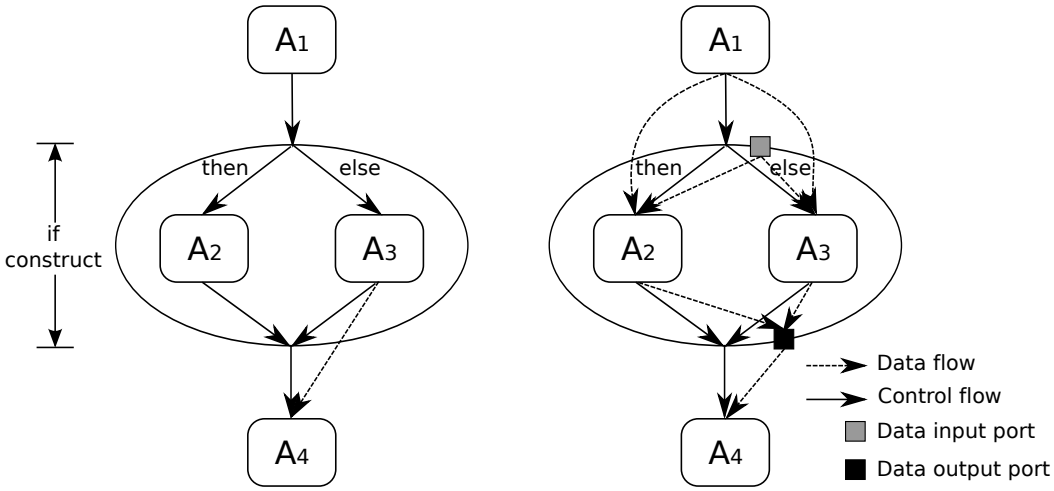
The `parallelFor/parallelForEach` activity is similar to the `for/forEach` activity with the difference that in `parallelFor/parallelForEach`, all loop iterations can be executed simultaneously. It is assumed that the input data of any iteration is independent of output data produced by other iterations of the same activity.

Data-flows. For each activity in AGWL, it must be guaranteed that whenever the control flow reaches the activity, all data input ports of the activity have been assigned to well-defined values (valid data packages). When the control flow leaves, all its data output ports must be well-defined as well. If the activity refers by name to the output port of another activity, a data flow link is established. Data flow can be expressed among arbitrary activities which are not necessarily connected by any control flow link. For example, a data flow link between activities A and B is simply established by defining the output of A as `<output name="oA">` and the input of B as `<input name="oA">`. If more than one activity declares `oA` as an input, the output will be sent to all those activities.

The data flow in AGWL is more complex when considering compound activities (*e.g.* conditional, sequential loop, parallel loop activities). AGWL defines a fixed pattern for each activity to allow the data exchange between activities. Each compound activity has a data output port which is linked to data output port of inner activities. Considering an example of the `if` activity presented in figure 2.2, activity A_4 is not allowed to be linked to a data output port of an inner activity of the `if` activity (*i.e.* A_2 and A_3).

In addition to the data flow among activities, AGWL supports the data flow between activities and special entities called *repositories*, which are abstractions for data containers. They are used to model, for instance, saving intermediate results or querying data resources without knowing any details about how repositories are implemented (*e.g.* file servers, databases, etc).

Properties and Constraints. In AGWL, properties and constraints can be defined by users to provide additional information for a workflow runtime environment to optimize and steer the execution of workflow applications. Properties provide hint about the behavior of activities (*e.g.* expected size of the input data, estimated computational complexity). Constraints should be complied by the underlying workflow runtime environment (*e.g.* to minimize the execution time, to provide as much memory as possible, to



(a) Illegal data flow (b) Legal data flow

Figure 2.2: The data flow in conditional activity [Fahringer et al., 2005]

run on the specific host architecture). User can also specify properties and constraints for activities and data flow dependencies.

To execute the AGWL workflow, ASKALON uses another language, Concrete Grid Workflow Language (CGWL) to have a task-graph representation of the workflow. Before the execution, the workflow manager performs a mapping from AGWL to CGWL.

2.2.3 SwiftScript

SwiftScript is a simple high-level scripting language with a functional, data-driven execution model. Used in Swift engine [Zhao et al., 2007], this scripting language is particularly well suited for specifying workflow applications involving loosely coupled collections of services which communicate by reading and writing files. Dependencies between services of the application are expressed through dataflow constraints: when an output of a service is the input of one more other services, the first service must complete the execution before other services can begin.

SwiftScript treats file system data as first class objects. The XML Data Type and Mapping (XDTM) [Moreau et al., 2005] is used to describe mappings between typed SwiftScript variables and file system variables. Considering the example of medical image applications, the input data of these applications are usually image files stored in a directory. SwiftScript uses an array of objects with type `Image` to map all medical images to objects. The SwiftScript `foreach` operator can then be used to specify a particular operation to be applied to each element of such arrays. The use of *futures* [Halsted, 1985] for every variable defined in the script makes SwiftScript completely data-driven. Futures are non-blocking assignment variables, using a proxy to ensure immediate execution of assignment and performing lazy blocking on variable value accesses only. The execution progresses asynchronously as long as a data access is not blocking. The availability of data enables blocked thread to restart execution. Thus, SwiftScript becomes easy to specify, compose and type check applications which execute on large volume of data maintained in complex file system structures.

Calls to services of workflow applications are specified via *atomic procedures*. An atomic procedure

encompasses an executable which has been installed on computing infrastructures, documented in a *site catalog*. The atomic procedure defines data flow of the executable, the input data to enable the execution, and the format of the call to the executable. At runtime, the Swift engine chooses from the site catalog a resource which has the specified executable installed, and executes it.

A SwiftScript *compound procedure* can be used to specify the application logic involving multiple calls to either atomic procedures or other compound procedures. For instance, a foreach operation can be used to apply a particular atomic procedure for each array element. A SwiftScript application typically composes of:

- a set of type definitions;
- one or more atomic and compound procedure definitions;
- a set of data definition statements defining the connections between SwiftScript variables and file system structures; and
- calls to atomic and compound procedures that operate on those variables. These calls can be viewed as equivalent to the `main()` procedure of a C program.

SwiftScript mapper constructs connect the logical data representation to a physical entity containing the data. This information allows the Swift engine to virtualize data resources, in the sense that both their physical location and physical format is abstracted away from SwiftScript programmer. When executing SwiftScript applications, the Swift engine identifies all *executable* tasks. An executable task is a task for which all its inputs are available. The execution of executable tasks generates outputs, other workflow tasks depending on these outputs will become executable.

2.2.4 The GWENDIA language

The GWENDIA language[4] [Montagnat et al., 2009] is a data-driven language that aims at easing the description of the complex application data flows from a user point of view while ensuring good application performances and distributed resources usage. GWENDIA is represented in XML using the tags and syntax defined below:

Types. Values flowing through the workflow are typed. Basic types are `integer`, `double`, `string` and `file`.

Processors. A `processor` is a data production unit. A regular processor invokes a service through a known interface. Special processors are workflow `source` (a processor with no inbound connectivity, delivering a list of externally defined data values), `sink` (a processor with no outbound connectivity, receiving some workflow output) and `constant` (a processor delivering a single, constant value).

[4] GWENDIA is defined in the context of the ANR-06-MDCA-009 GWENDIA project: http://gwendia.polytech.unice.fr

Processor ports. Processor input and output ports are named and declared. A port may be an input (<in> tag) or an output (<out> tag). The input ports also define iteration strategies that control the number of invocation of the processor as a function of its inputs.

A simple example is given below:

```
<?xml version="1.0" encoding="UTF-8"?>
<workflow name="Workflow Example">
   <interface>
     <constant name="size" type="string" value="1.5" cardinality="scalar"/>
     <source name="floating" type="string" />
     <source name="reference" type="string" />
     <sink name="Results" type="string" />
 </interface>
 <processors>
      <processor name="CrestLines" >
         <in name="input0" type="string" depth="0" />
         <in name="input1" type="string" depth="0" />
         <in name="input2" type="string" depth="0" />
         <out name="result0" type="string" depth="0" />
         <iterationstrategy>
             <cross>
                <dot>
                    <port name="input0" />
                    <port name="input1" />
                </dot>
                <port name="input2" />
             </cross>
        </iterationstrategy>
        <gasw descriptor="/home/truonghuu/bronze-standard/crestLines.xml"/>
    </processor>
      <processor name="CrestMatch" >
         <in name="input0" type="string" depth="0" />
         <in name="input1" type="string" depth="0" />
         <in name="input2" type="string" depth="0" />
         <out name="result0" type="string" depth="0" />
      <iterationstrategy>
        <dot>
          <port name="input0" />
          <port name="input1" />
          <port name="input2" />
        </dot>
       </iterationstrategy>
      <gasw descriptor="/home/truonghuu/bronze-standard/cmatch.xml" />
    </processor>
  </processors>
</workflow>
```

Data link. A data link is a simple connection between a processor output port and a processor input port as exemplified below:

```
<links>
   <link from="floating" to="CrestLines:input0" />
   <link from="reference" to="CrestLines:input1" />
   <link from="CrestLines:result0" to="CrestMatch:input0" />
</links>
```

Data structures. The data manipulated in the GWENDIA language is composed from scalar typed data items. The basic types are mentioned above (`integer`, `double`, `string` and `file`). Fixing the drawback of the Scufl language, the GWENDIA language allows the data to be grouped into different depth arrays. An array is an ordered collection of data items with the same type. A simple array is a collection of scalars (*e.g.* $a = \{-2, 3, 1\}$ is an array of integers). A one-dimension index designates each of its data item (a_0 designates the integer -2). An array may be empty or may contain other arrays at any nesting level. An array of array is further referred to a *2-nesting levels* array, etc. A scalar data item corresponds to a *0-nesting level* array while a singleton $\{s\}$ corresponds to a *1-nesting level* array. The special value $\emptyset$ represents the absence of data.

Iteration strategies. While extending the dot and the cross iteration strategies implemented in the Scufl language to handle the $\emptyset$ value, the GWENDIA language introduces two other iteration strategies:

The *flat cross product* matches inputs identically to a regular cross product. The difference is in the indexing scheme of the data items produced: it is computed as a unique index value by flattening the nested-array structure of regular cross produces (a_i and b_j received on two input ports produces a data item c_k with index $k = i \times m + j$ where m is the size of array b), thus preserving the input data nesting depths. As consequence, the flat cross product may be partially synchronous. As long as the input array dimension are not known, some indices cannot be computed. Similarly as the cross product, ports of a flat cross product are associative but not commutative. A $\emptyset$ value received on a flat cross product port behaves as in the case of a regular cross product. It matches with all possible combinations of data items received in other ports and produces a $\emptyset$ output without firing the activity.

The *match product* matches data items carrying one or more identical user-defined tag, independently of their index scheme [Montagnat et al., 2006]. Similarly to a cross product, the output of a match is indexed in a multiple nesting levels array item whose index is the concatenation of the input indices. A match product implicitly defines a boolean valued function match(u_i, v_j) which evaluates to true when tags assigned to u_i and v_j match. The output array has a value at index i, j if match(u_i, v_j) is true. It is completed with $\emptyset$ values if match(u_i, v_j) is false and then $w_{i,j} = \emptyset$. The port of a match is therefore associative but not commutative. A $\emptyset$ value received on a match product input does not match any other data item and does not cause activity firing.

Control structures. The data-driven and graph-based approach adopted in the GWENDIA language makes the parallelism expression straightforward for end users. The data parallelism is completely hidden through the use of arrays. Advanced data composition operators are available through activities

port depth definitions and iteration strategies. Consequently, complex data parallelism patterns and data synchronization can be expressed without any additional control structures.

The only control structures considered in the GWENDIA language are therefore conditionals and loops. Conditionals and loops are expressed using the java language and interpreted each time the activity is fired. The data received on the input ports of a control structure is mapped to java variables (basic types or java ArrayLists depending on the input port depths). The detail of conditional and loops specification can be found in [Montagnat et al., 2009].

2.2.5 Application use case

Most experiments reported in this thesis are using the Bronze Standard application [Glatard et al., 2006b] as a representative use case of workflow-based medical applications.

Medical image *registration* algorithms are playing a key role in a large number of medical image analysis procedures and therefore their accuracy is critical. Image registration consists in estimating the 3D transformation between two images, so that the first one can superimpose on the second one in a common 3D frame. A difficult problem, as for many other medical image analysis procedures, is the assessment of these algorithms robustness, accuracy and precision [Jannin et al., 2002]. Indeed, there is no well established *gold standard* to compare to the algorithm results. The Bronze Standard algorithm is a statistical procedure that aims at estimating the accuracy of a given number of algorithms.

The idea is to compute the registration of a maximum number of image pairs with a maximum number of registration algorithms in order to obtain a largely overestimated system of transformation estimates (observations). From this redundant system, the Bronze Standard can be estimated by minimizing a specific criterion in the space of transformations to determine the transformations that better explain the observations. The accuracy of a given algorithm is then computed as the distance between its results and the Bronze Standard. The higher the number of independent registration algorithms considered and the number of images processed, the more accurate the procedure. It makes this application very data-intensive.

Figure 2.3 shows a representative workflow application, described in the GWENDIA language, named *Bronze Standard* [Glatard et al., 2006b]. The input data set of this application composes of two list of images named `Floating` and `Reference`, respectively. They are used in four image registration services `CrestLines`, `CrestMatch`, `Yasmina` and `Baladin`. Let n be the number of images in the list of `Floating` and `Reference`. The *dot* product of the iteration strategy between these inputs for each registration service results in n pairs of images to be processed. Many workflow services also have some constant parameters. These parameters are used for all input image pairs, as expressed by the *cross* product. The number of invocations of `CrestLines` is thus n. Similarly, for other services, the number of invocations of each service can be determined depending all the number of input and the iteration strategy (n invocations).

Combining the input data set and the application description, a DAG will be generated at runtime and submitted to the infrastructure. The generated DAG of the Bronze Standard workflow shown in figure 2.3 is presented in figure 2.4. $\{I_1, I_2, ..., I_n\}$ represent n input image pairs to be processed, and $\{R_1, R_2, ..., R_n\}$ stands for the output results. This DAG, typically composed of hundreds of tasks, is represented in a compact and user readable framework through a workflow language abstraction.

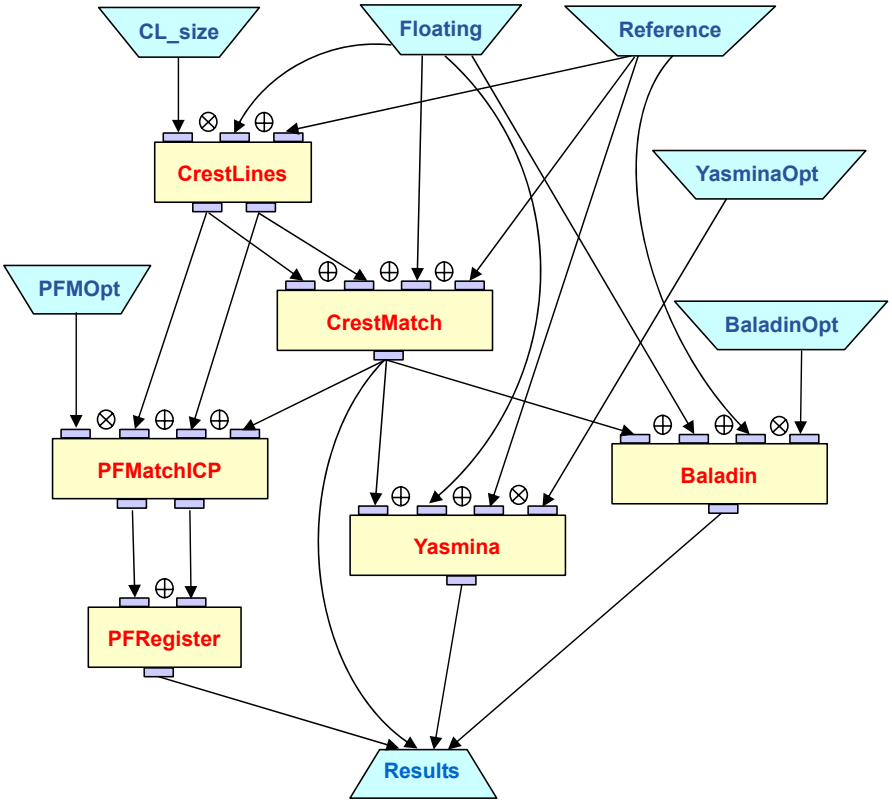


Figure 2.3: Bronze Standard workflow. Arrows denote precedence constraints. $\oplus$ and $\otimes$ represent the dot and cross product iteration strategies, respectively. Six workflow services named `CrestLines`, `CrestMatch`, `PFMatchICP`, `Yasmina`, `Baladin`, `PFRegister`, are iterated according to the their iteration strategy on the input data sets. `Floating` and `Reference` represent a pair of input images of the workflow. `CL_size`, `PFMOpt`, `YasminaOpt` and `BaladinOpt` are constant parameter of workflow services.

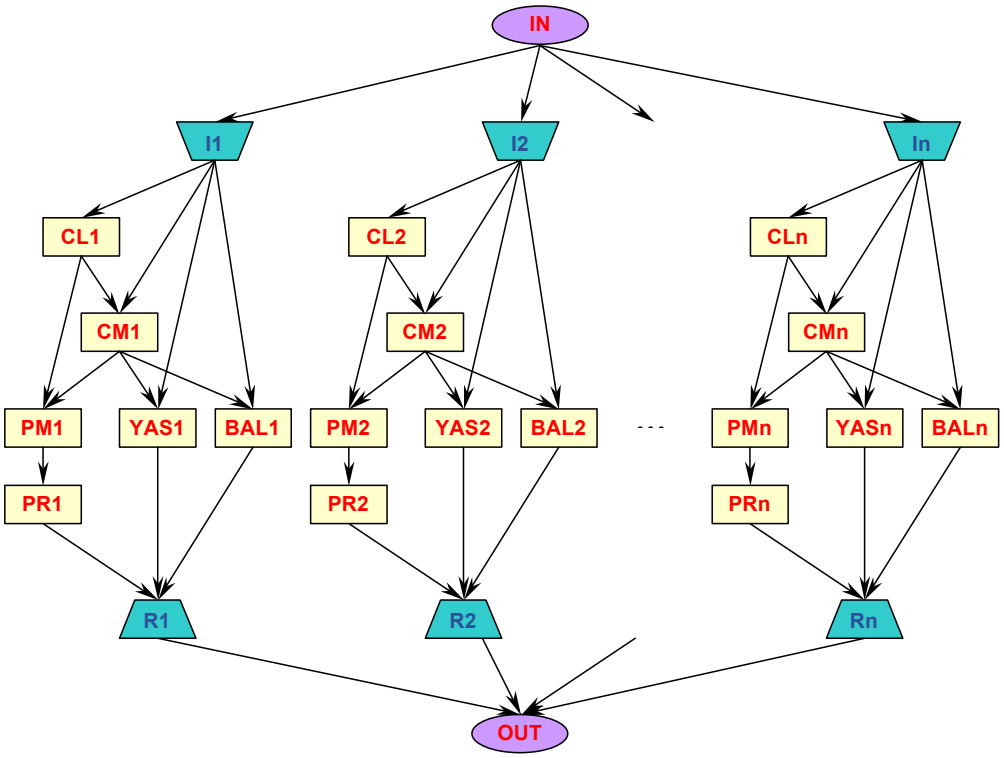


Figure 2.4: DAG jobs of the Bronze Standard application for n inputs

2.2.6 Conclusions

In this section, a review on workflow languages has been provided. We focussed on the data-driven languages to highlight their features for data intensive applications such as medical image analysis applications. Existing languages and their corresponding implementations offer a very diverse and complete set of tools providing to users the required facilities to build their applications. They provide a high-level abstraction to users to describe the application logic. With the data-driven approach, these languages make the parallelism expression straightforward to end users. They hide the complexity of underlying execution environment from users.

Depending on users profile and application characteristics, different language could be chosen. For instance, users who do not have programming background can use languages with graphical interface support such as Scufl or AGWL. Conversely, users who are familiar with scripting languages can use SwiftScript which offers all constructs and data types facilities of a traditional scripting language. Combining the idea of the Scufl language on iteration strategies and SwiftScript language on the array management, the GWENDIA language brings to users a complete and simple tool for describing data-intensive applications. A workflow description in the GWENDIA language can be executed on different data sets without any modification. Being an XML-based language, GWENDIA implementation also provides a graphical interface via MOTEUR workflow engine for users who want to use graphical tool to design their applications. Based on requirements and characteristics of medical images analysis applications, the GWENDIA language has been chosen for all experiments presented in this thesis.

After having a description of the application logic, the next step is to interpret this description combining with the input data set and execute it on a distributed system. Managing the data transfer between application services is a difficult task. It is even more complicated when executing the application on a shared infrastructure. Network resources are shared by multiple users and can significantly impact applications performance. In the next section, we discuss existing approaches addressing workflow data management.

2.3 Workflow data management on distributed systems

During the execution of workflow-based applications, the input data needs to be staged to computing resources before executing tasks. Similarly, the output data may be required by subsequent tasks which are processed on different resources. Intermediate data has to be staged out to corresponding computing resources. While some workflow systems provide *automatic* mechanisms, others require users to manage intermediate data transfer in the workflow description. We refer to the *user-directed* approach. We focus on the automatic approach and we present in this section several systems supporting the data management for workflow-based applications, especially for data-intensive applications whose intermediate data transfer mechanism significantly impact the overall performance. These approaches can be divided into two categories:

- use smart strategies to schedule the workflow tasks on the same computing resource (*e.g.* [Arnold et al., 2000]);
- optimize the data transfers on disk by using a central storage server (*e.g.* GASS [Bester et al., 1999], OmniStorage [Aida et al., 2006], Kangaroo [Thain et al., 2001]).

2.3.1 Request Sequencing in NetSolve

NetSolve [Casanova and Dongarra, 1997] is a Grid middleware built at the University of Tennessee. An instance of NetSolve Grid is a set of possibly heterogeneous computer hosts accessible over the Internet via some networking infrastructures. The system uses a client/agent/server model and is available for all popular operating systems (variants of Unix and Windows). The major components of the NetSolve system are the *NetSolve agent*, an information service and resources scheduler, the *NetSolve server*, a network resource that serves up computational hardware and software resources, and the *NetSolve client* libraries which allow users to instrument their application code with calls for remote computational services. A NetSolve client sends a request to the NetSolve agent. The NetSolve agent chooses the best NetSolve server according to the size and nature of the problem to be processed.

The first approach used in NetSolve to tackle the problem of sending large data sets on the infrastructure is *Request Sequencing* [Arnold et al., 2000]. To allow users to group two or more regular NetSolve requests into a sequence, two functions were implemented in the interface of this approach. Their purpose is to mark the beginning and end of a sequence of requests. `begin_sequence()` takes no arguments and returns nothing, it notifies the system to begin the data analysis. `end_sequence()` marks the end of the sequence and takes a variable number of arguments describing which output pa-

rameters not to return. This is a part of the API as it is users who determine which results are mandatory and which are not necessary.

When users submit a sequence of requests to the platform, a DAG is produced. Then, the entire sequence is sent to a computing resource which first runs tasks with satisfied dependencies. When one task (at least) ends, the computing resource updates its dependencies and repeats the same process while there are tasks left to execute. By using this approach, dependent tasks in an application can be grouped into a sequence. Consequently, the data transfer between these tasks will be eliminated.

However, this approach has some drawbacks. It does not handle multiple computing resources because no data redistribution is possible between computing resources. The whole DAG of all NetSolve requests within the sequence is built before being sent to the chosen computing resource. This can lead to sub-optimal utilization of computing resources when, within a sequence, two or more requests can be solved in parallel on two different resources. Consequently we cannot use this approach to workflow-based applications since we aim at dividing applications to exploit the parallelism on distributed systems. Second, the control structure is forbidden within a sequence since the condition clause of this control structure may depend on the result of a prior request in the sequence which is not yet scheduled for execution until the end of the sequence. Finally, statements that would change the value of any input parameter of any component of the sequence are also forbidden within the sequence since only references to the data are stored during the data analysis. So if changed, the data transferred at the end of the sequence will not be the same as the data that was present when the request was originally made.

In conclusion, this approach can be considered as a smart strategy but it has many drawbacks and it is not convenient to our objectives.

2.3.2 Distributed Storage Infrastructure

To make a data persistent and to take advantage of its placement in the infrastructure, NetSolve has proposed the second data management approach called *Distributed Storage Infrastructure* (DSI) [Beck and Moore, 1998]. DSI allows a program to manage data stored remotely. It means that users can control the placement of data that will be requested by computing resources. Instead of multiple transfers of the same data, DIS enables the transfer of the data once from the client database to a storage server. Considering these storage servers as being closer to computing resources than to the client database, the cost of data transfer will be lower when data is reused.

The *Internet Backplane Protocol* (IBP) [Plank et al., 1999] is an example DSI that has been incorporated into NetSolve. This integration allows users to allocate, destroy, read and write data objects to remote storage servers via IBP client API such as `IBP_allocate`, `IBP_store`, `IBP_load`, `IBP_copy`... To run client computational requests, computing resources connect to these storage servers to find the data that they need. Users can thus run their applications on the remote computing resources in using remote data and retrieve only relevant results.

The data items managed by a DSI system are called DSI objects. To generate a DSI object, users have to know the storage server in which they want to store their data. It is noted that the data location is not a criteria for the choice of a computing resource. NetSolve maintains its own *File Allocation Table* (FAT) to manage DSI objects. Each DSI object has a key by which it is cataloged in the FAT. When users send a request to the NetSolve infrastructure, the input and output references in the request are

checked against keys of the FAT to see if they represent a remote object. If not found, they are assumed to be referring to local data, in-core or on disk.

In conclusion, although DSI improves the data transfer that is explicit at users level, useless transfers between DSI storage servers and computing resources remain. It means that intermediate results need to be returned to storage servers before being sent to other computing resources for other request. This may lead to the over-utilization of network resources. Indeed, we have used this approach for many experiments to show that it may lead to the overloading of the storage server and network since the volume of the input and intermediate data in medical applications is usually very large.

2.3.3 GASS in Globus Toolkit

The Globus Toolkit [Foster and Kesselman, 1997] provides middleware services for grid infrastructures. It comprises a set of modules such as *resource location and allocation*, *communication*, *unified resource information service*, *authentication interface*, *process creation* and *data access*. Each module defines an *interface* which high-level services use to invoke that module mechanisms, and provides an *implementation* which uses appropriate low-level operations to implement these mechanisms in different environments. The *Authentication interface* includes the Grid Security Infrastructure (GSI) which provides public-key-based authentication and authorization services. The *Resource location and allocation* provides a language for specifying application requirements, mechanisms for immediate and advance reservations of grid resources and for remote job management. And, the *unified resource information service* is used for the distributed publication and retrieval of information about grid resources.

The default data management service in Globus Toolkit is the RIO (remote I/O) system [Foster et al., 1997] which uses striping to support high-performance remote access, but requires that applications to adopt the MPI-IO parallel I/O library. To overcome the limitation of the remote I/O system, *Global Access to Secondary Storage* (GASS) [Bester et al., 1999], a data movement and access service is designed, implemented and integrated in the Globus Toolkit. The goal of the GASS service is to support for the file access patterns (I/O patterns) common in distributed applications and the ability for users controlled management of network bandwidth.

There are four I/O patterns considered in the GASS service: read-only access to entire file assumed to contain constant data, shared write access to an individual file whose value is produced by the last writer, append-only access, and unrestricted read/write access to an entire file with no other concurrent accesses. These patterns are distinguished by particularly simple access patterns and coherency requirements.

GASS addresses bandwidth management issues by providing a file *cache*, a local secondary storage area, in which copies of remote files can be stored. By default, data is moved into and out of this cache when files are opened and closed according to two standard strategies: *fetch and cache on first read open* and *flush cache and transfer on last write close*. GASS also provides mechanisms that allow users to refine default data movement strategies and to manage how they are applied in particular cases. These mechanisms fall into two general classes: relatively high-level mechanisms concerned with pre-staging data into the cache prior to program access and with post-staging of data subsequent to program access; and low-level mechanisms that can be used to implement alternative data movement strategies.

Distributed applications access remote files using GASS by opening and closing files with specialized open and close calls: `globus_gass_open`, `globus_gass_fopen`,

`globus_gass_close`, `globus_gass_fclose`. These calls trigger the GASS cache management operations described above to optimize performance based on the default data movement strategies. From an application point of view, the GASS open and close calls act like their standard Unix I/O counterparts, except that a URL rather than a file name is used as an argument of these calls. A URL used in a GASS open call specifies a remote file name, the physical location of the data source on which the file is located, and the protocol required to access the resource (HTTP, HPSS, DPSS, FTP). GASS also exposes additional low-level APIs which can be used to implement specialized data movement and access strategies.

The main drawback of this approach is that applications must be specifically re-written to use GASS. One must find all of instances of `open` in a program and replace them with `globus_gass_open`. Although this is not a difficult problem for the technically minded, it would be laborious to re-build each application that wishes to use GASS. And, it is only possible when source code is available and recompilation feasible.

In conclusion, GASS is just a data access API which aims at reducing the latency of data transfer and optimizing the bandwidth consumption by using a local secondary storage server located in grid infrastructures. Apply to medical applications with the assumption that the secondary server is closer to computing resources, the input data stored in the remote server can be cached after the first read into secondary storage server. The same mechanism can be used for intermediate results of the workflow. When intermediate results have just been created, they will be cached into the secondary storage server. The subsequent tasks of the workflow will find the data already present in the cache (secondary storage server). Because of these mechanisms, GASS does not implement the direct data transfer between computing resources.

2.3.4 OmniStorage in OmniRPC

OmniRPC [Sato et al., 2003] is a GridRPC system which allows seamless parallel programming from a cluster to a grid environment. OmniRPC inherits its API and basic architecture from Ninf [Sato et al., 1997]. A client and remote computing resources which execute remote procedures are connected via a network. Remote libraries are implemented as an executable program which contains a network stub routine as its main routine.

OmniRPC provides a partial data persistence facility called automatic-initializable remote module to hold only data given by an initialize function of the remote executable module. This allows multiple transmission of the same initial data to be avoided. However, the data must be sent directly from client to each computing resource when remote module is invoked. If the initial data is large, the client may be a bottleneck. Furthermore, since remote modules are invoked on demand, the invocation of computing resources is sometimes delayed, resulting in poor scalability. To solve this problem, a data transfer layer for OmniRPC called OmniStorage [Aida et al., 2006] is designed and implemented.

The main component of OmniStorage is a relay host that relays data transfer between client host and computing resources. When a cluster is used for a pool of computing resources, it is useful for relay host to be set up at the master node of the cluster. In OmniStorage, the connection to each computing resource forms a tree topology without any cycle. Data is transferred from the root, and all computing resources can cache the received data. By using the OmniStorage's API, the client can register data to

the cache in the relay host. When the remote program is invoked by OmniRPC, the invoked remote program checks first whether the requested data is in local cache. If the data is not found, a process for requesting the data from the relay host is then performed. The relay host checks if the requested data is in the local cache and sends to computing resource. If the data is not found, the relay host sends a request to client host. The client host retrieves the data from the local cache and sends to relay host. The data is stored in the cache of relay host.

In conclusion, OmniStorage achieves efficient initial data transfer to computing resources. Using the OmniStorage together OmniRPC can improve the performance of applications with large-scale initial data transfer. However, OmniStorage does not include a function to collect the data of computation results.

2.3.5 Kangaroo in Condor

Condor [Tannenbaum et al., 2001] is a specialized workload management system for compute-intensive jobs. Similarly to other full-featured batch systems, Condor provides a job queueing mechanism, scheduling policy, priority scheme, resources monitoring, and resource management. Users submit their jobs to Condor. Condor places them into a queue, chooses when and where to run them based on policy, monitors their progress, and ultimately informs users upon completion.

While providing functionality similar to that of a more traditional batch systems, Condor's novel architecture allows it to succeed in areas where traditional scheduling fail. Condor can be used to manage a cluster of dedicated Beowulf nodes. In addition, several unique mechanisms enable Condor to effectively harness wasted resources power from otherwise idle desktop workstations. Condor can be used to seamlessly combine all of a organization's computational power into one resource.

Kangaroo [Thain et al., 2001] is a wide-area data movement system developed at University of Wisconsin-Madison. Kangaroo improves the throughput and reliability of distributed applications by hiding network storage devices behind memory and disk buffers. Kangaroo allows unmodified applications to overlap computation with I/O. By removing the burden of data movement from the application, Kangaroo helps reduce the application makespan.

Kangaroo uses a TCP-based message-oriented protocol. Servers exchange the information by passing well-defined message to each other. Different file operations are encoded as Kangaroo messages and may contain control and data information. Kangaroo also offers a highly reliable data movement mechanism by using a write-ahead log and retransmitting messages in case of network failures or when a server downstream runs out of spool space.

The Kangaroo architecture is centered around a chainable series of servers that implement a simple interface:

- `void kangaroo_put(host, path, offset, length, data);`
- `int kangaroo_get(host, path, offset, length, data);`
- `int kangaroo_commit();`
- `int kangaroo_push(host, path).`

All above functions except `kangaroo_put` are Remote Procedure Calls (RPCs). `kangaroo_put` and `kangaroo_get` allow servers to fetch data from any reachable host/-filesystem. A host is reachable if it is running the Kangaroo server to which the machine that requests the data can authenticate. `kangaroo_commit` ensures that all outstanding puts have been accepted for delivery. In practice, this is achieved by returning to the caller only after ensuring that all messages are sent prior to a commit have been logged on persistent storage at the next hop. `kangaroo_push` blocks until all outstanding puts have been transferred to their ultimate destination. We can think of this as a recursive RPC, in which, each caller invokes push and returns when the server downstream returns.

A `kangaroo_push` call on the destination returns when all data has made it to the proper file. In other words, a `kangaroo_commit` guarantees that all previously sent messages have been successfully spooled at the next hop Kangaroo server, whereas `kangaroo_push` returns only after a successful commit/push.

However, the Kangaroo prototype uses a static single route. This route is the first match that it finds in the Kangaroo routing table. Since it uses a single route, data cannot be routed around failures, even if alternate routes exist. This can affect the availability of data at the destination. The Kangaroo implementation is also not able to identify operations that can be performed in parallel, which results in wasted bandwidth.

Again, Kangaroo uses the same approach as GASS and OmniStorage. It means that Kangaroo uses several storage servers located in the execution environment to aim at hiding errors and latency of the data transfer. Workflow-based applications can be executed on this infrastructure since the input data and intermediate results can be replicated on storage servers. Subsequent tasks of the workflow can communicate with any storage server, preferably the closest, to accomplish the data request. However, applications must know one of these storage servers to send and request the data. In addition, the time between two dependent tasks of the workflow is often small while the application has to wait for the accomplishment of data replications. This can lead to an increase in the application makespan.

2.3.6 Conclusions

Many data management approaches have been proposed in the literature to satisfy users needs. Some of them use a local centralized storage server located in the execution environment. This approach reduces the data transfer time from client to computing resources in assuming that computing resources are closer to the storage server than the client. Other approaches use the smart scheduling mechanism to avoid useless data transfers between two dependent tasks of the application. However, each of them has its drawbacks. Most systems do not consider bandwidth optimization. Moving to cloud infrastructures which enable the network virtualization, new data management systems need to be studied to optimize the bandwidth reservation for data transfer.

The overall performance of medical applications is significantly impacted by data transfer mechanisms due to their data-intensive characteristic. With the workflow description, the overall performance is also impacted by workflow scheduling mechanisms. The dependency between workflow tasks makes the scheduling process more complex. Since this is an NP-complete problem, finding a “near-optimal” solution is already a difficult process. In the next section, we present existing approaches of workflow scheduling to motivate for a new workflow scheduling which can adapt to the diversity of the new

generation of distributed systems.

2.4 Scheduling and resource allocation for workflow-based applications

Workflow scheduling is a process that maps and manages the execution of workflow tasks on distributed resources [Yu and Buyya, 2005]. It allocates suitable resources to workflow tasks so that the execution can be completed to satisfy objective functions imposed by users. Proper scheduling can have significant impact on the performance of the system. In general, the problem of mapping tasks on distributed services belongs to a class of problems known as NP-complete problems [Ullman, 1975]. For such problems, no known algorithms are able to generate the optimal solution within polynomial time. Solutions based on exhaustive search are impractical as the overhead of generating schedules is very high. In distributed multi-users environments, scheduling decisions must be made in the shortest possible time since there are many users competing for resources, and time slots desired by one user could be taken up by another user at any moment. Many heuristics and meta-heuristics based algorithms have been proposed to schedule workflow applications in heterogeneous distributed environments.

Based on the taxonomy presented in [Yu et al., 2008], this section provides a summary of workflow scheduling algorithms which can be grouped into two categories: *best-effort based* and *QoS-constraint based* scheduling (see sections 2.4.1 and 2.4.2, respectively). Best-effort based algorithms focus on minimizing the execution time of the application (makespan) and ignore other criteria (*e.g.* execution costs). QoS constraints based algorithms take into account not only the application makespan but also other users requirements such as the execution cost, reliability, security or robustness of scheduling algorithms [Yu et al., 2008, Jeannot et al., 2008, Shi et al., 2006].

2.4.1 Best-effort based workflow scheduling

In distributed multi-user systems, resources are shared by different VOs. In this model, monetary cost is not considered during the application execution. Best-effort based scheduling algorithms aim at minimizing the application makespan. Many heuristic-based approaches have been proposed to solve a particular problem. Research effort has also been invested in meta-heuristic scheduling algorithms which are a general solution for developing a specific algorithm to solve a particular problem. In this section, several best-effort algorithms are given.

2.4.1.1 Individual task scheduling with Myopic

Individual task scheduling is the simplest scheduling algorithm for scheduling workflow applications. It makes the schedule decision based on only one individual task. The workflow engine is responsible for managing the dependency between workflow tasks and submitting tasks to the scheduler. The *Myopic* algorithm is an example of individual task scheduling presented in [Wieczorek et al., 2005]. In Algorithm 1, we show the detail of this approach. Considering Γ the set of unmapped ready tasks to be scheduled, this algorithm aims at mapping a task t belonging to Γ on the resource which is expected to finish t earliest. A task is ready when its precedent tasks have been finished. The algorithm repeats until all tasks in Γ have been scheduled.

Algorithm 1 Myopic scheduling algorithm

while $\exists t \in \Gamma$ **do**
  get t from Γ
  get resource r which can complete t at the earliest time
  schedule t on r
  remove t from Γ
end while

In conclusion, this algorithm is simple but it does not consider the overall performance of the application. It does not provide any full-graph analysis and does not consider the order of task execution. Additionally, it does not consider the optimization of resources usage.

2.4.1.2 List scheduling

A list scheduling algorithm prioritizes workflow tasks and schedules them based on their priorities. Consequently, there are two phases in a list scheduling, the *task prioritizing* and the *resource selection* phase. The task prioritizing phase sets the priority of each task based on task characteristics. This phase results in a scheduling list by sorting tasks according their priority value. The resource selection phase maps tasks in the order of their priority on the optimal resource.

Min-Min, Max-Min and Sufferage. The first instance of list scheduling algorithms is *Min-Min*, *Max-Min* and *Sufferage* proposed by Maheswaran *et al* in [Maheswaran et al., 1999]. These three heuristics are based on the performance estimation for task execution and data transfer. It has been employed in several workflow systems such as vGrADS [Berman et al., 2005] and Pegasus [Blythe et al., 2005].

In Algorithm 2, the detail of the Min-Min heuristic is given. Let Γ be the set of independent tasks needed to be scheduled. For each iterative step, the algorithm computes the *Estimated Completion Time* (ECT) of each task on every available resource r in the list of available resources ($availResources$) of this task. From all ECT values, the algorithm gets the *Minimum Estimated Completion Time* (MCT) for each task which is estimated to be executed on the resource R_t. The task τ with the minimum MCT over all tasks will be chosen to be scheduled first at this iteration. τ will be assigned to the resource R_τ which is expected to complete the task at the earliest time.

The Max-Min operates similarly to the Min-Min heuristic. The only difference is that the Max-Min heuristic schedules the task with the maximum ECT rather than minimum ECT in the Min-Min heuristic. After obtaining the MCT value for each task, the task having maximum MCT is chosen to be scheduled on the resource which is expected to complete the task at the earliest time.

Instead of using minimum MCT or maximum MCT, the Sufferage heuristic compute the task priority based on its *sufferage* value. The sufferage value of a task is the difference between its earliest completion time and its second earliest completion time. The task with the maximum sufferage value is chosen to be scheduled on the resource which is expected to complete the task at the earliest time.

Heterogeneous Earliest Finish Time (HEFT). This algorithm has been proposed by Topcuoglu *et al* in [Topcuoglu et al., 2002] and applied in the ASKALON project [Wieczorek et al., 2005]. It consists of

Algorithm 2 Min-Min, Max-Min and Sufferage scheduling algorithms

while $\exists t \in \Gamma$ **do**
  for all $t \in \Gamma$ **do**
    $availResources \leftarrow$ get all available resources for t
    for all $r \in availResources$ **do**
      compute ECT(t, r)
    end for
    get the resource R_t having MCT(t) = min ECT(t, r), $r \in availResources$
  end for
  // Min-Min: get the task having minimum ECT(t, r)
  $\tau \leftarrow$ min ECT(t, R_t), $t \in \Gamma$
  // Max-Min: get the task having maximum ECT(t, r)
  $\tau \leftarrow$ max ECT(t, R_t), $t \in \Gamma$
  // Sufferage: get the task having maximum sufferage suf_t
  get the resource R_t^1 having earliest ECT(t, r), $r \in availResources$
  get the resource R_t^2 having second earliest ECT(t, r), $r \in availResources/\{R_t^1\}$
  // compute the sufferage value for task t
  suf_t = ECT(t, R_t^2) - ECT(t, R_t^1)
  $\tau \leftarrow$ max $suf(t)$, $t \in \Gamma$
  schedule τ on R_τ
  remove τ from Γ
end while

three phases in which two first phases correspond to the task prioritizing phase of other list scheduling algorithms:

- the weight phase: assign the weight to workflow nodes;
- the ranking phase: create a sorted list of tasks in the order of how they will be executed; and
- the mapping phase: assign tasks on resources.

Algorithm 3 Heterogeneous Earliest Finish Time (HEFT) algorithm

compute the average execution time of each task t according to equation 2.1
compute the average data transfer time between tasks and their successors according to equation 2.2
compute the rank value of each task t according to the 2.3 and 2.4
sort the task in a scheduling list Γ by decreasing order of the task rank value
while $\exists t \in \Gamma$ **do**
  $t \leftarrow$ remove the first task from Γ
  get resource r which can complete t at the earliest time
  schedule t on r
end while

We present the detail of the HEFT in Algorithm 3. Denote $\overline{w_i}$ as the weight of task t_i and R_i as the set of available resources for t_i. The estimated execution time of task t_i on the resource r is $time(t_i, r)$. The weight of task t_i is the average estimated execution time on all available resources for this task, computed by 2.1.

$$\overline{w_i} = \frac{\sum_{r \in R_i} time(t_i, r)}{|R_i|} \tag{2.1}$$

Let $time(e_{i,j}, r_i, r_j)$ be the data transfer time on the edge $e_{i,j}$ between resources r_i and r_j which execute the task t_i and t_j, respectively. The average data transfer time from t_i to t_j is defined by:

$$\overline{c_{i,j}} = \frac{\sum_{r_i \in R_i, r_j \in R_j} time(e_{i,j}, r_i, r_j)}{|R_i||R_j|} \tag{2.2}$$

The ranking phase is performed traversing the workflow upwards. Workflow tasks are ordered in HEFT based on a rank function. For the exit task t_i, the rank value is:

$$rank(t_i) = \overline{w_i} \tag{2.3}$$

The rank value of other tasks is computed recursively based on 2.1, 2.2 and 2.3.

$$rank(t_j) = \overline{w_j} + max(\overline{c_{i,j}} + rank(t_i)), t_i \in succ(t_j) \tag{2.4}$$

where $succ(t_j)$ is the set of immediate successors of the task t_j. After having rank values, workflow tasks are sorted by decreasing the order of their rank value. The task with higher rank value is given higher priority.

In the mapping phase, workflow tasks are scheduled in the order of their priority, each task is assigned to the resource which is expected to complete the task at the earliest time.

Cluster- and duplication-based scheduling. Both cluster- and duplication-based scheduling are designed to reduce the data transfer time between dependent tasks of the workflow. This leads to the decrease of the application makespan. Cluster-based scheduling aims at grouping several dependent tasks into a cluster and executing this cluster on the same resource. The duplication-based scheduling aims at duplicating some parent tasks on idle resources. Children tasks are then scheduled on the same resource to avoid the data transfer.

The *Task duplication-based Algorithm for Network Heterogeneous systems* (TANH) is a hybrid of cluster-based and duplication-based scheduling [Bajaj and Agrawal, 2004]. The overview of this algorithm is show in Algorithm 4. It first traverses the task graph to compute parameters of each node including the earliest start and completion time, latest start and completion time, critical immediate successor tasks, best resource and the level of task. After that it clusters workflow tasks based on these parameters. Tasks in the same cluster are scheduled on the same resource. If the number of clusters is greater than the number of available resources, it scales down the number of clusters to the number of available resources by merging some clusters. Otherwise, it uses the idle resources to duplicate tasks.

Algorithm 4 Task duplication-based Algorithm for Network Heterogeneous systems (TANH) algorithm

```
compute parameter of each task node
cluster workflow tasks
if the number of clusters is greater than number of available resources then
    reduce the number of clusters to the number of available resources
else
    perform the duplication of tasks
end if
```

Conclusions. All list scheduling approaches only focus on minimizing the application makespan. Min-Min schedules tasks having shortest execution time first, it results in the higher percentage of tasks assigned to their best choice than Max-Min. However, in the case where there are many more short tasks than long tasks, Max-Min has better performance than Min-Min [Braun et al., 2001]. The Sufferage approach can have good performance in high heterogeneity environment where there is large performance difference between resources. However, Casanova *et al* argue in [Casanova et al., 2000] that Sufferage can perform worst in case of data intensive applications in multiple cluster environments. Consequently, this heuristic is not the best choice for medical image analysis applications which usually process large input data sets. HEFT and TANH analyze the dependency between tasks of the workflow to achieve better overall performance. However, they again only focus on the optimization of the application makespan and can have the different performance on different applications [Zhao and Sakellariou, 2003].

2.4.1.3 Greedy Randomized Adaptive Search Procedure (GRASP)

GRASP [Feo and Resende, 1995] is a meta-heuristic scheduling approach based on the interactive randomized search technique. Algorithm 5 describes GRASP. In GRASP, a number of iterations are conducted to search a possible optimal solution for scheduling tasks on resources. Each iteration is made up a construction phase and a local search phase.

The construction phase generates a feasible scheduling solution. It uses a *Restricted Candidate List* (RCL) to record the best resources for processing each task. There are two mechanisms to generate the RCL: *cardinality-based* and *value-based* mechanism. The cardinality-based mechanism recordes the k best resources, while the value-based mechanism records all resources whose performance is better than a given threshold.

In the local search phase, the neighborhood of the current solution is searched to generate a new solution. The new solution will replace the current solution if its overall performance is better than that of the current solution (*i.e.* its makespan is shorter than that generated in the construction phase).

In GRASP, resource allocated to each task is randomly selected from its RCL. After allocating a resource to a task, the resource information is updated and the scheduler continues to process other unmapped tasks.

Algorithm 5 Greedy Randomized Adaptive Search Procedure (GRASP) algorithm

while stopping criterion not satisfied **do**
  $schedule \leftarrow$ **createSchedule**($workflow$)
  if $schedule$ is better than $bestSchedule$ **then**
    $bestSchedule \leftarrow schedule$
  end if
end while
procedure createSchedule($workflow$)
$solution \leftarrow$ **constructSolution**($workflow$)
$newSolution \leftarrow$ **localSearch**($solution$)
if $newSoulution$ is better than $solution$ **then**
  return $newSolution$
end if
return $solution$
end procedure
procedure constructSolution($workflow$)
while schedule is not complete **do**
  $\Gamma \leftarrow$ get all unmapped ready tasks
  make a RCL for each task $t \in \Gamma$
  $subSolution \leftarrow$ select a resource randomly for each $t \in \Gamma$ from its RCL
  $solution \leftarrow solution \cup subSolution$
  update information for further RCL making
end while
return $solution$
end procedure
procedure localSearch($solution$)
$newSolution \leftarrow$ find an optimal local solution
return $newSolution$
end procedure

2.4.1.4 Conclusions

In this section, the overview of best-effort scheduling algorithms is given. In general, heuristic algorithms can produce a reasonable solution in polynomial time. Among them, individual task scheduling is the simplest but only suitable for simple workflows such as a pipeline in which several tasks are required to be executed sequentially. Most medical image analysis applications are complex applications which have many tasks competing for a limited number of resources. List scheduling algorithms can make a more efficient schedule for such applications.

Although data transfer time has been considered in list scheduling algorithms, they still may not provide an efficient scheduling for data intensive workflow applications, in which a significant fraction of computation time is used for transferring the input data and results between dependent tasks. List

scheduling algorithms focus on finding an execution order for a set of parallel tasks, and the definition of the best resource for each task is based on the information of the task. Therefore, they may not assign data dependent tasks on resources which can optimize the data transfer. Both cluster- and duplication-based algorithms aim at reducing the data transfer time between dependent tasks. Cluster-based algorithms group dependent tasks and schedule them on the same resources, while duplication-based algorithms duplicate several tasks on idle resources. However, the cluster-based optimization is not practical since workflow tasks in medical image analysis applications are usually heterogeneous in computation. Grouped tasks may require different type of resources. Similarly to the cluster-based optimization, idle resources in the duplication-based optimization may not satisfy the requirements of tasks to be duplicated.

The meta-heuristic based workflow scheduling uses guided random search technique and exploits the feasible solution space iteratively. GRASP generates a randomized schedule at each iteration and keeps the best solution as the final solution. Compared with heuristic-based algorithms, meta-heuristic algorithms provide an optimized scheduling solution based on the performance of an entire workflow. However, the scheduling time required for finding the optimal solution is usually higher than the heuristic-based approach. Therefore, meta-heuristic algorithms can be usually applied for simple workflows only.

2.4.2 QoS constraint based workflow scheduling

Many workflow applications require some assurances of Quality of Service (QoS) such as deadline or execution cost constraints. For such applications, the workflow scheduler has to be able to analyze users QoS requirements and map workflow tasks on suitable resources. However, whether users requirements can be satisfied depends not only on the scheduling decision of the workflow scheduler but also on the local resource allocation mechanism. If every single task cannot be completed as expected, it is not possible to guarantee the entire workflow execution. A *Service Level Agreement* (SLA) should be established to allow the workflow scheduler to negotiate with the infrastructure providers. An SLA is a contract specifying users expectations and obligations between users and infrastructure providers. The scheduler needs to determine the QoS for each workflow task, so that the QoS of the whole workflow is satisfied.

Within SLA context, users usually have to pay for the resource access and service pricing based on the QoS level. Therefore, the goal of workflow scheduling algorithms is to find out the trade-off between the execution cost and achievable performance rather than to complete the execution at the earliest time. To date, popular QoS constraints are the execution cost and deadline. In this section, we present the representative QoS constraints based scheduling algorithms.

2.4.2.1 Deadline constraint scheduling

Many workflow applications are time critical and require the execution to be completed before a certain deadline. The deadline constraint scheduling is designed for such applications.

Back-tracking algorithm. The *Back-tracking* algorithm [Menascé and Casalicchio, 2004] is a representative of the deadline constraint scheduling approach. This heuristic assigns workflow ready tasks

to the least expensive computing resources. If there is more than one ready task, the algorithm assigns the task with the largest computational demand to the fastest resource in its available resources list. The heuristic repeats the procedure until all tasks have been mapped. After each iterative step, the execution time of current assignment is computed. If the execution time exceeds the deadline, the heuristic back-tracks the previous step, removes the least expensive resource from its resources list and reassigns it with the reduced resource set. If the resource list is empty, the heuristic keeps back-tracking to the previous step, reduces corresponding resource list and reassigns tasks.

Deadline/Time Distribution (TD). Instead of back-tracking and repairing the initial schedule, the TD heuristic [Yu et al., 2005] partitions the workflow and distributes the overall deadline into each workflow task based on the task workload and dependencies. After the deadline distribution, the whole workflow scheduling problem has been divided into several sub-workflow scheduling problems.

To partition the workflow, the TD defines two categories of workflow tasks: *synchronization* and *simple* task. A synchronization task is a task which has more than one parent or child task. For instance, in the workflow shown in figure 2.5a, T_1, T_{10} and T_{14} are synchronization tasks. Other tasks which have only one parent or child task are simple tasks. For example, $T_2 - T_9$ and $T_{11} - T_{13}$ are simple task. Simple tasks are then clustered into a *branch*. A branch is defined as a set of inter-dependent tasks which are executed sequentially between two synchronization tasks. As shown in figure 2.5b, branches of the workflow in figure 2.5a are $\{T_2, T_3, T_4\}$, $\{T_5, T_6\}$, $\{T_7\}$, $\{T_8, T_8\}$, $\{T_{11}\}$ and $\{T_{12}, T_{13}\}$. The overall deadline is then distributed into branches and synchronization tasks. Based on the deadline of each branch, a sub-deadline will be assigned to tasks of this branch.

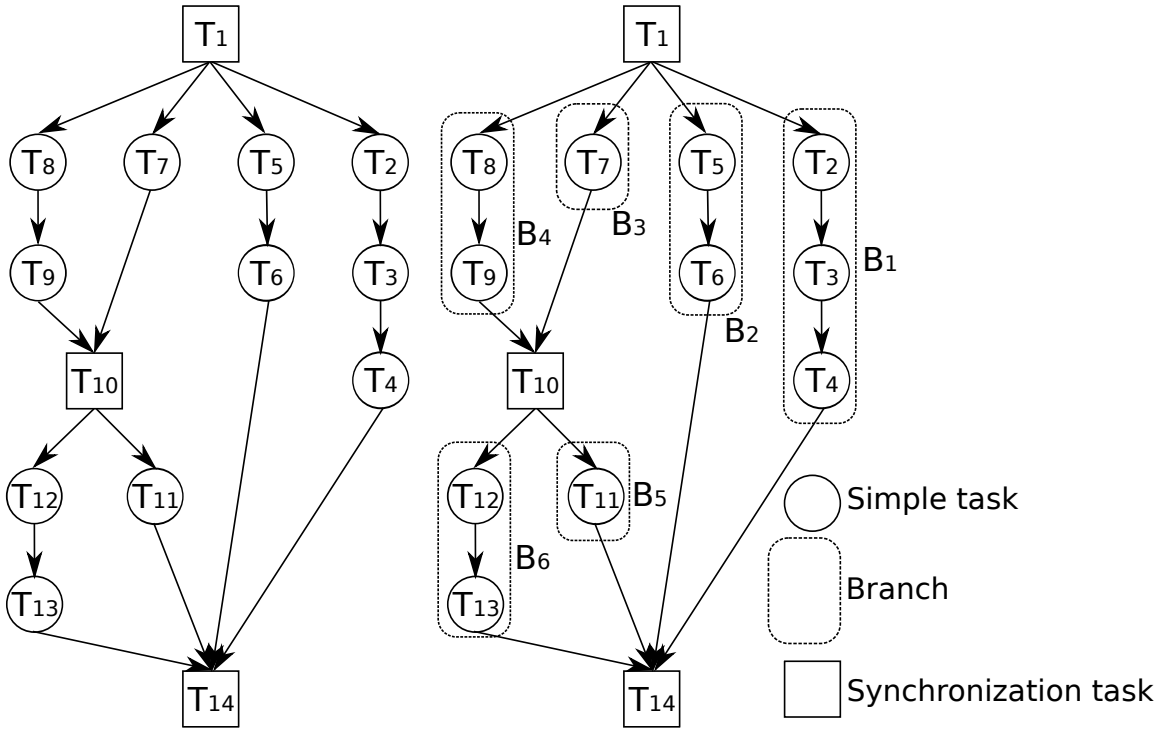
Once each task has its own sub-deadline, a local optimal schedule can be generated for each task. If each local schedule guarantees that its task execution can be completed within its sub-deadline, the whole workflow execution will be completed within overall deadline. Similarly, the result of the cost minimization solution for each task leads to an optimized cost solution for the entire workflow. Therefore, an optimized workflow schedule can be constructed from all local optimal schedules. The scheduler allocates every workflow task to a selected resource which can meet the sub-deadline of the workflow task at the low execution cost.

2.4.2.2 Cost constraint scheduling

As users have to pay to achieve the QoS level, they sometimes would like to execute the workflow with the minimized execution cost while guaranteeing the QoS level. It is noted that the execution cost is proportional to the execution time. Cost constraint scheduling therefore intends to minimize the application makespan while meeting users cost constraint. Sakellariou *et al* [Sakellariou et al., 2005] present a cost constraint scheduling called *LOSS* and *GAIN*.

The *LOSS* and *GAIN* approach iteratively adjust a schedule which is generated by a time optimized heuristic or a cost optimized heuristic to meet users cost constraints. A time optimized heuristic aims at minimizing the application makespan while a cost optimized heuristic aims at minimizing the execution cost.

If the total execution cost of the schedule generated by time optimized heuristic is not greater than the cost constraint, this schedule can be used as the final assignment. Otherwise, *LOSS* is applied. The



(a) Before partitioning (b) After partitioning

Figure 2.5: Workflow task partition with the TD algorithm

idea of *LOSS* is to gain a minimum loss in execution time for the maximum cost savings to satisfy the cost constraint. The algorithm repeats to reassign tasks with the smallest loss in execution time until the cost constraint is satisfied.

If the total execution cost of the schedule generated by the cost optimized heuristic is less than the cost constraint, *GAIN* is applied to reduce the execution time. The idea of *GAIN* is to gain a maximum benefit in execution time for the minimum increase of execution cost. The algorithm repeats to reassign tasks with the biggest gain in execution time until the total cost exceeds the cost constraint.

2.4.2.3 Genetic algorithm

A genetic algorithm based approach [Yu and Buyya, 2006] is also developed to solve the workflow scheduling problem within the deadline and execution cost constraints. As the goal of the scheduling is to maximize the performance based on two factors: execution time and cost, a fitness function is defined and separated into two parts: *cost-fitness* and *time-fitness*. Both functions use two binary variables, α and β. If users specify a cost constraint, then $\alpha = 1$ and $\beta = 0$. If users specify a deadline constraint, then $\alpha = 0$ and $\beta = 1$. For the cost constraint scheduling, the cost-fitness component encourages the formation of the solution which satisfies the cost constraint while minimizing the execution time. For the deadline constraint scheduling, the time-fitness component encourages the algorithm to choose the schedule with less cost.

In conclusion, QoS based scheduling algorithms solve the problem of scheduling workflow applications within QoS constraints. The back-tracking heuristic is more naive. It is like a constraint based

Myopic algorithm since it makes a greedy decision for each task without planning for the entire workflow. Conversely, the deadline distribution makes a scheduling decision for each task based on a planned sub-deadline according to dependencies between workflow tasks and the overall deadline. Therefore, it has better schedule for current task and does not need to trace back the assigned schedule.

The *LOSS* and *GAIN* approach addresses the cost constraint. It takes advantage of heuristics designed for a single constraint optimization problem to solve a multi-constraints optimization problem. It adjusts the schedule optimized for one constraint to satisfy other constraints in the manner that it can gain the maximum benefit or minimum loss. Even though the original heuristics are targeted at the cost constraint scheduling problem, such concept can be used for other constraint scheduling. However, large scheduling computation time could occur for data-intensive applications due to the re-computation of the loss and gain after adjusting a task assignment.

Unlike best-effort scheduling in which only one single objective is considered, QoS constraint scheduling has to consider more objectives which are usually conflicting. It becomes infeasible to develop a heuristic to solve QoS constraint scheduling optimization problems. However, this situation often happens in distributed systems, especially cloud infrastructures where users have to pay for resources usage and specify their requirements for resources reservation. For this reason, the meta-heuristic like genetic algorithm will play an important role for the multi-objective based scheduling.

2.4.3 Conclusions

In this section, we presented an overview of workflow scheduling. Two scheduling approaches have been described. The best-effort approach aims at minimizing the application makespan and does not consider other requirements while the QoS constraint approach tries to achieve multiple objectives. In general, workflow scheduling on existing distributed systems has to face many practical problems such as:

- Resources are shared on infrastructures and many users compete for the resources;
- Resources are not under control of the scheduler;
- Resources are heterogeneous and may not all perform identically for a given task; and
- Workflow-based applications are usually data-intensive.

Therefore, workflow scheduling has to consider non-dedicated and heterogeneous executions environments. Existing scheduling algorithms can still be applied to cloud infrastructures which can have different characteristics from existing distributed infrastructures. Resources are dedicated, fully controllable thanks to the virtualization technology. Physical resources can be adjusted the performance according to users requirements. Existing algorithms therefore need to be improved to adapt to such characteristics. Furthermore, users can expect a new scheduling model which can allow them to optimize the resource allocation and execution of their applications while guaranteeing the desired performance. That has motivated us to do the research on workflow scheduling on cloud infrastructures.

2.5 Conclusions and motivation for the following

In this chapter, we presented the process of porting a medical application on a distributed system. As concluded in section 2.1, workflow languages are sufficiently expressive to describe the majority of medical applications. Among existing workflow languages, the data-driven approach is appealing for designing medical image analysis applications and executing them on a distributed system. Executing data-intensive applications on distributed systems is significantly influenced by the data management mechanism. We reviewed existing approaches to motivate our study when porting applications on cloud infrastructures. In addition to the data management problem, using workflows for describing medical applications leads to scheduling problems which are very difficult.

Moving to cloud infrastructures, existing approaches on data management and workflow scheduling are still applicable. However, to adapt to the diversity of cloud infrastructures and usage, these approaches need to be improved. With the concept of "Infrastructure as a Service", cloud infrastructures bring to users the facility and flexibility to design and configure their execution environment. Additionally, users usually have to pay for resources access on cloud infrastructures. They need new tools and operation models to optimize the execution cost. The forthcoming study presented in this thesis therefore focuses on the execution optimization of workflow-based medical analysis applications on cloud infrastructures. We present in the next chapter the cost function model for such applications. The model takes into account both network and computing resources. Based on this model, several optimizations on the resources allocation strategies will be given.

Chapter 3

Execution optimization on cloud infrastructures

Focussing on cloud infrastructures, this chapter presents the main contributions of this thesis on the execution optimization of workflow-based applications. An infrastructure cost model is given, taking into account both computing and network resources. The model is based on the application makespan estimated by a workflow planner derived from the workflow execution engine. Four resource allocation strategies are then proposed exploiting this cost model. This chapter also takes well-known distributed computing-related problems which impact application performance and therefore the infrastructure usage cost such as the data transfers overhead and the low reliability of large scale distributed systems.

Etant intéressés par les infrastructures de cloud, nous présentons dans ce chapitre les contributions principales de cette thèse sur l'optimisation d'exécution des flots applicatifs sur ce type d'infrastructure. Un modèle de coût d'utilisation d'infrastructure est proposé en tenant compte à la fois des ressources de calcul et de communication. Le modèle est basé sur le temps d'exécution estimé par un ordonnanceur dérivé du gestionnaire d'exécution de flots applicatifs. Quatre stratégies d'allocation de ressources sont ensuite proposées en exploitant ce modèle de coût. Ce chapitre prend en compte également des problèmes connus des infrastructures à grande échelle qui ont un impact sur la performance des applications et donc le coût d'utilisation d'infrastructures telles que le sur-coût de transfert des données et la faible fiabilité.

3.1 Introduction

Cloud computing infrastructures are being increasingly exploited for tackling the computation needs of large-scale distributed applications. They provide resources on demand to address the computation needs of such applications. The virtualization technologies exploited ease the migration of heavyweight applications by adapting the execution environment to the specific application requirements. However, state of the art cloud solutions delivered today hardly address the problem of estimating performance and cost related to a given platform provisioned. This chapter addresses three key problems which are correlated to each other when executing applications on cloud infrastructures:

- resources allocation and execution cost estimation,
- data transfer optimization, and
- infrastructure reliability.

3.1.1 Resources allocation and execution cost estimation

Estimating the "optimal" amount of cloud resources (network links and nodes) needed to run a specific complex distributed application given an input data set is a challenging problem, both for users who aim at minimizing their cost and infrastructure providers who aim at controlling their resources allocation and account (financial or not) for resources usage. In the commercial cloud offers, various business models have been developed to bill resources usage. They are usually based on a coarse-grained a posteriori metering of the amount of CPU and disk space consumed (*e.g.* Amazon EC2[1] charges users per hour of resources usage, per GB/month of storage and for the generated traffic in network). The estimation of the proper amount of resources to allocate is left to the responsibility of the users, although such an estimation is far from trivial, especially when considering distributed applications. From an infrastructure provider point of view, this practice is less suitable for dedicated infrastructures, such as academic clouds or intra-enterprise clouds, for which providers are not only interested in billing but also aim at improving quality of services and optimizing resources sharing. From a user point of view, assistance in resources consumption planing and cost management is highly desirable. Therefore, a finer grain model has to be proposed to (i) decide on the amount of resources to allocate to each application and (ii) compute the resources usage cost.

Determining the amount of computational and storage resources needed for each application run is often not sufficient when considering distributed applications. Communication network bandwidth is also a critical resource, shared among the infrastructure users, which may impact application performance significantly. Nowadays, the virtualization paradigm can be applied and combined to both network and computing resources and the Infrastructure as a Service can be extended to the network. This advanced cloud computing paradigm enables the definition of confined execution environments, including the amount of virtual resources needed, virtual network topology and network links bandwidth. The global cloud infrastructure manager is able to create multiple, isolated and protected environments

[1] http://aws.amazon.com/ec2/

for multiple users concurrently while sharing the same set of physical resources, without interfering with each other.

Research community has put forward a lot of effort on the cost-based resources allocation. For instance, Buyya *et al* present in [Buyya et al., 2005] a summary of existing research systems which have explored the use of economy models for trading resources in different application domains: CPU cycles, storage space, database query processing, and distributed computing. The main purpose of these systems is to provide mechanisms and tools that allow users and infrastructure providers to express their requirements and facilitate the realization of their goal. The final goal of these systems is to satisfy the largest possible number of users, thus maximizing the benefit of infrastructure providers. Additionally, they have been designed for CPU or storage management. Few systems consider the network resource which is also an important one when executing data intensive applications.

Motivated for both end users and infrastructure providers, this chapter proposes a fine-grained model to help (i) users estimating the amount of resources needed and thus optimizing the execution cost, and (ii) infrastructure providers controlling their resources allocation and billing the resources usage [Truong Huu and Montagnat, 2010]. This fine-grained model takes into account both computing and network resources [Truong Huu et al., 2011] (see sections 3.2 and 3.3).

3.1.2 Data transfer optimization

Data transfer optimization is the second problem addressed in this chapter. During an execution of data intensive applications, a considerable amount of time is spent on data transfer. Therefore, optimizing the data transfer leads to the optimization of application performance and execution cost. From an economy point of view, optimizing the data transfer reduces network bandwidth reserved for the execution, thus reducing the execution cost paid for network resources. From the application performance point of view, optimizing the data transfer reduces the application makespan. The reservation duration for computing and storage resources is therefore also reduced.

The underlying infrastructure impacts the implementation of the data transfer mechanism. On traditional systems (*e.g.* batch systems) where computing resource will be freed after finishing the computational task, users have to use a central storage server for input data and intermediate results which are the data exchanged between workflow services (*e.g.* GASS [Bester et al., 1999], OmniStorage [Aida et al., 2006]). This is sub-optimal because these intermediate results have to be copied back to the storage server before being transferred to other computing resources.

On GridRPC systems or infrastructures supporting resources reservation, several techniques can be used to optimize the data transfer as presented in [Desprez and Jeannot, 2004, Caron et al., 2005]. Among them, Data Tree Manager (DTM [Del-Fabbro et al., 2007]) is an example which provides to users a set of operations to manage data (*e.g.* addition, deletion, search and replication). Users can decide whether a data item must be persistent inside the execution platform or not. If a data item is persistent, an identifier will be assigned to it. Users must know this identifier to use it for further computations. Therefore, contrarily to the approach presented in previous paragraph, by using DTM, input data which is used several times during the execution or the intermediate results can be defined as persistent and the transfer to the central server can be avoided. Furthermore, intermediate results can stay cached inside the platform. A computing node can request for a data item through its identifier.

DTM is responsible for looking and transferring this data item to computing node. However, this service is only distributed as a part of the DIET platform [Caron and Desprez, 2006]. The users have to adopt their application and run it on the DIET infrastructure to take avantage of this data transfer optimization.

On cloud infrastructures in particular, storage resources play an important part, users have to reserve an amount of storage space for storing their data (*e.g.* Amazon S3 released storage service for persistent data). Therefore, a central storage space can be reserved for the input and output data. Temporary data generated during execution usually does not need to be backed up on permanent storage resources. A data transfer mechanism is thus needed to exploit the standard storage capacity of computing resources. Related to workflow-based applications, workflow services can be executed on different computing resources in parallel. The result of each workflow service can be cached on the computing resource and transferred directly to other computing nodes for further computation if needed. Only output data of the workflow will be transfer to storage server or users machine. Based on this approach, this chapter presents in section 3.5 our techniques to optimize the data transfer between workflow services without requiring laborious modifications to adapt the application. The results obtained assess the performance of their improvement.

3.1.3 Reliability

The last problem addressed in this chapter is the low reliability of distributed systems. We describe an extension of the "Infrastructure as a Service" paradigm to provide reliable cloud infrastructures. Distributed systems are subject to failures which happen on computing nodes and network links. Recent experimental studies show that jobs submitted by users to large-scale, multi-institutional grid infrastructures often fail to complete successfully. For example, data collected and analysed by the WISDOM project [Breton et al., 2009] which submits tens of thousands of jobs to EGEE infrastructure[2] in the context of a drug-design effort [Jacq et al., 2008], indicate that only 65% of submitted jobs were executed successfully. Recovering from the failures to enable application to continue running is therefore highly desirable.

Recovering from failures has been well studied in the literature. On large-scale infrastructures, the re-submission is one of the solutions used for recovering from failures [Berten and Jeannot, 2009, Lingrand et al., 2009a, Lingrand et al., 2009b]. The underlying infrastructure supports APIs for users to monitor the job status and detect the failure (*e.g.* OAR middleware [Capit et al., 2005] provides `oarstat` command to get status of job submitted to the infrastructure). Users are responsible for monitoring and detecting whether a failure occured, and resubmit their jobs. The application makespan is longer in this case especially when the execution time of individual tasks is long. Optimizing the application makespan and reliability is therefore a major issue as studied in [Jeannot et al., 2008], in which the authors propose a scheduling approach that minimizes both the makespan and the probability of faillure of the execution. Multi-submission is another solution to reduce chance of failure [Casanova, 2007]. For each job to be executed, a collection of k copies of a job is submitted. As soon as one job of the collection is running, all the other ones are canceled. If none of the jobs starts executing before the timeout value (t_∞), the whole collection is canceled and resubmitted. On the virtual networked environ-

[2] http://www.eu-egee.org/

ment, Kangarlou *et al* propose to periodically save static snapshots of the entire execution environment to disk, while the execution is in progress [Kangarlou et al., 2009]. The live snapshots are reloaded as a new submission if failures are encountered in the current execution. The application makespan then depends on the re-submission interval or the snapshot interval, which may be long due to disk-access time.

On cloud infrastructures, reliability becomes a *service* provided by infrastructure providers. Users have to pay for enhancing the reliability of their execution environment. The existing mechanisms are still applicable but do not provide sufficient transparency against failures. Re-initiating or resuming applications to recover from failures will impact any time-sensitive application and therefore new failover mechanisms are needed. These mechanisms will be used for both infrastructure providers to transparently provide the reliability to users, and users to efficiently provision the desired reliability level. Thanks to the virtualization technology, the reliability on cloud infrastructures can be enabled through the allocation of virtual backup nodes and links. The live migration mechanism such as Remus [Cully et al., 2008] or Kemari [Tamura et al., 2008] can be used. In this approach, the memory state of a protected node is continuously "synchronized" with a backup node, as with checkpointing. When a failure in protected node occurs, the backup node can resume the execution immediately, and the failover can be made transparently to other nodes in the execution environment. However, as discussed above, the reliability level is proportional with the execution cost. The higher the reliability level, the larger the number of backup computing resources and network link is reserved, and thus the higher the execution cost has to be paid. A trade-off between the reliability level and the execution cost needs to be found. We present in section 3.6 our analysis and solution to enhance the reliability while optimizing the execution cost [Koslovski et al., 2010].

3.2 Cost model of application execution

This section deals with the problem of estimating the amount of resources needed for a specific run of a complex distributed application. At first, we introduce the Virtual Private eXecution Infrastructure (VPXI) concept as a confined execution environment and then we formulate the cost function model used to determine which VPXI(s) better fulfill the application needs, based on information extracted from the application logic and the execution infrastructure.

3.2.1 Virtual Private eXecution Infrastructure (VPXI)

As opposed to clusters and grids, to execute distributed applications on clouds users first have to reserve resources to compose their execution environment. Theoretically, the cost of an infrastructure deployment and usage scenario may be quantitatively estimated by the system if sufficient information on the application and the infrastructure is known. In the general case though, it is hardly feasible to anticipate the precise needs of a parallel application or the behavior of such an application given a determined size infrastructure. Restraining the problem a bit more, it appears that workflow-based applications have good properties for such a quantitative estimation. Workflow-based applications represent a large class of coarse-grained distributed applications [Glatard et al., 2008b]. Taking advantage of the workflow formalism, the application logic can be interpreted and exploited to produce an execution schedule estimate.

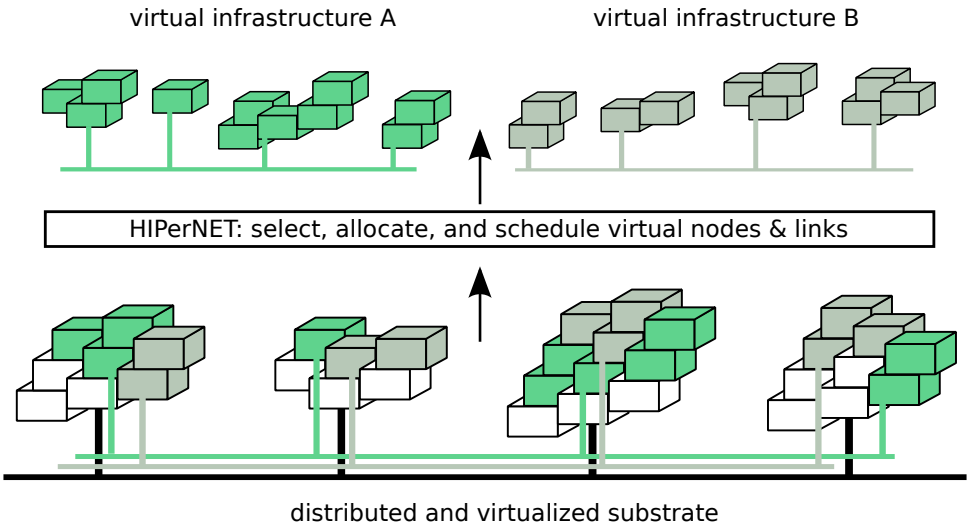
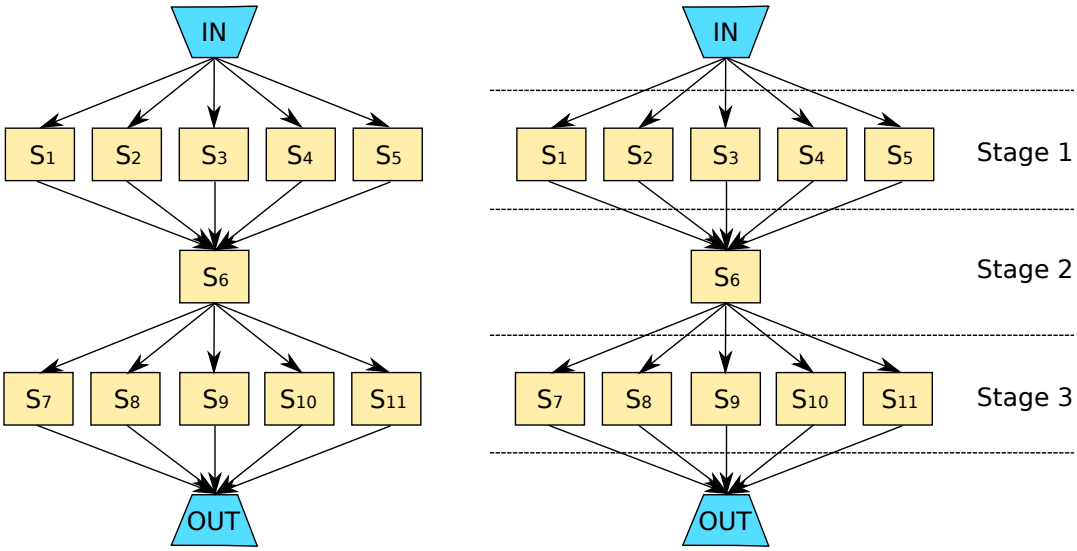


Figure 3.1: Example of a VPXI allocation on a distributed and virtualized HIPerSpace (image courtesy of HIPCAL project [Vicat-Blanc Primet et al., 2009b]).

For instance, the Bronze Standard application described in section 2.2.5 can be analyzed as illustrated in figure 2.4 to create an execution plan.

The information extracted from the application becomes useful to help users estimating the amount of resources needed for their application given an input data set. The execution environment is defined taking into acount specific requirements on the amount of resources (computing and network), the performance of computing resources and the network topology. The execution environment is specified through the *Virtual Private eXecution Infrastructure* (VPXI [Koslovski et al., 2009]) defined as a time-limited interconnection of a set of virtual computing resources through a virtual private overlay network. Cloud infrastructures provide a description language (*e.g.* VXDL [Koslovski et al., 2008]) allowing users to describe their requirements. The cloud middleware is responsible for interpreting the user description, deploying the user image on virtual machines and adjusting the performance of these machines according to the user specification. The final goal is to make the confined execution environment (VPXI) available from the execution start time until the estimated execution completion time. Figure 3.1 presents an example of two virtual execution infrastructures (VPXI A and VPXI B) managed by the HIPerNet virtualization middleware [Vicat-Blanc Primet et al., 2009b]. Each application can execute, confined in a VPXI dedicated for a defined time period. The platform pre-deployment phase is a time-consuming process needed to make the VPXI ready for execution. Additionally, several cloud middlewares (*e.g.* HIPerNet) enable the reconfiguration of the infrastructure during the execution. A VPXI reservation can be divided into several *stages*. For each stage, the VPXI can be reallocated with respect to a specific configuration, to perform the execution of one part of the application. After completing the execution, allocated resources are returned to the cloud. The VPXI reconfiguration between different stages, which may involve redeployment of resources, is also time-consuming. One extreme condition, is to create a static execution environment for the whole duration of the complete application execution, thus sparing the redeployment cost. Another extreme, is to allocate new resources one by one on demand.

Specific VPXI can be allocated for the need of each application. Workflow-based application ex-



(a) Original workflow (b) Divided into 3 stages

Figure 3.2: Workflow example and its execution divided into stages

ecution plans can be decomposed into several stages as illustrated in figure 3.2. The synchronization service S_6 has to wait for services S_{1-5} to finish before launching the execution. After S_6 finishes its execution, all services S_{7-11} can begin their execution. Consequently, the execution of this application can be divided into 3 stages, as shown in figure 3.2b. Furthermore, the number of invocations in each stage can be different, depending on the iteration strategy of each service and the number of services in each stage. Stages 1 and 3 require 5 invocations, while stage 2 requires only 1 in this case. This leads to the variation of number of computing resources needed between stages. Additionally, due to the computation heterogeneity and variety of data volume of workflow services, the desired performance of computing and network resources for each workflow service can also differ. These characteristics make workflow-based applications suitable for porting their execution on cloud infrastructures with a fine-grained model that maps the workflow execution on to several sequential stages of a VPXI. In the next section, we present the cost model which helps users estimating the amount of resources needed to describe the VPXI description.

3.2.2 Cost model formalization

The cost model proposed below makes a fine-grained estimate of the resources that will be consumed for each application run. All parameters used in the cost model are summarized in table 3.1. Let $m_{\max}$ be the maximum number of computing nodes available on the infrastructure and s be the number of execution stages of the application. The vector $m = (m_1, m_2, ..., m_s)$ is the number of nodes used in each execution stage with $\forall i, m_i \leq m_{\max}$. If we assume the per-second cost of a resource is c_r, then the total computing cost of the infrastructure allocated for the application is:

$$C_r = c_r \sum_{i=1}^{s} m_i \left(Td_i + T_i(m_i, n, b)\right) \tag{3.1}$$

where Td_i is the deployment time (including resource reservation and initialization time) and

$m_{\max}$	maximum number of computing nodes available on the infrastructure
n	number of input data items
s	number of execution stages of the application
$m = (m_1, m_2, ..., m_s)$	number of computing nodes used in each execution stage with $\forall i, m_i \leq m_{\max}$
c_r	per-second cost of a computing resource
c_b	per-Mbps cost of the bandwidth
Td_i	the deployment time of stage i (in seconds)
$T_i(m_i, n, b)$	the execution time of stage i (in seconds)
$b = (b_{i,1}, b_{i,2}, ..., b_{i,ki}), i \in [1..s]$	link bandwidth used in stage i (in Mbps)

Table 3.1: Notations used in the cost function model.

$T_i(m_i, n, b)$ is the execution time at stage i. T_i depends both on computing time and data transfer time involved within stage i. It is parameterized by the number of resources reserved (m_i), the number of input data items to process (n) and the bandwidth ($b = (b_{i,1}, b_{i,2}, ..., b_{i,ki}), i \in [1..s]$) of the network links used for data transfer. The computation of T_i is possible using the application logic described through the workflow.

The total infrastructure cost is also impacted by the data transfer time. If the per-Mbps cost of the reserved bandwidth is c_b, then the total data transfer cost is:

$$C_b = c_b \times \sum_{i=1}^{s} (Td_i + T_i(m_i, n, b)) \sum_{j=1}^{k_i} b_{i,j} \tag{3.2}$$

This cost applies to an infrastructure where the amount of network bandwidth allocated is controlled (*e.g.* HIPerNet [Koslovski et al., 2009]). It sums all data transfer costs involved in the workflow execution, including workflow input data transferred from outside the cloud (at stage 1), the temporary data generated during workflow execution (at all stages) and the output data transferred to external resources (at stage s).

From formulas 3.1 and 3.2, the total infrastructure cost to execute the application can be computed:

$$C = C_r + C_b \tag{3.3}$$

This cost has to be optimized considering a maximum admissible cost and the application performance scalability. C depends on the value of T_i at each execution stage. A trade-off has to be found between the amount of computing resources and network resources allocated (which impacts T_i), and the resulting cost. The computation of T_i is possible using the application logic described through the workflow. The workflow engine used in our experiment, MOTEUR [Glatard et al., 2008b], was seminally designed to produce an execution schedule and control the distribution of an application at runtime. It was enriched with a resource allocation and scheduling planner that is used to estimate T_i, given that information on the workflow services execution time and transferred data amount is available. The workflow engine simulates the execution by resolving the application execution flow as it would do during a regular run. Instead of actually invoking the application processes though, the workflow planner just

keeps track of the estimated tasks execution duration, in virtual execution time, thus determining the exact number of tasks to be executed of each instance after execution start.

3.2.3 Comparison to a commercial offer

The cost model described in equations 3.1 and 3.2 can be used for cost estimation both from an infrastructure provider and an infrastructure user point of view. Depending on the intended usage, it may be tuned. For instance, Amazon EC2 cloud computing offer charge users per hour, day or week of usage. The times estimated are therefore rounded at the ceil value in the unit considered. In addition, Amazon EC2 does not account for infrastructure deployment time in billing ($Td_i = 0$). This cloud infrastructure also does not make it possible to adapt nor guarantee the network bandwidth allocated. The amount of network resources is therefore billed on the basis of the total amount of data transferred rather than the amount of bandwidth consumed. Finally, Amazon charges for workflow input and output data transfers (data transfer from and to the storage resources outside the cloud) additionally, while in the model proposed above this transfer is accounted for in C_b (equation 3.2).

Consequently, the cost billed for the EC2 computing resources usage is one of:

$$C'_r = \begin{cases} c'_r \times m_{\max} \times \left\lceil \dfrac{\sum_{i=1}^{s} T_i(m_i, n, b)}{3600} \right\rceil & \text{(3.4a)} \\ c'_r \times \sum_{i=1}^{s} m_i \times \left\lceil \dfrac{T_i(m_i, n, b)}{3600} \right\rceil & \text{(3.4b)} \end{cases}$$

where c'_r is the Amazon EC2 per-hour unit cost of computing resources. Case 3.4a applies if a single reservation is made for the whole duration of the workflow execution. In that case, there is a single stage and the maximum number of resources ($m_{\max}$) will be reserved. Case 3.4b applies if one reservation is made for each stage. Compared to equation 3.1, the cost computed in equation 3.4 is impacted by rounding to the next hour. In particular in case of multiple reservations (case 3.4b), the rounding at each stage may be penalizing. A trade-off has to be found between reserving the maximum number of resources for the whole duration of the computation (case 3.4a) and adapting the number of resources at each stage, at the expense of an over-estimated platform usage time (case 3.4b).

Similarly, the cost charged for usage of network resources when transferring input/output data in Amazon EC2 is:

$$C'_b = c'_b \times V_D \tag{3.5}$$

where V_D is the total amount of data transferred between EC2 and other data sources (*e.g.* the user machine or a database server on Amazon S3), and c'_b is a per-volume unit cost. Unlike equation 3.2, this cost cannot be adapted to specific network usage requirements. This reflects the fact that this infrastructure does not provide any bandwidth control mechanism.

The total Amazon EC2 cost is:

$$C' = C'_r + C'_b \tag{3.6}$$

3.3 Resources allocation strategies for workflow-based applications

The application execution time for each stage (T_i) depends on the amount of resources allocated within each VPXI. To determine VPXIs and estimate the corresponding execution times, we describe below four strategies named *naive*, *FIFO*, *optimized* without services grouping optimization and *optimized* with services grouping optimization.

3.3.1 Naive strategy

Given p the number of services composing an application workflow and t_i the benchmarked execution time of service $i \in 1..p$, a set of $m_{\max}$ virtual computing nodes is allocated and split proportionally to each service execution time: $m_{\max} t_i / \sum_j t_j$ nodes are dedicated to the service i. The network bandwidth is similarly allocated proportionally to the amount of data to transfer between each pair of services, or the same bandwidth is reserved for all links in the infrastructure. This strategy is naive in the sense that it only considers a single execution stage and the resources are statically allocated to each service even though a service may not be involved during the whole duration of the workflow execution. This strategy serves as a performance base-line.

3.3.2 FIFO strategy

In this approach, we make the simplifying assumption that all services can be deployed on every computing resource. These resources are thus indistinguishable and the scheduler may request any task to be executed on any resource. A *FIFO* scheduling strategy is optimal in this case and a single stage is considered since infrastructure redeployment is unnecessary ($T = T_1$). In addition, the same bandwidth is reserved for all links in the infrastructure ($b_1 = b_2 = ... = b_k$). As an example, figure 3.3 displays the estimated execution time and the total cost of the workflow from figure 2.3 on page 48 with regard to the bandwidth (for $n = 32$ input data items and unit costs $c_r = c_b = 0.2$). When the bandwidth is small, the total cost is high due to the data transfer time. When the bandwidth increases, the execution time and cost both decrease. However, after a 2.0Mbps threshold, the execution time only slightly reduces while the bandwidth allocation cost increase dominates. The optimization method used to numerically approximate the optimal bandwidth leads to 0.6517Mbps.

3.3.3 Optimized strategy

The *FIFO* strategy can only apply to identical resources and without optimizing the bandwidths between each pair of resources. Conversely, the optimized strategy described below considers dividing the workflow execution in multiple stages and allocating resources and bandwidth independently for each stage. The cost minimization algorithm is executed for each stage to allocate an optimal number of virtual resources to the services involved in this stage.

An algorithm is needed to decide on the number of stages and when infrastructure reconfiguration should happen. Firstly, the workflow of services is transformed into a *Directed Acyclic Graph* (DAG), using the second composition approach presented in [Zhao and Sakellariou, 2006] for instance. Secondly, the DAG is divided in execution stages, each of them meant to be executed on a specific virtual

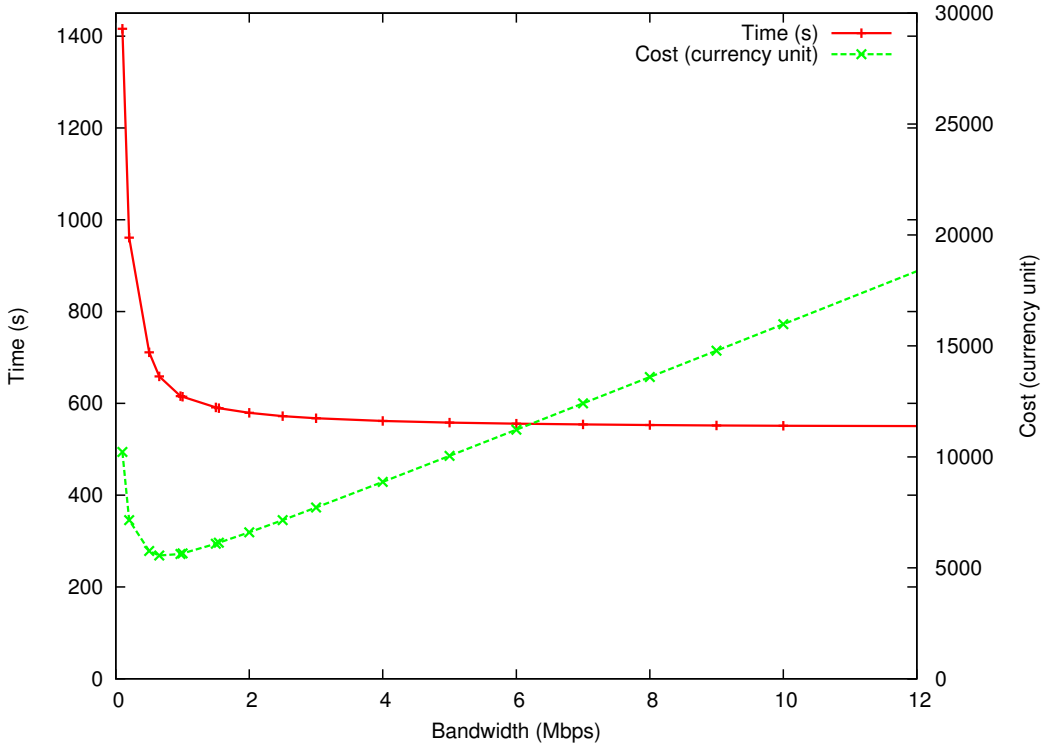


Figure 3.3: Estimation of the execution time and total cost with regard to the bandwidth of the *FIFO* strategy

infrastructure. An example execution DAG for the workflow of figure 2.3 is shown in figure 3.4, where `IN` and `OUT` are special entry and exit nodes that are not accounted for in the execution and data transfer times estimation. The pseudo-code of the DAG split into stages is presented in algorithm 6. An execution stage is defined as the set of invocations which have the same depth in the DAG.

Note that the DAG generation is only possible for workflows without unbounded loops (the exact number of invocations of each service needs to be known) so that the workflow planner can determine a complete execution schedule. Workflows including *while* kind of loops, or *foreach* constructs iterating over unknown size data structures make the workflow unresolvable prior to execution. This is limiting the class of applications that can be planned. Yet, this represents a broad category of workflow applications in e-Science (many data-intensive, scientific workflow languages do not support loops [Deelman et al., 2003]). A solution for dealing with workflows with unresolvable constructs is to divide them into smaller resolvable sub-workflows. This generation process has to be revised dynamically though (*e.g.* each time a loop is iterated, the loop body sub-workflow can be generated). Such a strategy was implemented in the workflow manager of the DIET middleware (MA DAG) for instance, to deal with workflows which could not be represented by DAGs[3].

At each execution stage, the infrastructure is reconfigured for only deploying the specific services involved in that stage. The resources are allocated proportionally to the number of invocations needed for each service. In a typical data intensive application execution, there are more data items to process (n) than resources available ($m_{\max}$). For instance, in the case of a stage i with only one service S (*e.g.*

[3]DIET MA DAG: `http://graal.ens-lyon.fr/~diet/workflow.html`

Algorithm 6 Execution DAG split into stages

Require: **processedServices** list initialized with all workflow inputs.
Require: **stage** = 1

```
while There are still services to process do
  stage-services = empty list
  for each service S in workflow do
    if all inputs of S come from the list of processed services then
      add S into stage-services
      set stage of service S to stage
    end if
  end for
  add list stage-services to list processedServices
  increment the stage counter (stage = stage + 1)
end while
```

stage 1, 2 or 4 in figure 3.4), $m_{\max}$ data items are processed concurrently by S and the process is repeated $n/m_{\max}$ times, leading to the execution time:

$$T_i = \left\lceil \frac{n}{m_{\max}} \right\rceil \times T_S \tag{3.7}$$

where T_S is the execution time for S.

More generally, the optimal resources and bandwidth allocation strategy, taking into account the number of service invocations, the execution time and the data transfer time in each stage is computed using the multi-criteria *Downhill Simplex* minimization method. Let $inv_j, j = 1..s$ be the number of invocations of service j at stage i where s is the number of services being executed at this stage. Let vector $m = (m_1, m_2, ..., m_s)$ be a combination of number of resources allocated to the service j. This combination must satisfy the condition $\sum_{j=1}^{s} m_j \leq m_{\max}$. The resulting optimal execution time to complete inv_j invocations of service j is:

$$T_j = \left\lceil \frac{inv_j}{m_j} \right\rceil \times T_{uj} \tag{3.8}$$

where T_{uj} is the unit execution time of service j.

3.3.4 Services grouping optimization

The total execution cost also depends on the infrastructure deployment time of each stage. We present in this section an optimization of the total resources reservation and redeployment time by extending the job grouping strategy without loss of parallelism introduced in [Glatard et al., 2008a]. The seminal strategy of grouping without loss of parallelism minimizes the application makespan by grouping services which would have been executed sequentially, thus reducing data transfers and the number of job invocations needed. Applying this strategy to the Bronze Standard workflow (described in section 2.2.5), two service groups are identified which do not cause loss of parallelism as shown in figure 3.5a. The number of execution stages can also be reduced as shown in figure 3.5b.

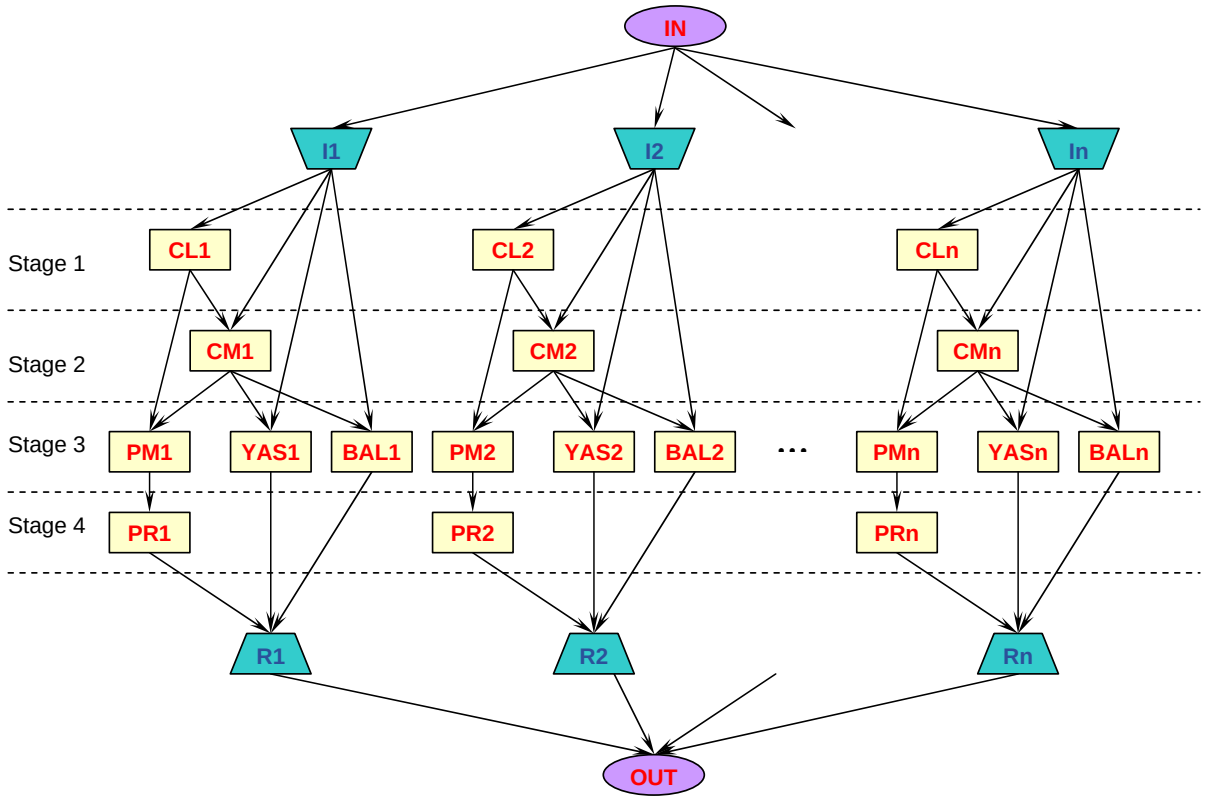
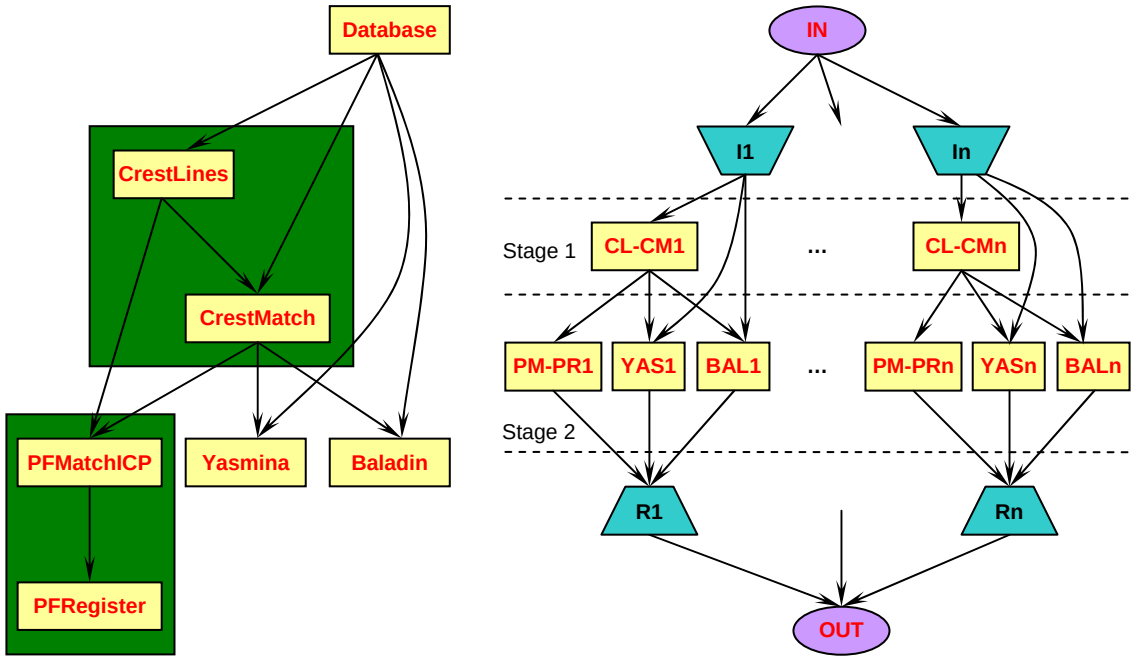


Figure 3.4: DAG jobs of the Bronze Standard application for n inputs



(a) Grouping without parallelism loss (b) Resulting execution DAG considering n input data items

Figure 3.5: Services grouping without parallelism loss of the Bronze Standard application

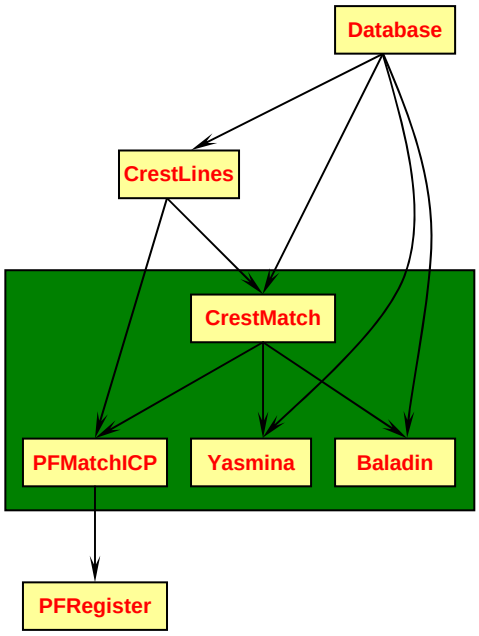


Figure 3.6: Grouping CrestMatch, PFMatchICP, Yasmina and Baladin

A limitation of this algorithm is that it only exploits workflow topology information but not the actual execution cost of the services, although it might be preferable to loose some degree of parallelism, when the grouping gain is higher. The trade-off can be found thanks to the execution planner developed for the allocation strategies. Starting from the execution DAG split into stages, job invocation groups are evaluated for each consecutive pair of stages. For each service A of the workflow involved in the stage i, let $B_0, B_1, ..., B_j$ be all children from A in stage $i+1$. All possible combinations of grouping A with one or more of the B_k services is tested and the resulting execution cost is evaluated by optimizing the number of resources and the bandwidth allocated. In the Bronze Standard application, the best solution is shown in figure 3.6.

3.4 Handling the uncertainty and exception in real execution

As discussed in section 3.3, to allocate resources to the application, the workflow planner exploits the information extracted from the application: start time, execution time and data volume to be transferred. Among them, the execution time plays the important role. It is usually estimated or predicted based on history of executions or benchmarks. However, due to the variations of data volumes and the non-deterministic nature of some algorithms, the estimated execution time that is usually an average value may differ from the actual value at runtime. Users will tend to overestimate their application to ensure that the application execution is not aborted if the actual runtime is higher than the estimated one. This leads to the degradation of infrastructure performance and also the increase of the execution cost that users have to pay. In [Canon and Jeannot, 2008], the authors propose a scheduling algorithm that can absorb part of the uncertainty and gives an allocation whose execution time is still close to the predicted value. However, it does not take into account the execution cost when executing the application on a pay-per-use platform.

We propose in this section a technique to handle the uncertainty on the actual execution duration. Prior to the execution, the workflow planner uses the estimated execution time to allocate resources. During the execution, the workflow planner monitors the progress of each task on each computing node and reevaluates the execution plans dynamically accordingly.

At stage i, if the real execution time (Tr_i) happens to be shorter or longer than the estimated execution time (T_i):

$$Tr_i = \rho \times T_i \tag{3.9}$$

then, the associated cost according to the cost model introduced in section 3.2.2 would be:

$$C_r = c_r \times m_i \times (Td_i + Tr_i) \tag{3.10}$$

However, in that case, the cost reduction or increase caused by improper estimation of the execution time should be strengthened to avoid abuses of the infrastructure. Without extra cost, a malicious user could reserve for an arbitrary long time and later on resize the reservation without penalty. An extra cost ($Cextra_r$) is therefore added to the real usage cost:

$$C_r = c_r \times m_i \times (Td_i + Tr_i) + Cextra_r \tag{3.11}$$

If the real execution time (Tr_i) is shorter than the estimated execution time (T_i), the extra cost is proportional to the estimated time deviation:

$$Cextra_r = \lambda_1 \times c_r \times m_i \times (T_i - Tr_i) \tag{3.12}$$

where the per-unit cost of resources is $\lambda_1 \times c_r$, $\lambda_1 \in [0, 1]$.

The problem is more challenging when the real execution time (Tr_i) is longer than the estimated execution time (T_i). We propose two approaches for two kinds of infrastructure: *non-shared infrastructure* and *shared infrastructure*. On a non-shared infrastructure, resources are reserved exclusively to the application during its execution. One or more constant duration time interval τ will be allocated to the stage to finish its execution. The start time of the next stage will be postponed until all tasks of the previous stage finish. On a shared infrastructure, the infrastructure manager has to consider the execution of other applications. If the additional time allocated for the task of this application does not violate the start time constraint of other applications, the schedule is confirmed. Otherwise, the task is cancelled. If we assume that the per-unit cost of resources increases linearly with the number of additional time intervals to meet the real execution time ($j \times \lambda_2 \times c_r$ for the j^{th} additional time interval), the extra cost would be:

$$Cextra_r = \lambda_2 \times c_r \times \tau \times m_i \times \sum_{j=1}^{\chi} j \tag{3.13}$$

where $\chi = \left\lceil \frac{Tr_i - T_i}{\tau} \right\rceil$ is the number of additional time intervals added.

It is to be noted that each time an additional time interval is allocated to the task, the workflow planner needs to reevaluate the execution plan. Therefore, if the value of τ is much smaller than the actual error in the underestimated execution time, more overheads will be imposed to the workflow planner. On the other hand, if the value of τ is larger than the actual error, the number of tasks cancelled may

increase. This leads to the degradation of performance of the infrastructure. In [Farooq et al., 2009], authors present an approach to calculate τ value. If we assume that the error in the estimation is proportional to the estimated execution time, we can choose the value of τ as: $\tau = \sigma \times T_i$. If the mean error in the underestimated execution time is some proportion θ of the mean estimated execution time T_i, the average number of time χ, the additional time should be added to complete the execution of task is given by:

$$\chi = \frac{\theta \times T_i}{\sigma \times T_i} = \frac{\theta}{\sigma} \tag{3.14}$$

The value of θ is calculated based on the past history. The value of σ is then adjuted accordingly to ensure that the value of χ stays near a desired value (*i.e.* which does not cause the overhead imposed to the workflow planner while minimizing the number of tasks cancelled).

The same approach can be applied for the network bandwidth.

$$C_b = c_b \times (Td_i + Tr_i) \times \sum_{j=1}^{k_i} b_{i,j} + Cextra_b \tag{3.15}$$

In conclusion, this section presents our approach to handle the uncertainty during the real execution. An execution stage could be overestimated or underestimated and does not finish at the expected time. Both users and cloud providers can benefit from our approach. In the case the application is underestimated, the application can still execute if it does not violate any constraint. Cloud providers can increase the performance of the infrastructure which can be defined as the number of applications executed successfully. In any case, users will be charged for extra cost if the application does not finish at the expected time. A "near-optimal" estimation is therefore needed.

3.5 Improving data transfer performance

As discussed in the previous chapter, the data transfer mechanism plays an important role in the optimization process. It impacts the application makespan, the network resources consumed by the execution and therefore the execution cost.

Depending on underlying infrastructures, suitable data transmission mechanism can be used. Intelligent scheduling strategies have been used to schedule dependent tasks on the same computing resources to avoid the data transfer between dependent tasks (*e.g.* Request Sequencing [Arnold et al., 2000]). This approach is sub-optimal since the parallelism exploitation is limited. Other solutions are to use relay hosts in the infrastructure or a central server to store the data (*e.g.* DSI [Beck and Moore, 1998], GASS [Bester et al., 1999]). On traditional systems (*e.g.* batch system), it is mandatory to use this approach since the computing resource does no longer belong to the user after finishing the task. Considering a simple example as shown in figure 3.7, the workflow has two dependent services S_1 and S_2, executing on a batch system. The service S_1 copies the input data a from the storage server and executes on a computing node. After finishing the execution, the computing node has to send the result $S_1(a)$ to the storage server since it will be released. The service S_2 in turn has to copy the result $S_1(a)$ from the storage server and executes on another resource, the result $S_2(S_1(a))$ has to be sent to the storage server.

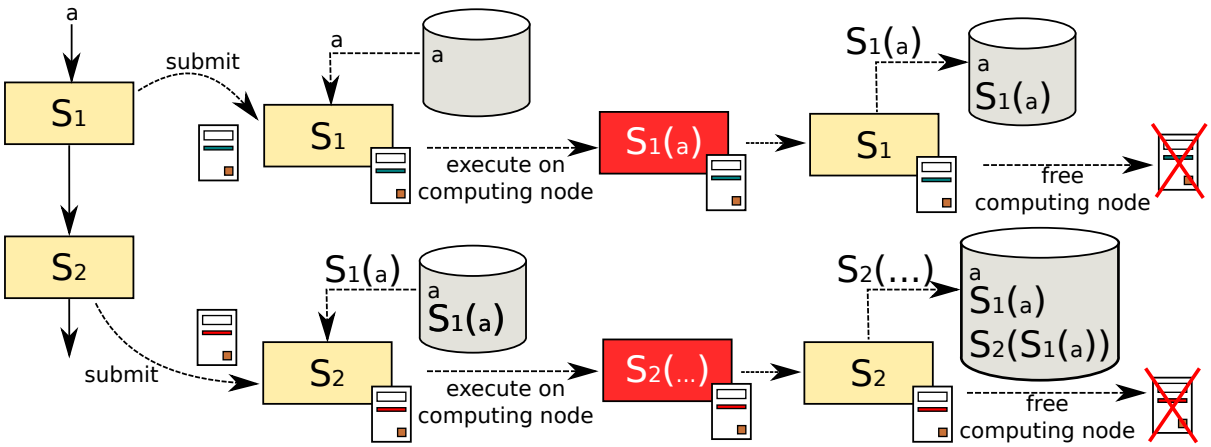


Figure 3.7: Using a central storage server for input data and intermediate results, the default execution mode of batch systems.

Although this mechanism is sub-optimal because intermediate results have to be transferred twice before arriving to other computing resources, users have no better mean to reduce the data transfer time.

On systems which are subject to advance reservation (*e.g.* Grid'5000 or cloud infrastructures), users reserve a set of resources for a given duration to execute their application. Resources are reserved exclusively to the application execution during this time. Intermediate results therefore can stay cached on computing resources until the expiration of the reservation. Thanks to this feature, a direct transfer mechanism can be implemented. Again, considering the simple workflow shown in figure 3.9a, the service S_1 is executed on a computing resource and generates an intermediate result $S_1(a)$, this result stays on the computing node and waits for further invocations. Instead of connecting to the central storage server, the service S_2 connects to the computing node of the service S_1 to copy its input data. After finishing the execution, the result of S_2 either stays on its computing node or is sent to the storage server as a final result of the application.

In figure 3.9, we propose two techniques to implement the direct transfer mechanism. The original workflow is diagrammed in figure 3.9a, the first technique is described in figure 3.9b and the second one is presented in figure 3.9c. As the input data of medical images analysis applications are usually image file and described through a description file, the first technique proposes indicating the location of the data as shown in figure 3.9b. An input data item is described in input file as `login@database:/filename`. Workflow services installed on computing nodes need to be modified to generate intermediate results using the same kind of identifiers, `login@computing_node:/result`. With this technique, the workflow description stays the same for any execution but the input data description must be modified to adapt to the change of data base server.

The second approach proposes changing the description of the workflow as presented in figure 3.9c. For each original input port of each workflow service, we add two input ports representing the login and host of the input data location. All outputs of this workflow service are generated from the same computing resource. Therefore, for each workflow service, we add two output ports describing the login and host of the current computing resource. However, this technique violates the philosophy of the workflow language. The first reason is that the workflow language describes only the application logic

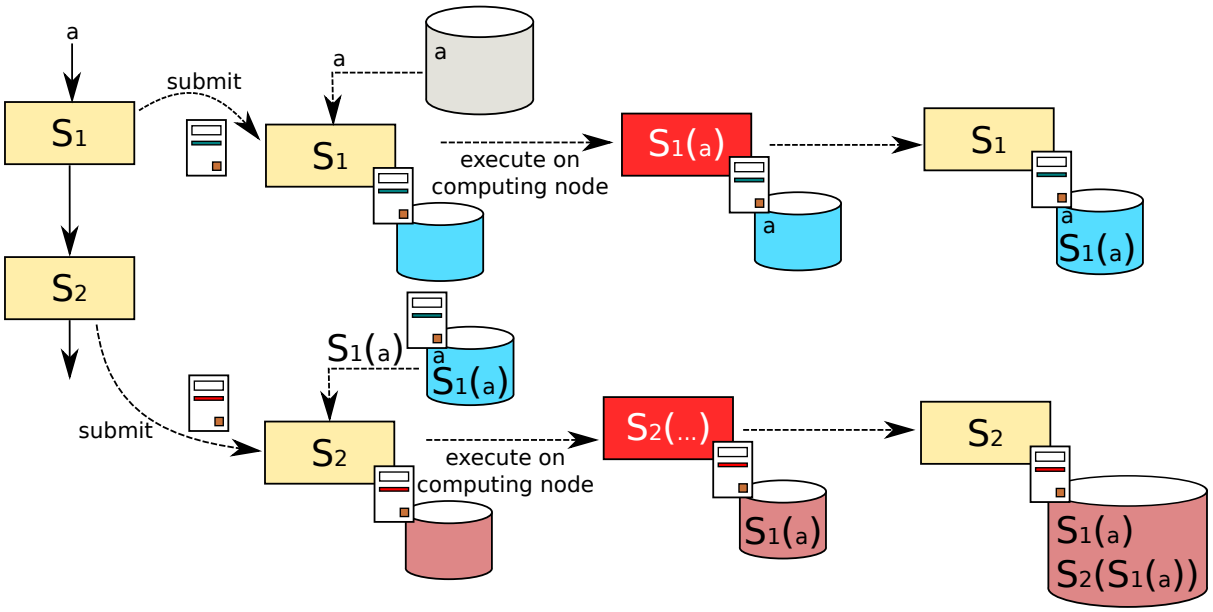


Figure 3.8: On systems supporting the advance reservation, intermediate results can be stored on computing nodes and transferred to other computing nodes directly.

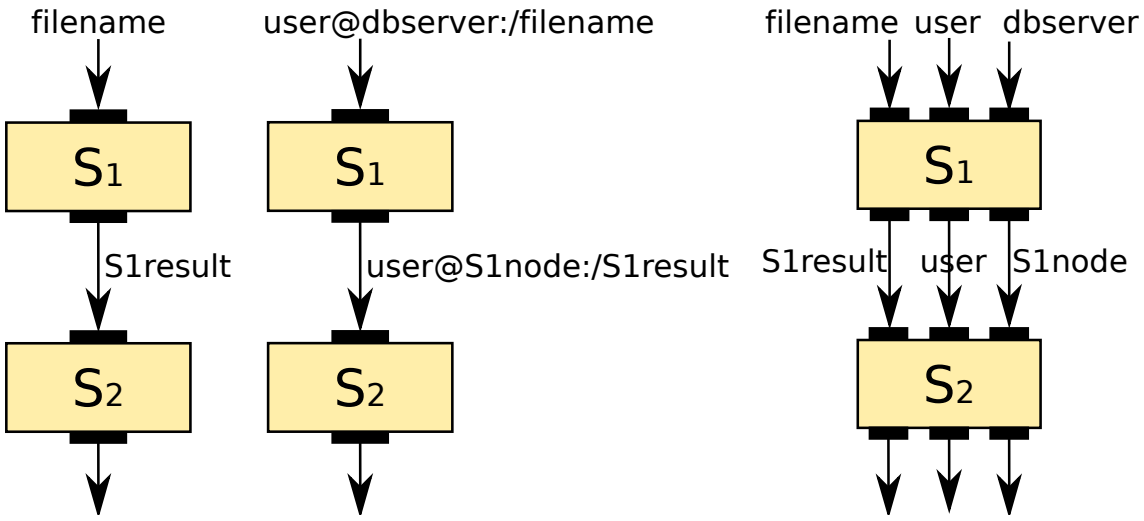
and independently from the underlying infrastructure, the detail on how to transfer the data is the responsibility of the workflow engine and underlying infrastructure. The second reason is that the addition of two input port representing the location of each input data port makes the workflow description more complex. Therefore, we use the first technique for the experimental evaluation in the next chapter.

3.6 Reliability support in cloud infrastructure

Networking and computing infrastructures are subject to random failures of nodes and links. These failures are not rare in the case where the number of physical entities are large, especially in distributed systems. The reliability of a system may be evaluated quantitatively and qualitatively. The Mean Time Between Failures (MTBF) is a statistical metric to determine the failure rate of the underlying infrastructure, which can be evaluated by the infrastructure management system. Already, the impact of a node failure to a distributed application can be different; a failed worker node amongst hundreds of others is less significant than the failure of a database server.

The virtualization technology enables the definition of confined execution environment, with a user-specified amount of virtual resources including computing and network resources. The virtualization also enables the specification of the reliability to be provided to different resources within the execution environment. The database server requires a higher level of reliability than a computing node. The reliability becomes a service provided by infrastructure providers. The higher the reliability level, the higher the cost users have to pay.

The goal of this section is to summarize the solution presented in [Koslovski et al., 2010] that we used to support the reliability for the execution environment (section 3.6.1). We then formulate the cost model to calculate the extra cost that users have to pay for reliability (section 3.6.2).



(a) Original workflow

(b) Modify the input data description and output of each workflow service

(c) Add two new input ports representing the user and hostname of data source

Figure 3.9: Improving the data transfer, direct transfer between computing resources

3.6.1 Providing transparent reliability

We summarize in this section the solution presented in [Koslovski et al., 2010] to provide transparent reliability. We used a live protection mechanism such as Remus [Cully et al., 2008] or Kemari [Tamura et al., 2008]. The main idea of Remus and Kemari is to continuously "synchronize" the memory state of a protected (critical) node with a replica (backup node), similarly to checkpointing. When a failure in the protected node occurs, the backup node can resume execution immediately, and the failover process can be made transparent to other nodes in the VPXI.

The key difference between Remus and Kemari is that Kemari initiates a checkpoint only when external events occur, such as disk writing and network-packet sending, whereas Remus checkpoints at a regular interval. One important feature of Remus is that, at every checkpoint, the external output is buffered locally in the critical node until it is assured that the backup node completes that checkpoint update. This ensures that any failover operation will be transparent to other unaffected nodes. Moreover, the protected node continues execution in parallel until the next checkpoint, thereby increasing system performance over classical lock-step checkpointing. Kemari, on the other hand, does not perform any buffering and relies on pausing the protected node to achieve the required transparency. We chose to use Remus over Kemari in our proof of concept as it provides a finer and customizable granularity between checkpoints, which can be as frequent as tens of milliseconds. As of Xen 4.0.0, Remus is included in the official Xen releases.

3.6.2 Extending the cost function model to reliability

Given a confined virtual execution environment with a user-specified amount of computing resources interconnected by a private overlay network, this section deals with the problem of determining the number of backup nodes and backup links to provide the reliability level required by users while optimizing the

execution cost.

Extending equation 3.1 and 3.2, if in each execution stage i, the number of backup nodes and links are mb_i and lb_i, respectively, the total execution cost for computing resources (including the extra cost for backup nodes) is:

$$C_r = c_r \times \sum_{i=1}^{s} (m_i + mb_i) \times (Td_i + T_i(m_i + mb_i, n, b + lb_i)) \tag{3.16}$$

and respectively for link bandwidth:

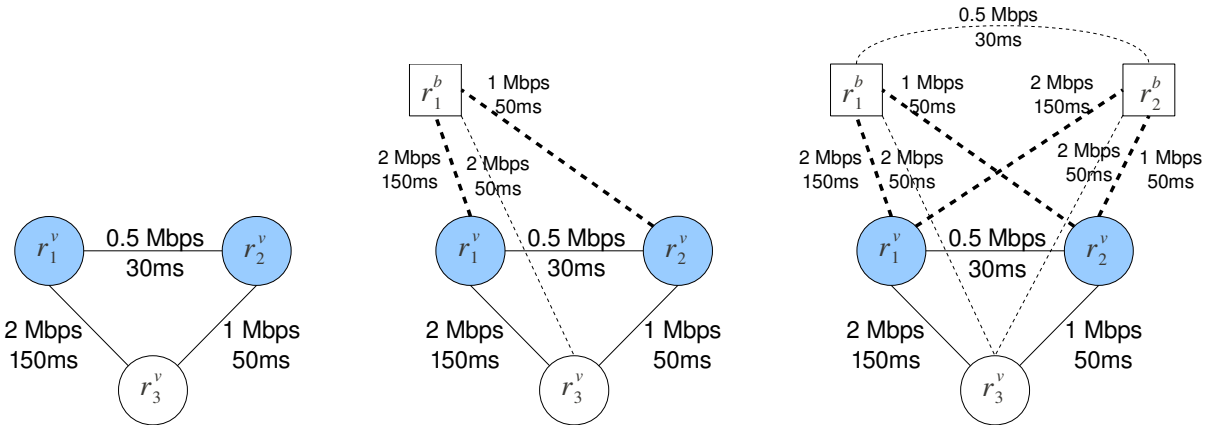
$$C_b = c_b \times \sum_{i=1}^{s} (Td_i + T_i(m_i + mb_i, n, b + lb_i)) \sum_{j=1}^{k_i+lb_i} b_{i,j} \tag{3.17}$$

The total execution cost is therefore:

$$C = C_r + C_b \tag{3.18}$$

The challenging problem is to determine the number of backup nodes and links at each stage of the execution. An optimized number of backup resources results in an optimized execution cost while the desired reliability can be achieved. A critical node with a low MTBF will require more backup nodes on standby (synchronized through Remus) than another node with a higher MTBF for the same reliability level, if physical failures are independent. As noted in [Yeow et al., 2010], backup nodes can be shared among different groups of critical nodes to minimize the total number of backup nodes (and hence, minimize the execution cost). For example, let us assume that stage i of the execution has two services needing n_1 and n_2 critical nodes. To assure the reliability level at least r_1 and r_2, these two services require k_1 and k_2 backup nodes, respectively. The higher the reliability level, the higher number of backup nodes needed. It is possible to share the backup nodes for $n_1 + n_2$ nodes such that the total number of backup nodes is lower than $k_1 + k_2$ provided that every backup node is a standby for all other critical nodes. In [Yeow et al., 2010], the Opportunistic Redundancy Pooling (ORP) mechanism imposes a sharing policy between groups of critical nodes such that it is possible to have $\min(k_1, k_2)$ backup nodes so long as the reliability of every service is satisfied. We use the ORP approach to evaluate the number of backup nodes required. Since ORP assumes independent physical failures, it also generates additional physical-embedding constraints such that the physical locations of all shared backup nodes and critical nodes validate that assumption. For example, virtual nodes may not be embedded onto the same physical host, or rack that is connected to the same switch, or power supply.

The backup nodes must be connected to the rest of VPXI through backup links which have to be pre-allocated. The higher the number of backup links reserved, the higher the reservation cost. Therefore, the number of backup links must be optimized. We used the approach presented in [Koslovski et al., 2010] to generate the backup links. Figure 3.10 shows an example of how backup links are generated. A new link between a backup node and some other node is created if it is a neighbor of a critical node. Hence, in figure 3.10b, node r_1^b connects to all three nodes. Furthermore, the attributes of link (r_1^b, r_3^v) can function as links (r_1^v, r_3^v) or (r_2^v, r_3^v). Links (r_1^b, r_1^v) and (r_1^b, r_2^v) are reused for synchronization. With one more backup node (as in figure 3.10c), the backup links between node r_2^b and the rest of VPXI (node



(a) Original VPXI (b) Add one backup (c) Add two backups

Figure 3.10: The figures show the steps (from left to right) as each backup node is added to the original VPXI for reliability. Nodes r_1^v and r_2^v are critical nodes, and nodes r_1^b and r_2^b are backup nodes. Backup links are reused for synchronization (in bold dotted lines). The non-bold dotted lines are backup links assuring the connection between backup nodes and other nodes which are neighbor of critical nodes. The respective attributes (bandwidth and latency) are determined by the existing links in the original VPXI specified by the user (image courtesy of [Koslovski et al., 2010]).

r_3^v) are the same as those of node r_1^b, and with a backup link between node r_1^b and r_2^b to function as the link (r_1^v, r_2^v) when both critical nodes fail.

3.7 Conclusions

In this chapter, the main contributions of this thesis were given. We described a cost model which allows users to scale their confined virtual execution environment. Based on the estimation of the application makespan, this cost model helps users finding the trade-off between the desired performance and the execution cost that they have to pay. Using this cost model, we enriched the MOTEUR workflow manager with four resource allocation strategies applied for workflow-based applications. The real execution of distributed applications is usually different from the estimation due to the variation of computational process execution time with input data or stochastic processes causing unforeseeable execution time, we proposed in this chapter our technique to handle this problem. Both users and infrastructure providers can benefit from these strategies. In the case the application execution time has been underestimated, users can continue executing their application if their extended resource reservation does not violate the constraints of other reservations in the infrastructure. Infrastructure providers can increase the performance of the infrastructure while minimizing the number of aborted applications.

Taking advantage of the virtualization, we extended the Infrastructure as a Service concept to reliability. Our approach allows both users to specify their desired reliability level, and infrastructure providers to transparently provide the reliable execution environment.

In the next chapter, we will present our validation of all proposals presented in this chapter. Experiments are conducted on the Aladdin/Grid'5000 testbed through HIPerNet framework.

Chapter 4

Evaluation experiments on the Aladdin/Grid'5000 infrastructure

This chapter provides experimental validation of the optimization techniques described in chapter 3. All experiments have been conducted on the Aladdin/Grid'5000 infrastructure, using the HIPerNet virtual platform management middleware. Before performing the experiments, the overview of the testbed infrastructure and tools is given. Following the evaluation of the impact of the virtualization, the four resources allocation strategies proposed are then evaluated. The experiments assess the performance of the optimized strategy with services grouping. Other experiments on data transfer improvement and the reliability support are also presented.

Ce chapitre présente la validation expérimentale des propositions décrites dans le chapitre 3. Toutes les expériences ont été menées sur l'infrastructure d'Aladdin/Grid'5000 à l'aide de l'intergiciel HIPerNet, un gestionnaire d'infrastructure virtuelle. Avant d'effectuer les expériences, l'infrastructure expérimentale et des outils impliqués dans les expériences sont introduits. Après l'évaluation de l'impact de la virtualisation, les quatre stratégies d'allocation de ressources sont ensuite évaluées. Les expériences démontrent la performance de la stratégie optimisée avec l'optimisation de groupement de services. D'autres expériences sur le transfert des données et l'amélioration de fiabilité de l'infrastructure sont également présentées.

4.1 Aladdin/Grid'5000, experimental infrastructure

Aladdin/Grid'5000[1] is a dedicated experimental infrastructure composed of 13 clusters distributed in 9 French cities. It was recently extended with one cluster in Brazil not used in the experiments presented in this chapter. Sites are linked with 1Gbps or 10Gbps connections dedicated to this platform. Within each cluster, nodes are located in the same geographic area and communicate through Gigabyte Ethernet links. Communications between clusters are made through the Renater French academic network.

The platform is fully configurable. It allows users to reserve part of the resources and deploy their own system and environment, with administrator privilege access. User defined system images and environment are deployed on reserved machines by the *kadeploy*[2] tool. Thanks to this feature, users can reserve a pool of physical machines and deploy the virtual machine monitor (*e.g.* Xen, KVM) on these machines to create the virtualized substrate. Bandwidth control is also supported on Aladdin/-Grid'5000 [Vicat-Blanc Primet et al., 2009a]. This allows users to specify the bandwidth needed for their applications.

Resources allocation on the Aladdin/Grid'5000 platform is made through OAR [Capit et al., 2005] which provides all the basic mechanisms of classical batch schedulers. Users can reserve "intenstive" resources for an immediate purpose if there are some available or use advance reservation to preserve for a future run.

4.2 HIPerNet virtualization middleware

HIPerNet provides a framework to build and manage private, dynamic, predictable, and large-scale virtual computing environments, that high-end challenging applications can use through traditional APIs such as standard POSIX calls, sockets, and Message Passing (*e.g.* MPI and OpenMP) communication libraries. With this framework, a user preempts and, for a given timeframe, virtually interconnects a pool of virtual resources from a distributed physical substrate, in order to execute her application. VPXIs (section 3.2.1) correspond to the HIPerNet management unit.

The HIPerNet framework aims at partitioning a distributed physical infrastructure (computers, disks, and networks) into dedicated virtual private computing environment composed dynamically. When a new machine joins the physical resource set, HIPerNet prepares its operating system to enable several virtual machines (VMs) to be instantiated dynamically when required. This set of potential virtual machines is called an HIPerSpace and it is represented in the HIPerSpace database. The HIPerSpace is the only entity that sees the physical entities. A resource, volunteer to join the pool of resources, is automatically initiated and registered in the HIPerSpace database. The discovery of all the devices of the physical node is also automatic. An image of the specific HIPerNet operating system is deployed on it. In our current HIPerNet implementation, the operating system image basically contains the Xen Hypervisor and its domain of administration called domain 0 (Dom 0). The HIPerSpace registrar (operational HIPerVisor) collects and stores data persistently, and manages accounts (*e.g.* the authentication database). It is therefore hosted by a physical machine outside of the HIPerSpace itself. For the sake of

[1] https://grid5000.fr
[2] http://kadeploy.imag.fr/

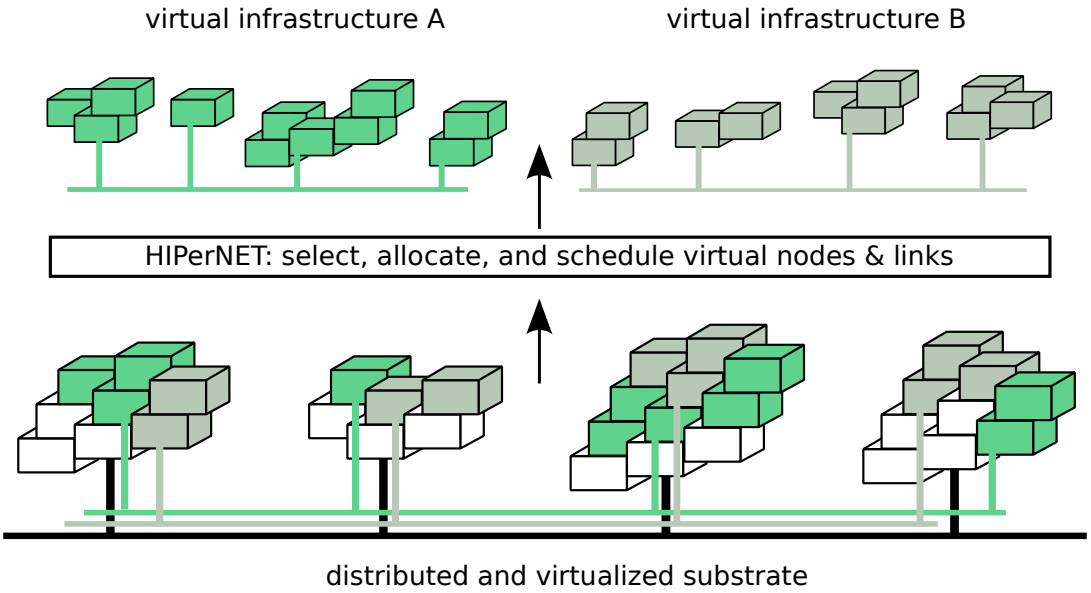


Figure 4.1: Example of a VPXI allocation on a distributed and virtualized HIPerSpace.

robustness and scalability, the HIPerSpace registrar can be replicated or even distributed.

When a user submits a VPXI request specified using the VXDL language, the HIPerNet allocator examines the request and executes an embedding algorithm to map the virtual infrastructure on the physical one. Then if the request is accepted, HIPerNet deploys or reconfigures the virtual resources of the VPXI according to this specification. Using the bandwidth control concept in Grid'5000 [Vicat-Blanc Primet et al., 2009a], HIPerNet allocates links bandwidth to all the virtual links whose bandwidth was explicitly specified in VXDL during the request submission. When a VPXI is created, virtual links are provisioned according to the VXDL request. Within this request, the user can specify several stages for the VPXI, involving different configurations of bandwidth to best fit the application requirements. While the VPXI is running, the user can change its configuration moving from one stage to another.

Figure 4.1 presents an example of two virtual execution infrastructures (VPXI A and VPXI B) managed by the HIPerNet framework. Each application can execute, confined in a VPXI dedicated for a defined time period.

For all experiments presented below, a system image was created, which contains the operating system – based on a standard Debian *Etch* Linux distribution with a kernel version 2.6.18-8 for *AMD64* –, the domain-specific image processing services, and the middleware components (the MOTEUR workflow engine and DIET middleware).

4.3 Infrastructure description using VXDL

4.3.1 VXDL language

The VPXI of each experiment is described through the *Virtual eXecution Description Language* (VXDL) [Koslovski et al., 2008]. VXDL is an XML-based language which allows users to describe not only the end resources, but also the virtual network topology, including virtual routers and timeline

representation. The VXDL grammar is divided into *Virtual Resources*, *Virtual Network Topology*, and *Virtual Timeline* description as described below. Note that these descriptions are partially optional: it is possible to specify a simple communication infrastructure (a virtual private overlay network) or a simple aggregate of end resources without any network topology description (a virtual cluster or grid).

Virtual Resources Description. This part of VXDL grammar enables users and applications to describe, in a simple and abstract way, all the required end hosts and host groups. VXDL allows the basic resource parametrization (*e.g.* minimum and maximum acceptable values for RAM and CPU frequency). An important feature of VXDL is that it proposes cross-layer parameters. With the specification of *anchor* and *the number of virtual machines allocated per physical host* users can directly interact with lower layers and transmit application-specific information. The *anchor* parameters corresponds to a physical allocation constraint of a VPXI. Indeed, in theory a VPXI can be allocated anywhere in a virtualized substrate, but sometimes it is desirable that a virtual end host (or group) be positioned in a given physical location (*e.g.* a site or a machine - URL, IP) for an application-specific reason. On the other hand, in a virtualized substrate, multiple virtual machines can be allocated in the same physical host, sharing the real resources. VXDL enables the definition of a maximum number of virtual machines that must be allocated in a physical host, enabling users to interact directly with the allocation algorithm.

Virtual Network Topology Description. VXDL brings two original aspects within the network topology description: (i) the joined specification of network elements and computing elements and (ii) the link-organization concept, which permits a simple and abstract description of complex structures. Links can define connections between end hosts, between end hosts and groups, inside groups, between groups and VXrouters, and between VXrouters. In VXDL grammar, the definition of *source - destination pairs* for each link is proposed. The same link definition can be applied to different pairs, simplifying the specification of complex infrastructures. For example, links used to interconnect all components of an homogeneous group, as a cluster, can all be defined in a same link description. Each link can be defined by attributes such as latency, bandwidth, and direction. Latency and bandwidth can be defined by the maximum and minimum values.

Virtual Timeline Description. Any VPXI can be permanent, semi-permanent, or temporary. The VPXI are allocated for a defined lifetime in time slots. Time slots duration is specific to the substrate-management framework and consequently this parameter is configured by the manager of the environment. Often the VPXI components are not used simultaneously or all along the VPXI lifetime. Thus, the specification of an internal timeline for each VPXI can help optimizing the allocation, scheduling, and provisioning processes. Periods can be delimited by temporal marks. A period can be activated after the end of another period.

4.3.2 Translation of VPXI into VXDL

For each experiment presented in this chapter, its VPXI is described through VXDL. This VPXI is composed of two parts: a *generic* part and *variable* one. The *generic* part is used to describe mandatory nodes to execute an application (*e.g.* middleware, database). Figure 4.2 presents the description of

```
<vxdl:vNode id=''database''>
    <vxdl:memory>
        <vxdl:simple>512</vxdl:simple>
        <vxdl:unit>MB</vxdl:unit>
    </vxdl:memory>
</vxdl:vNode>
<vxdl:vNode id=''MOTEUR''>
    <vxdl:memory>
        <vxdl:simple>512</vxdl:simple>
        <vxdl:unit>MB</vxdl:unit>
    </vxdl:memory>
</vxdl:vNode>
<vxdl:vNode id=''DIET''>
    <vxdl:memory>
        <vxdl:simple>512</vxdl:simple>
        <vxdl:unit>MB</vxdl:unit>
    </vxdl:memory>
</vxdl:vNode>
```

Figure 4.2: Generic part description in VXDL language

a database node. The *variable* part composing of computing resources is generated according to the design strategies.

For example, the *naive* strategy presented in section 3.3 divides the set of $m_{\max}$ virtual computing resources proportionally to the execution time of workflow services. We use the `<vxdl:group>` tag to describe a cluster of virtual computing resources corresponding to a workflow service. This cluster composes of $m_i = m_{\max} t_i / \sum_j t_j$ resources with a minimum amount of RAM. Figure 4.3 presents the description of this strategy.

Similarly to the *naive* strategy, the *FIFO* strategy runs the application in a single stage and assumes that all services can be deployed on every computing resource. Therefore, the VXDL description has only one group.

The *optimized* strategy has a more complex description which uses the *Virtual Timeline Description* of VXDL language. Figure 4.4 presents an example of the application which has two stages. The first stage has one service which executes in t_i seconds. The second stage has three services starting at the same time after first stage has finished.

The virtual network topology is specified by depending on each application. The more dependence between workflow services, the more complicated network topology. Each link is specified by a minimum amount of bandwidth and one or more pairs of source/destination. Figure 4.5 shows a typical link between the database storing the workflow input and the computing resource cluster of a workflow service.

VXDL has been extended to enable the specification of reliable virtual infrastructures. The extension proposes the identification of the required reliability level for each virtual resource (nodes and links). Users can set the requirement individually or for a group of resources. To illustrate the flexibility of the

```
<vxdl:vGroup id=''workers'' multiplicity=''m_i''>
    <vxdl:vNode id=''worker''>
      <vxdl:memory>
          <vxdl:simple>512</vxdl:simple>
          <vxdl:unit>MB</vxdl:unit>
      </vxdl:memory>
      <vxdl:cpu>
          <vxdl:cores>1</vxdl:cores>
          <vxdl:frequency>
              <vxdl:simple>1.0</vxdl:simple>
              <vxdl:unit>GHz</vxdl:unit>
          </vxdl:frequency>
      </vxdl:cpu>
    </vxdl:vNode>
</vxdl:vGroup>
```

Figure 4.3: VXDL description of one cluster of computing resources

```
<vxdl:virtualTimeline>
   <vxdl:id>ApplicationTimeline</vxdl:id>
   <vxdl:timeline>
      <vxdl:id>T1</vxdl:id>
      <vxdl:activate>Service_i</vxdl:activate>
      <vxdl:until>
          <vxdl:totalTime>ti</vxdl:totalTime>
          <vxdl:unit>s</vxdl:unit>
      </vxdl:until>
   </vxdl:timeline>
   <vxdl:timeline>
      <vxdl:id>T2</vxdl:id>
      <vxdl:after>T1</vxdl:after>
      <vxdl:activate>Service_j1</vxdl:activate>
      <vxdl:activate>Service_j2</vxdl:activate>
      <vxdl:activate>Service_j3</vxdl:activate>
      <vxdl:until>
          <vxdl:totalTime>tj</vxdl:totalTime>
          <vxdl:unit>s</vxdl:unit>
      </vxdl:until>
    </vxdl:timeline>
</vxdl:virtualTimeline>
```

Figure 4.4: Virtual Timeline Description for *optimized* strategy

```
<vxdl:virtualTopology>
  <vxdl:id>VirtualNetwork</vxdl:id>
  <vxdl:link>
    <vxdl:id>lv 1</vxdl:id>
    <vxdl:bandwidth>
      <vxdl:min>2</vxdl:min>
      <vxdl:unit>Mbps</vxdl:unit>
    </vxdl:bandwidth>
    <vxdl:direction>bi</vxdl:direction>
    <vxdl:pair>
      <vxdl:source>database</vxdl:source>
      <vxdl:destination>cluster_i</vxdl:destination>
    </vxdl:pair>
  </vxdl:link>
</vxdl:virtualTopology>
```

Figure 4.5: Virtual network topology description

specification language, the example below is part of a VXDL file and describes a group of 30 virtual nodes with a reliability specification of 99.9% (among others parameters).

4.4 Medical image analysis application use case

The experiments presented in this chapter are performed using the *Bronze Standard* (BS) a real workflow-based application from the area of medical image analysis [Glatard et al., 2006b] introduced in section 2.2.5.

The Bronze Standard application exhibits the complex workflow illustrated in figure 4.7. On this figure, each rectangle represents an application service to be executed for every image in the tested database. There are six services in the Bronze Standard application: `CrestLines`, `CrestMatch`, `PFMatchICP`, `Yasmina`, `Baladin`, `PFRegister`. The arrows between services represent data dependencies (the output of a service is piped into the input of the following one). In addition to the computational services, the diagram represents data sources (`Floating` and `Reference` representing two input image sets to be registered) and sinks (output collectors `Results`). `CL_size`, `PFMOpt`, `YasminaOpt` and `BaladinOpt` are constant parameter of workflow services. $\oplus$ and $\otimes$ represent two iteration strategies, the dot and cross product, respectively. The *dot* product make the service fire once for each pair of input image. It corresponds to a traditional *one-to-one* execution semantic. For instance, if `Floating` has n images and `Reference` has m images, `CrestLines` will fire $\min(n, m)$ times. The *cross* product corresponds to an *all-to-all* execution semantic. In the Bronze Standard application, each service has an option parameter. This parameter is used for all input image pairs. Thus, a *cross* product is used for them. The graph represents the flow of processings, described in the GWENDIA language. The data to process is described independently. The workflow is interpreted and enacted by the MOTEUR workflow engine [Glatard et al., 2008b]. From the application graph of processings and given input data sets, the engine dynamically determines the data flows to be processed. Using the

```
<vxdl:vGroup id=''workers'' multiplicity=''30''>
    <vxdl:vNode id=''worker''>
    <vxdl:reliability >99.9%</vxdl:reliability >
    <vxdl:memory>
        <vxdl:simple >512</vxdl:simple>
        <vxdl:unit>MB</vxdl:unit>
     </vxdl:memory>
    <vxdl:cpu>
        <vxdl:cores >1</vxdl:cores>
        <vxdl:frequency>
            <vxdl:simple >1.0</vxdl:simple>
            <vxdl:unit>GHz</vxdl:unit>
         </vxdl:frequency>
      </vxdl:cpu>
   </vxdl:vNode>
</vxdl:vGroup>
```

Figure 4.6: Specification of the reliability level in VXDL

Services	Time (seconds)	Input data	Produced data
CrestLines	31.06 ± 0.57	15MB	10MB
CrestMatch	3.22 ± 0.51	25MB	4MB
PFMatchICP	10.14 ± 2.41	10.2MB	240kB
PFRegister	0.64 ± 0.22	240kB	160kB
Yasmina	52.94 ± 12.96	15.2MB	4MB
Baladin	226.18 ± 19.36	15.2MB	4MB

Table 4.1: Benchmark of the Bronze Standard services execution time and data transfer volumes.

rich semantics of the data composition operators discussed in chapter 2, section 2.2, this results in the production of a very large number of computation tasks, many of which can be executed in parallel although some dependencies have to be taken into account in the scheduling. To exploit the parallelism, three parallelism levels discussed in the section 2.1.2 are enabled in the MOTEUR workflow engine.

4.5 Experiment runs condition

In the experiments reported below, unless explicit mention of another database size, a clinical database with 59 pairs of patient images was used for the execution of the Bronze Standard application. For each run, 354 computing tasks were generated.

As a baseline, the execution time and the data volume transferred for each Bronze Standard processor have been measured in initial microbenchmarks out of the virtualized system. These benchmarks were also used for the needs of the MOTEUR workflow engine in the VPXI design step. The results are reported in table 4.1. It can be seen that the algorithm execution time is rather reproducible (with a standard deviation in the order of 1-5% of the average value).

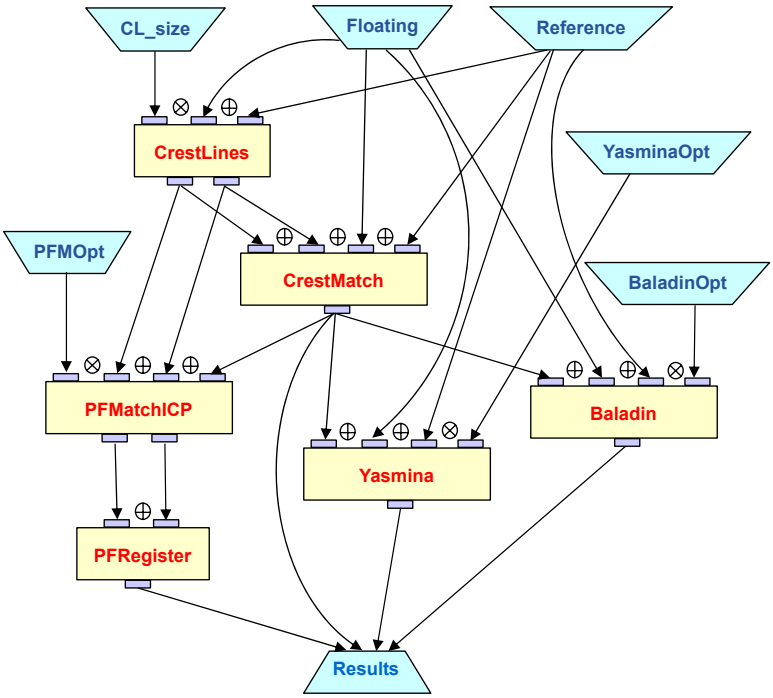


Figure 4.7: Bronze Standard workflow.

In all experiments, the MOTEUR workflow engine [Glatard et al., 2008b] is used for interpreting the workflow description which is in the format of the GWENDIA language. It is responsible for handling the composition strategies in each workflow service. It enables three parallelism levels presented in section 2.1.2 (workflow, data and service parallelism). It implements several interfaces to large-scale infrastructures (EGI[3] through gLite middleware[4], Aladdin/-Grid'5000 [Cappello et al., 2005] through the OAR middleware [Capit et al., 2005] and GridRPC through the DIET middleware [Caron and Desprez, 2006]).

The DIET middleware is a scalable grid scheduler based on a hierarchy of agents communicating through CORBA. The leaves of the hierarchy are called Service Daemons (SeDs) which can offer any number of application specific computational service. Thus, the number of computing resources for each workflow service can be controlled by pre-installing the computational algorithm of workflow service on any number of SeDs. The MOTEUR workflow engine submits a workflow task to a DIET Master Agent which is responsible for finding a suitable SeD to solve this task. If the number of tasks is larger than the number of SeDs, the *Round-Robin* scheduling policy is applied. If there is more than one task arriving on each SeD, the *First Come First Served* policy is applied. In the first experiments (for the measurement of the virtualization impact and data improvement, sections 4.6 and 4.7, respectively), the DIET middleware was used. In the latter experiments (validation of the four resources allocation strategies and reliability support, sections 4.8 and 4.10, respectively), we enriched the MOTEUR workflow engine with a scheduling module to replace the DIET middleware. Details on the characteristics of physical resources reserved on the Aladdin/Grid'5000 platform are given for each experiment.

[3] http://www.egi.eu/
[4] http://glite.web.cern.ch/glite/

It is to be noted that, all experiments reported below are subject to variation due to concurrent usage on the Aladdin/Grid'5000 infrastructure by other users. Especially for network resources, the experiments performed on two remote interconnected clusters are more impacted by external network traffics than a on local clusters.

4.6 Infrastructure virtualization impact measurement

System and network virtualization has an impact on applications performance. The precise topology of the Virtual Private eXecution Infrastructure (VPXI) allocated causes some additional performance variations. In this section, an evaluation of the impact of virtualization on application performance is given.

Let us consider a request for a VPXI composed of 35 nodes to execute the Bronze Standard application. Three nodes are dedicated to the middleware services: 1 node for the MOTEUR workflow engine, 1 node for the database server and 1 node for the DIET middleware. The 32 nodes left are dedicated to application services. The *naive* strategy is considered in this experiment: the computing nodes are allocated proportionally to the execution time of the workflow processors: 3 nodes for `CrestLines`, 1 node for `CrestMatch`, 1 node for `PFMatchIP`, 1 node for `PFRegister`, 22 nodes for `Baladin`, and 4 nodes for `Yasmina`. Figure 4.8 represents the distribution of virtual resources for application services and administrative tools (MOTEUR, DIET and database). The `CrestMatch` node is data-sensitive. It requires larger data transfers than other services. Thus the links between `CrestMatch` and `PFMatchIP`, `PFRegister`, `Baladin` need more network resources.

For this same computing resource set, several variants of VPXI descriptions with different network topologies can be expressed. In **VPXI 1** the network is composed by two links type, one with low latency (0.2ms, *e.g.* intra cluster) and the other one with a maximum latency of 10ms (*e.g.* remote clusters interconnection). In **VPXI 2** the network comprises three virtual links: one with a low intra-cluster latency (maximum latency of 0.2ms), another one with a latency of 10ms interconnecting the components except one asking for a maximum latency of 0.2ms to interconnect CrestMatch (dark blue) with the components PFMatchICP, Yasmina and Baladin (blue) which require larger data transfers than other services.

Let us now illustrate how each VPXI description can be embedded in a physical substrate. Depending on the infrastructure availability, the VPXI specification may be mapped differently on the available resources. We propose two different solutions for each VPXI, resulting in four different physical allocations as represented in figure 4.9. In this example, *Site 1* and *Site 2* represent two geographically distributed resources sets:

- In **VPXI 1 - Allocation I** in this allocation one virtual machine is hosted by a physical node.
- In **VPXI 1 - Allocation II** each physical node in clusters CrestMatch, PFRegister, Yasmina, and Baladin are allocated 2 virtual machines.
- **VPXI 2 - Allocation III** respects the required interconnection allocating corresponding resources in the same physical set of resources (such as a site in a grid). This embedding solution explores the allocation of 1 virtual machine per physical node.

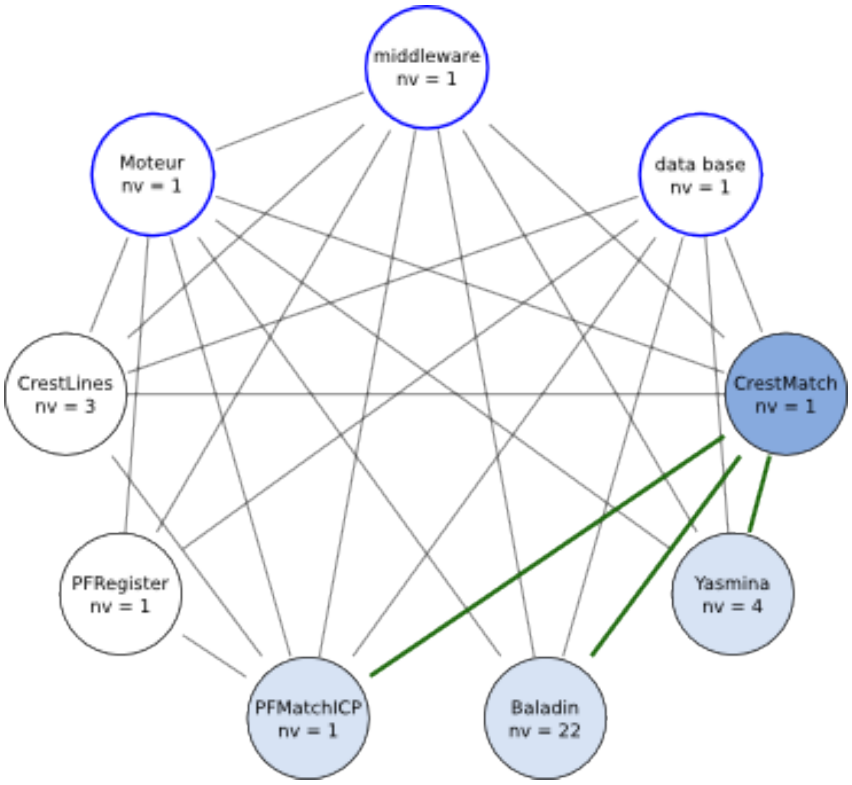


Figure 4.8: VPXI description of the Bronze Standard application as generated by the *naive* allocation strategy.

- **VPXI 2 - Allocation IV** explores the same physical components as *Allocation III* but allocates 2 virtual machines per physical node in the CrestMatch, PFRegister, Yasmina, and Baladin clusters.

The physical infrastructure for these experiments is reserved on the Grid'5000 clusters: *capricorne* (Lyon), *bordemer* (Bordeaux) and *azur* (Sophia), the CPUs of which are 2.0 GHz dual-cores Opterons. The distance between clusters is approximately 500km and they are connected through 10 Gbps links. For the experiments reported in this section, we used a clinical database of 32 pairs of images. In each experiment, we repeated the application 10 times to measure the average and standard deviation of the application makespan, the data transfer, and the task execution time. Each VPXI is composed of 35 nodes divided in *generic* and *variable* parts: 3 nodes are dedicated to the *generic* part (MOTEUR, DIET, file server) using 1 CPU per node, and the remaining 32 nodes of the *variable* part are allocated dependently on the VPXIs (*VPXI 1 - Allocation I* and *VPXI 2 - Allocation III* used 1 CPU per node while *VPXI 1 - Allocation II* and *VPXI 2 - Allocation IV* used 1 CPU core per node).

Co-allocating resources on one grid site: the application makespan on the *VPXI 2 - Allocation III* and *VPXI 2 - Allocation IV* is 11min 44s ($\pm$49s) and 12min 3s ($\pm$50s) respectively. This corresponds to a +3.8% makespan increase, due to the execution overhead when there are two virtual machines collocated on the same physical resource. Indeed, we present in table 4.2 the average execution time of application services on the *VPXI 2 - Allocations III and IV*. We can observe that the average execution overhead is 5.17% (10.53% in the worst case and 1.28% in the best case).

Resources distributed over 2 sites: when porting the application from a local infrastructure to a large scale infrastructure, the data transfer increases. Table 4.3 presents the data transfer time (s)

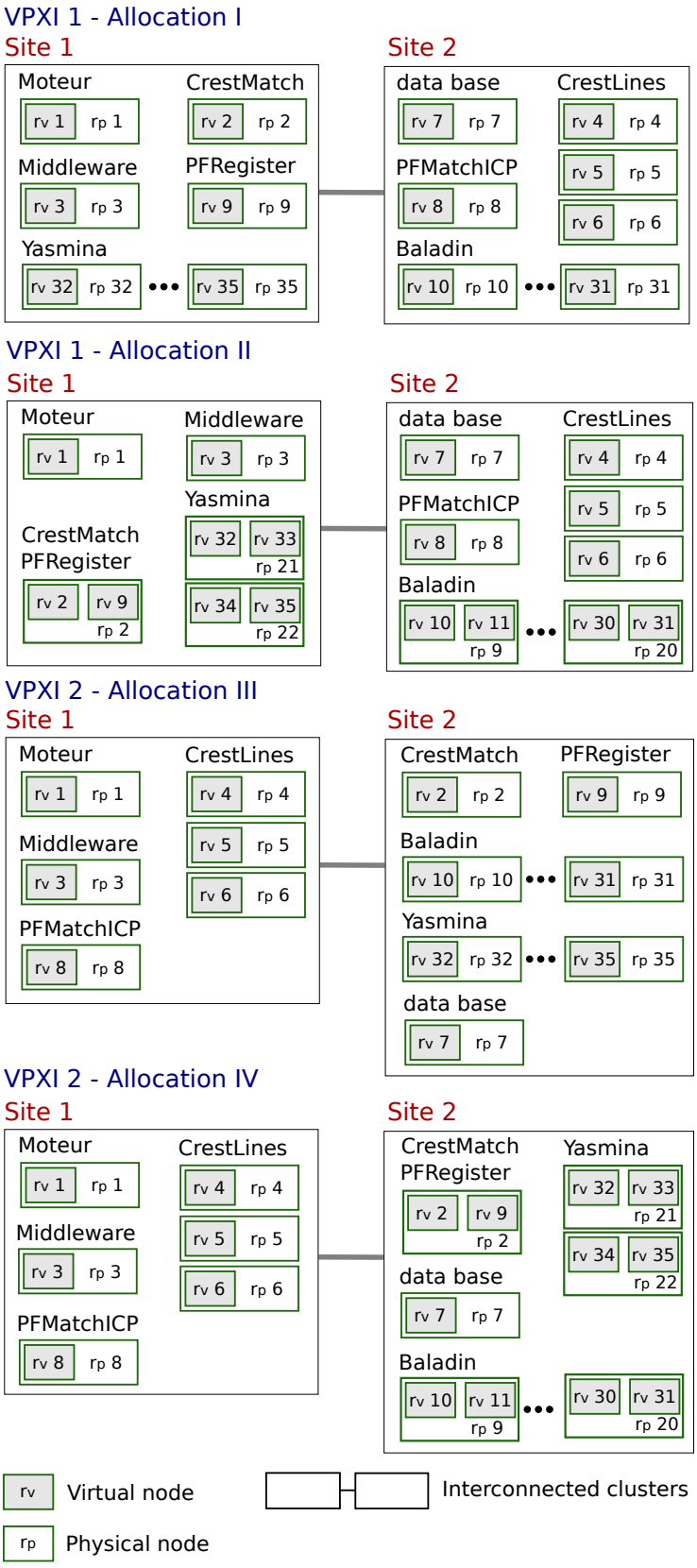


Figure 4.9: Allocations of descriptions **VPXI-1** and **VPXI-2**.

Services	Allocation III	Allocation IV	Variation
CrestLines	34.12 ± 0.34	36.84 ± 5.78	+7.97%
CrestMatch	3.61 ± 0.48	3.99 ± 0.63	+10.53%
PFMatchICP	11.93 ± 2.76	12.75 ± 5.35	+6.87%
PFRegister	0.78 ± 0.18	0.79 ± 0.18	+1.28%
Yasmina	59.72 ± 14.08	61.53 ± 13.98	+3.03%
Baladin	244.68 ± 16.68	247.99 ± 19.51	+1.35%

Table 4.2: Execution time on VPXI 2 - Allocations III and IV.

of the application services on *VPXI 2 - Allocation IV* (local) and *VPXI 1 - Allocation II* (distributed over 2 sites). The measured overhead is 150% in the worst case. Conversely, some local transfers may be slightly reduced. In the case of our application however, this overhead has little impact on the application makespan since it is compensated for by the parallel data transfer and computations. Indeed, the makespan is 12min (±12s) and 12min 11s (±20s) on *VPXI 1 - Allocation I* and *VPXI 1 - Allocation II* respectively, very similar to the performance of *VPXI 2 - Allocation IV*.

Services	Allocation IV	Allocation II	Variation
CrestLines	2 ± 0.45	3.01 ± 1.6	+50.5%
CrestMatch	1.99 ± 0.34	1.83 ± 0.36	-8.04%
PFMatchICP	1.3 ± 0.4	3.25 ± 0.13	+150%
PFRegister	0.51 ± 0.23	0.43 ± 0.09	-15.69%
Yasmina	1.19 ± 0.27	1.16 ± 0.21	-2.52%
Baladin	1.17 ± 0.38	1.81 ± 1.03	+54.7%

Table 4.3: Data-transfer time on the local VPXI 2 - Allocation IV and large scale VPXI 1 - Allocation II infrastructures.

Resources distributed over 3 sites: further distributing computational resources causes an additional increase of the data-transfer overheads. An additional experiment with *VPXI 1 - Allocation II* the *generic* part of which is located in Lyon while the *variable* part is randomly distributed in Lyon, Bordeaux and Sophia leads to a makespan of 12min 13s (± 30s) with a data-transfer overhead of 176% in the worst case.

Conclusions: although system virtualization has a performance impact on the application, the execution time variation is small (within 10%) and acceptable when considering the other advantages of system virtualization for most non time-critical applications. The data transfer time over virtual link might impact performance much more however, and virtual network links deployment should be thought carefully.

4.7 Data transfer improvement evaluation

In this section, we present experiments to evaluate the improvement of data transfer mechanism described in section 3.5. The impact of the data transfer mechanism was evaluated on both physical and

Services	Batch mode	Direct mode	Variation
CrestLines	13.817 ± 2.349	9.59 ± 0.25	-30.59%
CrestMatch	17.263 ± 3.490	9.18 ± 1.96	-46.82%
PFMatchICP	2.875 ± 2.406	1.32 ± 0.53	-54.09%
PFRegister	0.649 ± 0.164	0.72 ± 0.12	+10.94%
Yasmina	4.172 ± 3.057	3.37 ± 2.19	-19.22%
Baladin	4.689 ± 3.688	3.19 ± 2.00	-31.97%

Table 4.4: Data transfer time of six services in the Bronze Standard application on the physical infrastructure, comparing batch mode and direct mode.

virtual infrastructure. On each infrastructure, two kinds of experiment were conducted:

- run the application in batch mode using a central database server to store the intermediate results);
- run the application in *direct* mode, intermediate results being transferred directly between computing nodes.

The virtual infrastructure was created by specifying a VPXI. We reserved 16 physical machines on the *capricorne* Aladdin/Grid'5000 cluster in Lyon, France. The reserved nodes are IBM e325 workstation with AMD Opteron 246, 2 cores × 2.0GHz, 2.0GB RAM. Each physical machine hosts two virtual nodes. One virtual node was deployed per core. Each virtual node has 512 MB of memory. Therefore, a VPXI whose *variable* part composes of 32 virtual computing nodes is created. The *FIFO* strategy was considered in this experiment. All computing nodes are able to process any workflow service. The *generic* part of the VPXI was hosted on three physical machines including DIET, MOTEUR and the database server similarly to previous experiments.

The matching physical infrastructure is deployed on the same cluster. We reserved the same number of physical resources as in the virtual infrastructure and we submitted up to two processes to each computing node concurrently (thus equally making use of each CPU two cores). For all experiments, a clinical database of 32 pairs of images was used (192 tasks were submitted to the infrastructure).

In table 4.4, we present the data transfer time the six services involved in the Bronze Standard application. The results show that the data transfer time with the direct mode is significantly reduced compared to the batch mode. The best case, the PFMatchICP service, achieves a decrease in data transfer time by over 54.09%.

Moving to the virtual infrastructure, the data transfer improvement is even higher. Table 4.5 presents the data transfer time on the virtual infrastructure when executing the application with two batch mode and direct mode. The best case reduces the data transfer time by 68.01%. However, the results in this table show that the network bandwidth sharing on virtual infrastructure is less fair. The standard deviation of some workflow services is very high (*e.g.* CrestMatch, Yasmina, Baladin). The control bandwidth mechanism is therefore very useful to fairly share network among services, thus helping in controlling their execution time. In the next section, we present the experiments dealing with the resources allocation including computing and network resources.

Services	Batch mode	Direct mode	Variation
CrestLines	31.140 ± 4.366	22.37 ± 0.74	-28.16%
CrestMatch	46.237 ± 7.918	20.07 ± 3.67	-56.59%
PFMatchICP	5.316 ± 5.784	1.70 ± 1.17	-68.02%
PFRegister	0.620 ± 0.057	0.63 ± 0.05	+1.60%
Yasmina	8.158 ± 8.483	7.00 ± 6.94	-14.19%
Baladin	8.368 ± 8.363	7.47 ± 6.85	-10.73%

Table 4.5: Data transfer time of six services in the Bronze Standard application on the virtual infrastructure, comparing batch mode and direct mode.

4.8 Virtual resources allocation strategies evaluation

In this section, we present our experiments for the evaluation of four resources allocation strategies presented in section 3.3. The experimental infrastructure is diagrammed in figure 4.10. For all experiments, 36 physical computers were reserved. The MOTEUR workflow engine, as a client of the HIPerNet engine, was hosted on one physical host, outside of the virtual infrastructure. The 35 remaining computers were registered in the HIPerSpace. The HIPerNet engine deploys and manages virtual machines on these computer on demand (dark arrows), to host the input database server (1 node) or the application services (34 remaining nodes). In our experiments, each physical computer hosts a single virtual machine. MOTEUR produces VXDL descriptions that are required by the HIPerNet engine (blue connection). After receiving all virtual machines allocated to the VPXI, MOTEUR connects to the computing nodes to invoke the application services (red connections). The computing nodes connect to the database host to copy the input data, intermediate results, and send the final results to MOTEUR (green connections). For each experiment, the application was executed 5 times and the makespan was averaged to minimize the execution time variations encountered in distributed computing. The standard deviation is also reported.

For each strategy, the planner optimizer was executed to determine the configuration with the minimal execution cost. The number of virtual machines allocated to the application and the bandwidth between the database node and computing nodes are specified by corresponding VXDL documents.

The *naive* and *FIFO* strategies are single-stage. They use all available computing resources (34 computing machines) with an optimal bandwidth yielding to a minimal execution cost. Conversely, the *optimized* strategies are multi-stages, optimize bandwidth needed, and may allocate less resources than the maximum available when there is no gain in doing so.

We also measured the deployment time of the virtual infrastructure before running the application and the reconfiguration time between stages of the *optimized* strategies. The reconfiguration time takes into account bandwidth reconfiguration between the database host and computing nodes allocated to application services in each stage. The virtual machines in stage n are reused in stage $n + 1$. If the stage $n + 1$ uses more virtual machines than stage n, additional virtual machines are deployed during the execution of stage n.

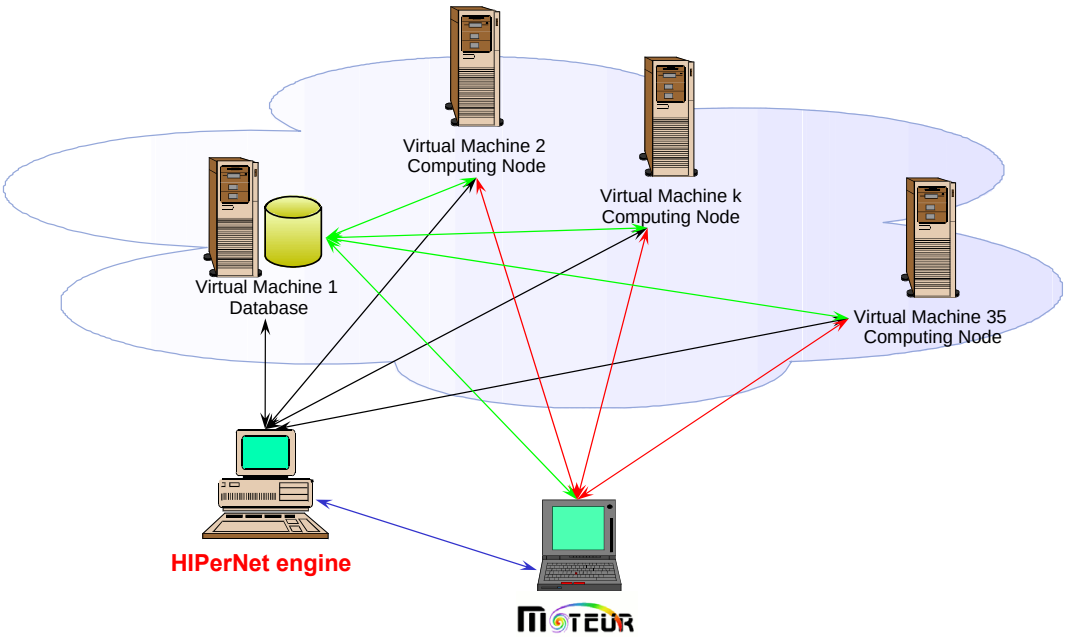


Figure 4.10: Experimental infrastructure

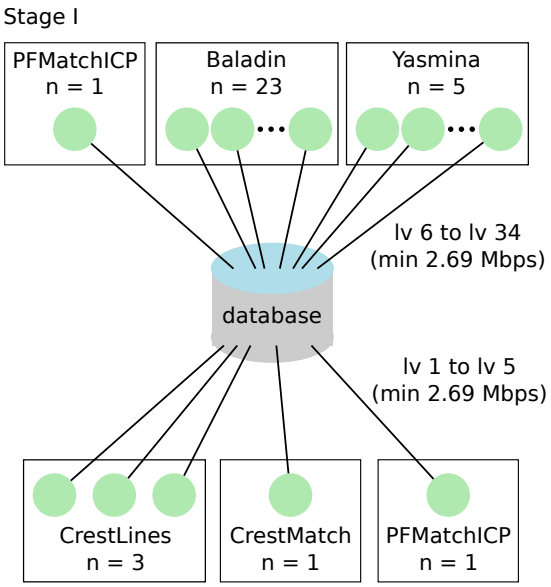


Figure 4.11: Virtual infrastructure composition considering the *naive* strategy.

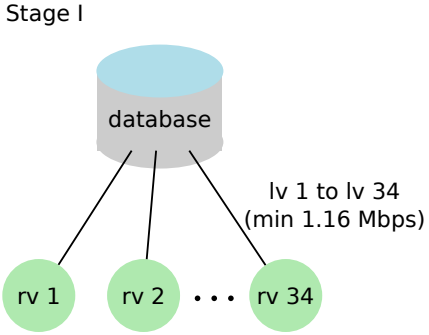


Figure 4.12: Virtual infrastructure composition considering the *FIFO* strategy.

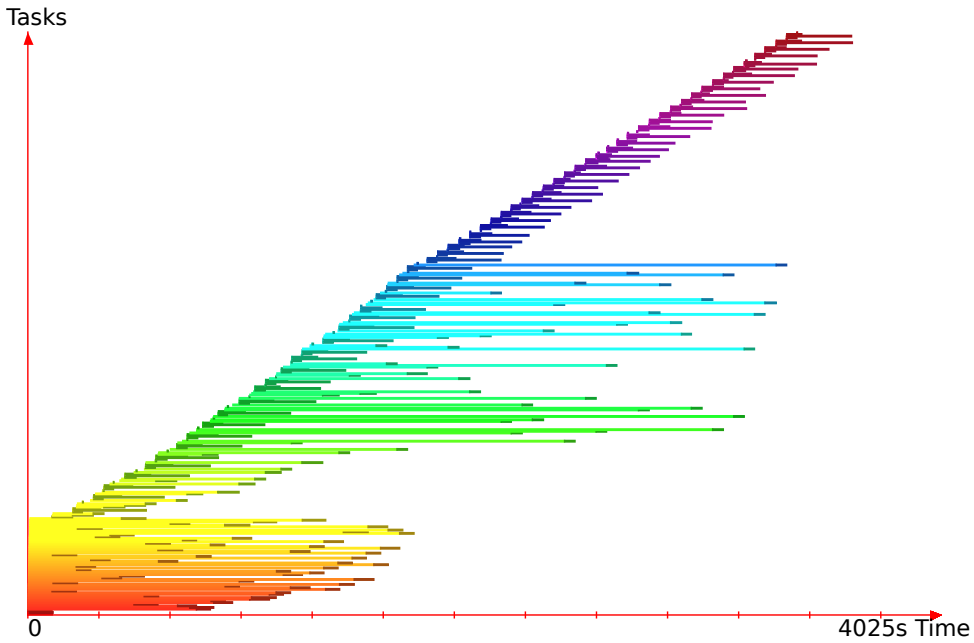


Figure 4.13: Tasks schedule with the *naive* strategy

4.8.1 Single stage strategies

The virtual infrastructures of the *naive* and *FIFO* strategies are represented in figures 4.11 and 4.12, respectively. The *naive* allocation strategy allocated the 34 computing nodes to application services as follows: 3 nodes for CrestLines, 1 node for CrestMatch, 1 node for PFMatchICP, 1 node for PFRegister, 5 nodes for Yasmina, and 23 nodes for Baladin. The same bandwidth, 2.69Mbps, is used for all computing nodes. The application makespan is 67.08min $\pm$ 0.10min. This experiment shows that the virtual resources are not well exploited during the execution. Figure 4.13 shows a schedule of this strategy. Each colored line represent one task duration: it starts once the corresponding task has been submitted and stops at the end of its execution. The first, brighter part of the line represents the task waiting time spent from submission until a resource becomes available for execution. Colors are arbitrary and just help to distinguish the different tasks. As can be seen, at the beginning of the execution, only three nodes are used to execute the CrestLines service. Other resources are wasted. Similarly, the result of CrestMatch is needed for three services: PFMatchICP, Yasmina and Baladin but there is only one resource allocated to this service according to this strategy and it becomes a bottleneck.

The makespan of the *FIFO* strategy is lower: 46.88min $\pm$ 0.78min with the optimal bandwidth (1.16Mbps). The standard deviation of this strategy is higher due to the variable arriving order of the tasks. Some long tasks can be executed on the same computing resource, leading to the increase of the application makespan. Figure 4.14 shows a typical task schedule for this strategy.

4.8.2 Multi-stages strategies

For the *optimized* strategies, the workflow planner determines the number of virtual resources and the bandwidths yielding to a minimal execution cost. Without services grouping there are 4 execution stages which are represented in figure 4.15. According to the optimization results: only 30 nodes were

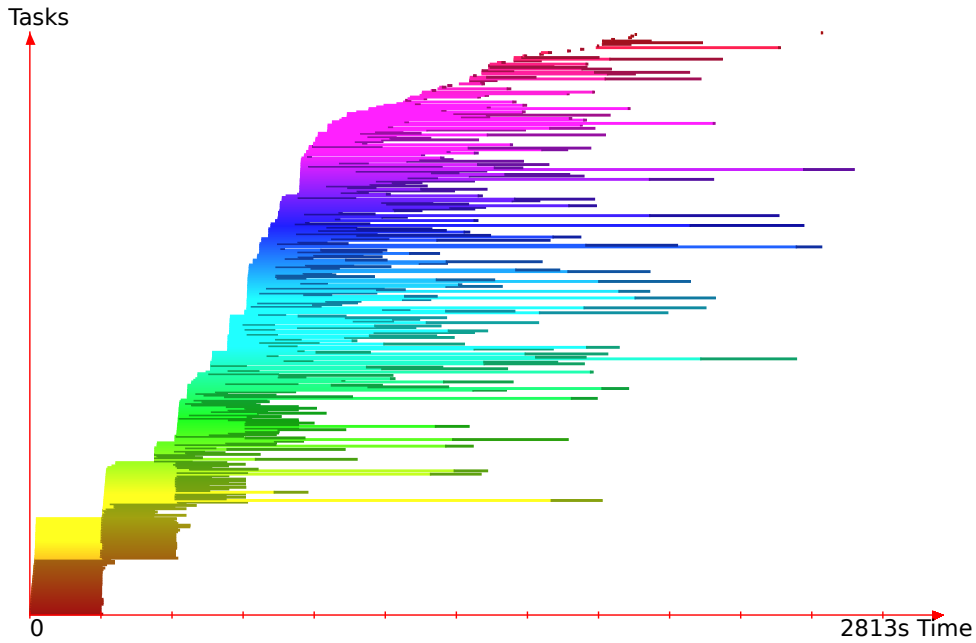


Figure 4.14: Tasks schedule with the *FIFO* strategy

allocated for the first, second and fourth stages (additional resources would be wasted). The bandwidths are 4.62Mbps, 14.74Mbps and 3.87Mbps, respectively. For the third stage, 4 nodes were allocated to PFMatchICP, 6 nodes for Yasmina and 20 nodes for Baladin. The bandwidth for each service in this stage is 0.87Mbps, 1.36Mbps and 1.29Mbps, respectively. The corresponding application makespan is 37.05min $\pm$ 0.25min.

Further grouping the application services as shown in figure 3.6, the application is divided into three stages only, using 30 nodes each. As presented in figure 4.16, the bandwidth allocated for each stage is 4.90Mbps, 1.95Mbps and 3.87Mbps, respectively. The application makespan is then 22.93min $\pm$ 0.35min. Besides the execution time improvement, the number of resources consumed is also lowered. As we can observe in figure 4.17, all tasks of the same stage do not finish exactly at the same time though, due to some variations of the image analysis tools execution time depending on the exact processed image content. This has an impact as the tasks of stage n have to wait for the longest task of stage $n-1$ before the system can be reconfigured.

4.8.3 Summary

In conclusion, table 4.6 compares the performance of the strategies presented above and the associated platform cost computed using equation 3.3. The worst case is the *naive* strategy that uses the maximum number of resources for a very large makespan and a long deployment. The *FIFO* strategy spends the same time to deploy the infrastructure but it has a better makespan than the *naive* strategy. The *naive* and *FIFO* strategies reconfiguration time is null since they are single-stage. The *optimized* strategy without grouping services has better results both in terms of application makespan and number of resources consumed than the *naive* and *FIFO* strategies, although it has to spend time to reconfigure the infrastructure after each stage. The best case is obtained for the *optimized* strategy with services grouping. It uses less resources, spends less time to reconfigure the infrastructure and returns the results faster. In terms

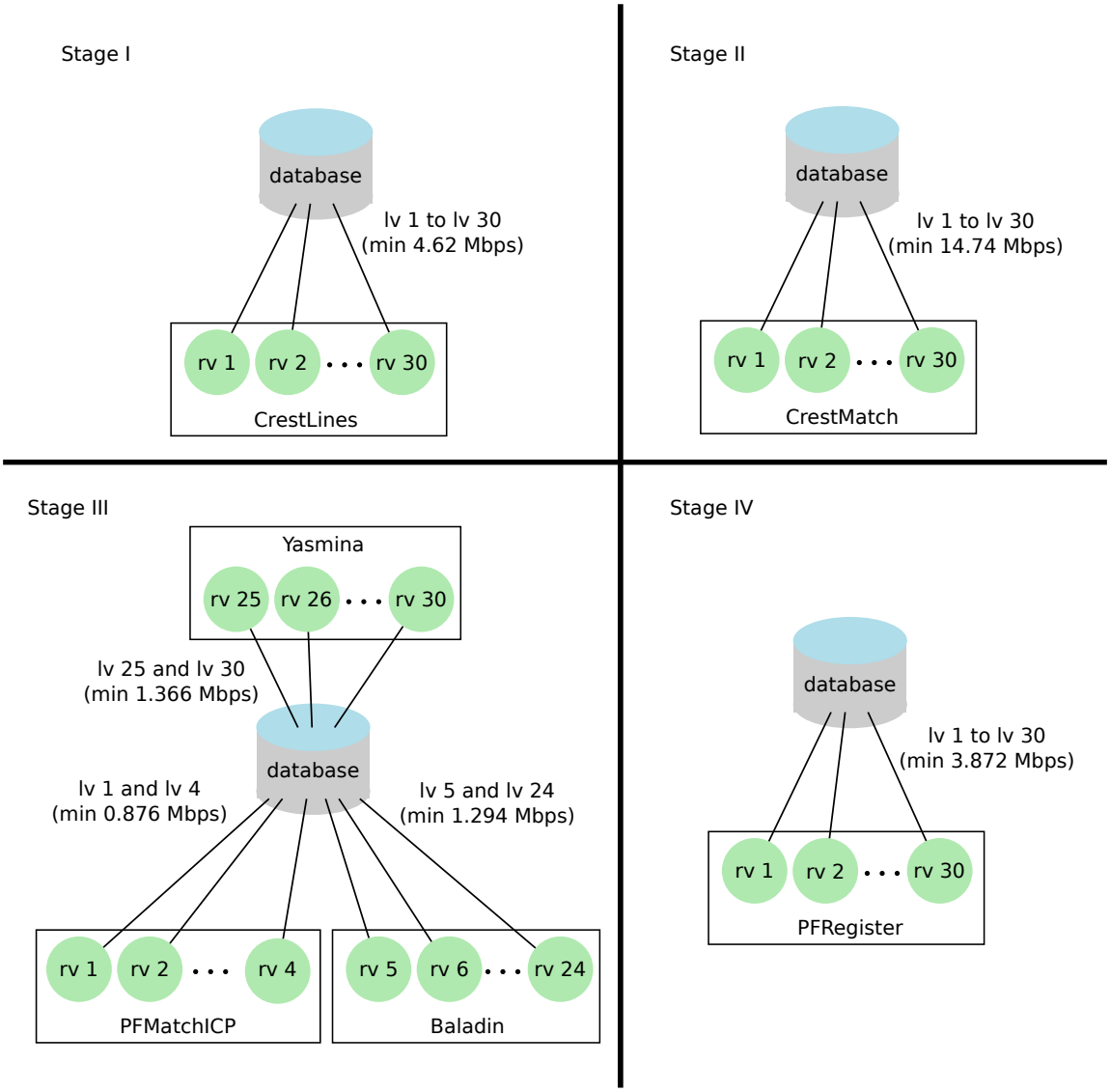


Figure 4.15: Virtual infrastructure composition with the *optimized* strategy without services grouping

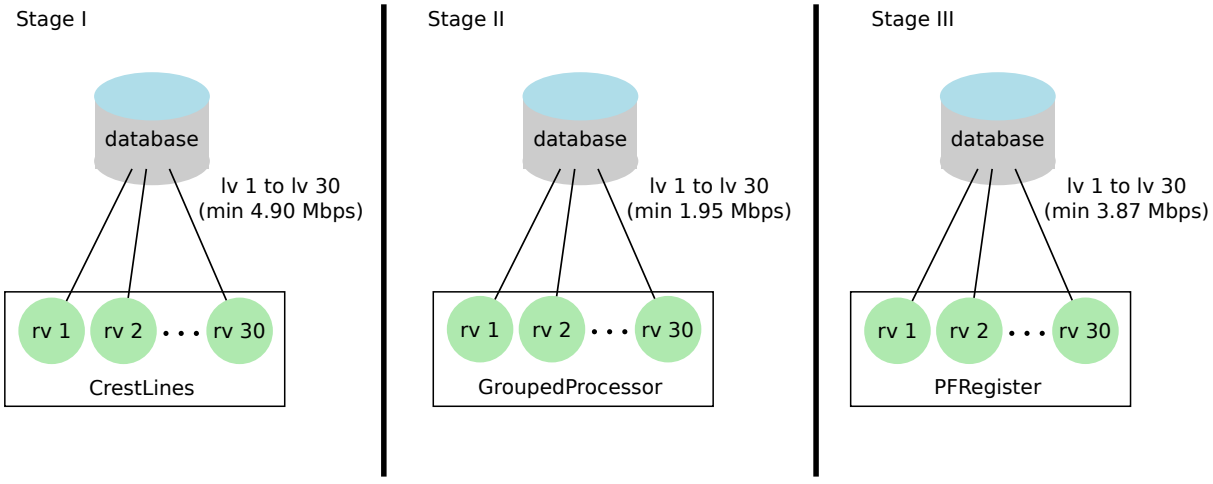


Figure 4.16: Virtual infrastructure composition with the *optimized* strategy with services grouping

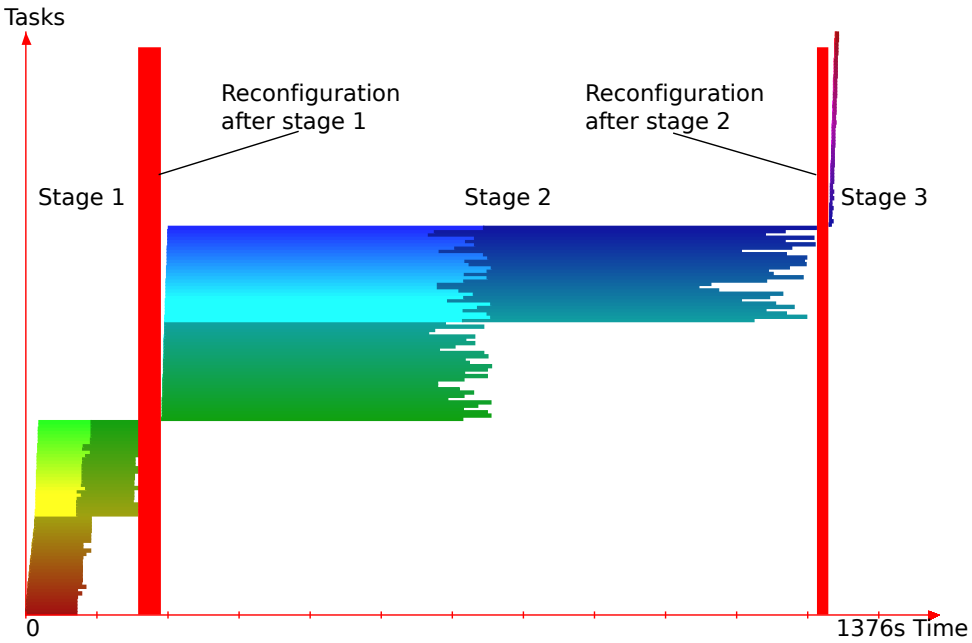


Figure 4.17: Tasks schedule with the *optimized* strategy with services grouping

Strategy	Makespan	#VM	Deployment time	Reconfiguration time	Execution cost ($\times 10^5$)
Naive	67.08min $\pm$ 0.10	35	29.83min	0	$1.40 \times c_r + 3.68 \times c_b$
FIFO	46.88min $\pm$ 0.78	35	29.83min	0	$0.98 \times c_r + 1.10 \times c_b$
Optimized (without grouping)	37.05min $\pm$ 0.25	31	25.68min	79.29s	$0.69 \times c_r + 1.40 \times c_b$
Optimized (with grouping)	22.93min $\pm$ 0.58	31	25.68min	52.86s	$0.42 \times c_r + 0.96 \times c_b$

Table 4.6: Performance comparison between the four strategies

of the deployment time, the *naive* and *FIFO* strategies take 29.83min to deploy 35 virtual machines. It is to be noted that HIPerNet does not enable the parallel deployment of resources yet. This duration corresponds to the time needed to copy the OS images (319MB) from the HIPerNet engine to the virtual machines and start them sequentially. The *optimized* strategies use only 31 machines, reducing the deployment time to 25.68min. In the future, parallel deployment is expected to lower this redeployment overhead. As expected, the cost estimated is lowered for higher performing strategies to the reduction of the application makespan and of the network bandwidth consumed.

4.8.4 Comparison with a commercial offer

Table 4.7 presents the cost billed by Amazon EC2 (equation 3.6) as a function of the unit costs (currently in Europe, c'_r =\$0.10 / VM / hour, and c'_b =\$0.15 / day / GB). For these computations we made the hypothesis of the same running times on Amazon EC2 nodes as on the Aladdin/Grid'5000 platform. While Amazon EC2 data transfer cost is the same for all strategies (transfer of 1GB input and output data), the cost paid for computing resources varies. The *naive* strategy, executing in more than one hour, dominates the reservation cost for computing resources ($2\ \text{hours} \times 35\text{VMs} \times c'_r$). The reservation

Strategy	Makespan	#VM	Execution cost	
			HIPerNet ($\times 10^5$)	Amazon EC2
Naive	67.08min $\pm$ 0.10	35	$1.40 \times c_r + 3.68 \times c_b$	$2\text{h} \times 35\text{VMs} \times c'_r + 1\text{GB} \times c'_b$
FIFO	46.88min $\pm$ 0.78	35	$0.98 \times c_r + 1.10 \times c_b$	$1\text{h} \times 35\text{VMs} \times c'_r + 1\text{GB} \times c'_b$
Optimized (without grouping)	37.05min $\pm$ 0.25	31	$0.69 \times c_r + 1.40 \times c_b$	$1\text{h} \times 31\text{VMs} \times c'_r + 1\text{GB} \times c'_b$
Optimized (with grouping)	22.93min $\pm$ 0.58	31	$0.42 \times c_r + 0.96 \times c_b$	$1\text{h} \times 31\text{VMs} \times c'_r + 1\text{GB} \times c'_b$

Table 4.7: Comparison with Amazon EC2

duration reduces to one hour for other strategies. Since the *FIFO* strategy uses 35VMs, its cost is higher than the *optimized* strategy with and without grouping optimization which use less resources (31VMs). Compared to Amazon EC2 cost, the cost model introduced in this paper is not rounded to the next hour, thus showing a decrease of the execution cost following the application makespan decrease. Moreover, the exact amount of bandwidth allocated is taken into account, thus showing a decrease on the data transfer cost for higher performing strategies. This cost is closer to a real measurement of the amount of resources consumed on the platform.

4.8.5 Impact of bandwidth control on application cost

Further experiments to evaluate the bandwidth control mechanism were also performed. The application was executed using the *optimized* strategy with service grouping under two additional network bandwidth configurations: lower and higher bandwidth values than the optimal found were tested (1 Mbps and 10 Mbps respectively). Table 4.8 displays for each configuration: the data transfer time in each stage (in seconds), the application makespan (in minutes) and the corresponding cost. Comparing the results with the optimized bandwidth allocation, it appears that using a low bandwidth, the makespan increases as expected. However, the cost increases as well because the cost gain on network bandwidth is compensated by the loss on computing nodes reservation time. With the high bandwidth, the application makespan can be reduced (-22.72% in this case) at a higher cost (+175% computed with $c_r = c_b = 0.10$).

Bandwidth	Stage 1 (s)	Stage 2 (s)	Stage 3 (s)	Makespan (min)	Execution cost ($\times 10^5$)
Low (1 Mbps)	222.59 $\pm$ 2.51	316.57 $\pm$ 40.37	2.91 $\pm$ 0.50	34.78 $\pm$ 0.67	$0.65 \times c_r + 0.63 \times c_b$
Optimized	53.8 $\pm$ 4.56	171.72 $\pm$ 24.66	1.53 $\pm$ 0.23	22.93 $\pm$ 0.58	$0.42 \times c_r + 0.96 \times c_b$
High (10 Mbps)	30.79 $\pm$ 3.85	42.68 $\pm$ 9.55	1.09 $\pm$ 0.18	17.72 $\pm$ 0.23	$0.33 \times c_r + 3.19 \times c_b$

Table 4.8: Bandwidth control mechanism evaluation

4.9 Handling the uncertainty in real execution: a simulation result

In this section, we present a simulation result of the technique for handling the uncertainty on services execution time introduced in section 3.4. We used the *optimized* strategy with services grouping opti-

mization applied to the Bronze Standard application (see figure 3.6). Thus, there are three stages per execution, each one uses 30 vitual computing resources connected to the database server through a 10Mbps link. The estimated total execution time is 779.52s with a simulation input data set of 59 image pairs resulting in 59 invocations of each service. For each task invocation, we made the hypothesis that the execution time is distributed according to a Gaussian distribution whose mean (μ) is the measured average execution time (for stage $i \in [1,3]$, $\mu_1 = 131.2$s, $\mu_2 = 645.52$s and $\mu_3 = 2.8$s). An execution stage is shorter than its expected time only if the execution time of all its task invocations is shorter than the estimated one. In the case of Gaussian distribution, there is a high probability that at least one random execution time is greater than its mean (μ). Therefore, the over-estimation case was always observed during the experiments. The extra cost caused by improper estimation is computed using equations 3.12 and 3.13 with $c_r = 0.10$, $c_b = 0.001$, $\lambda_1 = 90\%$ and $\lambda_2 = 10\%$. Two experiments were conducted to study the impact of the error in estimating the execution time:

1. The Gaussian distribution's standard deviation is set to increasing values, thus worsening the estimation;

2. The time slot extension period τ is increased, thus studying the impact of this parameter on cost computation.

Each experiment was repeated 20 times to measure the average and standard deviation of the application makespan and execution cost, respectively.

In the first experiment, the distribution's standard deviation varies from 0 to 0.8 of the mean value (μ). The τ value is set to 5% of the estimated execution time of each execution stage ($\tau_1 = 6.56$s, $\tau_2 = 32.276$s and $\tau_3 = 0.14$s). As shown in figure 4.18, the execution cost increases with the estimation error. The higher the error value, the higher the number of additional time slots of duration τ added to complete the execution. The total execution time also increases because the tasks belonging to the next stage have to wait for the longest task in the previous one to finish. Additionally, the standard deviation of the makespan and execution cost are higher when the error in the estimation increases. Although users could have to pay a very high cost when the estimation is not correct, their application can at least continue to execute until the end without being interupped.

In the second experiment, we fixed the standard deviation and changed the τ value. As presented in section 3.4, $\tau = \sigma \times T_i$, where T_i is the estimated execution time of stage i. The σ value is increased from 0.01 to 0.80, thus varying the τ value from 1% to 80% of T_i. The result displayed in figure 4.19 shows that if the τ value is small, the number of additional time τ needs to be added to complete the execution is high. The execution cost dominates due to the per-unit cost of resources that increases proportionally to this number. The execution cost decreases when τ increases (left part of the curves). However, when τ is greater than the error value, τ is added only once but its duration is increasing according to the σ value, thus increasing the execution cost paid for the this additional time slot (right part of the curves).

As intended, improper estimation is penalized by a cost increasing with the magnitude of the over estimation. The τ parameter can be used to tune the magnitude of the overhead.

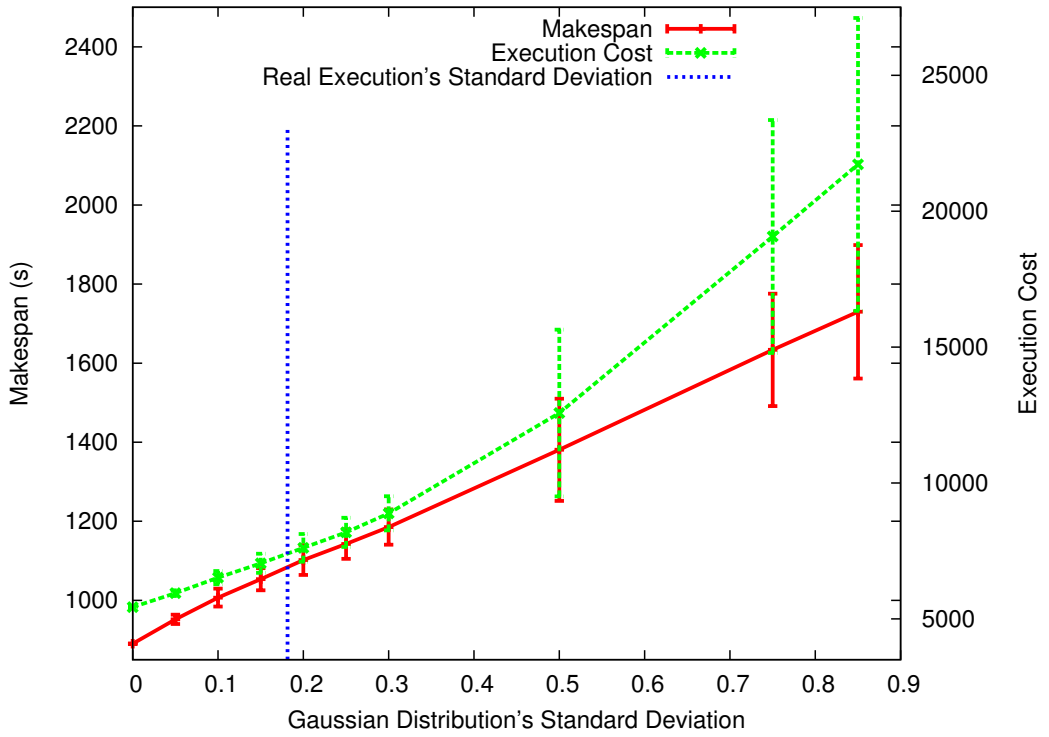


Figure 4.18: Variation of the execution cost respect to the error in the estimation.

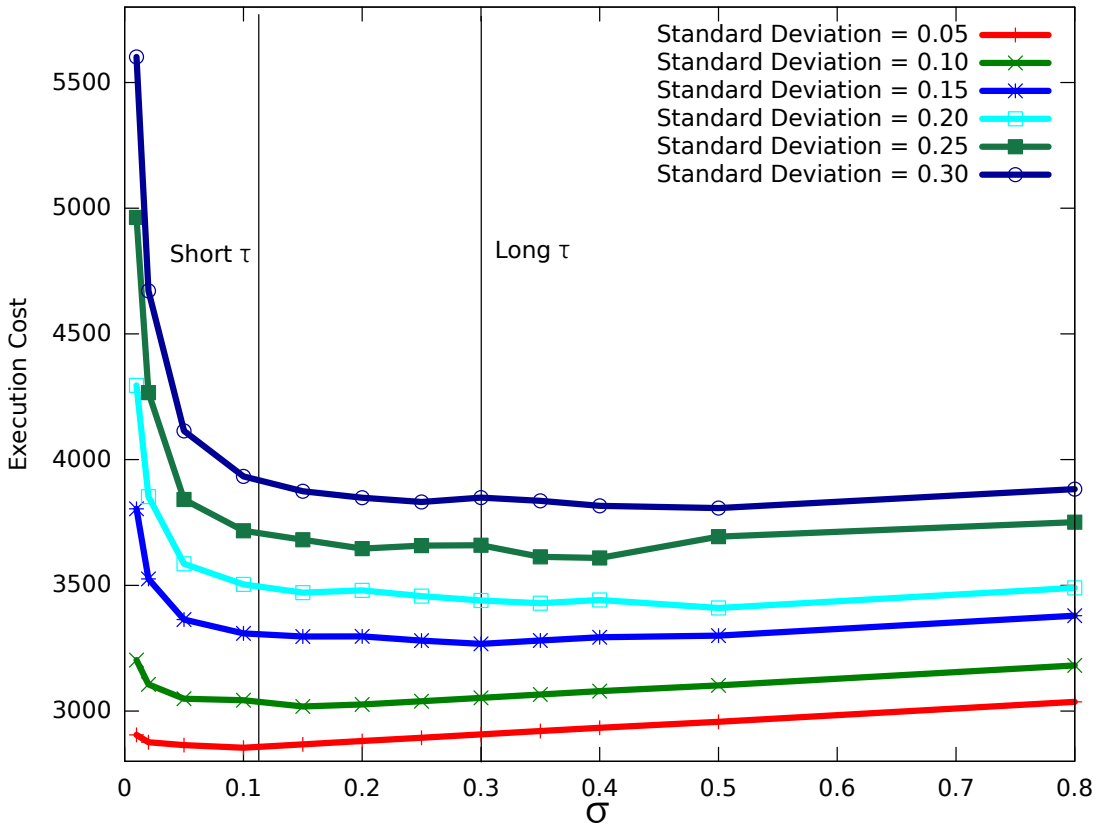


Figure 4.19: Variation of the execution cost respect to the τ value. $\tau = \sigma \times T_i$, where T_i is the estimated execution time of stage i.

4.10 Execution with redundant resources for improved reliability

The goal of the experiments presented in this section is to demonstrate the efficiency of the live migration mechanism to support the reliability on the virtual infrastructure described in section 3.6. We simulated faults by shutting down physical machines respecting the MTBF parameter which was set to 600s, 300s and 150s, respectively. After a MTBF period, a random physical machine in the set of physical resources will be shut down. Assuming the largest makespan of the application to be 30min, the failure probability of each node (P_{FAIL}) is then 0.03, 0.06 and 0.12, respectively. The initial value (600s) is based on failure rate of servers (with a probability between 0.02 and 0.04) identified by [Atwood and Miner, 2008].

For the experiments in this section, the VPXI is designed based on the *optimized* strategy with services grouping optimization presented in section 3.3.4 [Truong Huu and Montagnat, 2010] with the modification of the link bandwidth value. 31 virtual resources are configured with 512MB of RAM, and 1GHz of CPU. A link bandwidth of 10Mbps is reserved for each virtual link between the database and the computing nodes.

The application makespan is 1205s $\pm$ 40s when the application is executed on a substrate without simulated failures. It serves as the base-line computing time. In this case, the execution cost, computed by the equation 3.3, is $(0.37 \times c_r + 3.62 \times c_r) \times 10^5$, serving as base-line to analyze the execution cost.

The first experiment examines the protection of the database node which is a central point of failure, and therefore the most critical node. In this case, the database is the unique component protected, and faults are simulated in accordance with the MTBF value. Table 4.9 summarizes the application makespan of this scenario according to the MTBF values. The application makespan increases proportionally to the number of failures detected on the database node. Compared to the base-line, the application makespan increases by +16%, +26% and +40% with regard to the MTBF values, 600s, 300s and 150s, respectively.

In our experimental set-up, we provided the reliability by backing-up the database 1:1. A unique backup node is used for the database node for all the values of the MTBF. However, while 1:1 replication made our proof-of-concept implementation feasible[5], it does not keep the required reliability at the specified level (99.99%). To calculate the theoretical execution cost of each VPXI with the proper reliability support, we compute the number of backup nodes required to provide the reliability level of 99.99% as a function of the MTBF, computed according to [Koslovski et al., 2010]. Table 4.9 presents the execution cost computed by equation 3.18 according to the MTBF values. The lower the value of MTBF, the more the failures detected on the infrastructure, and thus the more backup nodes needed to guarantee the required reliability level. Consequently, the reservation cost for computing resources increases. The reservation cost for link bandwidth dominates significantly due to the increase of number of backup links. Especially in the case of database protection, for each backup node, the same number of links between the database node and other computing nodes needs to be reserved.

Each workflow service has a pre- and post-processing stage where the input data is copied to the execution worker node and the results are sent to the database. The more the failures happen during these two stages, the more the application makespan increases. In table 4.10, we present the data transfer time (in seconds) of this scenario. The data transfer time increase dominates when there are more failures detected on the database node.

[5] The current Remus implementation for Xen 3.4 is limited to a 1:1 protection.

MTBF	P_{FAIL}	Makespan	Makespan variation	Number of backup nodes	Execution cost ($\times 10^5$)
∞		1205s			$0.37 \times c_r + 03.62 \times c_b$
600s	0.03	1401s	+16.26%	2	$0.46 \times c_r + 12.61 \times c_b$
300s	0.06	1524s	+26.47%	3	$0.52 \times c_r + 18.29 \times c_b$
150s	0.12	1688s	+40.08%	4	$0.59 \times c_r + 25.32 \times c_b$

Table 4.9: Summarize of the database protection scenario. ∞ stands for the case where there is not failure simulation.

MTBF	Total data transfer time
∞	165.02s $\pm$ 44.30s
600	190.20s $\pm$ 96.75s
300	292.96s $\pm$ 115.38s
150	299.61s $\pm$ 128.26s

Table 4.10: Total data transfer time of six application services running with the critical database protection scenario.

The second experiment analyzes the protection of worker nodes. The MTBF values were the same as in the database protection scenario (600s, 300s and 150s, respectively). After an MTBF, a random physical machine will be crashed. The backup virtual machine is automatically started and continues running the same workflow task. As presented in table 4.11, the application makespan only slightly increases with regard to the number failures detected on the infrastructure. The delay on the backup node activation is partly compensated for by other parallel executions and the variation of input data. Table 4.11 shows the increase of the execution cost due to reliability for the different values of the MTBF. The reservation cost for computing nodes increases due to the larger number of backup nodes needed. Similarly to computing resources, the reservation cost for network bandwidth also increases but it does not dominates as in the database protection scenario. For each backup node, new backup link needs to be reserved to connect it to the database node.

In both, database protection and computing nodes protection cases, the application ran normally, the failures detected on the infrastructure are recovered transparently from the application point of view.

We also performed the experiments using the task resubmission mechanism (application level recovery mechanism) to compare with the VPXI reliability service. After MTBF, a failure will occur on

MTBF	P_{FAIL}	Makespan	Makespan variation	Number of backup nodes	Execution cost ($\times 10^5$)
∞		1205s			$0.37 \times c_r + 3.62 \times c_b$
600s	0.03	1208s	+0.2%	5	$0.43 \times c_r + 4.23 \times c_b$
300s	0.06	1225s	+1.7%	8	$0.48 \times c_r + 4.66 \times c_b$
150s	0.12	1244s	+3.2%	12	$0.53 \times c_r + 5.22 \times c_b$

Table 4.11: Summary of the computing nodes protection scenario.

MTBF	Reliability makespan	Resubmission makespan	Makespan variation	Reliability cost ($\times 10^5$)	Resubmission cost ($\times 10^5$)
600s	1208s	1366s	+13.08%	$0.43 \times c_r + 4.23 \times c_b$	$0.49 \times c_r + 4.78 \times c_b$
300s	1225s	1466s	+19.67%	$0.48 \times c_r + 4.66 \times c_b$	$0.57 \times c_r + 5.57 \times c_b$
150s	1244s	1520s	+22.19%	$0.53 \times c_r + 5.22 \times c_b$	$0.65 \times c_r + 6.38 \times c_b$

Table 4.12: Comparison of the reliability approach with the resubmission mechanism on the makespan and execution cost.

a worker node and this node will be unavailable for the computation. Consequently, new worker nodes must be provisioned, and the task executed on the failed node has to be restarted on a new node. We assumed that the number of backup nodes for task resubmission mechanism was the same as the one identified in the previous scenario (5, 8, and 12 for MTBF of 600s, 300s, 150s, respectively). These nodes are reserved, deployed and configured before starting the execution similarly to the reliability experiments. Even considering the activation time of a backup node as null, our experimental results show that the application makespan increases significantly in comparison with the virtual infrastructure reliability service, +13.08%, +19.67% and +22.19% with respect to 600s, 300s and 150s of the MTBF, as presented in table 4.12. Consequently, the execution cost with the resubmission mechanism increases significantly as well.

4.11 Conclusions

This chapter presented our experiments assessing the performance of the virtual infrastructure. Although the virtual infrastructure has a slight performance impact on the application execution (within 10%), this increase is acceptable when considering the other advantages of the virtual infrastructure such as the execution cost, security and reliability. The data transfer on virtual link might significantly impact the application performance though. This drawback was lowered thanks to the direct transfer and bandwidth control mechanisms. Additionally, the cost function model allows users to estimate the execution cost for an application run, given an input data set. The experiments demonstrate the usability of the resources allocation strategies and assess the performance of the optimized strategy with services grouping optimization. Finally, the live migration mechanism improves the virtual infrastructure reliability, transparently recovering from failures from the user point of view. All experimental results assess the validity of the approach in terms of infrastructure cost and application performance control. Our contributions both facilitate the exploitation of cloud infrastructures, delivering a higher quality of services to end users, and help the planning of cloud resources delivery.

Chapter 5

Conclusions and future work

5.1 Conclusions

The work presented in this thesis addresses the problem of performance and execution cost optimization of workflow-based applications on cloud infrastructures. Many commercial clouds provide to users a quasi-unlimited amount of computing, network and storage resources for dealing with the needs of large-scale applications. However, the sizing of each reservation is completely left under the user responsibility. Such an estimation is far from trivial though, especially when considering large-scale distributed applications. We focused on workflow-based applications which provide an application description framework with enough information to enable this estimation. We introduced a cost function model whose granularity is finer than commercial offers to bring to users a "near-optimal" estimation. The model is based on the expertise captured from the application including the data volume to exchange, the execution time of each workflow service and the application logic represented by the precedence contraints between workflow services.

Resources allocation has been studied since distributed computing infrastructures started being used for large-scale applications. Most of existing approaches focus on minimizing the application makespan and do not take into account the execution cost on a pay-per-use platform. Motivated by finding a trade-off between the resources reservation cost and application performance, we proposed four resources allocation strategies, based on the cost function model. A *naive* resources distribution strategy leads to sub-optimal performance and the waste of idle resources. The *FIFO* strategy makes the assumption that all services can be deployed on every computing resource. Computing resources are thus indistinguishable and the scheduler may request any task to be executed on any resource. The *FIFO* strategy is better exploiting available resources. It is also optimal in this case and a single stage is considered since the infrastructure redeployment is unnecessary. However, it does not optimize the bandwidth between each pair of resources. Towards a better optimization in both computing and network resources, we proposed the *optimized* strategy which considers dividing the workflow execution into multiple stages and allocating resources and bandwidth independently for each stage. The cost minimization algorithm is executed for each stage to allocate an optimal number of resources to the services involved at this stage. It reduces the application makespan by a factor 2, compared with the *naive* strategy, thus reducing the execution cost. These results were obtained taking into account the fact that the more stages involved in

an execution, the higher the reconfiguration cost between stages. We extended an existing service grouping approach [Glatard et al., 2008a] to reduce the number of execution stages. The experimental results applied to the Bronze Standard application show that reducing the number of execution stages from 4 to 3 leads to a reduction of the application makespan of 38%. The overall optimization, as compared to the *naive* strategy, leads to an improvement of the application makespan over 65%.

A limitation of these strategies is that they apply only to workflow-based applications which can be represented by an execution DAG (the exact number of invocations of each service needs to be known), so that the workflow planner can determine a complete execution schedule. Consequently, workflows including *while* kind of loops, or *foreach* constructs iterating over unknown size data structures make the workflow unresolvable prior to execution. Yet, this represents a broad category of workflow applications in e-Science.

During the experiments, we observed that the real execution differs from the estimated one due to the variation of data volumes and the non-deterministic nature of some algorithms. The real execution time becomes under- or overestimated. On a cloud platform, the execution could be aborted before completion if the reservation expires. We proposed a technique to handle this problem by resizing the resource reservation dynamically when the real execution differs from the estimated one. However, to avoid abuse of the infrastructure, the cost function model was extended to compute an extra cost resulting from the improper estimation. Both users and infrastructure providers can benefit from our technique. In the case the execution time is underestimated, the user application can continue running while infrastructure providers can dynamically control their resources allocation.

Beside resources allocation, we considered other problems related to distributed computing infrastructures that impact application performance, and therefore execution cost, such as the data transfer overhead and low reliability. We introduced a mechanism to improve the data transfer between workflow services. As resources on cloud infrastructures are reserved exclusively to the whole application execution, intermediate results of workflow services can stay cached on computing resources. They will be transferred directly to other computing resources if needed, instead of being copied to a central storage server back and forth. The experimental results on physical infrastructure show that the data transfer time with the proposed mechanism reduces by 54.09% for the best case. The improvement is even higher on virtual infrastructure (with a reduction of 68.01%) where the network virtualization causes extra overheads.

Finally, to address the low reliability problem, a live migration mechanism was used. Critical resources are synchronized with backup resources which will resume execution immediately when failures happen on critical resources. The cost function model was extended to infrastructure reliability by computing the cost for backup resources. The amount of backup resources is optimized by using the approach presented in [Koslovski et al., 2010, Yeow et al., 2010]. Compared to the resubmission approach, a traditional failure recovering mechanism, the experimental results show that the live migration mechanism reduces significantly (18.15% for the best case) the application makespan and thus reduces the execution cost.

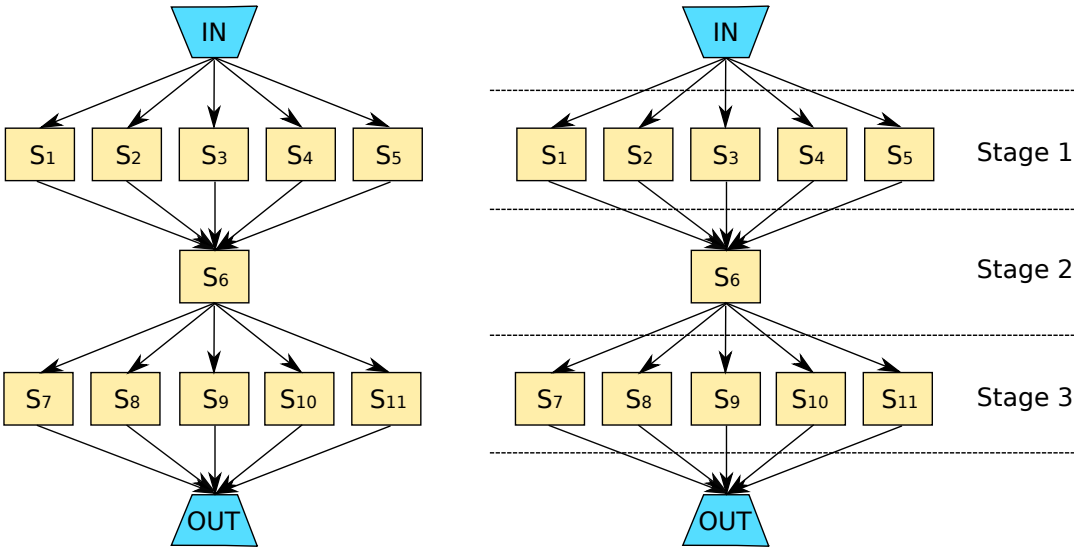
5.2 Future work

Extending the cost model to storage resources. An important part of cloud infrastructures is storage resources which are used for both long and short term data storage. The long term storage is used for persistent data such as input data or output of the application that needs to be preserved after the execution finishes. Users are responsible for determining the duration and amount of disk space to reserve, independently of the resources used for execution. However, the short term storage that is used for temporary data generated during the execution may impact the total execution cost. In this thesis, we assumed that the disk space on a standard computing resource is sufficient for storing all intermediate data. However, large-scale distributed applications, especially data intensive applications, consume significant amounts of temporary storage that could exceed the storage capacity of regular computing nodes. In this case, users should recruit additional storage. This leads to the extra cost which should be integrated in the cost model. Furthermore, reserving an additional storage server results in the data transmission cost between the storage server and computing resources. For instance, Amazon charges users for the network bandwidth between the storage resources delivered by Amazon S3 and computing resources provided by Amazon EC2 while the data transfer inside Amazon EC2 cluster is free. Therefore, it would be interesting to extend the cost function model to this kind of resources and propose the allocation strategy to optimize the amount of resources needed.

Towards the dynamic resources allocation. The resources allocation strategies presented in chapter 3 only apply to the workflow-based applications whose complete execution DAG can be generated prior to execution. In the general case though, many applications cannot be represented as static DAGs due to the use of conditionals, unbounded loops or variable data structures size. A solution for dealing with workflows with unresolvable constructs is to divide them into smaller resolvable sub-workflows and apply these resources allocation strategies to them. This solution was implemented in DIET MA DAG [Amar et al., 2006], which is a DAG-based workflow scheduler, to accomodate to complex workflows written with the GWENDIA language. However, this implementation only considers the workflow scheduling part. Further development of existing resources allocation strategies is an interesting perspective towards a dynamic approach for workflow-based applications.

Experimental validation of the mechanism handling the improper estimation. Dynamically handling improper estimations during the execution is a difficult task. The approach presented in chapter 3 is evaluated by a simulation based on a probabilistic model. Even if the simulation results assess the validity of the approach, a validation by a real execution is required to envisage the application of this technique on a cloud platform. To perform this experiment, some implementation work is needed. At the application level, the monitoring mechanism should be implemented to detect whether the estimated execution is under- or overestimated. At the infrastructure level, the cloud middleware should be instrumented to include the infrastructure resizing module which is used to resize the reservation if needed.

Prediction of the number of pilot agents to submit. As presented in chapter 1, the pilot job mechanism is used to reserve computing resources to execute computing tasks. Estimating the number of pilots needed for each application run is a difficult task. Without assistant, users tend to submit pilots



(a) Original workflow

(b) Divided into 3 stages requiring different number of pilot jobs

Figure 5.1: Workflow example and its execution divided into stages

to all infrastructure sites to get as many computing resources as possible. This practice is aggressive from an administrator point of view, which aims at fairly sharing resources among users and increasing the infrastructure performance. Additionally, in many applications, the number of computing resources needed varies during the execution. This leads to the variation of the number of necessary pilots along time. For example, considering the application presented in figure 5.1a, its execution is divided into 3 stages requiring different number of pilot agents (figure 5.1b). The first and third stage require more pilot agents than the second stage which has only one service (S_6). Assuming that the execution time of S_6 is longer than the duration that a pilot can stay idle before terminating without processing any computing task, all pilots which are idle during the second stage will terminate before the third stage starts. The third stage will be a bottleneck due to the lack of computing resources. Therefore, it could be useful if the resources allocation strategies presented in chapter 3 are applied to predict the number of pilots submitted to the infrastructure during the execution.

Towards the control of a complete cloud platform. The approach presented in this manuscript is user-oriented in the sense that each user can make use of the cost function model to locally optimize its cost. Applying it to the infrastructure administrators level requires further development to reach a well-operational cloud platform with high quality of services. Maximizing the benefit of the infrastructure is the main objective of commercial cloud providers but fairly sharing resources among users is also crucial, *e.g.* for academic clouds. Although the amount of resources on clouds is "quasi-unlimited", the user needs are unbounded and might cause resources shortage. A solution that could be used if there are multiple simultaneous user requests totalizing more resources than the infrastructure can provide is to virtualize physical resources into as many virtual machines as needed. However, the number of virtual machines hosted on the same physical one cannot increase seamlessly due to the virtualization impact on resource performance. To maximize the benefit of cloud providers while guaranteeing the quality of

services, a new cost model is needed. Similarly to many real cost booking systems, the model presented in this thesis could be extended to dynamically adapt to the amount of resources available. When the sum of user requests becomes larger than the capacity of the infrastructure, the user competes against each other and pay a higher price for resources usage. Otherwise, the cost may reduce to attract users.

Bibliography

[Ahn et al., 2008] Ahn, S., Namgyu, K., Seehoon, L., Soonwook, H., Dukyun, N., Koblitz, B., Breton, V., and Sangyong, H. (2008). Improvement of Task Retrieval Performance Using AMGA in a Large-Scale Virtual Screening. In *4th International Conference on Networked Computing and Advanced Information Management (NCM'08)*, pages 456–463, Gyeongju, Korea.

[Aida et al., 2006] Aida, Y., Nakajima, Y., Sato, M., Sakurai, T., Takahashi, D., and Boku, T. (2006). Performance Improvement by Data Management Layer in a Grid RPC System. In *1st International Conference on Grid and Pervasive Computing (CGP2006)*, pages 324–335, Taiwan. Springer Verlag.

[Albing, 1993] Albing, C. (1993). Cray NQS: production batch for a distributed computing world. In *11th Sun User Group Conference and Exhibition*, pages 302–309, Brookline, MA, USA.

[Alhusaini et al., 2000] Alhusaini, A. H., Prasanna, V. K., and Raghavendra, C. S. (2000). Framework for Mapping with Resource Co-Allocation in Heterogeneous Computing Systems. In *9th Heterogeneous Computing Workshop (HCW'00)*, pages 273–286, Cancun, Mexico. IEEE Computer Society.

[Alt and Hoheisel, 2005] Alt, M. and Hoheisel, A. (2005). A grid workflow language using high-level Petri Nets. In *6th international conference on Parallel Processing and Applied Mathematics (PPAM'05)*, pages 715–722, Poznan, Poland.

[Amar et al., 2006] Amar, A., Bolze, R., Bouteiller, A., Chis, A., Caniou, Y., Caron, E., Kaur Chouan, P., Le Mahec, G., Dail, H., Depardon, B., Desprez, F., Gay, J.-S., and Su, A. (2006). DIET: New Developments and Recent Results. Technical Report RR-6027, INRIA Rhône-Alpes.

[Amendolia et al., 2004] Amendolia, S. R., Estrella, F., Hassan, W., Hauer, T., Manset, D., Macclatchey, R., Rogulin, D., and Solomonides, T. (2004). MammoGrid: A Service Oriented Architecture based Medical Grid Application. In *3rd International Conference on Grid and Cooperative Computing (GCC 2004)*, volume 3251, pages 939–942, Wuhan, China.

[Arnold et al., 2000] Arnold, D., Bachmann, D., and Dongarra, J. (2000). Request Sequencing: Optimizing Communication for the Grid. In *6th International Euro-Par Conference-Parallel Processing (Euro-Par 2000)*, volume 1900/2000, pages 1213–1222, Munich, Germany. Springer Verlag.

[Atwood and Miner, 2008] Atwood, D. and Miner, J. G. (2008). Reducing data center cost with an air economizer. Technical report, Intel Coporation.

[Awadallah and Rosenblum, 2002] Awadallah, A. and Rosenblum, M. (2002). The vMatrix: A Network of Virtual Machine Monitors for Dynamic Content Distribution. In *7th International Workshop on Web Content Caching and Distribution (WCW)*, Boulder, Colorado. Elsevier's Computer Communications.

[Bajaj and Agrawal, 2004] Bajaj, R. and Agrawal, D. P. (2004). Improving Scheduling of Tasks in A Heterogeneous Environment. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 15(2):107–118.

[Barham et al., 2003] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the Art of Virtualization. In *19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 164–177, Bolton Landing, NY, USA. ACM.

[Beck and Moore, 1998] Beck, M. and Moore, T. (1998). The Internet2 Distributed Storage Infrastructure Project: An architecture for internet content channels. *Computer Networks and ISDN Systems*, 30(22-23):2141–2148.

[Berman et al., 2005] Berman, F., Casanova, H., Chien, A., Cooper, K., Dail, H., Dasgupta, A., Deng, W., Dongarra, J., Johnsson, L., Kennedy, K., Koelbel, C., Liu, B., Liu, X., Mandal, A., Marin, G., Mazina, M., Mellor-Crummey, J. M., Mendes, C. L., Olugbile, A., Patel, M., Reed, D. A., Shi, Z., Sievert, O., Xia, H., and YarKhan, A. (2005). New Grid Scheduling and Rescheduling Methods in the GrADS Project. *International Journal of Parallel Programming (IJPP)*, 33(2-3):209–229.

[Berten and Jeannot, 2009] Berten, V. and Jeannot, E. (2009). Modeling Resubmission in Unreliable Grids: the Bottom-Up Approach. In *7th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar'2009)*, volume 6043 of *LNCS*, Delft, The Netherlands.

[Bester et al., 1999] Bester, J., Foster, I., Kesselman, C., Tedesco, J., and Tuecke, S. (1999). GASS: A Data Movement and Access Service for Wide Area Computing Systems. In *6th Workshop on Input/Output in Parallel and Distributed Systems (IOPADS'99)*, pages 78–88, Atlanta, Georgia. ACM.

[Blanquer Espert et al., 2005] Blanquer Espert, I., Hernández García, V., and Segrelles, D. (2005). Creating virtual storages and searching DICOM medical images through a grid middleware based in OGSA. In *IEEE International Symposium on Cluster Computing and the Grid (CCGRID'2005)*, pages 504 – 511, Cardiff, UK.

[Blythe et al., 2005] Blythe, J., Jain, S., Deelman, E., Gil, Y., Vahi, K., Mandal, A., and Kennedy, K. (2005). Task Scheduling Strategies for Workflow-based Applications in Grids. In *International Symposium on Cluster Computing and the Grid (CCGrid'05)*, pages 759–767, Cardiff, UK.

[Bode et al., 2000] Bode, B., Halstead, D. M., Kendall, R., Lei, Z., and Jackson, D. (2000). The portable batch scheduler and the maui scheduler on linux clusters. In *4th Annual Linux Showcase & Conference (ALS'00)*, volume 4. USENIX Association Berkeley, CA, USA.

[Braun et al., 2001] Braun, T. D., Siegel, H. J., Beck, N., Bölöni, L. L., Maheswaran, M., Reuther, A. I., Robertson, J. P., Theys, M. D., Yao, B., Hensgen, D., and Freund, R. F. (2001). A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing (JPDC)*, 61(6):810–837.

[Breton et al., 2009] Breton, V., Kim, D., and Rastelli, G. (2009). *WISDOM: A Grid-Enabled Drug Discovery Initiative against Malaria*, chapter 14, pages 353–381. CRC Press.

[Bugnion et al., 1997] Bugnion, E., Devine, S. W., Govil, K., and Rosenblum, M. (1997). Disco: running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447.

[Buyya et al., 2000] Buyya, R., Abramson, D., and Giddy, J. (2000). Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid. In *The Fourth International Conference/Exhibition on High Performance Computing in Asia-Pacific Region (HPC-ASIA 2000)*, volume 1, pages 283–289, Beijing, China.

[Buyya et al., 2005] Buyya, R., Abramson, D., and Venugopal, S. (2005). The Grid Economy. *Proceedings of the IEEE*, 93(3):698–714.

[Cancio et al., 2004] Cancio, G., Fisher, S. M., Folkes, T., Giacomini, F., Hoschek, W., Kelsey, D., and Tierney, B. L. (2004). The DataGrid Architecture Version 2. In *EDMS 439938. CERN*.

[Canon and Jeannot, 2008] Canon, L.-C. and Jeannot, E. (2008). Scheduling Strategies for the Bicriteria Optimization of the Robustness and Makespan. In *11th International Workshop on Nature Inspired Distributed Computing (NIDISC 2008)*, pages 1–8, Miami, Florida, USA. IEEE CS Press.

[Capit et al., 2005] Capit, N., Da Costa, G., Georgiou, Y., Huard, G., and Marti, C. (2005). A batch scheduler with high level components. In *IEEE/ACM International Conference on Cluster Computing and Grid 2005 (CCGrid'05)*, volume 2, pages 776– 783, Cardiff, UK.

[Cappello et al., 2005] Cappello, F., Desprez, F., Dayde, M., Jeannot, E., Jegou, Y., Lanteri, S., Melab, N., Namyst, R., Vicat-Blanc Primet, P., Richard, O., Caron, E., Leduc, J., and Mornet, G. (2005). Grid'5000: A Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform. In *6th IEEE/ACM International Workshop on Grid Computing (Grid'2005)*, Seattle, Washington, USA. IEEE Computer Society.

[Caron et al., 2005] Caron, E., Del-Fabbro, B., Desprez, F., Jeannot, E., and Nicod, J.-M. (2005). Managing Data Persistence in Network Enabled Servers. *Scientific Programming Journal*, 13(4):333–354.

[Caron and Desprez, 2006] Caron, E. and Desprez, F. (2006). DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid. *International Journal of High Performance Computing Applications (IJHPCA) Special issue on on Clusters and Computational Grids for Scientific Computing*, 20(3):335–352.

[Casajus et al., 2010] Casajus, A., Graciani, R., Paterson, S., Tsaregorodtsev, A., and Team, L. D. (2010). DIRAC Pilot Framework and the DIRAC Workload Management System. *Journal of Physics: Conference Series*, 219(6).

[Casanova, 2007] Casanova, H. (2007). Benefits and Drawbacks of Redundant Batch Requests. *Journal of Grid Computing (JOGC)*, 2(5):888–903.

[Casanova and Dongarra, 1997] Casanova, H. and Dongarra, J. (1997). NetSolve: A Network-Enabled Server for Solving Computational Science Problems. *International Journal of High Performance Computing Applications (IJHPCA)*, 11(3):212–223.

[Casanova et al., 2000] Casanova, H., Legrand, A., Zagorodnov, D., and Berman, F. (2000). Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *9th Heterogeneous Computing Workshop (HCW'00)*, pages 349–363, Cancun.

[Cully et al., 2008] Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., and Warfield, A. (2008). Remus: High Availability via Asynchronous Virtual Machine Replication. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*, pages 161–174, San Francisco, CA, USA. ACM.

[Czajkowski et al., 1998] Czajkowski, K., Foster, I., Karonis, N. T., Kesselman, C., Martin, S., Smith, W., and Tuecke, S. (1998). A Resource Management Architecture for Metacomputing Systems. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, Orlando, Florida, USA. Springer-Verlag.

[Czajkowski et al., 1999] Czajkowski, K., Foster, I., and Kesselman, C. (1999). Resource Co-Allocation in Computational Grids. In *8th International Symposium on High Performance Distributed Computing (HPDC-8)*, pages 219–228, Redondo Beach, California. IEEE Computer Society.

[Deelman et al., 2003] Deelman, E., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Blackburn, K., Lazzarini, A., Arbree, A., Cavanaugh, R., and Koranda, S. (2003). Mapping Abstract Complex Workflows onto Grid Environments. *Journal of Grid Computing (JOGC)*, 1(1):9–23.

[Deelman et al., 2005] Deelman, E., Singh, G., Su, M.-H., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G. B., Good, J., Laity, A., Jacob, J. C., and Katz, D. S. (2005). Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming Journal*, 13(3):219–237.

[Del-Fabbro et al., 2007] Del-Fabbro, B., Laiymani, D., Nicod, J.-M., and Philippe, L. (2007). DTM: a service for managing data persistency and data replication in network-enabled server environments. *Concurrency and Computation: Practice & Experience (CCPE)*, 19(16):2125–2140.

[Desprez and Jeannot, 2004] Desprez, F. and Jeannot, E. (2004). Improving the GridRPC Model with Data Persistence and Redistribution. In *3rd International Symposium on Parallel and Distributed Computing (ISPDC 2004)*, pages 193–200, Cork, Ireland. IEEE CS.

[Devine et al., 1998] Devine, S. W., Bugnion, E., and Rosenblum, M. (1998). Virtualization system including a virtual machine monitor for a computer with a segmented architecture. United States Patent 6397242.

[Elmroth and Tordsson, 2009] Elmroth, E. and Tordsson, J. (2009). A standards-based Grid resource brokering service supporting advance reservations, coallocation, and cross-Grid interoperability. *Concurrency and Computation: Practice & Experience (CCPE)*, 21(18):2298–2335.

[Enslow, 1978] Enslow, P. H. (1978). What is a "Distributed" Data Processing System? *Computer*, 11(1):13–21.

[Fahringer et al., 2007] Fahringer, T., Prodan, R., Duan, R., Hofer, J., Nadeem, F., Nerieri, F., Podlipnig, S., Qin, J., Siddiqui, M., Truong, H.-L., Villazon, A., and Wieczorek, M. (2007). ASKALON: A Development and Grid Computing Environment for Scientific Workflows. In [Taylor et al., 2007], chapter 27, pages 450–471.

[Fahringer et al., 2005] Fahringer, T., Qin, J., and Hainzer, S. (2005). Specification of Grid Workflow Applications with AGWL: An Abstract Grid Workflow Language. In *fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, volume 2, pages 676–685, Cardiff, UK. IEEE Computer Society.

[Farooq et al., 2009] Farooq, U., Majumdar, S., and Parson, E. W. (2009). Achieving efficiency, quality of service and robustness in multi-organizational Grids. *Journal of Systems and Software (JSS)*, 82(1):23–38.

[Farooq et al., 2006] Farooq, U., Majumdar, S., and Parsons, E. W. (2006). A Framework to Achieve Guaranteed QoS for Applications and High System Performance in Multi-Institutional Grid Computing. In *35th International Conference on Parallel Processing (ICPP 2006)*, pages 373–380, Ohio, USA. IEEE Computer Society.

[Feo and Resende, 1995] Feo, T. A. and Resende, M. G. C. (1995). Greedy Randomized Adaptive Search Procedures. *Journal of Global Optimization*, 6(2):109–133.

[Foster and Karonis, 1998] Foster, I. and Karonis, N. T. (1998). A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems. In *Supercomputing (SC98)*. ACM Press.

[Foster and Kesselman, 1997] Foster, I. and Kesselman, C. (1997). Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications (IJSA)*, 11(2):115–128.

[Foster et al., 1999] Foster, I., Kesselman, C., Lee, C., Lindell, B., and Nahrstedt, K. (1999). A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. In *IEEE International Workshop on Quality of Service (IWQoS'99)*, pages 27–36, London, UK. IEEE Computer Society.

[Foster et al., 2001] Foster, I., Kesselman, C., and Tuecke, S. (2001). The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications (IJSA)*, 15(3):200–222.

[Foster et al., 1997] Foster, I., Kohr, D., Krishnaiyer, R., and Mogill, J. (1997). Remote I/O: Fast access to distant storage. In *5th Workshop on Input/Output in Parallel and Distributed Systems (IOPADS'97)*, pages 14–25, San Jose, CA, USA. ACM.

[Frey et al., 2002] Frey, J., Tannenbaum, T., Livny, M., Foster, I., and Tuecke, S. (2002). Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing (CC)*, 5(3):237–246.

[Gabriel et al., 1998] Gabriel, E., Resch, M., Beisel, T., and Keller, R. (1998). Distributed Computing in a Heterogeneous Computing Environment. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Lecture Notes in Computer Sciences*, pages 180–187. Springer.

[Gannon, 2007] Gannon, D. (2007). Component Architectures and Services: From Application Construction to Scientific Workflows. In [Taylor et al., 2007], chapter 12, pages 174–189.

[Germain et al., 2005] Germain, C., Breton, V., Clarysse, P., Gaudeau, Y., Glatard, T., Jeannot, E., Legré, Y., Loomis, C., Montagnat, J., Moureaux, J.-M., Osorio, A., Pennec, X., and Texier, R. (2005). Grid-enabling medical image analysis. In *proceedings of the IEEE/ACM International Symposium on Cluster Computing and the Grid (Biogrid'05), workshop*, volume 1, pages 487– 495, Cardiff, UK.

[Germain et al., 2008] Germain, C., Loomis, C., Mościcki, J. T., and Texier, R. (2008). Scheduling for Responsive Grids. *Journal of Grid Computing (JOGC)*, 6(1):15–27.

[Glatard et al., 2008a] Glatard, T., Montagnat, J., Emsellem, D., and Lingrand, D. (2008a). A Service-Oriented Architecture enabling dynamic services grouping for optimizing distributed workflows execution. *Future Generation Computer Systems (FGCS)*, 24(7):720–730.

[Glatard et al., 2008b] Glatard, T., Montagnat, J., Lingrand, D., and Pennec, X. (2008b). Flexible and efficient workflow deployement of data-intensive applications on grids with MOTEUR. *International Journal of High Performance Computing Applications (IJHPCA) Special Issue on Workflows Systems in Grid Environments*, 22(3):347–360.

[Glatard et al., 2005] Glatard, T., Montagnat, J., and Pennec, X. (2005). Grid-enabled workflows for data intensive medical applications. In *18th IEEE International Symposium on Computer-Based Medical Systems (CBMS)*, pages 537–542, Dublin, Ireland.

[Glatard et al., 2006a] Glatard, T., Montagnat, J., and Pennec, X. (2006a). Medical image registration algorithms assesment: Bronze Standard application enactment on grids using the MOTEUR workflow engine. In *HealthGrid conference (HealthGrid'06)*, pages 93–103, Valencia, Spain. IOS Press.

[Glatard et al., 2006b] Glatard, T., Pennec, X., and Montagnat, J. (2006b). Performance evaluation of grid-enabled registration algorithms using bronze-standards. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI'06)*, LNCS 4191, pages 152–160, Copenhagen, Denmark. Springer.

[Govil et al., 1999] Govil, K., Teodosiu, D., Huang, Y., and Rosenblum, M. (1999). Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. In *17th ACM SIGOPS*

Symposium on Operating Systems Principles (SOSP'99), volume 33, pages 154–169, Charleston, SC, USA. ACM.

[Gropp et al., 1996] Gropp, W., Lusk, E., Doss, N., and Skjellum, A. (1996). A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing (PARCO)*, 22(6):789–828.

[Hall et al., 2007] Hall, R., Rosenberg, A. L., and Venkataramani, A. (2007). A Comparison of Dag-Scheduling Strategies for Internet-Based Computing. In *International Parallel and Distributed Processing Symposium (IPDPS'07)*, pages 1–9, Long Beach, CA, USA. IEEE Computer Society.

[Halsted, 1985] Halsted, R. (1985). Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538.

[Huedo et al., 2004] Huedo, E., Montero, R. S., and Llorente, I. M. (2004). A framework for adaptive execution in grids. *Software: Practice and Experience*, 34(7):631–651.

[Jacq et al., 2008] Jacq, N., Salzeman, J., Jacq, F., Legré, Y., Medernach, E., Montagnat, J., Maass, J., Reichstadt, M., Schwichtenberg, H., Sridhar, M., Kasam, V., Zimmermann, M., Hofmann, M., and Breton, V. (2008). Grid-enabled Virtual Screening against malaria. *Journal of Grid Computing (JOGC)*, 6(1):29–43.

[Jannin et al., 2002] Jannin, P., Fitzpatrick, J., Hawkes, D., Pennec, X., Shahidi, R., and Vannier, M. (2002). Validation of Medical Image Processing in Image-guided Therapy. *IEEE Transactions on Medical Imaging (TMI)*, 21(12):1445–1449.

[Jeannot et al., 2008] Jeannot, E., Saule, E., and Trystram, D. (2008). Bi-Objective Approximation Scheme for Makespan and Reliability Optimization on Uniform Parallel Machines. In *14th International European Conference on Parallel and Distributed Computing (Euro-Par 2008)*, volume 5168 of *LNCS*, pages 877–886, Las Palmas de Gran Canaria, Spain.

[Kacsuk et al., 2003] Kacsuk, P., Dózsa, G., Kovács, J., Lovas, R., Podhorszki, N., Balaton, Z., and Gombás, G. (2003). P-GRADE: A Grid Programing Environment. *Journal of Grid Computing (JOGC)*, 1(2):171–197.

[Kacsuk et al., 2008] Kacsuk, P., Farkas, Z., and Fedak, G. (2008). Towards making BOINC and EGEE interoperable. In *4th eScience conference (eScience'08)*, pages 478–484, Indianapolis, USA.

[Kangarlou et al., 2009] Kangarlou, A., Eugster, P., and Xu, D. (2009). VNsnap: Taking Snapshots of Virtual Networked Environments with Minimal Downtime. In *39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009)*, Estoril, Lisbon, Portugal.

[Keahey et al., 2005] Keahey, K., Foster, I., Freeman, T., and Zhang, X. (2005). Virtual workspaces: Achieving quality of service and quality of life in the Grid. *Scientific Programming Journal*, 13(4):265–275.

[Koslovski et al., 2009] Koslovski, G., Truong Huu, T., Montagnat, J., and Vicat-Blanc Primet, P. (2009). Executing distributed applications on virtualized infrastructures specified with the VXDL language and managed by the HIPerNET framework. In *First International Conference on Cloud Computing (CLOUDCOMP 2009)*, Munich, Germany.

[Koslovski et al., 2008] Koslovski, G., Vicat-Blanc Primet, P., and Charão, A. S. (2008). VXDL: Virtual Resources and Interconnection Networks Description Language. In *The Second International Conference on Networks for Grid Applications (GridNets 2008)*, Beijing, China.

[Koslovski et al., 2010] Koslovski, G., Yeow, W.-L., Westphal, C., Truong Huu, T., Montagnat, J., and Vicat-Blanc, P. (2010). Reliablility Support in Virtual Infrastructures. In *2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2010)*, Indianapolis, USA. IEEE CS Press.

[Kottha et al., 2007] Kottha, S., Peter, K., Steinke, T., Bart, J., Falkner, J., Weisbecker, A., Viezens, F., Mohammed, Y., Sax, U., Hoheisel, A., Ernst, T., Sommerfeld, D., Krefting, D., and Vossberg, M. (2007). Medical image processing in MediGrid. In *German e-Science Conference (GES2007)*, Baden-Baden.

[Lingrand et al., 2009a] Lingrand, D., Montagnat, J., and Glatard, T. (2009a). Modeling user submission strategies on production grids. In *International Symposium on High Performance Distributed Computing (HPDC'09)*, pages 121–130, Munchen, Germany.

[Lingrand et al., 2009b] Lingrand, D., Montagnat, J., Martyniak, J., and Colling, D. (2009b). Analyzing the EGEE production grid workload: application to jobs submission optimization. In *14th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'09)*, volume LNCS 5798, pages 37–58, Roma, Italy. Springer.

[Litzkow et al., 1988] Litzkow, M. J., Livny, M., and Mutka, M. W. (1988). Condor - A Hunter of Idle Workstations. In *8th International Conference of Distributed Computing Systems (ICDCS)*, pages 104–111, San Jose, California.

[MacLaren et al., 2006] MacLaren, J., Keown, M. M., and Pickles, S. (2006). Co-Allocation, Fault Tolerance and Grid Computing. In *UK e-Science All Hands Meeting*, pages 155–162, Nottingham, UK.

[Maheswaran et al., 1999] Maheswaran, M., Ali, S., Siegel, H. J., Hensgen, D., and Freund, R. F. (1999). Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems. In *Eighth Heterogeneous Computing Workshop (HCW'99)*, pages 30–44, San Juan , Puerto Rico. IEEE Computer Society.

[Malewicz et al., 2006] Malewicz, G., Foster, I., Rosenberg, A. L., and Wilde, M. (2006). A tool for prioritizing DAGMan jobs and its evaluation. In *Proceedings of the 15th International Symposium on High Performance Distributed Computing (HPDC'06)*, pages 156–167, Paris, France.

[McIlroy, 1968] McIlroy, M. (1968). Mass-produced software components. In *Proceedings of the NATO Conference on Software Engineering*. NATO Science Committee.

[Menascé and Casalicchio, 2004] Menascé, D. A. and Casalicchio, E. (2004). A Framework for Resource Allocation in Grid Computing. In *12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS 2004)*, pages 259–267, Volendam, The Netherlands. IEEE Computer Society.

[Mohamed and Epema, 2008] Mohamed, H. and Epema, D. (2008). KOALA: a co-allocating grid scheduler. *Concurrency and Computation: Practice & Experience (CCPE)*, 20(16):1851–1876.

[Montagnat et al., 2004] Montagnat, J., Bellet, F., Benoit-Cattin, H., Breton, V., Brunie, L., Duque, H., Legré, Y., Magnin, I., Maigne, L., Miguet, S., Pierson, J.-M., Seitz, L., and Tweed, T. (2004). Medical images simulation, storage, and processing on the european datagrid testbed. *Journal of Grid Computing (JOGC)*, 2(4):387–400.

[Montagnat et al., 2006] Montagnat, J., Glatard, T., and Lingrand, D. (2006). Data composition patterns in service-based workflows. In *Workshop on Workflows in Support of Large-Scale Science (WORKS'06)*, Paris, France.

[Montagnat et al., 2009] Montagnat, J., Isnard, B., Glatard, T., Maheshwari, K., and Blay-Fornarino, M. (2009). A data-driven workflow language for grids based on array programming principles. In *Workshop on Workflows in Support of Large-Scale Science (WORKS'09)*, pages 1–10, Portland, USA.

[Moreau et al., 2005] Moreau, L., Zhao, Y., Foster, I., Voeckler, J., and Wilde, M. (2005). XDTM: The XML Data Type and Mapping for Specifying Datasets. In *Advances in Grid Computing - European Grid Conference*, pages 495–505, Amsterdam, The Netherlands. Springer Berlin / Heidelberg.

[Mościcki, 2003] Mościcki, J. T. (2003). Distributed analysis environment for HEP and interdisciplinary applications. *Nuclear Instruments and Methods in Physics Research A*, 502:426–429.

[MPI Forum, 1994] MPI Forum, M. P. I. (1994). MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, TN, USA.

[Nakada et al., 2005] Nakada, H., Matsuoka, S., Seymour, K., Dongarra, J., Lee, C., and Casanova, H. (2005). A GridRPC Model and API for End-User Applications. Technical report, Global Grid Forum (GGF).

[Nemo et al., 2007] Nemo, C., Glatard, T., Blay-Fornarino, M., and Montagnat, J. (2007). Merging overlapping orchestrations: an application to the Bronze Standard medical application. In *International Conference on Services Computing (SCC 2007)*, pages 364–371, Salt Lake City, Utah, USA. IEEE Computer Engineering.

[Oinn et al., 2007] Oinn, T., Li, P., Kell, D. B., Goble, C., Gooderis, A., Greenwood, M., Hull, D., Stevens, R., Turi, D., and Zhao, J. (2007). Taverna/myGrid: Aligning a Workflow System with the Life Sciences Community. In [Taylor et al., 2007], chapter 19, pages 300–319.

[Perera and Ranabahu, 2006] Perera, S. and Ranabahu, A. (2006). Axis2 - The Future of Web Services. Online: http://www.jaxmag.com/itr/Online_artikel/psecom,id,747,nodeid,147.html.

[Plank et al., 1999] Plank, J. S., Beck, M., Elwasif, W., Moore, T., Swany, M., and Wolski, R. (1999). The Internet Backplane Protocol: Storage in the Network. In *Network Storage Symposium (NetStore'99)*, Seattle, WA. Internet2.

[Sakellariou et al., 2005] Sakellariou, R., Zhao, H., Tsiakkouri, E., and Dikaiakos, M. D. (2005). Scheduling Workflows with Budget Constraints. In *CoreGRID Integration Workshop (CGIW2005)*, pages 347–357, Pisa, Italy. Springer-Verlag.

[Sametinger, 1997] Sametinger, J. (1997). *Software Engineering with Reusable Components*. Springer-Verlag New York, Inc., New York, NY, USA.

[Sato et al., 2003] Sato, M., Boku, T., and Takahashi, D. (2003). OmniRPC: a Grid RPC System for Parallel Programming in Cluster and Grid Environment. In *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'03)*, pages 206–213, Tokyo, Japan. IEEE Computer Society.

[Sato et al., 1997] Sato, M., Nakada, H., Sekiguchi, S., Matsuoka, S., Nagashima, U., and Takagi, H. (1997). Ninf: A Network Based Information Library for Global World-Wide Computing Infrastructure. In *International Conference and Exhibition on High-Performance Computing and Networking (HPCN Europe '97)*, pages 491–502. Springer-Verlag.

[Sfiligoi, 2008] Sfiligoi, I. (2008). glideinWMS - A generic pilot-based Workload Management System. *Journal of Physics: Conference Series*, 119(6).

[Shi et al., 2006] Shi, Z., Jeannot, E., and Dongarra, J. (2006). Robust Task Scheduling in Non-Deterministic Heterogeneous Systems. In *IEEE International Conference on Cluster Computing (Cluster 2006)*, pages 1–10, Barcelona, Spain. IEEE CS Press.

[Smith et al., 2000] Smith, W., Foster, I., and Taylor, V. (2000). Scheduling with Advanced Reservations. In *IEEE/ACM 14th International Parallel and Distributed Processing Symposium (IPDPS 2000)*, pages 127–132, Cancun, Mexico.

[Snell et al., 2000] Snell, Q., Clement, M. J., Jackson, D. B., and Gregory, C. (2000). The Performance Impact of Advance Reservation Meta-scheduling. In *International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'00)*, volume 1911, pages 137–153, Cancun, Mexico. Springer-Verlag.

[Sulistio and Buyya, 2004] Sulistio, A. and Buyya, R. (2004). A Grid Simulation Infrastructure Supporting Advance Reservation. In *16th International Conference on Parallel and Distributed Computing and Systems (PDCS 2004)*, pages 1–7, Boston, USA.

[Takefusa et al., 2007] Takefusa, A., Nakada, H., Kudoh, T., Tanaka, Y., and Sekiguchi, S. (2007). GridARS: An Advance Reservation-Based Grid Co-allocation Framework for Distributed Computing

and Network Resources. In *13th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'07)*, pages 152–168, Seattle, WA, USA. Springer Berlin / Heidelberg.

[Tamura et al., 2008] Tamura, Y., Sato, K., Kihara, S., and Moriai, S. (2008). Kemari: Virtual Machine Synchronization for Fault Tolerance. In *2008 USENIX Annual Technical Conference (USENIX'08)*, Boston, Massachusetts. Poster Session.

[Tannenbaum et al., 2001] Tannenbaum, T., Wright, D., Miller, K., and Livny, M. (2001). Condor: A Distributed Job Scheduler. *Beowulf Cluster Computing with Linux*, pages 307–350.

[Taylor et al., 2007] Taylor, I., Deelman, E., Gannon, D., and Shields, M. (2007). *Workflows for e-Science*. Springer-Verlag.

[Thain et al., 2001] Thain, D., Basney, J., Son, S.-C., and Livny, M. (2001). The Kangaroo Approach to Data Movement on the Grid. In *10th IEEE International Symposium on High Performance Distributed Computing (HPDC'01)*, pages 325–333, San Francisco, CA, USA. IEEE Computer Society.

[Topcuoglu et al., 2002] Topcuoglu, H., Hariri, S., and Min-You, W. (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. *International Journal of Supercomputer Applications (IJSA)*, 13(3):260–274.

[Truong Huu et al., 2011] Truong Huu, T., Koslovski, G., Anhalt, F., Montagnat, J., and Vicat-Blanc Primet, P. (2011). Joint Elastic Cloud and Virtual Network Framework for Application Performance-cost Optimization. *Journal of Grid Computing (JOGC)*, 9(1):27–47.

[Truong Huu and Montagnat, 2010] Truong Huu, T. and Montagnat, J. (2010). Virtual resources allocation for workflow-based applications distribution on a cloud infrastructure. In *2nd International Symposium on Cloud Computing (Cloud 2010)*, Melbourne, Australia. IEEE Computer Society.

[Turi et al., 2007] Turi, D., Missier, P., Goble, C., de Roure, D., and Oinn, T. (2007). Taverna Workflows: Syntax and Semantics. In *IEEE International Conference on e-Science and Grid Computing (eScience'07)*, pages 441–448, Bangalore, India.

[Ullman, 1975] Ullman, J. D. (1975). NP-complete scheduling problems. *Journal of Computer and System Sciences (JCSS)*, 10(3):384–393.

[Vadhiyar and Dongarra, 2002] Vadhiyar, S. S. and Dongarra, J. (2002). A Metascheduler For The Grid. In *11th IEEE International Symposium on High Performance Distributed Computing (HPDC 2002)*, pages 343–351, Edinburgh, Scotland. IEEE CS Press.

[Vicat-Blanc Primet et al., 2009a] Vicat-Blanc Primet, P., Anhalt, F., and Koslovski, G. (2009a). Exploring the virtual infrastructure service concept in Grid'5000. In *20th ITC Specialist Seminar on Network Virtualization*, Hoi An, Vietnam.

[Vicat-Blanc Primet et al., 2009b] Vicat-Blanc Primet, P., Roca, V., Montagnat, J., Gelas, J.-P., Mornard, O., Giraud, L., Koslovski, G., and Truong Huu, T. (2009b). A Scalable Security Model

for Enabling Dynamic Virtual Private Execution Infrastructures on the Internet. In *IEEE International Symposium on Cluster Computing and the Grid (CCGRID 2009)*, Shanghai, China. Shanghai Jiaotong University.

[Virtualization, 2009] Virtualization, O. (2009). Open Virtualization: More Innovation without Vendor Lock In. Technical report, Sun MicroSystems Inc.

[Whitaker et al., 2002] Whitaker, A., Shaw, M., and Gribble, S. D. (2002). Scale and performance in the Denali isolation kernel. In *5th Simposium on Operating Systems Design and Implementation (OSDI'02)*, pages 95–209, Boston, Massachusetts. ACM.

[Wieczorek et al., 2005] Wieczorek, M., Prodan, R., and Fahringer, T. (2005). Scheduling of scientific workflows in the ASKALON grid environment. *ACM SIGMOD records (SIGMOD)*, 34(3):56–62.

[Yeow et al., 2010] Yeow, W.-L., Westphal, C., and Kozat, U. C. (2010). Designing and Embedding Reliable Virtual Infrastructures. In *The Second ACM SIGCOMM Workshop on Virtualized Infrastructure Systems and Architectures (VISA 2010)*, New Delhi, India. ACM.

[Yu and Buyya, 2005] Yu, J. and Buyya, R. (2005). A Taxonomy of Workflow Management Systems for Grid Computing. *Journal of Grid Computing (JOGC)*, 3(3-4):171 – 200.

[Yu and Buyya, 2006] Yu, J. and Buyya, R. (2006). Scheduling Scientific Workflow Applications with Deadline and Budget Constraints using Genetic Algorithms. *Scientific Programming Journal*, 14(3-4):217–230.

[Yu et al., 2008] Yu, J., Buyya, R., and Ramamohanarao, K. (2008). Workflow Scheduling Algorithms for Grid Computing. In *Metaheuristics for Scheduling in Distributed Computing Environments*, chapter 5, pages 109–153. Springer.

[Yu et al., 2005] Yu, J., Buyya, R., and Tham, C. K. (2005). Cost-Based Scheduling of Scientific Workflow Application on Utility Grids. In *First International Conference on e-Science and Grid Computing (E-SCIENCE'05)*, pages 140–147, Melbourne, Australia. IEEE Computer Society.

[Zhao and Sakellariou, 2003] Zhao, H. and Sakellariou, R. (2003). An Experimental Investigation into the Rank Function of the Heterogeneous Earliest Finish Time Scheduling Algorithm. In *International Conference on Parallel and Distributed Computing (Euro-Par 2003)*, volume 2790 of *LNCS*, pages 189–194, Klagenfurt, Austria. Springer Berlin / Heidelberg.

[Zhao and Sakellariou, 2006] Zhao, H. and Sakellariou, R. (2006). Scheduling Multiple DAGs onto Heterogeneous Systems. In *15th Heterogeneous Computing Workshop (HCW 2006)*, Rhodes Island, Greece.

[Zhao et al., 2007] Zhao, Y., Hategan, M., Clifford, B., Foster, I., von Laszewski, G., Raicu, I., Stef-Praun, T., and Wilde, M. (2007). Swift: Fast, Reliable, Loosely Coupled Parallel Computation. In *IEEE International Workshop on Scientific Workflows*, Salt-Lake City, Utah, USA.

Workflow-based applications performance and execution cost optimization on cloud infrastructures

Tram Truong Huu

I3S - Projet Modalis, 930 Route des Colles, BP 145, 06903 Sophia-Antipolis Cédex, France.

Cloud computing is increasingly exploited to tackle the computing challenges raised in both science and industry. Clouds provide computing, network and storage resources on demand to satisfy the needs of large-scale distributed applications. To adapt to the diversity of cloud infrastructures and usage, new tools and models are needed. Estimating the amount of resources consumed by each application in particular is a difficult problem, both for end users who aim at minimizing their cost and infrastructure providers who aim at controlling their resources allocation. Although a quasi-unlimited amount of resources may be allocated, a trade-off has to be found between (i) the allocated infrastructure cost, (ii) the expected performance and (iii) the optimal performance achievable that depends on the level of parallelization of the application. Focusing on medical image analysis, a scientific domain representative of the large class of data intensive distributed applications, this thesis proposes a fine-grained cost function model relying on the expertise captured from the application. Based on this cost function model, four resources allocation strategies are proposed. Taking into account both computing and network resources, these strategies help users to determine the amount of resources to reserve and compose their execution environment. In addition, the data transfer overhead and the low reliability level, which are well-known problems of large-scale distributed systems impacting application performance and infrastructure usage cost, are also considered.

The experiments reported in this thesis were carried out on the Aladdin/Grid'5000 infrastructure, using the HIPerNet virtualization middleware. This virtual platform manager enables the joint virtualization of computing and network resources. A real medical image analysis application was considered for all experimental validations. The experimental results assess the validity of the approach in terms of infrastructure cost and application performance control. Our contributions both facilitate the exploitation of cloud infrastructures, delivering a higher quality of services to end users, and help the planning of cloud resources delivery.

Keywords: *Clouds, Workflows, Resources Allocation, Scheduling, IaaS, Virtualization, Bandwidth Control.*

Optimisation des performances et du coût de flots applicatifs s'exécutant sur des infrastructures de cloud

Les infrastructures virtuelles de cloud sont de plus en plus exploitées pour relever les défis de calcul intensif en sciences comme dans l'industrie. Elles fournissent des ressources de calcul, de communication et de stockage à la demande pour satisfaire les besoins des applications à grande échelle. Pour s'adapter à la diversité de ces infrastructures, de nouveaux outils et modèles sont nécessaires. L'estimation de la quantité de ressources consommées par chaque application est un problème particulièrement difficile, tant pour les utilisateurs qui visent à minimiser leurs coûts que pour les fournisseurs d'infrastructure qui visent à contrôler l'allocation des ressources. Même si une quantité quasi illimitée de ressources peut être allouée, un compromis doit être trouvé entre (i) le coût de l'infrastructure allouée, (ii) la performance attendue et (iii) la performance optimale atteignable qui dépend du niveau de parallélisme inhérent à l'application. Partant du cas d'utilisation de l'analyse d'images médicales, un domaine scientifique représentatif d'un grand nombre d'applications à grande échelle, cette thèse propose un modèle de coût à grain fin qui s'appuie sur l'expertise extraite de l'application formalisée comme un flot. Quatre stratégies d'allocation des ressources basées sur ce modèle de coût sont introduites. En tenant compte à la fois des ressources de calcul et de communication, ces stratégies permettent aux utilisateurs de déterminer la quantité de ressources de calcul et de bande passante à réserver afin de composer leur environnement d'exécution. De plus, l'optimisation du transfert de données et la faible fiabilité des systèmes à grande échelle, qui sont des problèmes bien connus ayant un impact sur la performance de l'application et donc sur le coût d'utilisation des infrastructures, sont également prises en considération.

Les expériences exposées dans cette thèse ont été effectuées sur la plateforme Aladdin/Grid'5000, en utilisant l'intergiciel HIPerNet. Ce gestionnaire de plateforme virtuelle permet la virtualisation de ressources de calcul et de communication. Une application réelle d'analyse d'images médicales a été utilisée pour toutes les validations expérimentales. Les résultats expérimentaux montrent la validité de l'approche en termes de contrôle du coût de l'infrastructure et de la performance des applications. Nos contributions facilitent à la fois l'exploitation des infrastructures de cloud, offrant une meilleure qualité de services aux utilisateurs, et la planification de la mise à disposition des ressources virtualisées.

Mots clés : *Infrastructures de Cloud, Flots applicatifs, Allocation de ressources, Ordonnancement, Infrastructure comme un Service, Virtualization, Contrôle de bande passante.*