Application Partitioning and Hierarchical Application Management in Grid Environments

Patrícia Kayser Vargas Mangan[1,2] Inês de Castro Dutra [1] Cláudio F. R. Geyer [3]

[1]COPPE/Engenharia de Sistemas e Computação
Universidade Federal do Rio de Janeiro
[2]Curso de Ciência da Computação
Centro Universitário La Salle
[3]Instituto de Informática
Universidade Federal do Rio Grande do Sul
{kayser,ines}@cos.ufrj.br, geyer@inf.ufrgs.br

June 2004

Resumo

A grid computing environment supports the sharing and coordinated use of heterogeneous and geographically distributed resources. A grid differs from conventional distributed computing in its focus on large-scale resource sharing, innovative applications, and, in some cases, high-performance and/or high-throughput orientation. Therefore, a grid presents challenges such as the control of huge numbers of tasks and their allocation to the grid nodes. In the last years, many works on grid computing environments have been proposed. However, most work presented in the literature can not deal properly with all issues related to application management. Usually, applications are composed of tasks and most systems deal with each individual task as if they are stand-alone applications. Very often, as for example in some application of High Energy Physics, aplications are composed of hierarchical tasks that need to be dealt with altogether. Some problems that are not properly solved include data locality, unnecessary migration of transient data files, and overload of the submit machines (i.e. the machine where the applications are launched). Our work deals with these limitations, focusing on applications that spread a very large number of tasks. The central idea of our work is to have a hierarchical application control mechanism. This control mechanism manages the execution of a huge number of distributed tasks preserving data locality while controlling the load of the submit machines.

Contents

1. Introduction

The main subject of this text is the control of applications in a grid computing environment. We present an ongoing work that deals with resource management limitations, focusing on applications that spread a huge number (thousands) of tasks and manipulate very large amount of data across the grid.

1.1. Motivation

The term *grid computing* [41, 39] was coined in the mid-1990s to denote a distributed computing infrastructure for scientific and engineering applications. The grid can federate systems into a supercomputer far beyond the power of any current computing center [9]. Several works on grid computing have been proposed in the last years [39, 6, 90, 12]. There are also several initiatives in grid Computing at Brazil [47, 23].

A grid computing environment supports sharing and coordinated use of heterogeneous and geographically distributed resources. These resources are made available transparently to the application independent of its physical location as if they belong to a single and powerful logical computer. These resources can be CPUs, storage systems, network connections, or any resource made available due to hardware or software. In the last years, many works on grid computing environments have been proposed [10, 62, 21, 82, 111]. However, most work presented in the literature can not deal properly with all issues related to application management.

Many applications have a high demand for computational resources such as CPU cycles and/or data storage. For instance, research in High Energy Physics (HEP) and Bioinformatics usually requires processing of large amounts of data using processing intensive algorithms. The major HEP experiments for the next years aim to find the mechanism responsible for mass in the universe and the "Higgs" particles associated with mass generation [15]. These experiments will be conducted through collaborations that encompass around 2000 physicists from 150 institutions in more than 30 countries. One of the largest collaborations is the Compact Muon Solenoid (CMS) project. The CMS project estimates that 12-14 Petabytes of data will be generated each year [24].

In Bioinformatics, genomic sequencing is one of the hot topics. The genomic sequences are being made public on a lot of target organisms. A great amount of gene sequences are being stored in public and private databases. It is said that the quantity of stored genomic information should double every eight months. The more the quantity of information increases, the more computation power is required [64].

There are several applications that can run in a grid computing environment. We consider only applications composed by several tasks which can have dependencies through file sharing. We classify those applications in three types, as we present in more detail later on: independent tasks (bag-of-tasks), loosely-coupled tasks (few sharing points), and tightly-coupled tasks (more complex dependencies).

Usually, applications are composed of tasks and most systems deal with each individual task as if they are stand-alone applications. Very often, as for example in some application of High Energy Physics (HEP), they are composed of hierarchical tasks that need to be dealt with altogether, either

because they need some feedback from the user or because they need to communicate. These applications can also present a large-scale nature and spread a very large number of tasks requiring the execution of thousands or hundreds of thousands of experiments. Most current software systems fail to deal with these two problems: (1) manage and control large numbers of tasks; and (2) regulate the submit machine load and network traffic, when tasks need to communicate. One work in the direction of item (1) is that of Dutra *et al.* [30] that reported experiments of inductive logic programming that generated over 40 thousand jobs that required a high number of resources in parallel in order to terminate in a feasible time. However, this work was concentrated on providing an user level tool to control and monitor that specific application execution, including automatic resubmission of failed tasks.

Dealing with huge amounts of data requires solving several problems to allow the execution of the tasks as well as to get some efficiency. One of them is data locality. Some applications are programmed using data file as a means of communication to avoid interprocess communication. Some of the generated data can be necessary only to get the final results, that is, they are intermediate or transient data which are discarded at the end of the computation. Transient data can exist in loosely- and tightly-coupled tasks. We consider that allocating dependent tasks grouped allows to keep data locality. The goal is to get a high data locality so that data transfer costs are minimized.

So, applications that spread a large number of tasks must receive a special treatment for submission and execution control. Submission of a large number of tasks can stall the machine. A good solution is to have some kind of distributed submission control. Besides, monitoring information to indicate application progress must be provided to make the system user friendly. Finally, automatic fault detection is crucial since handling errors manually is not feasible.

Our work deals with these application control limitations, focusing on applications that spread a very large number of tasks. These non trivial applications need a powerful distributed execution environment with many resources that currently are only available across different network sites. Grid computing is a good alternative to obtain access to the needed resources.

1.2. Goals and Contributions

Two open research problems in grid environments are addressed in this work. The first one is the task submission process. Executing in parallel an application which is embarrassingly parallel is usually not complex. However, if the application is composed by a huge number of tasks, the execution control is a problem. Tasks of an application can have dependencies. The user should not need to start such tasks manually. Besides, the number of tasks can be very large. The user should not need to control start and termination of each task. Moreover, tasks should not be started from the same machine to avoid (1) overloading the submit machine, and (2) that the submit machine becomes a bottleneck. In this work we employ a hierarchical application management organization for controlling applications with a huge number of tasks and distribute the task submission among several controllers. We believe that a hierarchy of adaptable and grid-aware controllers can provide efficient, robust, and resilient execution.

The second open problem is the data locality maintenance when tasks are partitioned and mapped to remote resources. In this work, the applications are distributed applications, i.e. applications composed by several tasks. Tasks can be independent or dependent due to file sharing. The application partitioning goal is to group tasks in blocks that are mapped to available processors. This process must keep data locality, which means to minimize data transfer through processors and to avoid unnecessary data transfers. This is fundamental to get good application performance.

Our main goal is to evaluate our hierarchical application management. Our specific goals are the following:

- to study the main works related to application control and resource management systems;
- to design our model aiming to keep it as simple as possible;
- to present it in detail in this text;
- to produce a simulation model to evaluate our design, checking its feasibility;

1.3. Text Organization

The remaining of this text is organized as follows. First, we present basic concepts related to grid computing (Section 2). Then, we analyze concepts and works related to application management (Section 3). We focus mainly in application partitioning. Since the applications will run in a grid environment, we also analyze concepts and works in resource management for grid (Section 4).

We present and analyze our hierarchical model in Section 5. We discuss some of the problems that need to be solved to satisfy user needs, which are

the motivation to our model. We also present our architecture.

Then, we describe some tools that can be used to simulate grid environments and MONARC, the tool chosen for our experiments (Section 6). It also presents our simulation model.

Finally, we conclude this text with our final remarks and future works (Section 7).

2. Grid Computing Systems Concepts

Grid computing is closely related to the distributed systems and network research areas. Grid computing differs from conventional distributed computing in its focus on large-scale resource sharing, innovative applications, and, in some cases, high-performance and/or high-throughput orientation [41, 65].

This section presents basic concepts related to grid computing. First, we present grid computing definitions (Subsection 2.1). Then, we discuss about grid applications (Subsection 2.2), and grid computational systems (Subsection 2.3).

2.1. Grid Computing Definition

An increasing number of research groups have been working in the field of network wide-area distributed computing [28, 49, 107, 11, 53, 74, 95, 29]. They have been implementing middleware, libraries, and tools that allow cooperative use of geographically distributed resources. These initiatives have been known by several names [6, 90] such as metacomputing, global computing, and more recently grid computing.

The term grid computing was coined by Foster and Kesselman [37] as a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities. Actual grid computing efforts can be seen as the third phase of metacomputing evolution as stated in the Metacomputing's paper of Smarr and Catlett [98]: a transparent network that will increase the computational and information resources available to an application. It is also a synonym of metasystem [52] which supports the illusion of a single machine by transparently scheduling application components on processors; managing data migration, caching, transfer, and the masking of data-format differences between systems; detecting and managing faults; and ensuring that users' data and physical resources are protected, while scaling appropriately.

The Global Grid Forum (GGF) [48] is a community-initiated forum of more than 5000 individual researchers and practitioners working on grid technologies. The GGF creates and documents technical specifications, user experiences, and implementation guidelines. The GGF's Grid Scheduling Dictionary [89] defines grid as following: "Grids are persistent environments that enable software applications to integrate instruments, displays, computational and information in widespread locations".

Since there is not a unique and precise definition for the grid concept, we present two attempts to define and check if a distributed system is really a grid system. First, Foster [36] proposes a three points checklist to define grid as a system that:

1. coordinates resources that are not subject to centralized control...
2. ... using standard, open, general-purpose protocols and interfaces...
3. ... to deliver nontrivial qualities of service.

Second, Németh and Sunderam in May 2002 [78] presented a formal definition of what a grid system should provide. They focused on the semantics of the grid and argue that a grid is not just a modification of "conventional" distributed systems but fundamentally differs in semantics. They present an interesting table comparing distributed environments and grids that we transcribe in Table 1 and analyze. A grid can present heterogeneous resources including, for example, sensors and detectors and not only computational nodes. Individual sites belong to different administrative domains. Thus, the user has access to the grid (the pool) but not to the individual sites and access may be restricted. User has little knowledge about each site due to administrative boundaries, and even due to the large number of resources. Resources in the grid typically belong to different administrative domains and also to several trust domains. Finally, while conventional distributed environments tend to be static, except due to faults and maintenance, grids are dynamic by definition.

Németh and Sunderam in April 2002 [77] also made an informal comparison of distributed systems and computational grids.

Besides, Foster *et al.* [41] presents the specific problem that underlies the grid concept as coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations. The sharing is, necessarily, highly controlled, with resource providers and consumers defining clearly and carefully what is shared, who is allowed to share, and the conditions under which sharing occurs. A set of individuals and/or institutions defined by such sharing rules form what they call a virtual organization.

	Conventional distributed environments	Grids
1	a virtual pool of computational nodes	a virtual pool of resources
2	a user has access (credential) to all the nodes in the pool	a user has access to the pool but not to individual sites
3	access to a node means access to all resources on the node	access to a resource may be restricted
4	the user is aware of the capabilities and features of the nodes	the user has little knowledge about each site
5	nodes belong to a single trust domain	resources span multiple trust domains
6	elements in the pool 10-100, more or less static	elements in the pool 1000-10000, dynamic

Table 1. Comparison of conventional distributed environments and grids [78]

Though there might be a strong relation among the entities building a virtual organization, a grid still consists of resources owned by different, typically independent organizations. Heterogeneity of resources and policies is a fundamental result of this [92].

There are several reasons for programming applications on a computational grid. Laforenza [65] presents some examples: to exploit the inherent distributed nature of an application, to decrease the turnaround/response time of a huge application, to allow the execution of an application which is outside the capabilities of a single (sequential or parallel) architecture, and to exploit affinities between an application component and grid resources with specific functionalities.

In the next subsections we give some of the classifications found in the literature for grid applications and systems.

2.2. Classifying Grid Applications

Foster and Kesselman [37] identify five major application classes for grid environment:

- *Distributed supercomputing* applications use grid to aggregate substantial computational resources in order to tackle problems that cannot be solved on a single system. They are very large problems, such as simulation of complex physical processes, which need lots of resources like CPU and memory.
- *High-Throughput Computing* uses grid resources to schedule a large number of loosely coupled or independent tasks, with the goal of putting unused processor cycles to work. The Condor system [107] has been dealing with this kind of application, as for example, molecular simulations of liquid crystal and biostatistical problems solved with inductive logic programming.
- *On-Demand Computing* applications use grid capabilities to meet short-term requirements for resources that cannot be cost-effectively or conveniently located locally. For example, one user doesn't need to buy a supercomputer to run an application once a week. Another example, the processing of data from meteorological satellites can use dynamically acquired supercomputer resources to run a cloud detection algorithm.
- *Data-Intensive Computing* applications, where the focus is on synthesizing new information from data that is maintained in geographically distributed repositories, digital libraries, and databases. This process is often computationally and communication intensive, as expected in future high energy physics experiments.
- *Collaborative Computing* applications are concerned primarily with enabling and enhancing human-to-human interactions. Many collaborative applications are concerned with enabling the shared use of computational resources such as data archives and simulations. For example, the CAVE5D [19] system supports remote, collaborative exploration of large geophysical data sets and the models that generated them.

Some examples of applications that are representative of these main classes of applications are the following:

Monte Carlo Simulation Monte Carlo experiments are sampling experiments, performed on a computer, usually done to determine the distribution of a statistic under some set of probabilistic assumptions [51]. The name Monte Carlo comes from the use of random numbers.

In HEP, there are several applications to this technique. The results can usually be obtained running several instances independently with different parameters, which can be easily executed in parallel.

Biostatistic Problems using ILP Techniques Dutra *et al.* [30] reported experiments of inductive logic programming (ILP) to solve biostatistic problems.

Since, it is a machine learning problem, it had two main phases: experimentation and evaluation. During the experimentation phase, the user wants to run a learner, adjusting the learner parameters, using several datasets, and sometimes, repeating the learning process several times. During the evaluation phase, the user is interested in knowing how accurate a give model is, and which one of the experiments gave the most accurate result.

Typically, these experiments, in both phases, need to run in parallel. Due to the highly independent nature of each experiment, all machine learning process can be trivially parallelized. However, all experiments of a phase must be finished before proceeding to the next phase.

Finite Constraint Satisfaction Problems Finite Constraint Satisfaction Problems (CSP) [3] usually describe NP-complete search problems. Algorithms exist, such as arc-consistency algorithms, that help to eliminate inconsistent values from the solution space. They can be used to reduce the size of the search space, allowing to find solutions for large CSPs.

Still, there are problems whose instance size make it impossible to find a solution with sequential algorithms. Concurrency and parallelization can help to minimize this problem because a constraint network generated by a constraint program can be split among processes in order to speed up the arc-consistency procedure. The dependency between the constraints can be represented usually as a complex and high-connected graph.

Considering the examples presented and other presented in the literature we certainly have a wide variety of applications that can profit from grid power. Because we intend to partition the applications to map their tasks to resources, we propose the following taxonomy for distributed applications in grid:

- **independent tasks** The simplest kind of application is the one usually called bag-of-tasks. It characterizes applications where all tasks are independents. One example are the Monte Carlo simulations typically used in HEP experiments.
- **loosely-coupled tasks** This kind of graph is characterized by few sharing points. It is typically characterized by an application divided in phases. One example are the ILP experiments mentioned.
- **tightly-coupled tasks** Highly complex graphs are not so often, but are more difficult to be partitioned. Constraint logic programming applications can fall in this category.

We present in Figure 1 visual representations of application graphs for each of the presented categories.

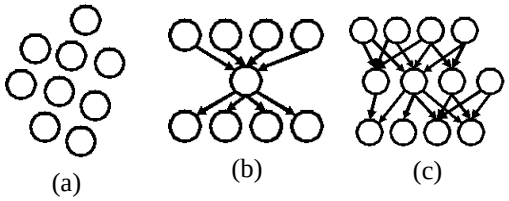

Figure 1. Grid application classes: (a) independent tasks, (b) loosely-coupled tasks, and (c) tightly-coupled tasks

2.3. Classifying Grid Systems

A current classification of grid computing systems is computational and data grid [82]. The *Computational Grid* focuses on reducing execution time of applications that require a great number of computer processing cycles or on execution of applications that can not be executed sequentially. The *Data Grid* provides the way to solve large scale data management problems. Data intensive applications such as High Energy Physics and Bio-informatics require both Computational and Data Grid features.

Krauter *et al.* [62] presents a similar taxonomy for grid systems, which includes a third category, the service grid:

- The *computational grid* category denotes systems that have higher aggregate computational capacity available for single applications than the capacity of any constituent machine in the system. It is subdivided into *distributed supercomputing* (application execution in parallel on multiple machines) and *high throughput* (stream of jobs). Computational grid can also be defined as a "large-scale high performance distributed computing environment that provide access to high-end computational resources" [89].
- The *data grid* is a terminology used for systems that provide an infrastructure for synthesizing new information from data repositories that are distributed in a wide area network.
- The *service grid* is the name used for systems that provide services that are not provided by any single local machine. This cat-

egory is further divided as *on demand* (aggregate resources to provide new services), *collaborative* (connect users and applications via a virtual workspace), and *multimedia* (infrastructure for real-time multimedia applications).

Currently, the most accepted technology for building computational grids are based on services.

A grid system can also be classified according to historical or other technical characteristics. Roure *et al.* [90] identify three generations in the evolution of grid systems. The **first generation** includes the forerunners of grid computing as we recognize it today and were projects to connect supercomputing sites. At the time this approach was known as metacomputing. The early to mid 1990s mark the emergence of the early metacomputing or grid environment.

Two representative projects in the first generation were FAFNER and I-WAY. FAFNER (Factoring via Network-Enabled Recursion) [33, 32] was created through a consortium to make RSA130 factorization using a numerical technique called Number Field Sieve. I-WAY (*The Information Wide Area Year*) [28] was an experimental high performance network that connected several high performance computers spread over seventeen universities and research centers using mainly ATM technology. These projects differ in some ways: (a) FAFNER was concerned with one specific application while I-WAY could execute different applications, mainly high performance applications; (b) FAFNER could use almost any kind of machine while I-WAY assumed high-performance computers with a high bandwidth and low latency network. Nevertheless, both had to overcome a number of similar obstacles, including communications, resource management, and the manipulation of remote data, to be able to work efficiently and effectively. Both projects are also pioneers on grid computing systems and helped to develop several second generation projects. FAFNER was the precursor of projects such as SETI@home (*The Search for Extraterrestrial Intelligence at Home*) [95] and Distributed.Net [29]. I-WAY was the predecessor of the Globus [93] and the Legion [53] projects.

The **second generation** projects have a focus on middleware to support large scale data and computation. The two most representative projects are Legion and Globus.

The Legion object oriented system [53, 21] was developed at the University of Virginia and it is now a commercial product of Avaki. Legion is a middleware with several components, such as, resource management, data management, security, etc. In Legion, active objects communicate via remote method invocation. Some system responsibilities are delegated to the user level, as for example, Legion classes create and locate their objects as well as are responsible for selecting the appropriate security mechanisms and the objects allocation policy.

The Globus Project has been developed by the Argonne National Laboratory, University of Southern California's Information Sciences Institute, and University of Chicago. The most important result of the Globus Project is the Globus Toolkit [49, 93]. The Globus Toolkit (GT) is an open source software. The GT version 2 can be classified as a second generation system since it is mainly a set of components that compose a middleware.

The GT2 design is highly related to the architecture proposed by Foster *et al.* [41]. This open and extensible architecture aims to identify requirements to a generic class of components and has four layers under the Application layer. Figure 2 [41] shows this architecture and its relationship with the Internet protocol architecture.

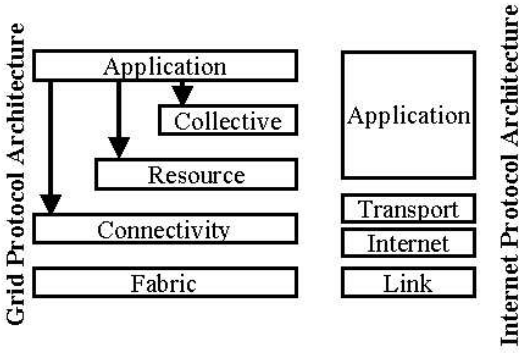

Figure 2. The layered grid architecture and its relationship to the Internet protocol architecture [41]

The *Resource* and *Connectivity* protocols are the main parts and aid the sharing of individual resources. The protocols of these layers are designed to be implemented on top of several resources defined in the lower layer, called *Fabric*, and to be used to build several global services and specific application behavior in the *Collective* layer. The *Collective* layer deals with the coordinated use of multiple resources.

During the period where projects were identified as belonging to the second generation, we

highlight two important publications. These publications emphasized the importance of middleware construction for a grid environment and presented the main guidelines and concepts for exploiting this kind of environment. One is the already referred "Grid Anatomy" [41], which starts using the term grid. The "Metasystems" paper [52] is the other paper that, actually, was published first. In this paper, the authors state that the challenge to the computer science community is to provide a solid, integrated middleware foundation on which to build wide-area applications.

Finally, the **third generation** is the current generation where the emphasis shifts to distributed global collaboration, a service oriented approach, and information layer issues.

In the context of the third generation is the *Open Grid Services Architecture* (OGSA) [110] proposal that aims to define a new common and standard architecture for grid-based applications. The version 3 of the Globus Toolkit (GT3) has a new philosophy of grid services and implements the *Open Grid Service Infrastructure* (OGSI) [40]. The OGSI is a formal and technical specification of the concepts described in OGSA, including Grid Services.

Similar to the GT3 philosophy is the Semantic Grid proposal [91]. This architecture adopts a service-oriented perspective in which distinct stakeholders in the scientific process, represented as software agents, provide services to one another, under various service level agreements, in various forms of marketplace.

2.4. Considerations

We conclude this section by emphasizing that grid is the result of several years of research. Figure 3 illustrates the main grid related systems and their time line. In this figure, PBS (Portable Batch System) [13] and Condor [104] are both cluster schedulers. They are presented to stress that most of the current local schedulers used in grid environments are mature projects being developed and maintained for several years. Y-Way and FAFNER are first generation systems while Globus, Legion, Seti@home and Distributed.net are second generation. Condor-G is an "evolution" of Condor and Globus and will be discussed later on.

There are several open research problems, and this work proposes solutions to some of them.

An important issue to allow the application execution in a grid environment is the application management. Application management, as we present in the next section, is concerned with questions such as how to partition and describe the application.

Once the application is partitioned, one of the main challenges is the mapping of the application to the large number of geographically distributed resources. Solutions to these problems will be discussed in the subsequent sections.

3. Application Management

In this section, we discuss application management related issues. According to our taxonomy presented in Subsection 2.2, an application can be represented as a graph, where each node represents a task and the edges represent the task precedence order. This graph needs to be represented in some format or syntax. In the literature, usually, the user uses description languages such as VDL [42] and the DAGMan language [107]. Once the application is properly represented as a graph, this needs to be managed to dispatch the tasks in the right order. In this section, we discuss how to partition the graph (Subsection 3.1) and how to represent it (Subsection 3.2).

3.1. Application Partitioning

One of the characteristics of grid applications is that they usually spread a very large number of tasks. Very often, these tasks present dependencies that enforce a precedence order. In a grid, as in any distributed environment, one important issue to get performance is how to distribute the tasks among resources. Although research in scheduling is very active, efficient partitioning of applications to run in a grid environment is an emerging research topic. A distributed application can be partitioned through grouping related tasks aiming to decrease communication costs. The partitioning should be such that dependencies between tasks and data locality are maintained. Few partitioning works deal specifically with the high resource heterogeneity and dynamic nature of grid environments.

In the next sections, we discuss several application partitioning related issues. First we present the partitioning problem definition in our work (Subsection 3.1.1). Then, we present some related works on graph partitioning techniques applied to the application partitioning problem (Subsection 3.1.2).

3.1.1. Problem Definition Usually, distributed applications can be expressed as a graph $G = (N, E)$, where N is a set of weighted nodes representing tasks and E is the set of weighted edges that represent dependencies between tasks [35, 63, 56]. This graph is usu-

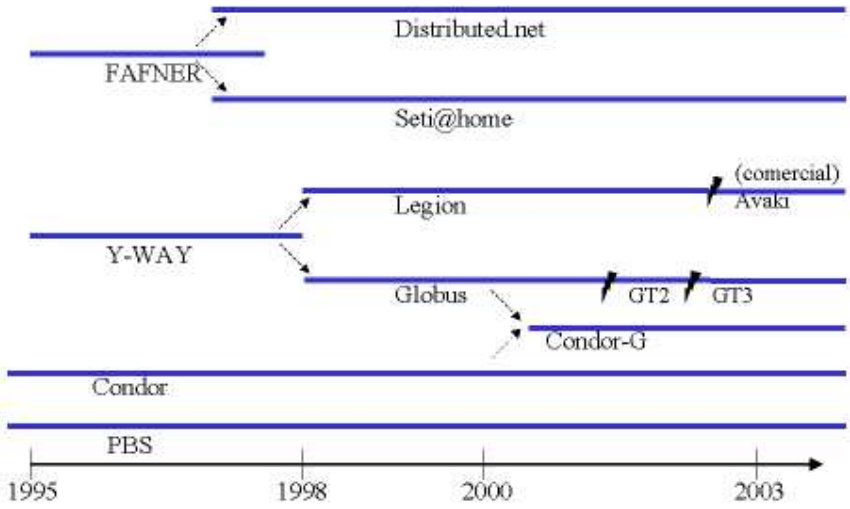

Figure 3. Main grid related systems timeline

ally undirected [63] and is sometimes a Directed Acyclic Graph (DAG) [107].

There are different applications that can be executed in a grid environment. Some applications have only independent tasks and are usually called *bag-of-tasks* applications. They are equivalent to graphs without edges. In this case, partitioning is simpler. Dependent tasks introduce complexity to the problem because tasks should be grouped under predefined criteria to obtain good performance.

Application partitioning can be defined as following: tasks must be placed onto machines in such a way as to minimize communication and achieve best overall performance of the application [57]. Thus, application partitioning, in our work, is to divide the application task graph in subgraphs such as they can be allocated to different available processors efficiently. Efficiency can be measured considering the interprocessor communication and the execution time. If they are kept low, the efficiency is high. Thus, application partitioning can be done using graph partitioning techniques.

Graph Partitioning can be defined as the problem of dividing a set of nodes of a graph into disjoint subsets of approximately equal-weight such that the number of edges with end points in different subsets is minimized. Graph partitioning is an NP-hard problem [46], thus heuristics must be applied.

3.1.2. Related Graph Partitioning Techniques The standard graph partitioning approach divides the nodes into p equally weighted sets while minimizing interprocessor communication (crossing edges between partitions). In other words, traditional graph partitioning algorithms compute a k-way partitioning of a graph such that the number of edges that are cut by the partitioning is minimized and each partitioning has an equal number of nodes.

Hogstedt *et al.* [57] consider a graph $G = (N, E, M)$, with a set of nodes N, a set of edges E, and a set of distinguished nodes $M \subset N$, denoting machine nodes. Each edge has a weight, denoted w_e for an edge $e \in E$. Each node $n \in N$ can be assigned to any of the machine nodes $m \in M$. The machine nodes M cannot be assigned to each other. Any particular assignment of the nodes of N to machines M is called a cut, in which the weight of the cut is the sum of the weights of edges between nodes residing on two different machines. The minimal cut set of a graph minimizes the interprocessor communication.

They present five heuristics to derive a new graph, which is then partitioned aiming to obtain the minimal cut (min-cut) set for the original graph. The five heuristics are the following: (1) *dominant edge*: there is a min-cut not containing the heaviest edge e, so we can contract [1] e to obtain a new graph G'; (2) *independent net*: if the communication graph can be broken into two or more independent nets, then the min-cut of the graph can be obtained by combining the min-cut of each net; (3) *machine cut*: let a machine cut M_i be the set of all edges between a machine m_i and non machine nodes N. Let W_i be the sum of the weight of all edges in the machine cut M_i. Let the W_i's

1 The contraction of $(x, y) \in N$ corresponds to replacing the vertex x and y by a new vertex z, and for each vertex v not in (x, y) replacing any edge (x, v) or (y, v) by the edge (z, v). The rest of the graph remains unchanged. If the contraction results in multiple edges from node z to another node v they are combined and their weights added. [57]

be sorted so that $W_1 \geq W_2 \geq W_3 \geq \ldots$. Then any edge which has weight greater than W_2 cannot be present in the min-cut. (4) *zeroing*: if a node n has edges to each of the m machines with weights $w_1 \leq w_2 \leq \ldots \leq w_m$, we can reduce the weights of each of the m edges from n to the machines by w_1 without changing min-cut assignment; (5) *articulation point*: nodes that would be disconnected from all machines if node n was deleted cannot be in min-cut, thus they can be contracted.

Hendrickson *et al.* [56] also consider that minimizing the number of graph edge cut minimizes the volume of communication. But they argue that the partitioning and mapping should be generated together to also reduce message congestion. Thus, they adapt an idea of the circuit placement community called terminal propagation to the problem of partitioning data structures among processors of a parallel computer. The basic idea of terminal propagation is to associate with each vertex in the subgraph being partitioned a value which reflects its net preference to be in a given quadrant board. In parallel computing, the quadrants represent processors or sets of processors. Their major contribution is a framework for coupling recursive partitioning schemes to the mapping problem.

Karypis and Kumar [61] extend the standard partitioning problem by incorporating an arbitrary number of balancing constraints. A vector of weights is assigned to each vertex, and the goal is to produce a k-partitioning such that the partitioning satisfies a balancing constraint associated with each weight, while attempting to minimize the edge-cut.

Hendrickson and Kolda [55] argue that the standard graph partitioning approach minimizes the wrong metrics and lacks expressibility. One of the metrics is minimizing edge cuts. They present the following flaws of the edge cut metric: (1) edges cuts are not proportional to the total communication volume; (2) it tries to (approximately) minimize the total volume but not the total number of messages; (3) it does not minimize the maximum volume and/or number of messages handled by any single processor; (4) it does not consider distance between processors (number of switches the message passes through). To avoid message contention and improve the overall throughput of the message traffic, it is preferable to have communication restricted to processors which are near each other.

Despite the limitations of edge cut metric, the authors arguee that the standard approach is appropriated to applications whose graph has locality and few neighbors.

Hendrickson and Kolda [55] also argue that the undirected graph model can only express symmetric data dependencies. An application can have the union of multiple phases, and this cannot generally be described via an undirected graph.

Kumar *et al.* [63] propose the MiniMax scheme. MiniMax is a multilevel graph partitioning scheme developed for distributed heterogeneous systems such as the grid, and differs from existing partitioners in that it takes into consideration heterogeneity in both the system and workload graphs. They consider two weighted undirected graphs: a workload graph (to model the problem domain) and a system graph (to model the heterogeneous system).

Finally, an interesting work is the ARMaDA framework proposed by Chandra and Parashar [20]. It does not perform DAG partitioning. ARMaDA is defined by its authors as an adaptative application-sensitive partitioning framework for structured adaptative mesh refinement (SAMR) applications. The goal is to adaptively manage dynamic applications on structured mesh based on the runtime state. The authors argue that no single partitioning scheme performs the best for all types of applications and systems:

> "Even for a single application, the most suitable partitioning technique and associated partitioning parameters depend on input parameters and the application's runtime state. This necessitates adaptative partitioning and runtime management of these dynamic applications using an application-centric characterization of domain-based partitioners." [20]

We believe that this statement applies to DAG partitioning problems as well.

3.2. Representing an Application as a Graph

As far as we know, in the grid literature, the only systems that deal with task precedence are Chimera [42] and DAGMan [107].

In these systems, the user needs to specify dependencies among tasks using a description language. In DAGMan, the user explicitly specifies the task dependence graph, where the nodes are tasks and edges represent dependencies that can be through data files or simply control dependencies. In Chimera, the user only needs to specify the data files manipulation.

We use in this section the DAG example of Figure 4 to illustrate how DAGs are expressed in both systems.

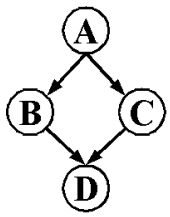

Figure 4. Simple DAG example

3.2.1. DAGMan The Directed Acyclic Graph Manager (DAGMan) [26, 107, 104] manages dependencies between tasks at a higher level invoking the Condor Scheduler to execute the tasks. Condor, as we will present in Subsection 4.4.3, finds resources for the execution of tasks, but it does not schedule tasks based on dependencies. DAGMan submits jobs to Condor in an order represented by a DAG and processes the results. Presently, DAGMan is a stand-alone command line application.

DAG Representation DAGMan receives a file defined prior to submission as input, which describes the DAG. Each node (task) of this DAG has a Condor submit description file associated to be used by Condor. As DAGMan submits jobs to Condor, it uses a single Condor log file to enforce the ordering required for the DAG. DAGMan is responsible for submitting, recovery, and reporting for the set of programs submitted to Condor.

The input file used by DAGMan specifies four items: (1) a list of the tasks in the DAG. This serves to name each program and specify each program's Condor submit description file; (2) processing that takes place before submission of any program in the DAG to Condor or after Condor has completed execution of any program in the DAG; (3) description of the dependencies in the DAG; and (4) number of times to retry if a node within the DAG fails. The items 2 and 4 are optional.

Figure 5 presents the DAGMan input file that describes the DAG example of Figure 4. The first four lines describe the tasks. Each task is described by a single line called a *Job Entry*: `job` is the keyword to indicate it is a job entry, the second element is the name of the task and the third is the name of the Condor submit file. Thus, a job entry maps a job name to a Condor submit description file.

A job entry can also have the keyword `DONE` at the end, which identifies a job as being already completed. This is useful in situations where the user wishes to verify results, but does not require that all jobs within the dependency graph to be executed. The `DONE` feature is also utilized when an error occurs causing the DAG to not be completed. DAGMan generates a Rescue DAG, a DAGMan input file that can be used to restart and complete a DAG without re-executing completed programs [26].

```
#
# first_example.dag
#
Job  A  A.condor
Job  B  B.condor
Job  C  C.condor
Job  D  D.condor
PARENT A CHILD B C
PARENT B C CHILD D
```

Figure 5. DAGMan input file

The last two lines in Figure 5 describe the dependencies within the DAG. The `PARENT` keyword is followed by one or more job names. The `CHILD` keyword is followed by one or more job names. Each child job depends on every parent job on the line. A parent node must be completed successfully before any child node may be started. A child node is started once all its parents have successfully completed.

Figure 6 presents four Condor job description files to describe the four tasks of the DAG example. In this example, each node in a DAG is a unique executable, each with a unique Condor submit description file.

Figure 7 is an alternative of code to implement the DAG example. This example uses the same Condor submit description file for all the jobs in the DAG. Each node within the DAG runs the same program (`/path/dag.exe`).

The `$(cluster)` macro is used to produce unique file names for each program's output. `$(cluster)` is a macro, which supplies the number of the job cluster. A cluster is a set of jobs specified in the Condor submit description file through a queue command. In DAGMan, each job is submitted separately, into its own cluster, so this provides unique names for the output files. The number of the cluster is a sequential number associated with the number os submissions the user had done. For example, in the tenth Condor submission the cluster number will be called "10", and the first task of this cluster "1" or more precisely "10.1". In our example, if task A receives a cluster number "10", its output file will be called "dag.out.10", and the output file for task B will be called "dag.out.11" due to the order of the task definitions in the DAGMan input file.

```
#
# task A
#
executable = A
input   = test.data
output  = A.out
log     = dag.log
Queue
```

```
#
# task B
#
executable = B
input   = A.out
output  = B.out
log     = dag.log
Queue
```

```
#
# task C
#
executable = C
input   = A.out
output  = C.out
log     = dag.log
Queue
```

```
#
# task D
#
executable = D
input   = B.out C.out
output  = final.out
log     = dag.log
Queue
```

Figure 6. Condor submit description files for DAG example

We could also use the `$(cluster)` macro to run a different program for each task. For example, the first line could be replaced by the following statement: `executable = /path/dag_${cluster}.exe`

This example is easier to code but less flexible. A problem is how to specify the inputs. For example, task B should receive as input the output of task A, whose name is dependent on the variable `cluster`. But, the Condor DAGMan description language does not support arithmetics to allow a code like `input=dag.out.{$(cluster)-1}`.

An alternative to allow the input description in this example is to use a VARS entry in the DAGMan input file. Each task would have a VARS entry to define an input file name (e.g. `VARS A inputfile="test.data"`). The Condor submit file would have an extra line: `input = ${inputfile}`.

Besides, two limitations exist in the definition of the Condor submit description file to be used by

```
#
# second_example.dag
#
Job  A  dag_job.condor
Job  B  dag_job.condor
Job  C  dag_job.condor
Job  D  dag_job.condor
PARENT A CHILD B C
PARENT B C CHILD D
```

```
#
# dag_job.condor
#
executable   = /path/dag.exe
output       = dag.out.$(cluster)
error        = dag.err.$(cluster)
log          = dag_condor.log
queue
```

Figure 7. Another solution for DAG example: DAGMan input file and Condor submit description file

DAGMan. First, each Condor submit description file must submit only one job (only one `queue` statement with value one or no parameter). The second limitation is that the submit description file for all jobs within the DAG must specify the same log. DAGMan enforces the dependencies within a DAG, as mentioned, using the events recorded in the log file produced by the job submission to Condor.

Partitioning The DAGMan literature does not present any algorithm for doing partitioning. A job is submitted using Condor as soon as it is detected that it does not have parent jobs waiting to be submitted.

Management A DAG is submitted using the program *condor_submit_dag*. The DAGMan program itself runs as a Condor job. The *condor_submit_dag* program produces a submit description file. An optional argument to *condor_submit_dag*, *-maxjobs*, is used to specify the maximum number of Condor jobs that DAGMan may submit to Condor at one time. It is commonly used when there is a limited amount of input file staging capacity. Thus the user must be aware of these limitations to avoid problems.

Before executing the DAG, DAGMan check the graph for cycles. If it is acyclic it proceeds. The next step is to detect jobs that can be submitted.

The submission of a child node will not take place until the parent node has successfully completed. There is no ordering of siblings imposed by the DAG, and therefore DAGMan does not impose an ordering when submitting the jobs to Con-

dor. For instance, in the previous example, jobs B and C will be submitted to Condor in parallel.

3.2.2. Chimera The Chimera Virtual Data System (VDS) [43, 42, 5, 27] has been developed in the context of the GriPhyN project [54]. The GriPhyN (Grid Physics Network) project has a team of information technology researchers and experimental physicists, which aims to provide infrastructure to enable Petabyte-scale data intensive computation.

Chimera handles the information of how data is generated by the computation. Chimera provides a catalog that can be used by application environments to describe a set of application programs ("transformations"), and then track all the data files produced by executing those applications ("derivations").

DAG Representation The Chimera input consists of transformations and derivations described in the Virtual Data Language (VDL). VDL comprises data definition and query statements. We will concentrate our description on data definition issues. Note that, in Chimera's terminology, a transformation is an executable program and a derivation represents an execution of a transformation (it is analogous, respectively, to program and process in the operating system terminology).

Figure 8 presents VDL code example that expresses the DAG example of Figure 4. The first two statements define transformations (`TR`). The first `TR` statement defines a transformation named `calculate` that reads one input file (`input a`) and produces one output file (`output b`). The `app` statement specifies the executable name. The keyword `vanilla` indicates one of the execution modes of Condor (Vanilla Universe as we present in Subsection 4.4.3). The `arg` statements usually describe how the command line arguments are constructed. But, in this example, the special argument `stdin` and `stdout` are used to specify a filename into which, respectively, the standard input and output of the application woulde redirected.

The `DV` statements define derivations. The string after a `DV` keyword names the transformation to be invoked. Actual parameters in the derivation and formal parameters in the transformation are associated. Note that these four `DV` statements define, respectively, tasks A, B, C, and D of the DAG example shown in Figure 4. However, there is no explicit task dependency in this code. The DAG can be constructed using the data dependency chain expressed in the derivation statements.

Partitioning and Management Chimera uses a description of how to produce a given logical file. The

```
TR calculate{ output b, input a} {
  app vanilla = "generator.exe";
  arg stdin = ${output:a};
  arg stdout = ${output:b};
}
TR analyze{ input a[], output c} {
  app vanilla = "analyze.exe";
  arg files  = ${:a};
  arg stdout = ${output:a2};
}
DV calculate { b=@{output:f.a},
               a=@{input:test.data} };
DV calculate { b=@{output:f.b},
               a=@{input:f.a} };
DV calculate { b=@{output:f.c},
               a=@{input:f.a} };
DV analyze{ a=[ @{input:f.b},
                @{input:f.c} ],
            c=@{output:f.d} };
```

Figure 8. DAG example expressed in Chimera VDL

description is stored as an abstract program execution graph. This abstract graph is turned into an executable DAG for DAGman by the Pegasus planner which is included in the Chimera VDS distribution. Chimera does not control the DAG execution: DAGMan is used to perform this control.

Pegasus (Planning for Execution in Grids) [84, 27] is a system that can map and execute workflows [2]. Pegasus receives an abstract workflow description from Chimera, produces a concrete workflow, and submits it to Condor's DAGMan for execution. The abstract workflow describes the transformations and data in terms of their logical names. The concrete workflow, which specifies the location of the data and the execution platforms, is optimized by Pegasus: if data described within an abstract workflow already exist, Pegasus reuses them and thus reduces the complexity of the concrete workflow.

3.3. Considerations

As we presented in this section, tasks launched by the user have some kind of dependence on each other. Systems like Chimera and DAGMan allow the user to specify, and, in the case of DAGMan, control, dependencies among tasks. Although they deal with task dependency, they do not handle data locality. Data locality should be preserved in order to avoid unnecessary data transfer, and consequently, reduce network traffic. As the available

2 Workfbw is usually related to automation of a business process. In Pegasus context, workfbw represent the computation sequence that needs to be performed to analyze scientifi c datasets.

systems do not deal with this issue, transient files can be unnecessarily circulating in the network.

In the next section we discuss resource management issues that is another topic fundamental for computational grid environments. We consider that each domain in the grid has a local RMS which can be accessed remotely to allow application scheduling decisions. Using an RMS support, an application can be executed in the grid through a metascheduler.

4. Resource Management Systems

A central part of a distributed system is resource management. In a grid, a resource manager manages the pool of resources that are available and that can include resources from different providers. The managed resources are mainly the processors, network bandwidth, and disk storage. Applications are executed using RMS information and control.

This section deals with the RMS subject. It presents some initial background (Subsections 4.1 and 4.2), and then some scheduling systems (Subsections 4.3 and 4.4). We conclude with an analysis of the presented concepts and systems (Subsection 4.5).

4.1. Resource Management Issues

Scheduling is one of the most important and interesting research topics in the distributed computing area. Casavant & Kuhl [18] consider scheduling as a resource management problem. This management is basically a mechanism or policy used to efficiently and effectively manage the access to and use of a resource by its various consumers. On the other hand, according to the GGF's Grid Scheduling Dictionary [89], scheduling is the "process of ordering tasks on computer resources and ordering communication between tasks". Thus, both applications and system components must be scheduled.

Two key concepts related to scheduling and resource management are task and job. They can be defined as follows according to Roehrig *at al.* [89]:

- **Task** is a specific piece of work required to be done as part of a job or application.
- **Job** is an application performed on high performance computer resources. A job may be composed of steps/sections of individual schedulable entities.

Implementation of the scheduling mechanisms can be part of a job management system. The Job Management System (JMS) [60] generally plays three roles, often as separate modules: Queuing, Scheduling, and Resource Management.

The *Queuing* role has traditionally been played by batch systems.

The *Scheduling* role is the process of selecting which jobs to run, according to a predetermined policy.

The *Resource Management* refers to the monitoring, tracking, and reservation of system resources; and enforcement of usage policy.

Gaj *et al.* [100] consider that the objective of a JMS is to let users execute jobs in a non-dedicated cluster of workstations. In this view, the queue element is not explicitly present in the system architecture and the Resource Management has two modules: *Resource Monitor* and *Job Dispatcher*. Figure 9 [60] presents the interaction of these components.

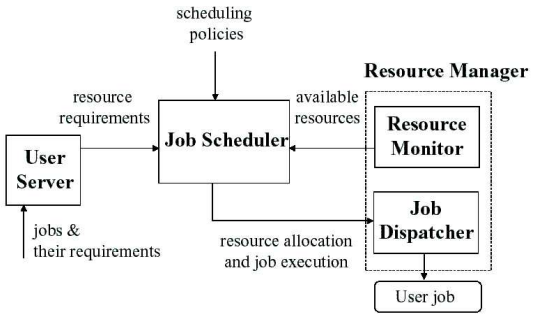

Figure 9. Job management system components [60]

Therefore, independent of the JMS architecture considered, the Resource Management System (RMS) is a central part of a distributed system. Note that resource management is traditionally an operating system problem. Resource management in a distributed system differs from that in a centralized system in a fundamental way [103]: centralized systems always have tables that give complete and up-to-date status information about all the resources being managed; distributed do not. The problem of managing resources without having accurate global state information is very difficult. Managing large-scale collections of resources poses new challenges. The grid environment introduces five resource management problems [25]: (1) site autonomy: resources are typically owned and operated by different organizations, in different administrative domains; (2) heterogeneous substrate: sites may use different local RMS or the same systems with different configurations; (3) policy extensibility: RMS must support development of new application domain-specific management mecha-

nisms, without requiring changes to code installed at participating sites; (4) co-allocation: some applications have resource requirements that can be satisfied only by using resources simultaneously at several sites; and (5) online control: RMS must support negotiation to adapt application requirements to resource availability.

An RMS for grids can be implemented in different ways. But, it may not be possible for the same scheduler to optimize application and system performance. Thus, Krauter *et al.* [62] states that a grid RMS is most likely to be an interconnection of RMSs that are cooperating with one another within an accepted framework. To make the interconnections possible, some interfaces and components were defined in an abstract model. Figure 10 [62] shows the abstract model structure. Krauter *et al.*'s model has four interfaces:

- *resource consumer interface*: used to communicate with actual applications or another RMS that implements a higher layer;
- *resource provider interface*: communicates with an actual resource or another RMS that implements a lower layer;
- *resource manager support interface*: obtains access to services such as naming and security;
- *resource manager peer interface*: provides protocols to connect with other RMSs. Several protocols may be supported by this interface, as for example resource discovery, resource dissemination, trading, resolution, and co-allocation.

Krauter *et al.* [62] also present a taxonomy that classifies RMSs by characterizing different attributes. We used some of these criteria and others to compare the systems we present in the next sections.

4.2. Scheduling Taxonomy

In this subsection we consider resource management concerning only job allocation to processors. Several solutions have been proposed to this problem, which is usually called the scheduling problem. Some authors studied solutions to this problem and presented classifications. We present in this subsection some of the most known well classifications.

Casavant and Kuhl [18] is one of the most referred scheduling taxonomies. They propose a hierarchical taxonomy that is partially presented in Figure 11, since we selected some of the classifications that are relevant to this study. Besides,

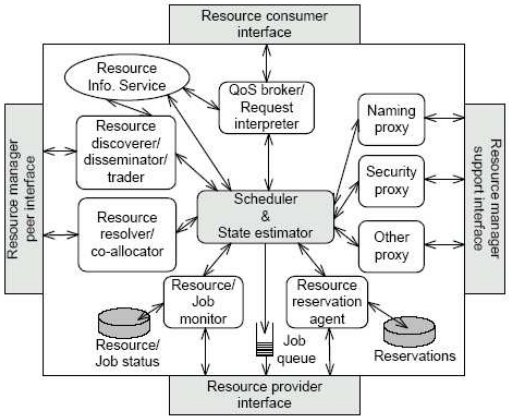

Figure 10. Resource Management System abstract structure [62]

Casavant and Kuhl also propose a flat classification from which we also selected just some issues.

At the highest level, they distinguish between *local* and *global*. Local Scheduling is related with the assignment of a process to the time-slices of a single processor. Global scheduling is the problem of deciding where (in which processor) to execute a process.

In the next level, beneath global scheduling, the time at which the scheduling or assignment decisions are made define *static* and *dynamic* scheduling.

Static scheduling can be *optimal* or *suboptimal*. In case that all information regarding the state of the system as well as the resource needs of a process are known, an optimal assignment can be made based on some criterion function (e.g. minimizing total process completion time). When computing an optimal assignment is computationally infeasible, suboptimal solutions may be tried. Within the realm of suboptimal solutions, there are two general categories: *approximate* and *heuristic*.

The next issue, beneath dynamic scheduling, involves whether the responsibility for the task of global dynamic scheduling should physically reside in a single processor (*physically non-distributed*) or whether the work involved in making decisions should be *physically distributed* among the processors. In this text, we use for simplicity non-distributed (or centralized) and distributed scheduling instead of physically non-distributed and physically distributed. Within the range of distributed dynamic global scheduling, there are mechanisms which involve cooperation between the dis-

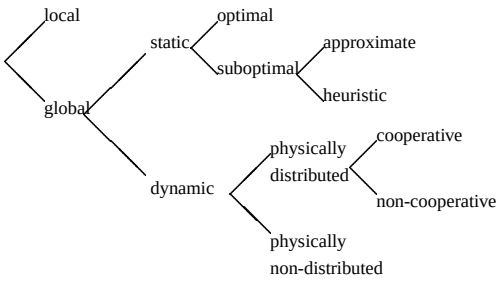

Figure 11. Part of the Casavant and Kuhl's taxonomy [18]

tributed components (*cooperative*) and those in which the individual processors make decisions independent of the actions of the other processors (*non-cooperative*).

In addition to the hierarchical portion of the taxonomy, Casavant and Kuhl [18] present other distinguishing characteristics which scheduling systems may have:

- *Adaptive versus Nonadaptive*: an adaptive solution to the scheduling problem is one in which the algorithms and parameters used to implement the scheduling policy change dynamically according to the previous and current behavior of the system in response to previous decisions made by the scheduling system. A nonadaptive scheduler does not necessarily modify its basic control mechanisms on the basis of the history of system activity.
- *One-Time Assignment versus Dynamic Reassignment*: One-time assignment can technically correspond to a dynamic approach, however it is static in the sense that once a decision is made to place and execute a job, no further decisions are made concerning the job. In contrast, solutions in the dynamic reassignment class try to improve on earlier decisions. This category represents the set of systems that (1) do not trust their users to provide accurate descriptive information, and (2) use dynamically created information to adapt to changing demands of user processes. This adaptation is related to migration of processes.

The scheduling algorithm has four components [97]: (1) transfer policy: *when* a node can take part of a task transfer; (2) selection policy: *which task* must be transferred; (3) location policy: *which node* to transfer to; and (4) information policy: *when to collect* system state information. Considering the location policy component, a scheduling algorithm can be classified as *receiver-initiated* (started by the task importer), *sender-initiated* (started by the task exporter), or *symmetrically initiated*. Their performance are closely related to system workloads. Sender-initiated gives better performance if workers are often idle and receiver-initiated performs better when the load is high.

Another important classification is *dedicated* and *opportunistic* scheduling [116, 86]. Opportunistic scheduling involves placing jobs on non-dedicated resources under the assumption that the resources might not be available for the entire duration of the jobs. Using opportunistic scheduling, resources are used as soon as they become available and applications are migrated when resources need to be preempted. Dedicated scheduling algorithms assume the constant availability of resources to compute fixed schedules. Most software for controlling clusters relies on dedicated scheduling algorithms. The applications that most benefit from opportunistic scheduling are those that require high throughput rather than high performance. Traditional high-performance applications measure their performance in instantaneous metrics like floating point operations per second (FLOPS), while high throughput applications usually use such application-specific metrics, and the performance might be measured in TIPYs (trillions of instructions per year).

In 1998, Berman [10] classified the scheduling mechanisms for grid in three groups. Nowadays, a current concept is the Metascheduler as presented by the GGF [89]. Thus, we consider that the scheduling mechanisms can be classified in four groups:

- *task schedulers*[3] (high-throughput schedulers) promote the performance of the system (as measured by aggregate job performance) by optimizing throughput. Throughput is measured by the number of jobs executed by the system.
- *resource schedulers* coordinate multiple requests for access to a given resource by optimizing fairness criteria (to ensure that all re-

3 Berman called this kind of scheduler as *job scheduler*. We changed it to be consistent with the terminology adopted in this text.

quests are satisfied) or resource utilization (to measure the amount of resource used).

- *application schedulers* (high-performance schedulers) promote the performance of individual applications by optimizing performance measures such as minimal execution time, speedup, or other application-centric cost measures.
- *meta-scheduler* is a scheduler that allows to request resources of more than one machine for a single job. May perform load balancing of workloads across multiple systems. Each system would then have its own local scheduler to determine how its job queue is processed. Requires advance reservation capability of local schedulers.

4.3. Job Management Systems for Clusters

We consider that a cluster may be made of a set of workstations, multiple CPU systems, or a set of nodes in a parallel computer. Usually, this set of execution nodes or hosts have a single batch server that manages batch jobs. The batch processing is related to the following concepts [89]:

- **Queue** is a collection of schedulable entities, e.g. jobs (or job-related tasks) within the (batch) queuing system. Each queue has a set of associated attributes that determine which actions are to be performed upon each job within the queue. Typical attributes include queue name, queue priority, resource limits, destination(s), and job count limits. Selection and scheduling of jobs are implementation-defined. The use of the term "queue" does not imply the ordering is "first in, first out".
- **Batch** is a group of jobs which are submitted for processing on a computer and the results of which are obtained at a later time.
- **Batch Processing** is the capability of running jobs outside the interactive login session and providing for additional control over job scheduling and resource contention.
- **Batch Queue** is an execution queue where the request actually is started from.
- **Batch Server** is a persistent subsystem (daemon) upon a single host that provides batch processing capability.
- **Batch System** is a set of batch servers that are configured for processing. The system may consist of multiple hosts, each with multiple servers.

The single batch system or centralized job management system lets users execute jobs on a cluster. This system must perform at least the following tasks [105]: (a) monitor all available resources; (b) accept jobs submitted by users together with resource requirements for each job; (c) perform centralized job scheduling that matches all available resources with all submitted jobs according to the predefined policies; (d) allocate resources and initiate job execution; (e) monitor all jobs and collect accounting information.

Some of the most well know centralized job management systems are **LSF** (Load Sharing Facility) [72, 73] from Platform Computing Corporation, **SGE** (Sun Grid Engine) [102] from Sun Microsystems, and **PBS** (Portable Batch System) [13]. The original version of PBS is a flexible batch queuing system developed for NASA in the early to mid-1990s. Nowadays, the Altair Grid Technologies offer two versions: OpenPBS [80], the unsupported older original version and PBS Pro [83], the commercial version.

These three systems are general purpose distributed queuing systems that unite a cluster of computers into a single virtual system to make better use of the resources on the network. Most of them perform only dedicated scheduling, but SGE is also capable of performing opportunistic scheduling.

Although they can automatically select hosts in a heterogeneous environment based on the current load conditions and the resource requirements of the applications, they are not suitable for grid environments. Actually, these systems are very important in our study since they will take part in a grid environment as a "grid node". However, they cannot be used to manage a grid environment since they have scalability problems, have a single point of failure, and not provide security mechanisms to schedule across administrative domains.

4.4. Scheduling Mechanisms for Grid Environments

Grid environments are composed by several trust domains. Usually, each domain has its private scheduler, which works isolated from each other. A resource manager grid aware is necessary to allow all these isolated schedulers to work together taking advantage of all grid potential. Several works in the literature present scheduling mechanisms for grids. In this subsection we present some of the most well know works: Legion (Subsection 4.4.1), Globus (Subsection 4.4.2), Condor-G (Subsection 4.4.3), MyGrid (Subsection 4.4.4), the GrADS's Metasched-

uler (Subsection 4.4.5), and ISAM (Subsection 4.4.6).

4.4.1. Legion The Legion Project of University of Virginia started in 1993. The main result is the Legion system [53, 21, 69], which is nowadays an Avaki commercial product. Legion is an object oriented infrastructure for grid environments layered on top of existing software services. It uses the existing operating systems, resource management tools, and security mechanisms at host sites to implement higher level system-wide services. The Legion design is based on a set of core objects.

In Legion, resource management is a negotiation between resources and active objects that represent the distributed application. In the allocation of resources for a specific task there are three steps [108]: decision (considers task's characteristics and requirements, the resource's properties and policies, and users' preferences), enactment (the class object receives an activation request; if the placement is acceptable, start the task), and monitoring (ensures that the task is operating correctly).

Figure 12 presents the main components that are responsible for resource management. The Legion object hierarchy rectangle presents the main classes that can be instantiated to represent applications and resources. The *LegionClass* object is the root class that is extended to create all other core classes, including user defined classes (e.g. *MyObjClass*). It defines that objects in Legion are active objects. A *LegionClass* is also a manager of its instances. Two main core objects represent the basic resource types in Legion: *Vaults* and *Hosts*.

The *Host* is a machine's representative to Legion: it is responsible for executing objects on the machine, protecting objects from each other, deactivating objects, and reporting object exceptions. It periodically updates its local state. When it receives a reservation request, it must ensure that the *Vault* is available, there are enough resources, and local allocation policy allows object instantiation. Hosts can grant reservation for future service.

The *Vault* represents persistent storage. It does not have a dynamic information.

The *Collection* acts as a repository for information describing the state of resources comprising the system. Each record is stored as a set of Legion object attributes. Users can obtain information about resources by issuing queries using a specific query language. Chapin *et al.* [21] presents the following query to find all hosts that run the IRIX operating system version 5.x:

```
match($host_os_name, "IRIX") and
     match($host_os_name, "5\..*")
```

The *Scheduler* computes the mapping of objects to resources. It obtains information by querying the *Collection* and computes a mapping of object instances to resources. Then, the mapping is passed to the *Enactor*.

The *Enactor* "executes" each entry in the received schedule. If all mappings succeed, then scheduling is complete. If not, then a *variant schedule*, that contains a new entry for the failed mapping, is selected.

The steps in object placement, as illustrated in Figure 12, are as follows [21]: **(1)** The Collection is populated with information describing the resources. **(2)** The Scheduler queries the Collection, and **(3)** based on the result and knowledge of the application, computes a mapping of objects to resources. This application-specific knowledge can either be implicit (in case of an application-specific Scheduler), or can be acquired from the application's classes. **(4)** This mapping is passed to the Enactor, which **(5)** invokes methods on hosts and vaults to **(6)** obtain reservations from the resources named in the mapping. **(7)** After obtaining reservations, the Enactor consults the Scheduler to confirm the schedule, and **(8)** after receiving approval from the Scheduler, **(9)** attempts to instantiate the objects through member function calls on the appropriate class objects. **(10)** The class objects report success/failure codes, and **(11)** the Enactor returns the result to the Scheduler. **(12)** If, during execution, a resource decides that the object needs to be migrated, it performs an outcall to a Monitor, **(13)** which notifies the Scheduler and the Enactor that rescheduling should be performed.

4.4.2. Globus Globus [38, 93] is one of the most well know projects on grid computing. As already mentioned in Section 2, the most important result of the Globus Project is the Globus Toolkit. The toolkit consists of a set of components that implement basic services, such as security, resource location, resource management, data management, resource reservation, and communication. From version 1.0 in 1998 to the 2.0 release in 2002 and the latest 3.0, the emphasis is to provide a set of components that can be used either independently or together to develop applications. The Globus Toolkit version 2 (GT2) design is highly related to the architecture proposed by Foster *et al.* [41]. The Globus Toolkit version 3 (GT3) design is based on grid services, which are quite similar to web services. GT3 implements the *Open Grid Service Infrastructure* (OGSI) [40].

The Globus Resource Allocation Manager (GRAM) [25] is one of the available services in Globus. Each GRAM is responsible for a set of resources operating under the same site-specific

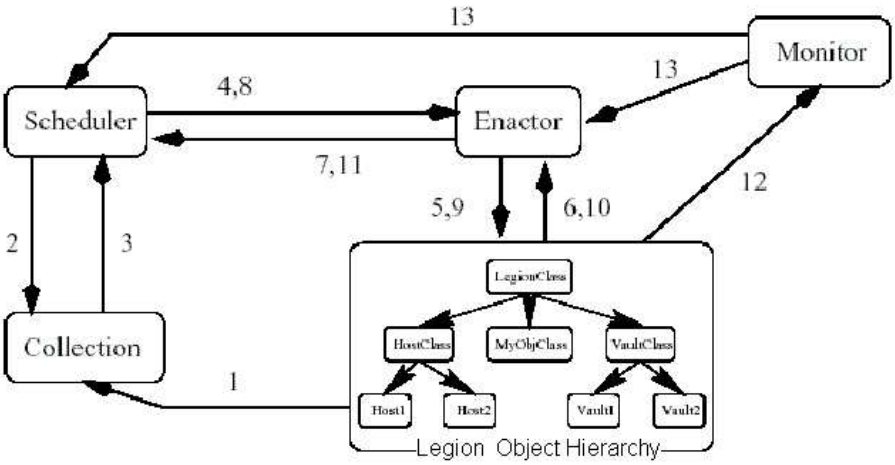

Figure 12. Legion Resource Manager [21]

allocation policy, often implemented by a local resource management system, such as LSF or Condor. GRAM provides an abstraction for remote process queuing and execution with several powerful features such as strong security and file transfer.

Thus GRAM does not provide scheduling or resource brokering capabilities but it can be used to start programs on remote resources, despite local heterogeneity due to the standard API and protocol. The Resource Specification Language (RSL) is used to communicate requirements.

To take advantage of GRAM, a user still needs a system that can remember what jobs have been submitted, where they are, and what they are doing. To track large numbers of jobs, the user needs queuing, prioritization, logging, and accounting. These services cannot be found in GRAM alone, but is provided by systems such as Condor-G [107].

The Globus research group and IBM started to work on the WS-Resource Framework [117]. They released an initial architecture and specification documents with co-authors from HP, SAP, Akamai, TIBCO and Sonic on January 20, 2004. The WS-Resource Framework (WSRF) is an extension of OGSI. WSRF is a set of six Web services specifications. To date, drafts of three of these specifications have been released, along with an architecture document that motivates and describes the WS-Resource approach to modeling stateful resources with Web services.

4.4.3. Condor and Condor-G Condor High Throughput Computing System [107, 104, 116], or simply Condor, is a specialized workload/resource management system for compute-intensive jobs. It can be used to manage a cluster of dedicated nodes as well as to harness wasted CPU power from otherwise idle desktop workstations. Using idle desktop workstation computational power is the main difference between Condor and the traditional RMS. The Condor scheduling policy can be classified as *opportunistic*.

Condor is composed of a collection of different daemons [116]. Figure 13 presents a schematic view of a Condor pool with the daemons. Condor exercises administrative control over a Condor pool. A pool is a set of resources that report to a single daemon called the `collector`. The `collector` is the central repository of information in the Condor system. Almost all Condor daemons send periodic updates to it. Each update is in the form of a *ClassAd*, a data structure consisting of a set of attributes describing a specific entity in the system. The machine where the `collector` runs is referred to as the *central manager*.

The central manager also runs the `negotiator` daemon, which periodically performs a *negotiation cycle*. This cycle is a process of *matchmaking*, where the `negotiator` tries to find matches between various ClassAds (resource requests and resource offers). Once a match is made, both parties are notified and are responsible for acting on that match.

The `startd` daemon runs on all machines of the pool and monitors the conditions of the resource where it runs, publishes resource offer ClassAds, and it is responsible for enforcing the resource owner's policy for starting, suspending, and evicting jobs. For example, owners can specify which users or jobs a resource most prefers to serve. Any machine running a `startd` can be referred to as an *execute machine*, since it is able to execute Condor jobs.

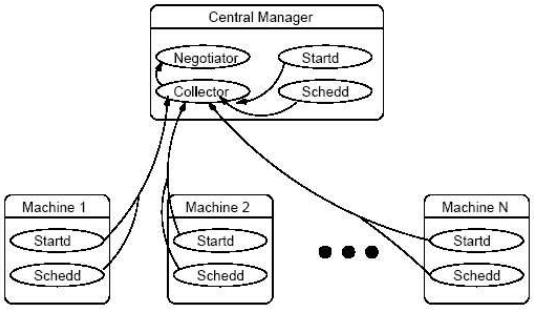

Figure 13. Architecture of a Condor pool with the daemons [116]

The user submit his/her jobs to a daemon called `schedd`. This daemon maintains a persistent job queue, publishes resource request ClassAds, and negotiates for available resources. After it receives a match for a given job, the `schedd` enters into a *claiming protocol* directly with the `startd`. Through this protocol, the `schedd` presents the job ClassAd to the `startd` and requests temporary control over the resource. Once it has claimed a given resource, the `schedd` performs its own local scheduling to decide what jobs to run. Any machine running a `schedd` can be referred to as a *submit machine*, since users are able to submit Condor jobs from that host.

When the `schedd` starts a job, it spawns a `shadow` process on the submit machine and the `startd` spawns a `starter` process on the corresponding execute machine. The `shadow` serves request for files to transfer, logs the job's process, and reports statistics. The `starter` sets up the execution environment and monitors the job.

Condor allows the user to run his/her jobs in different execution environments called universes. The main universes supported are the following [104]:

- *Vanilla*: is used to run serial (non-parallel) jobs. Any program that runs outside of Condor will run in the Vanilla Universe.
- *Standard*: allows a job running under Condor to handle system calls by returning them to the machine where the job was submitted. The standard universe also provides the mechanisms necessary to take a checkpoint and migrate a partially completed job. To use the standard universe, it is necessary to relink the program with the Condor library.
- *MPI*: allows parallel programs written with MPI (using the MPICH interface) to be managed by Condor.
- *PVM*: allows master-worker style parallel programs written for PVM to be used with Condor. Condor runs the master application on the machine where the job was submitted and workers in other available machines in the pool.

Checkpointing is done in the Standard Universe. A checkpointing of an executing program is a snapshot or the program's current state. It provides a way for the program to be continued from that state later on. It is desirable to ensure that only the computation done since the last checkpoint is lost in case of fault or preemption. Condor can be configured to periodically produce a checkpoint for a job.

Condor has a mechanism called *flocking* that allows jobs to be scheduled across multiple Condor pools [107]. Nowadays, it is implemented as direct flocking that only requires agreement between one individual and another organization, but accordingly only benefits the user who takes the initiative. A particular job will only flock to another pool when it cannot currently run in the pool of submission. It is a useful feature, but is not enough to enable jobs to run in a grid environment, mainly due to security issues.

Condor-G [107, 44] is the job management part of the Condor project to allow users to access grid resources. Instead of using the Condor-developed protocols to start running a job on a remote machine, Condor-G uses the Globus Toolkit to start the job on the remote machine. Thus, applications can be submitted to a resource accessible through a Globus interface.

Condor-G uses the protocols for secure interdomain communications and standardized access to a variety of remote batch systems from Globus. The user concerns of job submission, job allocation, error recovery, and creation of a friendly execution environment comes from Condor.

The major difference between Condor flocking and Condor-G is that Condor-G allows interdomain operation on remote resources that require authentication, and uses Globus standard protocols that provide access to resources controlled by other resource management systems, rather than the special-purpose sharing mechanisms of Condor [44].

4.4.4. MyGrid MyGrid [22, 81] enables the execution of bag-of-tasks parallel applications on all machines the user has access to. A bag-of-tasks application has completely independent tasks. The authors [81] argue that scheduling independent tasks, although simpler than tightly coupled parallel applications, is still difficult due

to the dynamic behavior and the intrinsic resource heterogeneity exhibited by most grids. The authors also indicate that it is usually difficult to obtain good information about the entire grid as well as about the tasks to make the scheduling plan. Thus, they propose a solution that does not require almost any kind of information: MyGrid uses a dynamic algorithm called Workqueue with Replication (WQR).

The WQR is a dynamic scheduling algorithm that is not based on performance information. It is an extension of the Workqueue algorithm. The Workqueue algorithm works as follows: Tasks are chosen in an arbitrary order in the "bag of tasks" and sent to the processors, as soon as they become available. After the completion of a task, the processor sends back the results and the scheduler assigns a new task to the processor. A problem with the Workqueue arises when a large task is allocated to a slow machine near the end of the schedule. When this occurs, the completion of the application will be delayed until the complete execution of this task.

The WQR algorithm uses task replication to cope with the heterogeneity of hosts and tasks, and also with the dynamic variation of resource availability due to the load generated by other users in the grid. Note that this strategy allows to deal only with the heterogeneity related to computational capacity. It works like the Workqueue algorithm until all tasks are assigned ("the bag-of-tasks becomes empty"). At this time, hosts that finished their tasks are assigned to execute replicas of tasks that are still running. Tasks are replicated until a predefined maximum number of replicas is achieved (in MyGrid, the default is one). This replication leads to wasted CPU cycles. Note that the replication assumes that the tasks do not cause side effects. If a task does not have side effects, it can be executed more than once producing always the same final results.

MyGrid executes at the user level. There is a machine called *home machine* that coordinates the execution of the applications through MyGrid. It is assumed that the user has good access to the home machine (often it will be the user's desktop). The home machine schedules tasks to run on the *grid machines* using the WQR algorithm. Grid machines do not necessarily share file systems with the home machine. To access grid resources, there is a *Grid Machine Interface* that has four services: (1) task start-up on a grid machine (remote execution); (2) cancellation of a running task; (3) file transfer from the grid machine to the home machine; and (4) file transfer from the home machine to the grid machine.

OurGrid [4] extends the MyGrid efforts, including the utilization of grid technology on commercial settings and the creation of large-scale community grids. OurGrid is a resource sharing system based on peer-to-peer technologies. The resources are shared according to a "network of favors model", in which each peer prioritizes those who have credit in their past history of interactions.

4.4.5. MetaScheduler in GrADS Project The GrADS system [11] is an application scheduler. Its execution model can be described as following. The user invokes the Grid routine component to execute his/her application. The Grid Routine invokes the component Resource Selector. The Resource Selector accesses the Globus MetaDirectory Service (MDS) to get a list of machines that are alive and then contact the Network Weather Service (NWS) to get system information for the machines. The Grid Routine then invokes a component called Performance Modeler with the problem parameters, machines and machine information. The Performance Modeler builds the final list of machines and send it to the Contract Developer for approval. The Grid routine then passes the problem, its parameters, and the final list of machines to the Application Launcher. The Application Launcher spawns the job using the Globus management mechanism (GRAM) and also spawns the Contract Monitor. The Contract Monitor monitors the application, displays the actual and predicted times, and can report contract violations to a rescheduler. All these components are presented in Figure 14 as white rectangles and ellipse.

Although the execution model is efficient from the application perspective, it does not take into account the existence of other applications in the system. Thus, Vadhiyar and Dongarra [111] proposed a metascheduling architecture in the context of the GrADS Project. The metascheduler receives candidate schedules of different application level schedulers and implements scheduling policies for balancing the interests of different applications. Figure 14 presents the modified GrADS architecture where the four new components are represented as colored rectangles and ellipses: Database Manager, Permission Service, Contract Negotiator, and Expander.

The Database Manager maintains a record for each application submitted to the grid system. Other components query this information to make scheduling decisions.

The Permission Service is a daemon that receives requests from the applications to grant them permission to proceed with the use of the grid system. It checks if the resources meet the applica-

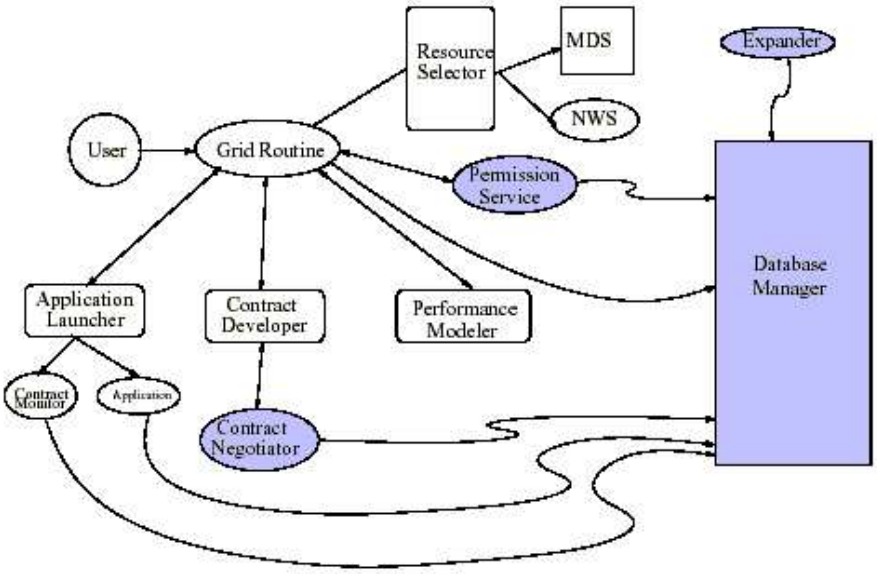

Figure 14. Modified GrADS architecture for metascheduler proposal [11]

tion's requirements. If not, it tries to accommodate the new application waiting other applications finish or stopping resource and time consuming applications.

The Contract Negotiator is a daemon that receives application level schedules from the applications. The schedules are the contracts submitted to the Contract Developer which presents the final list of machines that the application obtains from the Performance Modeler through the employment of the application specific execution model. The Contract Negotiator acts as a queue manager controlling different applications of the grid system.

If the Contract Negotiator approves the contract, the application can proceed to the launching phase, if not, the application restarts from the resource selection phase. It can reject the contract in the following cases: (1) application got its resource information before an executing application started; (2) the performance of the new application can be improved significantly in the absence of an executing application; (3) the already executing application can be severely impacted by the new application.

The Expander is a daemon that tries to improve the performance of the already executing applications. When an application completes, the Expander determines if performance benefits can be obtained for an already executing application by expanding the application to utilize the resources freed by a completed application.

4.4.6. ISAM ISAM (*Infra-estrutura de Suporte às Aplicações Móveis* – Support Infrastruture to Mobile Applications) [119, 118, 120] is a proposal of an integrated solution, from development to execution, for general purpose pervasive applications. These applications are distributed, mobile, adaptive an reactive to the context. Aiming at supporting the follow-me semantics (the application follows the user) for the pervasive applications, the ISAM middleware concerns with resource management in heterogenous, multi-institutional, networks.

The ISAM architecture is organized in layers with three abstraction level as we can see in Figure 15. At the high level, ISAM provides a programming language, ISAMadapt, which allows the application designer to express adaptions. ISAMadapt build on the execution model defined by Holo paradigm, being compiled to Java code which access the ISAM middleware services.

The intermediate level provides what can be called the ISAM middleware. It provides support the User Virtual Environment, a key component for supporting user's mobility. The scheduling is one of the main components in what concerns to the management of the pervasive execution.

TiPS (TiPS is a Probabilistic Scheduler) [88, 87] is part of the scheduling component. TiPS provides a scheduling strategy based on dynamic sensors, which includes a bayesian model of the environment to be managed.

The distributed environment is managed by ISAM in four levels: (a) *hosts* are the machines; (b) *network segments* are a set of hosts; (c) *computing cells* delimits the institutions boundaries; (d) *cell groups* are groups of computing

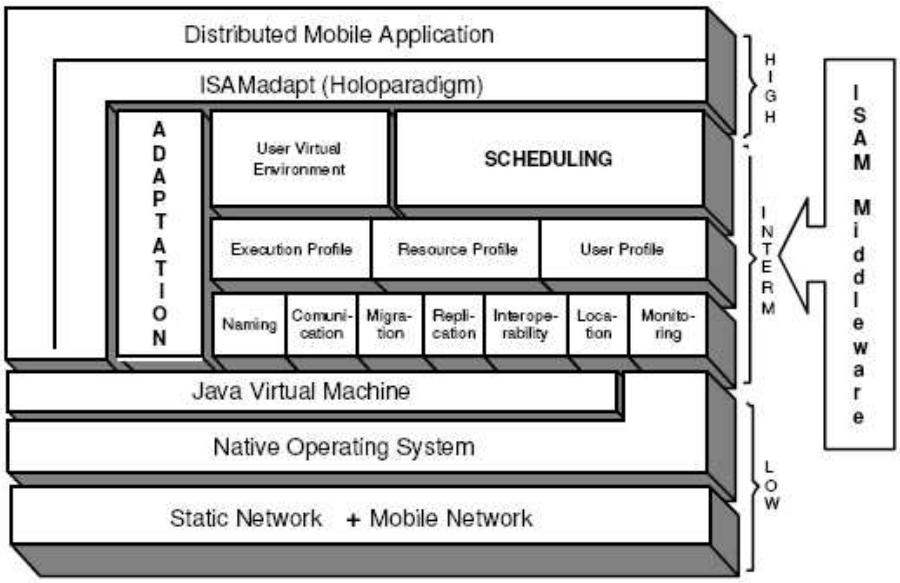

Figure 15. ISAM architecture [88]

cells. Scheduling decisions are made considering this hierarchy.

Finally, the ISAM low level is the fabric level with the physical components, the operating system, and the Java Virtual Machine.

4.5. Comparison

We presented some of the most representative RMSs used nowadays in grid research. We now present a brief comparison of them, summarized in Tables 2 and 3. We built this table using some of the characteristics we considered more important to our work.

In Table 2, the first column presents our classification presented in Section 4.1 which includes Berman's classification and the GGF metascheduler definition. PBS, LSF, and SGE are task schedulers. Condor and Legion are resource schedulers and both support at some extent scheduling over multiple domains. MyGrid is the only classified as application scheduler. However, in the literature, there are other systems that could be classified as application scheduler such as the EasyGrid framework [14, 31]. EasyGrid is a framework for the automatic grid enabling of MPI parallel applications. It provides services oriented towards the individual application in such a way that each application appears to have exclusive access to a virtual grid. Globus, Condor-G, the GrADS' Metascheduler, and ISAM can be classified as metaschedulers or high-level schedulers.

PBS, LSF, and Legion can also be classified as dedicated schedulers, SGE can work as a dedicate scheduler as well as an opportunistic scheduler, while Condor is an opportunistic scheduler. In our table, the symbol "–" means that the classification cannot be applied to the corresponding system.

The third column indicates that PBS, LSF, SGE, and Condor have centralized schedulers while Legion, Globus, Condor-G, GrADS' Metascheduler, and ISAM are hierarchical. MyGrid was classified as centralized, but actually there is one instance of scheduler for each running application.

Other criterion shown in the table (last column) is Open Source, i.e., if there is source code available for research purposes. Most of the RMS' source code are available as our table shows.

Table 3 presents other aspects of these systems and is a continuation of the previous table. All RMS use some kind of static information to take scheduling decisions. The first column indicates if the RMS also needs some dynamic information (usually related to resource utilization and availability). MyGrid is the only one that does not need any dynamic information to make its decisions. MyGrid and ISAM use the task replication technique to obtain fault tolerance and better performance at the cost of wasting some CPU cycles.

Since computational grids are dynamic, jobs should adapt themselves according to characteristics such as availability and load in order to obtain application performance. Execution must be flexible and adaptive to achieve either robust or even good performance due to heterogeneity of configuration, performance, and reliability in grid environments. So, one of the approaches is that jobs

System	Classification			Open Source
PBS	task scheduler	dedicated	centralized	OpenPBS – yes PBSPro – no
LSF	task scheduler	dedicated	centralized	no
SGE	task scheduler	dedicated and opportunistic	centralized	no
Condor	resource scheduler	opportunistic	centralized	yes
Legion	resource scheduler	dedicated	hierarchical	no
MyGrid	application scheduler	–	centralized (per application)	yes
Globus	metascheduler	–	hierarchical	yes
Condor-G	metascheduler	–	hierarchical	yes
GrADS' Metascheduler	metascheduler	–	hierarchical	no
ISAM	metascheduler	–	hierarchical	yes

Table 2. Comparison of presented schedulers – part I

System	Dynamic information	Migration	Replication
PBS	yes	no	no
LSF	yes	no	no
SGE	yes	no	no
Condor	yes	yes	no
Legion	yes	yes	no
MyGrid	no	no	yes
Globus	yes	no	no
Condor-G	yes	no	no
GrADS' Metascheduler	yes	yes	no
ISAM	yes	yes	yes

Table 3. Comparison of presented schedulers – part II

are able to migrate. Migration [89] describes the rearrangement of allocated resources within a resource pool. Several works deal with the so called "opportunistic" migration of jobs when a "better" resource is discovered [111, 112, 75, 2]. The Migration's column indicates if the RMS can migrate a job after its allocation. Condor, Legion, GrADS' metascheduler, and ISAM can migrate a running job for performance reasons. Condor can also migrate to avoid disturbing a user due to opportunistic scheduling. Note that in Globus and Condor-G migration in the local RMS can happen, but both cannot directly control this kind of mechanism.

The two most popular grid-aware systems are Condor and Globus. As mentioned before, Condor is a specialized workload management system for compute-intensive jobs. Like other full-featured batch systems presented, Condor provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to Condor, Condor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion. Among other things, Condor allows transparent migration of jobs from overloaded machines to idle machines and checkpointing, which permits that jobs can restart in another machine without the need to start from the beginning. These are typical tasks of a resource manager.

Globus, by its turn, is a whole framework that includes a set of services used, for example, to securely transfer files from one grid node to another (GridFTP), to manage meta data (MDS), and to allocate remote resources (GRAM). Condor-G [44] puts together Condor facilities to manage jobs with the Globus grid facilities such as secure authentication (GSI) and data transfer.

Condor applies an opportunistic scheduling policy that concentrates on allocating idle resources to take advantage of idle CPU cycles. Globus focuses on providing several different services to execute secure code on authorized machines.

Application management and control, load balancing and data locality are not their main focus.

The Condor scheduling system and the MetaScheduler in the GrADS Project present some similarities. Both support preemption of executing jobs to either accommodate other jobs or to transfer the control of the resources to the resource owners. However, the first is more concerned to free resources reclaimed by the owners whereas the second one tries to get performance. Besides, the Condor's Negotiator component has similar functionality to the Metascheduler's Contract Negotiator.

An aspect not covered in our tables is related to resource description. The resources must be represented in a somehow independent way to manage the inherent heterogeneity of grid environments. Legion has adopted the object model as a way to get a uniform way to access the resources. Globus and Condor-G have decided to adopt a description

language which is translated to an internal representation.

All systems present some way to deal with faults. One important issue that RMS must deal with is data loss. Most available software can not handle network traffic properly. For example, Condor, one of the software that we had experience with, can either loose jobs that were in the job queue, or generate corrupt data files, because of lack of network flow control. The user is responsible to manually control the number of jobs that will be simultaneously submitted in order to avoid network congestion. As a consequence of the little attention given to flow control and data management, data loss can occur due to overflow when too much traffic is generated on data and code transfers. Some experiments reported on [30] illustrate this problem. From 45,000 tasks launched, around 20% failed for several reasons, including data corrupted due to packet loss, and had to be resubmitted.

Other problem concerns application submission. In some systems, applications are launched in the user machine. Because most grid aware software create one connection or a new process to each launched job, the submit machine can become overloaded and have a very low response time.

On the light of this discussion about grid aware systems and their limitations, we discuss a promising solution to these limitations in the next section.

5. Toward an integrated system to manage applications and data in grid environments

This section aims to address some of the issues related to the construction of an integrated system to manage application and data in grid environment. More specifically, we are concerned with the execution management of applications that spread a high number of tasks (thousands) in a grid environment.

Figure 16 presents a schematic view of the steps necessary to execute a distributed application. The first step is called *partitioning*: an application composed by several tasks is divided in subgraphs. Partitioning aims to get together tasks that are dependent. Then, the obtained subgraphs can be grouped during the *mapping* step according to the available resources. Finally, the *allocation* step is responsable for ensuring the execution of the subgraphs on actual resources.

Before presenting our model to perform such steps, we present our premises (Subsection 5.1). Next, we introduce our application partitioning model (Subsection 5.2). We advocate that a hierarchy of managers that can dynamically distribute data and tasks can aid the task of application management for applications that spread a high number of tasks. Then, we present an overview of our task submission model (Subsection 5.3), including the model components (Subsection 5.3.1). We conclude this section presenting some final considerations (Subsection 5.4).

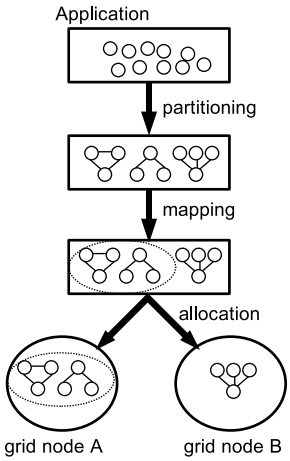

Figure 16. Steps to execute a distributed application

5.1. Premises

A premise is an assumption made or implied as a basis of argument. The main premises assumed to the conception of our model are the following:

Heterogeneous environment We assume that machines could have different software and hardware configurations, and probably will have as one could expect in any grid environment. This heterogeneity must be taken into consideration in our scheduling decisions.

A huge number of tasks can be submitted This assumption is fundamental in the definition of several details in our model. By a huge number of tasks we mean applications that will generate hundreds or thousands of processes.

Suppose the user submits in his/her home machine an application with, for example, ten thousand tasks. If all the submission and the control were done in the home machine, probably this machine would stall and the user would not continue to work there. This problem is already known in the literature. For example, Condor [107] allows the user to specify a limit of jobs that can be submitted in a specific machine. This is not the best solution, since users must have some previous ex-

perience with job submission in this environment to infer the appropriate limit avoiding stall his/her machine and getting a good degree of concurrency.

These kinds of applications could not be easily controlled by hand. Thus, our proposal needs to be scalable. It also must provide execution feedback to users since execution will probably take several hours or even days.

Tasks do not communicate by message passing Message passing introduces many aspects to be considered in the partitioning and allocation phases. We decided to assume that our applications do not communicate through messages. This is not a strong imposed restriction, i.e. we can treat parallel applications with message passing in the future. We just decided to postpone the analysis of this kind of application to narrow our initial scope, simplifying our model conception. Besides, we have at this moment some applications to use in our experiments that do not use message passing, e.g. Monte Carlo simulation for civil engineering [120] and experiments of inductive logic programming (ILP) to solve biostatistics problems [30].

Tasks can have dependencies with other tasks due to file sharing A *bag-of-tasks* is the simplest way an application can be organized: there are no dependencies between tasks, so they can be executed in any order. Some Monte Carlo simulations can be classified in this group. The grid system MyGrid [81] is an example of a system that deals properly with this kind of application, and has proved its usefulness.

Applications can also be modeled as a dependency *graph* of tasks due to file sharing. For example, if a task a produces an output file f_a that task b uses as its input file, then b must wait until task a finishes. In this example, a and b are nodes and there is an edge from a to b, therefore b can only be launched after a finishes its execution. This is a common assumption as presented in Condor's DAGMan [107] and Globus' Chimera [42].

We consider that we always have a graph of tasks. The simplest graph is a bag-of-tasks (only nodes, no edge) but more complex graphs must also be supported. Thus, our scheduling decisions also consider task precedence.

Huge number of files can be manipulated by tasks Tasks can communicate and synchronize through files, so, each task usually will manipulate at least two files (one input and one output). Since we assume a huge number of tasks, a very high number of files must be managed. Efficient algorithms to keep data locality and to efficiently transfer files are crucial to the success of the model under this assumption.

Huge files can be used in computation Huge file transfers could cause network congestion, package loss, and can make transfer times unbearable. So, some kind of action must be taken to control file transfer avoiding package loss and network saturation. Preserving data locality, and staging and caching techniques could help minimizing performance losses due to data transfer latency.

Underlying grid environment is secure We assume that a secure connection is available between grid nodes. We also assume that user authentication and authorization are available in the grid environment. For example, the Globus Security Infrastructure (GSI) [114] can be used to satisfy our requirements.

Each grid node has its local resource manager Local resource managers (lower-level schedulers) can have several attributes such as presented in [94]. We assume the following attributes will be available at each grid node:

- exclusive control: this attribute indicates if the local manager is in exclusive control of the resources. This information is useful to determine the reliability of the scheduling information;
- consideration of job dependencies: the lower-level scheduler takes dependencies between allocations into account if they are provided by the higher-level scheduling instance. For instance, in case of a complex job request the lower-level scheduling will not start an allocation if the completion of another allocation is required and still pending.

We also assume that once a task is allocated it will not be scheduled again or at least this will be transparent to the higher level resource managers.

5.2. Application Partitioning Model

We assume users submit applications to our metascheduler using our description language. Our description language GRID-ADL is an extension of DAGMan input description file language, as we describe next (Subsection 5.2.1). It also presents the main idea of Chimera's language: the user does not have to present explicitly the graph, but only the data files manipulated. Each application has several tasks and can be represented as a Directed Acyclic Graph (DAG). In our DAGs, the nodes represent tasks and edges represent dependencies between tasks through data files access. Our system

can infer automatically the DAG through the analysis of the data flow. The weight of a node denotes the required amount of computation. The user can specify these values, to allow a better partitioning. Otherwise, a same default value is set to all nodes.

Since we have a limited number of resources, nodes must be grouped to be executed in the same execution unit (processor, cluster or local network). We refer to this grouping and mapping as an application partitioning. We consider that a good partitioning is the one that minimizes data transfer (i.e. maximizing data locality) and maximizes application performance. Note that partitioning the DAG is a compromise between two conflicting forces: keeping nodes separate increases parallelism at the cost of communication, whereas clustering them causes serialization but saves communication [34]. We describe how applications are partitioned, considering our application taxonomy (defined in Subsection 2.2), in Subsection 5.2.2.

5.2.1. Description Language Generally, users run their jobs using some kind of description file that contains characteristics such as the task to be executed, the computational power required or the full path to the executable. As most users are acquainted with this kind of routine, we opted to maintain this classical approach, and provide to the user a simple description language that can quickly represent the user applications and needs.

Our description language is called GRID-ADL (*Grid Application Description Language*). GRID-ADL has the legibility and simplicity of shell scripts and DAGMan [107] language while presents data relations that allow to infer automatically the DAG in the same way Chimera [42] does. The user submits a file describing only the kind of application (independent, loosely-coupled, or tightly-coupled tasks) and its tasks indicating input and output files.

Our language syntax is represented in Backus-Naur-Form (BNF) in Figure 21. Since it is self explanatory, we will not explain all details of our description language. We will only highlight the main aspects using the three DAG examples presented in Figure 17.

As already mentioned, our language can be considered as an extension of the DAGMan description language. Some main differences are the following:

- the user can give a hint on how the task graph can be classified ("independent", "loosely-coupled", or "tightly-coupled"). This is useful to speedup the partitioning phase, since there are different algorithms for each type of graph as we present in the next section;
- the task description presents, besides a name and a submission file, input and output file names;
- some shell script like constructions are available to facilitate the description of tasks.

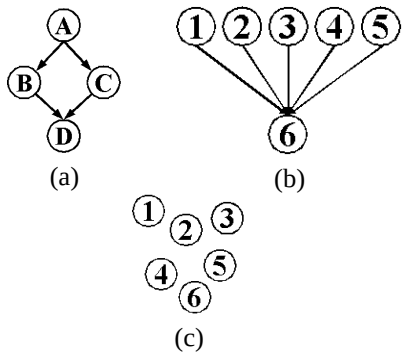

Figure 17. DAG used in input file examples

The first two differences can be seen in Figure 18. It presents an example where the user first indicates that his/her application has a loosely-coupled graph (first statement). The `graph` keyword is a hint the user gives to our system. It is an optional directive that should be used to get a better performance when trying to infer the DAG. For instance, if the user defines that the DAG represents independent tasks, it is not necessary to run the algorithm to infer the DAG since there is no precedence order between tasks.

The next four subsequent lines describe four tasks using the statement `task`. For each one the user needs to define a name, a submission file, one or more input files, and one or more output files.

```
1 graph loosely-coupled
2 task A A.sub -i data.in -o a.out
3 task B B.sub -i a.out -o b.out
4 task C C.sub -i a.out -o c.out
5 task D D.sub -i b.out c.out -o data.out
```

Figure 18. Example of input file for the simple DAG (17(a))

This example is simpler than the transformations and derivations used by the Chimera description language while being more powerful than the DAGMan specification language.

Besides these direct and simple statements, we added to GRID-ADL some shell script like constructions as illustrated in Figure 19. The second statement presents a string variable assignment. Next, an iteration command is used to declare five tasks as well to store in the string variable the output file names. Then, the string variable is used to indicate the input file of task 6. The final command defines that the files stored in the string variable would not be copied back to the user workspace, i.e, they are transient or temporary files.

```
1 graph loosely-coupled
2 OUTPUT = ""
3 foreach ${TASK} in 1..5 {
4   task ${TASK} ${TASK}.sub -i ${TASK}.in
5                            -o ${TASK}.out
6   OUTPUT = ${OUTPUT} + ${TASK}.out + " "
7   }
8 task 6 6.sub -i ${OUTPUT} -o data.out
9 transient ${OUTPUT}
```

Figure 19. Input file for DAG example 17(b)

The third example, in Figure 20, illustrates with only three lines how to define an arbitrary number of tasks. In this case, we define an independent graph (a bag-of-tasks application) with six tasks. This example shows a nice feature of our language not present in the specification languages of systems like Chimera or DAGMan.

```
1 graph independent
2 foreach ${TASK} in 1..6 {
3   task ${TASK} ${TASK}.sub -i ${TASK}.in
4                            -o ${TASK}.out
5 }
```

Figure 20. Input file for DAG example 17(c)

5.2.2. Application Partitioning Proposal In a first instance, we consider application partitioning without associating it to resources. Our objective with this first step is to maintain data locality. For this step, we propose the use of different partitioning algorithms according to the application taxonomy presented previously.

Therefore in the case of independent tasks, partitioning becomes simple, because tasks are not dependent on each other. We need to partition in blocks considering the computational power required for each task, but there is no restriction of which task will belong to each block.

For loosely-coupled graphs, where tasks can have a low degree of dependency, we intend to use classical graph partitioning algorithms to obtain subtrees of the graph and try to allocate sequences of dependent tasks in the same grid node. This will avoid unnecessary data transfers.

For tightly-coupled graphs, we intend to use and algorithm called Scheduling by Edge Reversal (SER) [8, 7, 45], that can produce good partitioning of complex graphs taking into account reduction of network traffic [85].

Once the application is conveniently partitioned, we need to solve a second problem that is how to map the partitions to the available grid nodes. As explained in Subsection 5.1, we assume that each grid node will have a local RMS to perform the schedule of tasks that belong to a partition.

Our mapping will be done taking into consideration:

- application requirements such as computational power and memory, disk, etc;
- node distance (latency should be minimized);
- link capacity (bandwidth should be maximized);
- node load;

Our intention is to use a predictive model based on history to choose resources. This will be developed during our thesis work. This second step, mapping, may require that some partitions be grouped.

5.3. Task Submission Model

Based on the types of applications we have run on grid environments, and motivated by experiments carried out by physicists and bioinformatics people, we designed an architectural model to be implemented at a middleware level.

Our proposal is a hierarchical management mechanism that can control the execution of a huge number of distributed tasks preserving data locality while reducing the load of the submit machines. The main idea is to balance the submission task control load into several machines. Our architectural model relies on already available software components to solve issues such as allocation of tasks within a grid node, or authorization control.

In our design we control the application through a hierarchical organization of controllers that is

```
<input_file> ::= [<graph_definition>]
                 <set_of_task_definition>
                 [<transient_file_definition>]

<graph_definition> ::= "graph" <graph_type>
<graph_type> ::= "independent"
             | "loosely-coupled"
             | "tightly-coupled"

<set_of_task_definition> ::= <task_definition>
             | <loop>
             | <assignment>
             | <task_definition> <set_of_task_definition>
             | <loop> <set_of_task_definition>
             | <assignment> <set_of_task_definition>

<task_definition> ::= "task" <task_name>
             "-i" <filenames>     "-o" <filenames>
             [ "-c" <number> ]    [ "done" ]
<task_name> ::=  <string> | <var>

<loop> ::= "foreach" <var> "in" <range> "{"
             <set_of_task_definition>
           "}"
<range> ::= <number> .. <number>
          | "{" <symbols> "}"
<symbols> ::= <string>
          | <string> ";" <symbols>

<assignment> ::= <string> "=" <assignment'>
<assignment'>  ::= <operator>
               | <operator> <operation> <operator>
               | ( <var>|<number> ) <math_operation> ( <var>|<number> )
<operator> ::=  <var>
               | <string>
               | <number>
<operation> ::= "+" | "-"
<math_operation> ::= "*" | "/" | "^"

<transient_file_definition> ::= "transient" <filenames>

<filenames> ::= <filename_unix>
             | <filename_windows>
             | <filename_unix> ";" <filenames>
             | <filename_windows> ";" <filenames>
<filename_unix> ::= <string>
             | <string> <filename_unix>
             | <string> "." <string>
             | "/" <string> <filename_unix>
<filename_windows>  ::= <char> ":" <filename_windows2>
             | "\\" <string> "\" <filename_windows2>
<filename_windows2> ::= <string>
             | <string> "." <string>
             | "\" <string> <filename_windows2>
<var> ::= "${" <string> "}"
<string> ::= <char> <string'>
<string'> ::= <char>
             | <special_char>
             | <digit>
             | <char> <string'>
             | <special_char> <string'>
             | <digit> <string'>
<char> ::= a..z | A..Z
<special_char> ::= "_" | "-"
<number> ::= <digit>
         | <number> <digit>
<digit>  ::= 0..9
```

Figure 21. GRID-ADL syntax in BNF

transparent to the user. We believe a distributed hierarchical control organization meets the applications needs, is scalable and can alleviate the load of the submit machine avoiding stalling the user working machine.

The application submission and control is done through the following hierarchy of managers: (level 0) the user submits an application in a submission machine through the `Application Submit`; (level 1) the `Application Submit` partition the application in subgraphs and send to some `Submission Managers` the task descriptions; (level 2) an `Application Submit` instantiates on demand the `Task Managers` that will control the task submission to the local RMSs on the grid nodes; (level 3) RMSs on grid nodes receive requests from our `Task Managers` and schedule the tasks to be executed locally.

Our task submission model can be classified as a high-level scheduler [94] since it queries other schedulers for possible allocations. It can also be considered as a Metascheduler [89] because it allows to request resources of more than one machine for a single job.

5.3.1. Model Components The higher level of the application control must infer the DAG and make the partitioning. The user submits his/her application through a *submit machine* that is a machine that has the `Application Submit` component installed, which is our higher level controller. The `Application Submit` is in charge of:

- processing the user submit file describing the tasks to be executed;
- partition the tasks into subgraphs. The partitioning algorithm is chosen according to the kind of the DAG; and
- showing, in a user friendly way, the status and monitoring information about the application.

Note that the `Application Submit` does not have information about individual resources available in the system. It only keeps track of the `Submission Manager` status to avoid communicating with a failure or overloaded node.

Subgraphs defined by the partitioning algorithm are assigned to the second level controllers, called `Submission Managers`, which will instantiate the `Task Manager` processes to deal with the actual submission of the tasks to the nodes of the grid. This third level is necessary to isolate implementation details related to specific local resource managers.

The `Submission Manager` main functions are:

- to create daemons called `Task Manager` to control actual task execution. Each daemon keeps control of a subgraph of tasks defined by the partitioning;
- to keep information about computational resources; and
- to supply monitoring and status information useful to the user. It stores in log files the information in a synthetic way. These information are sent to the `Application Submit` that has the responsibility to present data to the user. This periodic information flow is also used to detect failures.

The `Task Manager` is responsible for communicating with remote machines and launching tasks to remote nodes. It works similarly to a wrapper being able to communicate with a specific local resource manager. For example, a `Task Manager` is instantiated to communicate with a grid node that uses PBS while another `Task Manager` can be instantiated to communicate with another grid node that uses Condor.

We assume that our hierarchy of managers is running in the local network to (1) avoid forcing that other sites run our daemons, and (2) to minimize communication time between managers.

5.3.2. Model Components Interaction Figure 22 illustrates the three main components of our model and their relationship.

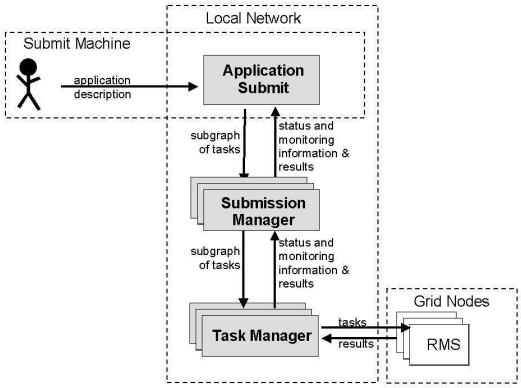

Figure 22. Hierarchical task management main components

When the user submits his/her application in the Submit Machine, the `Application Submit` can already be active or can be started due to the current request. When an `Application Submit`

becomes active, it broadcasts a message to its local network. All Submission Managers reply to this message to inform their location and status.

When an application submission request arrives to the active Application Submit, the application is partitioned in subgraphs. The Application Submit uses its local information and choose one or more Submission Managers to accomplish the required tasks. The choice is done based on the following criteria based on heuristics:

- the Submission Managers that have recently communicated with the Application Submit and reported that are not overloaded have preference to receive subgraphs. The periodical communication can detect when a Submission Manager is faulty or overloaded and thus not able to help in the tasks submission and control;
- the computational power of the machine, considering CPU and memory, determines the upper bound on the number of subgraphs a Submission Manager can receive. The more memory and CPU power a machine has, the more subgraphs its Submission Manager can handle. This aims at avoiding to overload the machine;
- the Application Submit keeps a weight for each Submission Manager. Greater values indicate powerful Submission Managers. This value is based on previous executions data and indicates how well the Submission Manager had accomplished the tasks it received.

The chosen Submission Managers will receive subgraphs in an internal representation. At this moment, there is no transfer of executables or input data files.

Then, periodically the Submission Managers will communicate to the Application Submit the execution progress. This communication allows online monitoring information to the user and also fault detection. Notice that we are assuming that all Application Submit and Submission Managers will run in machines that belong to the local network to reduce latency and network traffic when submitting tasks.

Communication between the Submission Managers can happen, since some tasks in different subgraphs can have dependencies. Therefore some synchronization points must be established. The Application Submit must send, included in the subgraph description, the identification of each manager that is related to this subgraph. For example, suppose a Submission Manager sm_2 has a task B which must be executed after task A assigned to Submission Manager sm_1. In this case, sm_1 must send a message to sm_2 when task A finishes.

Each Submission Manager must find the most suitable resources to run its subgraphs. A Submission Manager chooses a grid node using the following criteria:

- the Submission Managers keep a list of available grid nodes. Some subgraphs will have requirements that just some grid nodes can match. Thus, grid nodes must match tasks requirements to be selected;
- for each grid node, an upper bound on the number of subgraphs it can receive will be estimated. Besides, the ongoing submissions are taken into consideration;
- the Submission Manager keeps information about previous executions. It uses this information to calculate a weight based on the application characteristics. Greater values for the weight indicate "better" grid node candidates.

It is required a Task Manager, in the same machine of the Submission Manager, for each grid node a Submission Manager can access. Task Managers can be dynamically activated and deactivated according to the Submission Manager demands. The Submission Manager sends the subgraph to a Task Manager according to the grid node chosen.

The Task Manager is responsible for translating the internal subgraph description to the appropriate format for tasks submission. For example, a Task Manager that communicates with a Condor pool must prepare a Condor submit file and send the appropriate command to start tasks.

5.4. Discussion

The assumption of a huge number of tasks has important consequences to the scheduling architecture design. We cannot just submit tasks without controlling system parameters and flow control as some systems do. For example, MyGrid [81] considers that making a fast scheduling decision is more important. It is true for many applications, but for our target applications it is necessary to keep track of system information. For example, if the application takes several days to finish, two seconds to find a suitable cluster is not a problem.

Since it is generally hard to quantify analytically the efficacy of scheduling policies, experiments should be performed in different scenarios to perform the evaluation. However, repeatable experiments are very difficult to be conducted on highly distributed platforms. Specifically on grid environments, Buyya and Murshed [16] state that "it is impossible to create a repeatable and controlled environment for experimentation and evaluation of scheduling strategies. This is because resources in the grid span across multiple administrative domains, each with their own policies, users, and priorities." Then, simulation is presented as a good alternative for evaluating scheduling and distributed models in general. This is the approach we will be taking in this work. In the next section we will present some simulation tools and our simulation model.

6. Simulation Model

In this section, we discuss some of the available simulation tools suitable to grid related experiments (Subsection 6.1). We selected MONARC as our simulation platform, and we present some of its details (Subsection 6.2). Then, our designed and started to implement a simulation model (Subsection 6.3) is presented.

6.1. Grid Simulation Tools

Simulation is a powerful tool to test distributed models. It is cheaper, simpler, faster, and more flexible than running in a real environment, avoiding operational problems and implementation dependent issues. Besides, it is possible to reproduce experiments that otherwise would be non deterministic. It is easier to setup different parameters, simplifying debugging and monitoring of events.

Several tools exist for simulation of distributed systems that allow grid environment simulation [101]. Some are extensions of already available simulation tools while others were implemented from scratch.

We chose four tools to analyze in this work. All selected tools are ongoing works, but all present a stable distribution version for downloading. Besides all are free software. Note that some other tools are present in the literature such as Bricks [1, 106], but they are discarded since the code is not available for public download and thus could not be selected to our experiments.

We summarized some characteristics of the selected tools in Table 4, including their download addresses. We present and discuss in the next subsections these tools for grid simulation.

6.1.1. MicroGrid MicroGrid [99, 71], developed in the context of the GrADS Project [50, 11] from University of California at San Diego, enables evaluation of grid-oriented applications and services. Globus applications can run unmodified in MicroGrid.

Globus users can profit from MicroGrid to run their real applications into a virtual environment. Liu *et al.* [71] present experiments using the second version of MicroGrid, which includes peer-to-peer applications and applications using the GridFTP protocol. Several hardware and network configurations can be emulated using the actual user environment. For instance, the user can run a Globus application in any grid configuration while running the emulations on a homogeneous cluster.

The main concept to allow the emulation is the so called *virtualization* [99], which is the characteristic of applications perceiving only virtual grid resources independent of the physical resources being utilized. To provide virtualization, the MicroGrid intercepts all direct use of Globus' resources or information services.

MicroGrid allows the user to model computer, memory, and disk I/O performance. Since the network behavior is often a critical element of performance, one of the key components of MicroGrid is the packet-level network simulator *MaSSF*. MaSSF is a detailed network simulator, which models the behavior of each single IP packet. The second version uses *DaSSF* (Dartmouth Scalable Simulation Framework) [70] as the basis for MaSSF. MaSSF aims to include precise modeling of arbitrary network structure and on-line simulation (transferring of the communication traffic to the right destination with the right delay).

MicroGrid can also introduce in the simulation, link loss, router loss, and link speed variation.

6.1.2. SimGrid SimGrid [17, 66], presented as one of the projects of the Grid Research And Innovation Laboratory (GRAIL) in collaboration with the Laboratoire de l'Informatique du Parallélisme (LIP), was developed by researchers from University of California at San Diego and École Normal Supérieure de Lyon. SimGrid is a toolkit simulator that provides a C API for the simulation of distributed applications. In a single machine, SimGrid, and the next two simulators to be analyzed, can simulate any (heterogeneous) distributed environments.

SimGrid presents two interfaces or layers. **SG** is a low-level discrete-event simulation toolkit, that was deployed as the SimGrid first version [17]. In this interface, the simulation is done in terms of ex-

Simulation Tool	Free Software	Project Group	Download Address
MicroGrid	yes	GrADS (Grid Application Development Software Project)	`http://www-csag.ucsd.edu/projects/grid/MGridDownload.html`
SimGrid	yes	GRAIL (Grid Research And Innovation Laboratory) and LIP (Laboratoire de l'Informatique du Parallélisme)	`http://gcl.ucsd.edu/simgrid/dl/`
GridSim	yes	GRIDS Lab (Grid Computing and Distributed Systems Laboratory)	`http://www.cs.mu.oz.au/~raj/gridsim/gridsim2.1/`
MONARC 2	yes	MONARC (Models of Networked Analysis at Regional Centres for Large Hadron Collider Experiments)	`http://monalisa.cacr.caltech.edu/MONARC/Download/index.html`

Table 4. Simulation tools: software distribution

plicitly scheduling tasks on resources. It provides a set of core abstractions and functionalities that can be used to build simulators.

MSG is an application-oriented simulator [66] built using SG. This software layer presents a number of abstractions such as routing and a scheduling agent.

The basic abstractions provided by the MSG layer are the following:

- *agent* is an entity that makes scheduling decisions;
- *location* or host is the place in the simulated topology at which an agent runs;
- *task* is an activity of the simulated application (computation and/or data transfer);
- *path* interconnects locations and is an abstraction of message routing;
- *channel* is used to communication between agents.

With these abstractions, scheduling algorithms with SimGrid should always be described in terms of agents that run at locations and interact by sending, receiving, and processing simulated application tasks [66]. Agents do not have access to paths but can send a task to another location using a channel (mailbox number).

Legrand *et al.* [66] indicate that better simulations can be obtained if the network topology of a real computational grid is discovered using Effective Network View (ENV) [96]. Using the topology information, Network Weather Service (NWS) [115] can be used to record real-time traces which are directly usable in SimGrid.

We did not find any reference to faults simulation, such as link or router loss, but we believe that some kind of fault can be included into simulation through the trace facility.

6.1.3. GridSim GridSim [16, 76], developed in the Grid Computing and Distributed Systems Laboratory (GRIDS Lab) of the University of Melbourne, is a Java-based discrete-event grid simulation toolkit. GridSim is mainly concerned with the simulation of application schedulers for single or multiple administrative domains. Buyya and Murshed [16] present simulation results for a deadline and budget constrained scheduling system.

GridSim is built on top of SimJava. SimJava [58] is a general purpose discrete-event simulation package implemented in Java. The current version 2.2 presents facilities to simulate different grid testbeds and to generate default application scheduler source codes.

The main GridSim abstractions are:

- *user* represents a grid user;
- *broker* is a scheduler. Every job of a user is first submitted to its broker and the broker then schedules the parametric tasks according to the users' scheduling policy;
- *resource* is a grid resource. Brokers can query resources directly for their static and dynamic properties;
- *grid information service* provides resource registration services and keeps track of a list of resources available in the grid;
- *input and output* allow the flow of information among the GridSim entities.

We did not find any reference to faults simulation in the papers or in the API documentation.

6.1.4. MONARC 2 MONARC (Models of Networked Analysis at Regional Center) [67, 68] is a discrete-event simulation tool that allows distributed system simulation. It is part of the MONARC Project and has been implemented using Java technology involving a team

with members from CERN, Caltech, and Politehnica University of Bucharest. MONARC is now in its second version [109]. Legrand and Newman [67] report some experiments used to validate MONARC. They compare simulation results with theorical expected results based on queuing theory, and they conclude that the simulation is quite close to the theoretical model. They also show that the simulation tool reproduce results (job execution time) obtained with a real testbed.

MONARC is not intended to be a detailed simulator for basic components such as operating systems, data base servers or routers [68]. The simulation main abstractions are:

- *data container*: emulates a database file containing a set of objects;
- *database unit*: is a collection of containers;
- *job*: is a running task;
- *JobScheduler*: is the default scheduling policy, and different scheduling policies can be implemented extending this class;
- *regional centre*: is a complex entity containing a number of data servers and processing nodes, all connected to a LAN. It can contain a mass storage unit and can be connected to other regional centres.

We did not find any reference to faults simulation in the papers or in the API documentation.

6.1.5. Discussion We now discuss the main issues related to the four simulators presented. Table 5 shows the characteristics of the four simulation tools presented. We built it using some of the taxonomies proposed by Sulistio *et al.* [101]. The first column presents the name of the four simulators. The second column presents the usage taxonomy: simulator or emulator. *Simulator* is a tool that can model and represent the actual system. It runs at any speed relative to the real world. An *emulator* is a tool that acts like the actual system and is useful for accurate and reliable testing without having the real system. In our table, MicroGrid is classified as emulator and the others as simulators.

The simulation column presents the simulation taxonomy according to Sulistio *et al.*. Simulation comprises three properties:

- **presence of time**: A *static simulation* does not have real time as part of the simulation, in contrast to *dynamic simulation*;
- **basis of value**: Related to the values that the simulated entities can contain, a *discrete simulation* has entities only possessing one of many values within a finite range. A *continuous simulation* has entities possessing one of many values within an infinite range;
- **behavior**: Behavior defines how the simulation proceeds. A *deterministic simulation* has no random events occurring, so repeating the same simulation will always return the same simulation results. On the other hand, a *probabilistic simulation* has random events.

The platform column indicates the development language used. In our table, "gcc/Unix" means a C based implementation, which presents makefiles and scripts oriented to a Unix-like operating system. MicroGrid and SimGrid are implemented in C while GridSim and MONARC are implemented in Java. Both GridSim and MONARC 2 are multithreaded Java implementations. It is an advantage, since Java is portable and the use of threads could be used to get a scalable version of the simulators. SimGrid is implemented in C, which can possibly allow less time and memory consuming simulations although the system seems to have its portability limited.

Using Sulistio *et al.* programming framework taxonomy, MicroGrid and SimGrid are classified as *structured* because they implement a top-down structured programming design with control passing down the modules in a hierarchy. GridSim and MONARC are *object-oriented* since they express the program as a set of objects that communicate with one another to perform tasks. According to Sulistio *et al.* [101], the object-oriented framework is easier to create, maintain and reuse compared to the structured programming framework. However, the structured framework normally incurs less runtime overheads than the object-oriented.

Finally, the last column presents the design environment taxonomy. Design environment determines how the user uses the tool to design simulation models. All four tools are classified as *library*, since they provide a set of routines to be used in simulation model building.

We decided to use a simulator in our experiments, instead of the MicroGrid emulator, since we would have to port our examples to Globus. All three analyzed simulators have advantages that would justify its use in our experiments.

In our work, we chose to use MONARC as the simulation tool. MONARC is a popular tool among physicists. It is written in Java and thus is portable. Besides, MONARC presents high level abstractions suitable to our purposes. We also have access to its developers and programmers, through HEPGrid colaboration, which makes the task of modifying it easier.

Simulation Tool	Usage	Simulation	Platform	Programming framework	Design environment
MicroGrid	emulator	dynamic, continuous, deterministic	gcc/Unix	structured	library
SimGrid	simulator	static, discrete, deterministic	gcc/Unix	structured	library
GridSim	simulator	static, discrete, deterministic	Java	object-oriented	library
MONARC 2	simulator	static, discrete, deterministic	Java	object-oriented	library

Table 5. Simulation tools: comparison

6.2. MONARC 2 Overview

MONARC 2 is a Java object oriented simulator framework. It has seven main packages (set of related classes stored in the same directory):

- `engine`: main internal package, normally not used directly by simulation programmers. It contains the implementation of the core of the simulator. This package implements classes to manage active objects and provide communication between them. It also implements the class `Event` that defines simulation events.
- `network`: contains the classes used to simulate the network entities. There is also the package `network.protocol` containing the implementation of different transport protocols used in simulations.
- `monarc.center`: defines some of the base components of the cpu unit, farm and regional entities.
- `monarc.job`: contains different implementations of jobs used in the simulation, all being subclasses of `engine.AbstractJob` as for example `Job`, `JobFTP`, and `JobProcessData`
- `monarc.datamodel`: defines the database entities such as mass storage units and data base servers.
- `monarc.distribution`: defines statistical distributions that are used during simulation.
- `monarc.output`: is the main package for the classes that deal with the output of the simulation. There are also several other packages for dealing with output such as `monarc.output.FileClient` and `monarc.output.GraphicClient`.

Resource sharing is maintained between any discrete-event, including new job submission, through an interrupt driven mechanism. The interrupt driven mechanism is implemented into the simulation engine and it works transparently from the point of view of the simulation programmer.

Specific behavior of distributed data processing is mapped to threaded objects (also called active objects). There is a base class for threaded objects called `engine.Task` which implements the Java's `Runnable` Interface. It must be inherited by all the entities in the simulation that require a time dependent behavior [68]. Such entities are the running jobs, the database servers or the networks. There is a pool of thread objects to improve efficiency, but this is transparent to the user.

An important subclass of `engine.Task` is `monarc.center.Activity`. An Activity object is the base class for all activity processes and is used to estimate the time dependent job arrival patterns and correlation. These Activity objects are in fact the job injectors into the simulation framework.

The processing time of a task is evaluated when it starts and is implemented by setting the task object to a wait state. This wait state is not implemented with any 'sleep' or 'wait' operating system facility since the simulation engine must be able to change the time it remains in wait state any time another task starts or finishes using a shared resource.

MONARC receives a configuration file as input which is used in order to define the simulation. It is composed of sections, each one defining a simulation entity. Each section is defined by a name. The name of the section must appear as "[section_name]" and is the first line before the section. A section is composed of lines of the form "parameter=value". If a parameter name is wrong, the line is ignored. The first section is called the global section. This section can be defined without the name. The global section defines the basic parameters for the simulation such as queue type and maximum simultaneous threads that are running in parallel at any given time. To define a data unit, the name of the data unit must be declared in the global section. The data unit section must have the same name as the name declared there and spec-

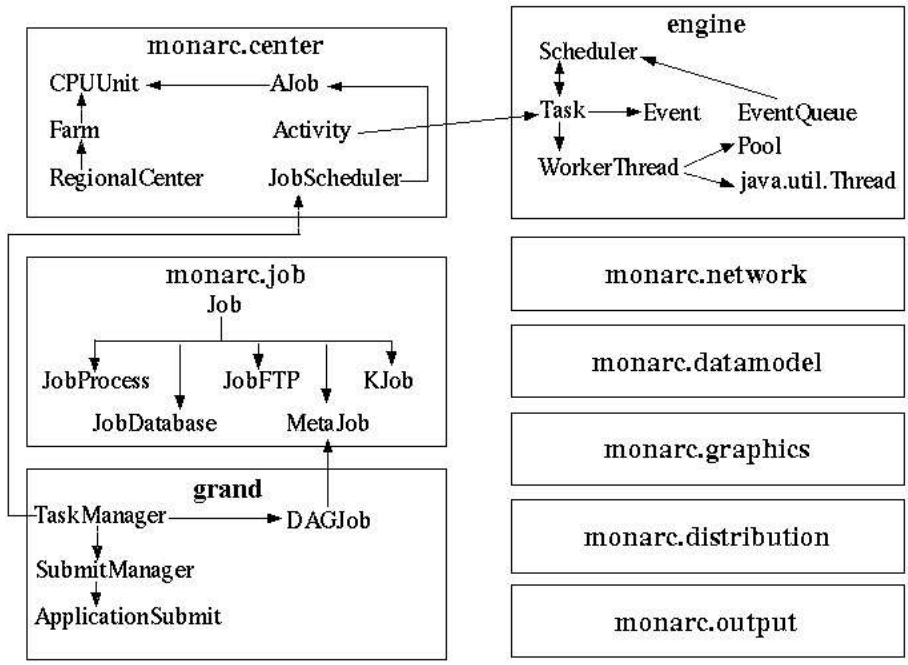

Figure 23. MONARC packages, the new package and the relation between the main classes

ify attributes such as CPU power and memory.

The regional center section is defined in order to specify what is inside a regional center. The name of the regional center must be the same as the name defined in the global section.

6.3. Simulation Model

To illustrate the main classes modeled, we adapted the figure presented in [68], which shows the main packages and the relation between the main classes. We created a new package grand (**g**rid **r**obust **a**pplicatio**n** **d**eployment).

We created a class for describing each of the application management levels of the hierarchy. They will communicate to make task dispatching. We also need to extend the `MetaJob` class, which represents a collection of jobs that are logically related and ought to be scheduled together, to have a directed acyclic graph structure.

7. Conclusion

This text presents an ongoing work that deals with application management in grid environments, focusing on applications that spread a very large number of tasks and data across the grid network.

We presented a general framework for grid environments, whose central idea is to have a flexible partitioning and a hierarchical organization where the load of the submit machine is shared with other machines. Our proposal wants to take advantage of hierarchical structures, because this seems to be the most appropriate organization for grid environments.

Based on the types of applications we can have been running on grid environments, and motivated by experiments carried out by physicists and bioinformatics people, we designed an architectural model to be implemented at a middleware level. In order to design our architecture, we consider that the resources and tasks are modeled as graphs.

Our architectural model handles three important issues in the context of applications that spread a huge number of tasks: (1) partitioning applications such that dependent tasks will be placed in the same grid node to avoid unnecessary migration of intermediate and/or transient data files, (2) partitioning applications such that tasks are allocated close to their required input data, and (3) distributing the submission process such that the submit machine do not get overloaded. As far as the author knows, this is the first proposal of a hierarchical application management system for grid envi-

ronments and is the first grid work that focuses on data locality to make scheduling decisions [113].

This architectural model relies on already available software components to solve issues such as allocation of tasks within a grid node, or authorization control.

Our model is concerned with scheduling tasks whose requirements are mainly memory and CPU. We are not considering special cases where tasks must have access to a specific resource such as a detector, a local database, or a special supercomputer. This kind of situation can occur in many real applications and must be considered in future works.

Our plan to the next months is (1) to refine our application partitioning model; (2) to refine our mapping model; and (3) to add our hierarchical model to the MONARC simulator. We intend to test our ideas using applications from Engineering, through a collaboration with the Laboratório de Projeto de Circuitos (LPC) at UFRJ, from High Energy Physics, through the HEPGrid collaboration, and from BioInformatics through a collaboration with the Department of Biostatistics and Medical Informatics at University of Wisconsin. As we defined the GRID-ADL application specification language, we have written a parser using JavaCC [59] for our language.

We also intend to obtain execution traces from real execution environments through the MonaLISA monitoring tool[79]. This will be very helpful to model the resources, resource load, application behavior and communication rate per link to be used as input to our simulation. The use of real data is fundamental to validate our simulation and to study how effective our hierarchical model is on a real environment.

References

[1] K. AIDA, A. TEKEFUSA, H. NAKADA, S. MATSUOKA, S. SEKIGUCHI, and U. NAGASHIMA. Performance evaluation model for scheduling in global computing systems. *The International Journal of High Performance Computing Applications*, 14(3):268–279, Fall 2000.

[2] G. ALLEN, D. ANGULO, I. FOSTER, G. LANFERMANN, C. LIU, T. RADKE, E. SEIDEL, and J. SHALF. The Cactus Worm: Experiments with dynamic resource discovery and allocation in a Grid environment. *The International Journal of High Performance Computing Applications*, 15(4):345–358, 2001.

[3] A. R. ANDINO, L. ARAÚJO, F. SÁENZ, and J. J. RUZ. Parallel execution models for constraint programming over finite domains. In *Proceedings of the International Conference Principles and Practice of Declarative Programming (PPDP'99)*, pages 134–151, Paris, France, September 29 – October 1 1999.

[4] N. ANDRADE, W. CIRNE, F. V. BRASILEIRO, and P. ROISENBERG. OurGrid: An approach to easily assemble grids with equitable resource sharing. In *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2003.

[5] J. ANNIS, Y. ZHAO, J. VOECKLER, M. WILDE, S. KENT, and I. FOSTER. Applying Chimera virtual data concepts to cluster finding in the sloan sky survey. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–14, Baltimore, Maryland, USA, November 16-22 2002.

[6] M. BAKER, R. BUYYA, and D. LAFORENZA. The Grid: International efforts in global computing. In *International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet (SSGRR 2000)*, Rome, Italy, July 31 – August 6 2000.

[7] V. C. BARBOSA. *An Introduction to Distributed Algoritms*. The MIT Press, London, England, 1996.

[8] V. C. BARBOSA and E. GAFNI. Concurrency in heavily loaded neighborhood-constrained systems. *ACM Transactions on Programming Languages and Systems*, 11(4):562–584, October 1989.

[9] G. BELL and J. GRAY. What's next in high-performance computing? *Communications of The ACM*, 45(2), February 2002.

[10] F. BERMAN. High-performance schedulers. In I. FOSTER and C. KESSELMAN, editors, *The Grid: Blueprint for a New Computing Infrastructure*, pages 279–309. Morgan Kaufmann, 1998.

[11] F. BERMAN, A. CHIEN, K. COOPER, J. DONGARRA, I. FOSTER, D. GANNON, L. JOHNSSON, K. KENNEDY, C. KESSELMAN, J. MELLOR-CRUMMEY, D. REED, L. TORCZON, and R. WOLSKI. The GrADS Project: Software support for high-level Grid application development. *The International Journal of High Performance Computing Applications*, 15(4):327–344, 2001.

[12] F. BERMAN, G. FOX, and T. HEY. *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., 1 edition, April 2003.

[13] B. BODE, D. M. HALSTEAD, R. KENDALL, Z. LEI, and D. JACKSON. The Portable Batch Scheduler and the Maui scheduler on linux clusters. In *Usenix Conference*, Atlanta, GA, USA, October 12–14 2000.

[14] C. B. BOERES and V. E. F. REBELLO. EasyGrid: Towards a framework for the automatic grid enabling of mpi applications. In *Proc of the 1st International Workshop on Middleware for Grid Computing (Middleware Workshops 2003)*, pages 256–260, Rio de Janeiro, Brazil, June 16–20 2003.

[15] J. J. BUNN and H. B. NEWMAN. Data intensive grids for high energy physics. In F. BERMAN, G. FOX, and T. HEY, editors, *Grid Computing: Making the Global Infrastructure a Reality*, pages 859–906. Wiley & Sons, 2003.

[16] R. BUYYA and M. MURSHED. GridSim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *The Journal of Concurrency and Computation: Practice and Experience (CCPE)*, 14(13–15), November–December 2002.

[17] H. CASANOVA. SimGrid: A toolkit for the simulation of application scheduling. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01)*, pages 430–437. IEEE, May 15–18 2001.

[18] T. L. CASAVANT and J. G. KUHL. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2), February 1988.

[19] Cave5D Release 1.4. `http://www.ccpo.odu.edu/~cave5d/`.

[20] S. CHANDRA and M. PARASHAR. ARMaDA: An adaptive application-sensitive partitioning framework for SAMR applications. In *Proceedings of the 14th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, pages 446–451, Cambridge, MA, USA, November 2002. ACTA Press.

[21] S. J. CHAPIN, D. KATRAMATOS, J. KARPOVICH, and A. GRIMSHAW. Resource management in Legion. In *Proceedings of the 5th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP '99), in conjunction with IPDPS '99*, April 1999.

[22] W. CIRNE and K. MARZULLO. OpenGrid: a user-centric approach for grid computing. In *Proceedings of the 13th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2001)*, September 2001.

[23] W. CIRNE, E. SANTOS-NETO, and L. BELTRÃO. Levantamento da comunidade grid no brasil, Novembro 2002. Disponível em `http://walfredo.dsc.ufpb.br/papers/levantamentoGridBrasil.v3.pdf`.

[24] The Compact Muon Solenoid (CMS) Project, 2003. `http://lcg.web.cern.ch/`.

[25] K. CZAJKOWSKI, I. FOSTER, N. KARONIS, C. KESSELMAN, S. MARTIN, W. SMITH, and S. TUECKE. A resource management architecture for metacomputing systems. In *Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, 1998.

[26] Directed Acyclic Graph Manager. `http://www.cs.wisc.edu/condor/dagman/`.

[27] E. DEELMAN, J. BLYTHE, Y. GIL, and C. KESSELMAN. Workfbw management in GriPhyN. In J. NABRZYSKI, J. M. SCHOPF, and J. WEGLARZ, editors, *Grid Resource Management: State of the Art and Future Trends*. Kluwer Academic Publishers, 2003.

[28] T. A. DeFANTI, I. FOSTER, M. E. PAPKA, R. STEVENS, and T. KUHFUSS. Overview of the I-WAY: Wide-area visual supercomputing. *The International Journal of Supercomputer Applications and High Performance Computing*, 10(2/3):123–131, Summer/Fall 1996.

[29] Distributed.Net, 2004. `http://www.distributed.net/`.

[30] I. d. C. DUTRA, D. PAGE, V. SANTOS COSTA, J. SHAVLIK, and M. WADDELL. Toward automatic management of embarrassingly parallel applications. In *26th International Conference on Parallel and Distributed Computing (Europar 2003)*, pages 509–516, Klagenfurt, Austria, August 2003.

[31] The EasyGrid project's research, reference and resource library. `http://easygrid.ic.uff.br/`.

[32] FAFNER overview, 2000. `http://www.npac.syr.edu/factoring/overview.html`.

[33] RSA130: Getting Started with FAFNER, 2000. `http://cs-www.bu.edu/cgi-bin/FAFNER/factor.pl`.

[34] D. G. FEITELSON and L. RUDOLPH. Parallel job scheduling: issues and approaches. In D. G. FEITELSON and L. RUDOLPH, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–18. Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[35] P.-O. FJÄLLSTRÖM. Algorithms for graph partitioning: A survey. *Linköping Electronic Articles in Computer and Information Science*, 3(010), 1998.

[36] I. FOSTER. What is the grid? a three poing checklist. *Grid Today*, 1(6), July 22 2002. Available at `http://www.gridtoday.com/02/0722/100136.html`.

[37] I. FOSTER and C. KESSELMAN. Computational Grids. In I. FOSTER and C. KESSELMAN, editors, *The Grid: Blueprint for a New Computing Infrastructure*, pages 15–52. Morgan Kaufmann, San Francisco, California, USA, 1 edition, 1998.

[38] I. FOSTER and C. KESSELMAN. The Globus project: A status report. In *Proc. IPPS/SPDP '98 Heterogeneous Computing Workshop*, pages 4–18, 1998.

[39] I. FOSTER and C. KESSELMAN. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Francisco, California, USA, 1 edition, 1998.

[40] I. FOSTER, C. KESSELMAN, J. NICK, and S. TUECKE. The physiology of the Grid: An open grid services architecture for distributed systems integration, June 2002. Available at `http://www.globus.org/research/papers/ogsa.pdf`.

[41] I. FOSTER, C. KESSELMAN, and S. TUECKE. The anatomy of the Grid: Enabling scalable virtual organizations. *The International Journal of High Performance Computing Applications*, 15(3):200–222, Fall 2001.

[42] I. FOSTER, J. VOECKLER, M. WILDE, and Y. ZHAO. Chimera: A virtual data system for representing, querying and automating data derivation. In *Proceedings of the 14th Conference on Scientifi c and Statistical Database Management*, Edinburgh, Scotland, July 2002.

[43] I. FOSTER, J. VOECKLER, M. WILDE, and Y. ZHAO. The virtual data grid: A new model and architecture for data-intensive collaboration. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, Asilomar, CA, USA, January 5-8 2003.

[44] J. FREY, T. TANNENBAUM, I. FOSTER, M. LIVNY, and S. TUECKE. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5:237–246, 2002.

[45] E. M. GAFNI and D. P. BERTSEKAS. Distributed algorithms for generating loop-free routes in networks with frequently changing topology. *IEEE Transactions on Communications*, 1(29):11–18, January 1981.

[46] M. R. GAREY and D. S. JOHNSON. *Computers and intractability: a guide to the theory of NP-Completeness*. Freeman, New York, 1979.

[47] C. F. GEYER, A. C. YAMIN, L. C. SILVA, P. K. VARGAS, I. d. C. DUTRA, M. PETEK, D. F. ADAMATTI, I. AUGUSTIN, and J. L. BARBOSA. Regional center and grid development in Brazil. In *LAFEX International School on High Energy Physics (LISHEP 2002) - GRID Workshop*, Rio de Janeiro, RJ, Brasil, February 7–8 2002.

[48] Global Grid Forum. `http://www.ggf.org/`.

[49] The Globus Toolkit, 2004. `http://www.globus.org/toolkit/`.

[50] The grid application development software (GrADS) project. `http://www.hipersoft.rice.edu/grads/`.

[51] D. A. GRIER. Systems for monte carlo work. In *Proceedings of the 19th Conference on Winter Simulation*, pages 428–433. ACM Press, 1987.

[52] A. GRIMSHAW, A. FERRARI, G. LINDAHL, and K. HOLCOMB. Metasystems. *Communications of the ACM*, 41(11):46–55, 1998.

[53] A. S. GRIMSHAW, A. FERRARI, F. KNABE, and M. HUMPHREY. Wide-area computing: Resource sharing on a large scale. *IEEE Computer*, 32(5):29–36, May 1999.

[54] GriPhyN – grid physics network. `http://www.griphyn.org/`.

[55] B. HENDRICKSON and T. G. KOLDA. Graph partitioning models for parallel computing. *Parallel Computing*, 26(12):1519–1534, 2000.

[56] B. HENDRICKSON, R. LELAND, and R. VAN DRIESSCHE. Enhancing data locality by using terminal propagation. In *29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture*, pages 565–584, Maui, Hawaii, USA, January 03–06 1996.

[57] K. HOGSTEDT, D. KIMELMAN, V. T. RAJAN, T. ROTH, and M. WEGMAN. Graph cutting algorithms for distributed applications partitioning. *ACM SIGMETRICS Performance Evaluation Review*, 28(4):27–29, 2001.

[58] F. HOWELL and R. McNAB. SimJava: A discrete event simulation package for Java with applications in computer systems modelling. In *Proceedings of the First International Conference on Web-based Modelling and Simulation*, San Diego, CA, USA, January 1998. Society for Computer Simulation.

[59] javacc: Javacc project home. `https://javacc.dev.java.net/`.

[60] J. P. JONES. NAS requirements checklist for job queuing/scheduling software. Technical Report NAS-96-003, NASA Ames Research Center, USA, April 1996. Available at `http://www.nas.nasa.gov/Research/Reports/Techreports/1996/nas-96-003-ab%stract.html`.

[61] G. KARYPIS and V. KUMAR. Multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pages 1–13. IEEE Computer Society, 1998.

[62] K. KRAUTER, R. BUYYA, and M. MAHESWARAN. A taxonomy and survey of grid resource management systems for distributed computing. *Software – Practice and Experience*, 32(2):135–164, 2002.

[63] S. KUMAR, S. K. DAS, and R. BISWAS. Graph partitioning for parallel applications in heterogeneous grid environments. In *International Parallel and Distributed Processing Symposium (IPDPS'02)*, Fort Lauderdale, CA, USA, April 15–19 2002.

[64] K.-i. KURATA, C. SAGUEZ, G. DINE, H. NAKAMURA, and V. BRETON. Evaluation of unique sequences on the european data grid. In *Proceedings of the First Asia-Pacifi c bioinformatics conference on Bioinformatics 2003*, pages 43–52. Australian Computer Society, Inc., 2003.

[65] D. LAFORENZA. Grid programming: some indications where we are headed. *Parallel Computing*, 28(12):1733–1752, December 2002.

[66] A. LEGRAND, L. MARCHAL, and H. CASANOVA. Scheduling distributed applications: the SimGrid simulation framework. In *Proceedings of the 3rd International Symposium on Cluster Computing and the Grid (CCGrid'03)*, pages 138–145. IEEE, May 12 - 15 2003.

[67] I. LEGRAND and H. B. NEWMAN. The MONARC toolset for simulating large network-

distributed processing systems. In *Winter Simulation Conference 2000*, pages 1794–1801, 2000.

[68] I. C. LEGRAND, C. M. DOBRE, and C. STRATAN. MONARC 2 (Models of Networked Analysis at Regional Centers) – distributed systems simulation. http://monalisa.cacr.caltech.edu/MONARC/Papers/MONARC_Implementation.zi%p.

[69] M. J. LEWIS, A. J. FERRARI, M. HUMPHREY, J. F. KARPOVICH, M. M. MORGAN, A. NATRAJAN, A. NGUYEN-TUONG, G. S. WASSON, and A. S. GRIMSHAW. Support for extensibility and site autonomy in the Legion Grid system object model. *Journal of Parallel and Distributed Computing*, (63):525–538, May 2003.

[70] J. LIU and D. M. NICOL. DaSSF 3.1 User's Manual, April 2001. http://www.cs.dartmouth.edu/~jasonliu/projects/ssf/papers/dassf-manual-%3.1.ps.

[71] X. LIU, H. XIA, and A. CHIEN. Network emulation tools for modeling grid behaviors. Submitted to The 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003). Available at http://www-csag.ucsd.edu/papers/ccgrid2003-final.pdf.

[72] Load Sharing Facility (LSF). http://accl.grc.nasa.gov/lsf/.

[73] Plataform LSF Family. http://www.platform.com/products/LSFfamily/.

[74] P. LU. The Trellis project. http://www.cs.ualberta.ca/~paullu/Trellis/.

[75] R. S. MONTERO, E. HUEDO, and I. M. LLORENTE. Grid resource selection for opportunistic job migration. In *26th International Conference on Parallel and Distributed Computing (Europar 2003)*, pages 366–373, Klagenfurt, Austria, August 2003.

[76] M. MURSHED and R. BUYYA. Using the GridSim toolkit for enabling grid computing education. In *Proceedings of the International Conference on Communication Networks and Distributed Systems Modeling and Simulation (CNDS 2002)*, San Antonio, Texas, USA, January 27–31 2002.

[77] Z. NEMETH and V. SUNDERAM. A comparison of conventional distributed computing environments and computational grids. In *Proceedings of the International Conference on Computational Science (ICCS2002)*, pages 729–738, Amsterdam, Netherlands, April 2002. LNCS 2329.

[78] Z. NEMETH and V. SUNDERAM. A formal framework for defi ning grid systems. In *Proceedings of the Second IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID2002)*, pages 202–211, Berlin, Germany, May 21–24 2002.

[79] H. B. NEWMAN, I. C. LEGRAND, P. Galvez, R. Voicu, and C. Cirstoiu. MonALisa: A distributed monitoring service architecture. In *Computing in High Energy and Nuclear Physics (CHEP03)*, La Jolla, California, USA, March 24-28 2003.

[80] Open PBS (Portable Batch System). http://www.openpbs.org/main.html.

[81] D. d. S. PARANHOS, W. CIRNE, and F. V. BRASILEIRO. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In *Proceedings of the 26th International Conference on Parallel and Distributed Computing (Euro-Par 2003)*, August 2003.

[82] S.-M. PARK and J.-H. KIM. Chameleon: A resource scheduler in a data grid environment. In *Proc of 3st International Symposium on Cluster Computing and the Grid*, pages 258–, Tokyo, Japan, May 12 - 15 2003.

[83] PBS Pro Home. http://www.pbspro.com/.

[84] Planning for execution in grids. http://www.isi.edu/~deelman/pegasus.htm.

[85] M. R. PEREIRA, P. K. VARGAS, F. M. G. FRANÇA, M. C. S. CASTRO, and I. d. C. DUTRA. Applying Scheduling by Edge Reversal to constraint partitioning. In *The 15th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2003)*, São Paulo, SP, November 10-12 2003.

[86] R. RAMAN, M. LIVNY, and M. SOLOMON. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, Chicago, USA, July 28-31 1998.

[87] R. A. REAL, A. YAMIN, I. AUGUSTIN, L. C. d. SILVA, G. FRAINER, J. L. BARBOSA, and C. GEYER. Tratamento da incerteza no escalonamento de recursos em pervasive computing. In *Conferência IADIS Ibero-Americana WWW/Internet*, pages 167–170, Algarve, Portugal, November 2003.

[88] R. A. REAL, A. YAMIN, L. d. SILVA, G. FRAINER, I. AUGUSTIN, J. BARBOSA, and C. F. R. GEYER. Resource scheduling on grid: handling uncertainty. In *Proceedings of the IEEE/ACM 4TH International Workshop on Grid Computing*, Phoenix, Arizona, November 2003.

[89] M. ROEHRIG, W. ZIEGLER, and P. WIEDER. Grid scheduling dictionary of terms and keywords. Document gfd-i.11, Global Grid Forum, Nov 2002. Available at http://forge.gridforum.org/projects/ggf-editor/document/GFD-I.11/en/1.

[90] D. D. ROURE, M. A. BAKER, N. R. JENNINGS, and N. R. SHADBOLT. The evolution of the

Grid. In F. BERMAN, G. FOX, and T. HEY, editors, *Grid Computing: Making the Global Infrastructure a Reality*, pages 65–100. Wiley & Sons, 2003.

[91] D. D. ROURE, N. R. JENNINGS, and N. R. SHADBOLT. The semantic Grid: A future e-science infrastructure. In F. BERMAN, G. FOX, and T. HEY, editors, *Grid Computing: Making the Global Infrastructure a Reality*, pages 437–470. Wiley & Sons, 2003.

[92] V. SANDER and *et al*. Networking issues of grid infrastructures. Document draft-ggf-ghpn-netissues-0, version 1, Global Grid Forum, June 2003. Available at `http://forge.gridforum.org/projects/ggf-editor/document/Networking_Issu%es_of_Grid_Infrastructures/en/1/Networking_Issues_of_Grid_Infrastructures.pdf`.

[93] T. SANDHOLM and J. GAWOR. Globus Toolkit 3 Core – A Grid Service Container Framework, May 2003. White paper. Available at `http://www-unix.globus.org/toolkit/3.0beta/ogsa/docs/gt3_core.pdf`.

[94] U. SCHWIEGELSHOHN and R. YAHYAPOUR. Attributes for communication between scheduling instances, December 2001. Available at `http://ds.e-technik.uni-dortmund.de/~yahya/ggf-sched/WG/sched_attr/Sch%edWD.10.6.pdf`.

[95] SETI@home – The Search for Extraterrestrial Intelligence at Home, 2004. `http://setiathome.ssl.berkeley.edu/`.

[96] G. SHAO, F. BERMAN, and R. WOLSKI. Using effective network views to promote distributed application performance. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, June 1999.

[97] M. SINGHAL and N. G. SHIVARATRI. *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems*. MIT Press, New York, 1994.

[98] L. SMARR and C. E. CATLETT. Metacomputing. *Communications of the ACM*, 35(6), June 1992.

[99] H. J. SONG, X. LIU, D. JAKOBSEN, R. BHAGWAN, X. ZHANG, K. TAURA, and A. CHIEN. The MicroGrid: a scientific tool for modeling computational grids. In *Proceedings of SC2000*, Dallas, Texas, USA, November 4-10 2000.

[100] A. V. STAICU, J. R. RADZIKOWSKI, K. GAJ, N. ALEXANDRIDIS, and T. EL-GHAZAWI. Effective use of networked reconfigurable resources. In *2001 MAPLD International Conference*, Laurel, Maryland, September 2001.

[101] A. SULISTIO, C. S. YEO, and R. BUYYA. A taxonomy of computer-based simulations and its mapping to parallel and distributed systems simulation tools. International Journal of Software: Practice and Experience, Wiley Press, 2003 (accepted in Sept. 2003 and in print) Available at `http://www.cs.mu.oz.au/~raj/papers/simulationtaxonomy.pdf`.

[102] Sun Microsystems. Sun cluster grid architecture. Sun one grid engine white papers, Sun Microsystems, 2002. Available at `http://wwws.sun.com/software/grid/SunClusterGridArchitecture.pdf`.

[103] A. S. TANENBAUM and R. VAN RENESSE. Distributed operating systems. *ACM Computing Surveys (CSUR)*, 17(4):419–470, December 1985.

[104] T. TANNENBAUM, D. WRIGHT, K. MILLER, and M. LIVNY. Condor – a distributed job scheduler. In T. STERLING, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2002.

[105] TAREK EL-GHAZAWI, P. I. et al. Conceptual comparative study of job management systems. Technical report, George Mason University, USA, February 21 2001. Available at `http://ece.gmu.edu/lucite/reports/Conceptual_study.PDF`.

[106] A. TEKEFUSA, O. TATEBE, S. MATSUOKA, and Y. MORITA. Performance analysis of scheduling and replication algorithms on grid datafarm architecture for high-energy physics applications. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12)*, pages 34–43, Seattle, Washington, June 22–24 2003.

[107] D. THAIN, T. TANNENBAUM, and M. LIVNY. Condor and the Grid. In F. BERMAN, G. FOX, and T. HEY, editors, *Grid Computing: Making The Global Infrastructure a Reality*. John Wiley, 2003.

[108] The Legion Group. Legion 1.8 – developer manual, 2001.

[109] The MONARC Project – Models of Networked Analysis at Regional Centres for LHC Experiments. Distributed computing simulation. `http://monarc.web.cern.ch/MONARC/sim_tool/`.

[110] S. TUECKE, K. CZAJKOWSKI, I. FOSTER, J. FREY, S. GRAHAM, C. KESSELMAN, and P. VANDERBILT. Grid service specification, 2002. Available at `http://www.gridforum.org/ogsi-wg/drafts/GS_Spec_draft03_2002-07-17.pdf`.

[111] S. S. VADHIYAR and J. J. DONGARRA. A metascheduler for the grid. In *11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11)*, pages 343–354, Edinburgh, Scotland, July 24 – 26 2002.

[112] S. S. VADHIYAR and J. J. DONGARRA. A performance oriented migration framework for the grid. In *3rd International Symposium on Cluster Computing and the Grid (CCGRID 2003)*, pages 366–373, Tokyo, Japan, May 12 – 15 2003.

[113] P. K. VARGAS, I. d. C. DUTRA, and C. F. GEYER. Hierarchical resource management and application control in grid environments. Technical report, COPPE/Sistemas - UFRJ, 2003. Relatório Técnico ES-608/03.

[114] V. WELCH, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke. Security for grid services. In *12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, Seattle, Washington, June 22–24 2003. Also available at `http://www.globus.org/security/GSI3/GT3-Security-HPDC.pdf`.

[115] R. WOLSKI, N. SPRING, and J. HAYES. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15(5–6):757–768, October 1999.

[116] D. WRIGHT. Cheap cycles from the desktop to the dedicated cluster: combining opportunistic and dedicated scheduling with condor. In *Proceedings of the Conference on Linux Clusters: The HPC Revolution*, Champaign - Urbana, IL, USA, June 2001.

[117] The WS-Resource Framework, 2004. `http://www.globus.org/wsrf/`.

[118] A. YAMIN, I. AUGUSTIN, J. BARBOSA, and C. F. GEYER. ISAM: a pervasive view in distributed mobile computing. In *Proceedings of the IFIP TC6 / WG6.2 & WG6.7 Conference on Network Control and Engineering for QoS, Security and Mobility (NET-CON 2002)*, pages 431–436, October 23–25 2002.

[119] A. YAMIN, I. AUGUSTIN, J. BARBOSA, L. C. d. SILVA, R. A. REAL, G. CAVALHEIRO, and C. F. GEYER. Towards merging context-aware, mobile and grid computing. *International Journal of High Performance Computing Applications*, 17(2):191–203, June 2003.

[120] A. YAMIN, J. BARBOSA, L. C. d. SILVA, R. A. REAL, C. F. GEYER, I. AUGUSTIN, and G. CAVALHEIRO. A framework for exploiting adaptation in high heterogeneous distributed processing. In *Proceedings of the XIV Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Vitória, ES, Brasil, October 28–30 2002.