Application Execution Management on the Integrade Opportunistic Grid Middleware

Francisco José da Silva e Silva[a], Fabio Kon[b], Alfredo Goldman[b], Marcelo Finger[b], Raphael Y. de Camargo[c], Fernando Castor Filho[d], Fábio M. Costa[e]

[a] *Federal University of Maranhão, Department of Informatics, São Luís, Brazil*
[b] *University of São Paulo, Department of Computer Science, São Paulo, Brazil*
[c] *Federal University of ABC, Santo André, Brazil*
[d] *Federal University of Pernambuco, Informatics Center, Recife, Brazil*
[e] *Federal University of Goiás, Institute of Informatics, Goiânia, Brazil*

Abstract

The InteGrade project is a multi-university effort to build a novel grid computing middleware based on the opportunistic use of resources belonging to user workstations, which is also known as volunteer computing. The InteGrade middleware currently enables the execution of sequential, bag-of-tasks, and parallel applications that follow the BSP or MPI programming models.

This article presents the lessons learned over the last five years of InteGrade development and describes the achieved solutions concerning the support for robust application execution. The contributions cover the related fields of application scheduling, execution management, and fault tolerance. We present our solutions, describing their implementation principles and evaluation through the analysis of several experimental results.

Key words: grid computing, volunteer computing, resource management, fault tolerance

Email address: `fssilva@deinf.ufma.br` (Francisco José da Silva e Silva)

1. Introduction

The success of grid systems can be verified by the increasing number of middleware systems, actual production grids, and dedicated forums that appeared in recent years. The use of Grid Computing technology is increasing rapidly, reaching more scientific fields and encompassing a growing body of applications (Foster and Kasselman , 2003).

A grid might be seen as a way to interconnect clusters that is much more convenient than the construction of huge clusters. Another possible approach for conceiving a grid is the opportunistic use of workstations of regular users, which is also known as volunteer computing. The focus of an opportunistic grid middleware is not on the integration of dedicated computer clusters (e.g., Beowulf) or supercomputing resources, but on taking advantage of idle computing cycles of regular computers and workstations that can be spread across several administrative domains.

During the last five years, our research group has been engaged on the development of the InteGrade project[1], a multi-university effort to build a robust and flexible middleware for opportunistic grid computing. InteGrade's main goal is to be an opportunistic grid environment with support for highly-coupled parallel applications. This goal may sound antagonistic at a first glance, but we intend to provide a parallel processing environment for institutions that have hundreds or thousands of personal computers, but do not possess the resources to acquire a powerful dedicated cluster or parallel machine. To do that, we focus on leveraging the idle computing power of

[1]Homepage: `http://www.integrade.org.br`

existing commodity workstations.

In this article, we focus on the support provided by InteGrade for application execution, covering three key related issues concerning the development of an opportunistic grid middleware: resource management and availability prediction for application scheduling, execution management, and fault tolerance.

Resource management encompasses challenges such as how to efficiently monitor a large number of highly distributed computing resources belonging to multiple administrative domains. On opportunistic grids, this issue is even harder due to the dynamic nature of the execution environment, where nodes can join and leave the grid at any time due to the use of the non-dedicated machines by their regular (non-grid) users. An effective monitoring infrastructure is crucial for performing appropriate application scheduling decisions and for timely detecting failures. Besides the current state of grid resources, the application scheduler should also take into consideration a prediction of the future availability of computing resources. This is particularly useful on opportunistic grids, as users of non-dedicated machines can resume local processing, forcing grid tasks to either migrate to other grid machine or abort and possibly restart at another machine.

With respect to application **execution management**, which also includes monitoring, there must be user-friendly mechanisms to execute applications in the grid environment, to control the execution of jobs, and to provide tools to collect application results and to generate reports about current and past situations. Application execution management should encompass all execution models supported by the middleware. The InteGrade middleware currently allows the execution of sequential, bag-of-tasks, and parallel applications that follow the BSP or MPI programming models. A related issue concerning application execution is the management of application data, which includes the application binaries, and input and output data. On grid environments, applications usually generate large amounts of data, which, combined with the environment's large scale, turns a centralized approach for data storage inappropriate. On the other hand, building a flexible distributed storage system that provides high data availability and fast data access in a dynamic environment, such as an opportunistic grid, is a challenging task.

Finally, **fault tolerance** comprises a major requirement for grid middleware as grid environments are highly prone to failures, a characteristic amplified on opportunistic grids due their dynamism and the use of non-dedicated machines, leading to a non-controlled computing environment. An efficient and scalable failure detection mechanism must be provided by the grid middleware, along with a means for automatic application execution recovery, without requiring human intervention.

InteGrade presents solutions to those problems. There are special modules in charge of controlling the resources for opportunistic grids, monitoring them, and providing tools to analyze the gathered data. There are special tools for managing the execution of applications, which guarantee secure mechanisms to store application binaries, their input and the generated output. InteGrade also provides mechanisms for fault tolerance, including both replication and checkpointing, in a configurable way, allowing multiple choices for data storage. The middleware also provides support for highly-coupled parallel applications using both the MPI and BSP standard programming models, as well as mobile agents for long-running sequential applications.

All these solutions, when used together, provide a powerful framework that allows for the execution of sequential and parallel applications in an opportunistic environment. Indeed, as all these solutions were designed to be decentralized and distributed, a good level of scalability is obtained.

This article concentrates on the new developments on the InteGrade project carried out since its prototype architecture was presented five years ago; for a detailed description of that early work, the reader should refer to (Goldchleger et al. , 2004).

In the next section we present an overview of the current InteGrade architectural model. In Section 3, we discuss the use of resource monitoring data to infer future availability of resources, which is taken into account during the scheduling process to minimize application migration and restart. Section 4 describes aspects related to application execution management, emphasizing the mechanisms that allow the execution of parallel applications that follow the MPI or BSP models; this section also addresses the management of application input and output data. In Section 5, we describe the available mechanisms used to guarantee application progress, circumventing failures. Finally, in Section 6 we conclude the article presenting open research problems and ongoing work.

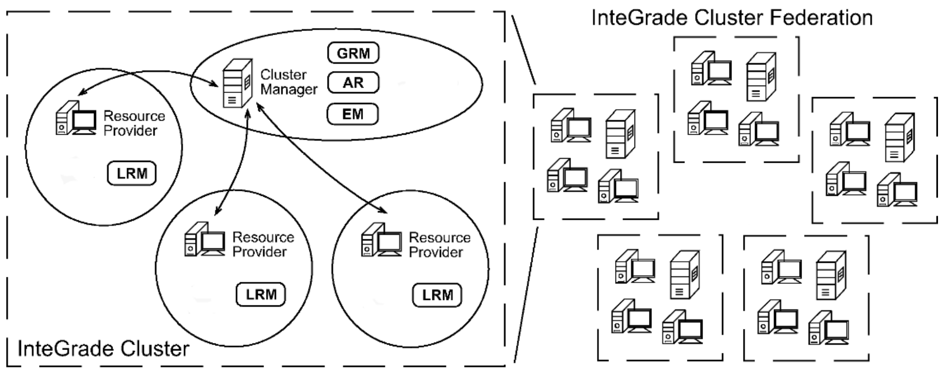

Figure 1: InteGrade architecture.

2. InteGrade Overview

The basic architectural unit of an InteGrade grid is a cluster, a collection of machines usually connected by a local network. Clusters can be organized in a hierarchy, enabling the construction of grids with a large number of machines. Each cluster contains a *Cluster Manager* node that hosts InteGrade components responsible for managing cluster resources and for inter-cluster communication. Other cluster nodes are called *Resource Providers* and export part of their resources to the grid. They can be either shared with local users (e.g., secretaries using a word processor) or dedicated machines.

InteGrade currently allows the execution of three application classes: (a) sequential applications, where the task to be run is assigned to a single grid node; (b) parametric or bag-of-tasks applications, where several copies of a task are assigned to different grid nodes, each of them processing a subset of the input data independently and without exchanging data; (c) parallel applications following the BSP or MPI models, whose processes exchange data among themselves using message passing or shared memory abstractions.

The InteGrade architecture comprises several components. Figure 1 shows the components that enable application execution, which are the following.

Application Repository (AR): before being executed, an application must be previously registered with the Application Repository. This component stores the application description (metadata) and binary code.

Application Submission and Control Tool (ASCT): a graphical user interface that allows users to browse the content of the Application Repository, submit applications, and control their execution. Alternatively, applications can be submitted via the **InteGrade Grid Portal**, a Web interface similar to ASCT.

Local Resource Manager (LRM): a component that runs on each cluster node, collecting information about the state of resources such as memory, CPU, disk, and network. It is also responsible for instantiating and executing applications scheduled to the node.

Global Resource Manager (GRM): manages cluster resources by receiving notifications of resource usage from the LRMs in the cluster (through an information update protocol) and runs the scheduler that allocates tasks to nodes based on resource availability; it is also responsible for communication with GRMs in other clusters, allowing applications to be scheduled for execution in different clusters. Each cluster has a GRM and, collectively, the GRMs form the Global Resource Management service.

Execution Manager (EM): maintains information about each application submission, such as its state, executing node(s), input and output parameters, submission and termination timestamps. It also coordinates the recovery process in case of application failures.

From this basic description, we can move on to InteGrade's core issues related to application execution, which include resource management and availability prediction for an appropriate scheduling, application execution management and fault tolerance, which are explained in the following sections.

3. Resource Availability Prediction

The success of an opportunistic grid depends on a good scheduler. An idle machine is available for grid processing, but whenever its local users need their resources back, grid applications executing at that machine must either migrate to another grid machine or abort and possibly restart at another machine. In both cases, there is considerable loss of efficiency for grid applications. A solution is to avoid such interruptions by scheduling grid tasks on machines that are expected to remain idle for the duration of the task.

InteGrade predicts each machine's idle periods by locally performing Use Pattern Analysis of machine resources at each machine on the grid. Currently, four different types of machine resources are monitored: CPU use, RAM availability, disk space and swap space.

Use pattern analysis deals with *machine resource use objects.* Each object is a vector of values representing the time series of a machine's resource use, as illustrated in Figure 1. The sampling of a machine's resource use is performed at a fixed rate (currently, once every 5 minutes) and grouped into objects covering 48 hours with a 24-hour overlap between consecutive objects. We employ 48-hour long objects so as to have enough past information to be used in the runtime prediction phase.

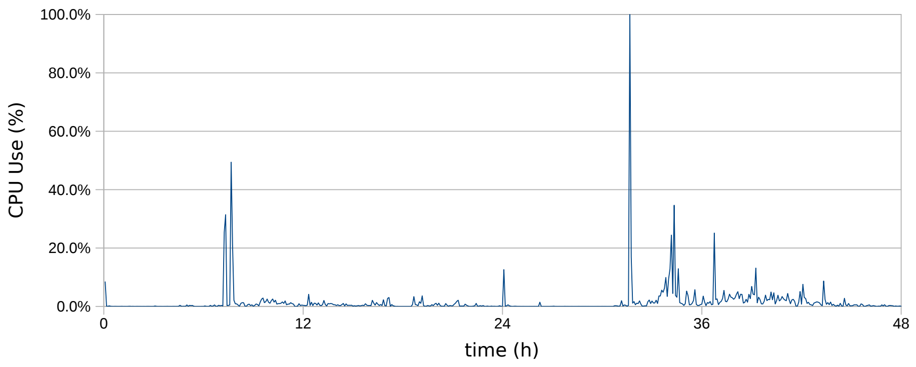

Figure 2: An object representing a machine's CPU use over a 48h period

Use Pattern Analysis performs unsupervised machine learning (Barlow , 1999; Theodoridi and Koutroumba , 2003) to obtain a fixed number of *use classes*, where each class is represented by its *prototypical* object. The idea is that each class represents a frequent use pattern, such as a busy work day, a light work day or a holiday. As in most machine learning processes, there are two phases involved in the process, which in the InteGrade architecture are implemented by a module called Local Use Pattern Analyzer (LUPA), as follows.

The Learning Phase. Learning is performed off-line, using 60 objects collected by LUPA during the machine regular use. A clustering algorithm (Sokal , 1996; Everitt, Landau and Leese , 2001) is applied to the training data, such that each cluster corresponds to a use class, represented by a prototypical object, which is obtained by averaging over the elements of the class. Learning can occur only when there is a considerable mass of data. In our case, we require at least two months of data. As data collection proceeds, more data and more representative classes are obtained.

The Decision Phase. There is one LUPA module per machine on the grid. Requests are sent by the scheduler specifying the amount of resources (CPU, disk space, RAM, etc.) and the expected duration needed by an application to be executed at that machine. The LUPA module decides whether this machine will be available for the expected duration, as explained below. LUPA is constantly keeping track of the current use of resources. For each resource, it focuses on the recent history, usually the last 24 hours, as illustrated in Figure 3, and computes a distance between the recent history and each of the use classes learned during the training phase. This distance takes into account the time of the day in which the request was made, so that the recent history is compared to the corresponding times in the use classes. The class with the smallest distance is the *current use class*, which is used to predict the availability in the near future. If all resources are predicted to be available, then the application is scheduled to be executed; otherwise, it is rejected.

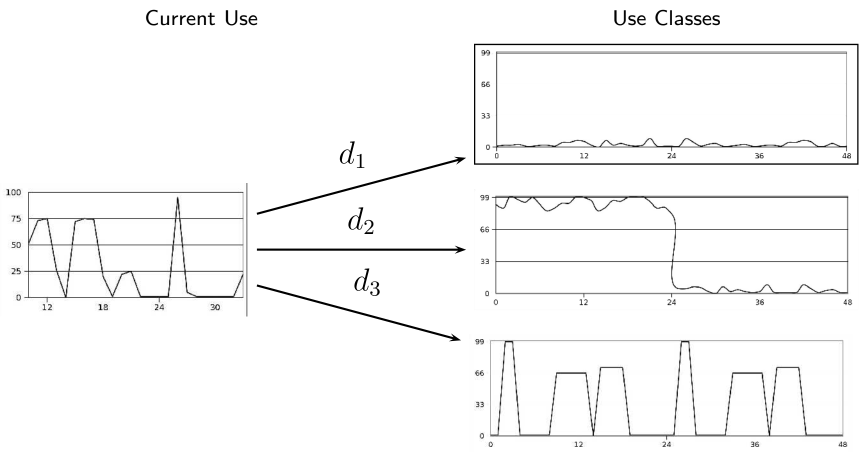

Figure 3: Estimation of class pertinence for use prediction

This scheduling strategy has been tested via simulation, using data collected from real machines on the grid, and was compared against two other strategies, namely: one that projects the availability at the time of the request into the future (alt1); and one that projects into the future a fixed valued obtained from the average use of the past 24 hours (alt2).

Table 1 presents a summary of the trace-based simulation results with data collected over a period of 7 months. Results are presented for two kinds of machines, single-user and multi-user machines, which have quite different use profiles; and for two types of resource availability prediction: memory and CPU. At random instants of time, a request is issued for a random amount of resources during a random amount of time, and a prediction for availability is made at that time; the prediction of availability is considered correct if the amount of resource required remains available for the full duration of the request; a prediction of unavailability is considered correct if at some point of this interval the requested amount of resources is unavailable. Three basic results are shown. First, the percentage of correct predictions. Second, the probability that the proposed strategy performs better than strategy alt1. Third, the probability the proposed strategy performs better than strategy alt2. Probabilities were computed assuming a normal distribution of requests, where the mean and standard deviation were obtained from the training data. All results in Table 1 were obtained by configuring the clustering algorithm such that 5 clusters were found. The same experiments were run for 10 clusters, with basically the same results.

Table 1 allows us to conclude that the proposed strategy is very successful in predicting idleness, with success rates above 90% at all times. It also shows

Resource	Machine	Correct Predictions	Beats alt1	Beats alt2
Memory	single user	$99,47\%$	$47,77\%$	$99,12\%$
Memory	multi user	$99,08\%$	$64,97\%$	$90,37\%$
CPU	single user	$97,57\%$	$50,21\%$	$87,89\%$
CPU	multi user	$92,52\%$	$91,23\%$	$91,35\%$

Table 1: Simulation results for 5 clusters

that single-user machines are easier to predict than multi-user machines. The proposed method also performs better than the proposed alternatives in all cases except one, namely the prediction of memory availability for single user machines, in which case the extremely simple strategy of projecting into the future the current availability seems to perform better.

A complete description of Use Pattern Analysis method can be found at (Finger, Bezerra and Conde , 2008).

4. Managing Application Executions

Once scheduled to proper grid resources, the tasks that comprise a submitted application can be instantiated and begin their execution. Executing computationally intensive parallel applications on dynamic heterogeneous environments, such as computational grids, is a daunting task. This is particularly true when using non-dedicated resources, as in the case of opportunistic computing, where one uses only the idle periods of the shared machines. In this scenario, the execution environment is typically highly dynamic, with resources periodically leaving and joining the grid. When a resource becomes unavailable, due to a failure or simply because the machine owner requests its use, the system needs to perform the necessary steps to restart the tasks on different machines. In the case of BSP or MPI parallel applications, the problem is even worse, since all processes that comprise the application may need to be restarted from a consistent distributed checkpoint.

The Execution Manager (EM) module is in charge of managing application execution, including sequential, bag-of-tasks, MPI, and BSP applications. The EM maintains information regarding each application executing on its cluster, such as the execution status, the nodes where the application processes are running and the names of the application input and output files. The EM is also responsible for restarting applications that had processes executing on a machine that became unavailable.

The following subsections describe how InteGrade implements the support for managing the execution of MPI and BSP parallel applications. Next, we describe the management of application data, which includes the application binaries, input, and output data.

4.1. Managing BSP Applications

InteGrade's support for executing BSP applications adheres to the Oxford BSP API [2], targeted for the C language. Thus, an application based on the Oxford BSPlib can be executed over InteGrade with little or even no modification of its source code, requiring only its recompilation and linkage with the appropriate InteGrade libraries.

A BSP computation proceeds in a series of global supersteps. Each superstep comprises three ordered stages: (1) *concurrent computation*: computa-

[2]The Oxford BSP Toolset `http://www.bsp-worldwide.org/implmnts/oxtool`

tions take place on every participating process. Each process only uses values stored on its local memory. Computations are independent in the sense that they occur asynchronously of all others; (2) *communication*: at this stage, the processes exchange data between themselves; (3) *barrier synchronization*: when a process reaches this point (the barrier), it waits until all other processes have finished their communication actions. The synchronization barrier is the end of a superstep and the beginning of another one.

Two inter-process communication models are available: (a) *Direct Remote Memory Access* (DRMA), which allows a process to read from and write to the remote address space of another process, and (b) *Bulk Synchronous Message Passing* (BSMP), that implements message passing between processes. In the DRMA model, an application process can register a local memory address as a virtual shared address using the `bsp_pushregister()` method. Once registered, processes are allowed to write and read from virtual memory locations through the `bsp_get()` and `bsp_put()` methods, respectively. For the BSMP model, messages are sent to other processors using `bsp_send()`. The sent messages are stored in a queue on the destination processor. Messages can be retrieved on the subsequent superstep using `bsp_move()`, which copies and removes the first message from the local queue.

InteGrade's implementation of the BSP model uses CORBA (OMG , 2008) for inter-process communication. CORBA has the advantages of an easier and cleaner communication environment, shortening development and maintenance time and facilitating system evolution. Also, since it is based on a binary protocol, the performance of CORBA-based communication is an order of magnitude faster than the performance of technologies based on

XML, requiring less network bandwidth and processing power. On the shared grid machines, InteGrade uses OiL (Maia, Cerqueira and Cosme (2006)), a very light-weight version of a CORBA ORB that imposes a small memory footprint. Nevertheless, CORBA usage is completely transparent to the InteGrade application developer, who only uses the BSP interface (Goldchleger et al. , 2005).

InteGrade's BSPLib associates to each process of a parallel application a *BspProxy*. The *BspProxy* is a CORBA servant responsible for receiving related communications from other processes, such as a virtual shared address read or write, or the receipt of messages signaling the end of the synchronization barrier. The creation of *BspProxies* is entirely handled by the library and is totally transparent to users.

The first created process of a parallel application is called *Process Zero*. Process Zero is responsible for assigning an unique identifier to each application process, broadcasting the CORBA IORs of each process to allow them to communicate directly, and coordinating synchronization barriers. Moreover, *Process Zero* executes its normal computation on behalf of the parallel application.

On InteGrade, the synchronization barriers of the BSP model are used to store checkpoints during execution, since they provide global, consistent points for application recovery. In this way, in the case of failures, it is possible to recover application execution from a previous checkpoint, which can be stored in a distributed way as described in Section 5.2. Application recovery is also available for sequential, bag-of-tasks, and MPI applications.

4.2. Managing MPI Applications

The Message Passing interface (MPI) has become the *de facto* standard for parallel applications programming. Its current version, MPI2, was released in 1997 and defines a comprehensive set of functions for point-to-point communication, management of process groups, group communication, query about the execution state of processes, support for distributed shared memory, and dynamic spawning of processes (MPI Forum , 1997).

Support for MPI applications is achieved through the MPICH-IG (Cardozo and Costa , 2008) module, which is based on MPICH2[3], an open source implementation of the MPI2 standard. MPICH-IG adapts MPICH2 to use InteGrade's LRM instead of the MPI daemon to launch and control MPI tasks running on the grid nodes. It also uses the application repository to retrieve the binaries of MPI applications, which are automatically installed and launched, instead of requiring them to be manually installed *a priori* as with MPICH2. MPI applications can thus be dispatched and managed in the same way as other kinds of applications in InteGrade (such as BSP and sequential applications).

Regarding communication among tasks, MPICH-IG uses the MPICH2 Abstract Device Interface (ADI) construct to abstract the details of actual communication mechanisms, enabling higher layers of the communication infrastructure to be independent of such mechanisms. A CORBA-based device was thus implemented, which enables tasks running on different clusters/networks to communicate seamlessly. However, when the communicat-

[3]MPICH2 Website: http://www.mcs.anl.gov/research/projects/mpich2/index.php

ing tasks reside in the same cluster, communication support switches to a device that uses a more efficient, sockets-based, mechanism, as is typical of MPICH2 applications.

In order to adapt MPICH2 to run on InteGrade, two of its interfaces had to be re-implemented: the *Channel Interface* (CI) and the *Process Management Interface* (PMI). The former is required to monitor the sockets channel to detect and treat failures through a mechanism of coordinated checkpointing and recovery. The latter is necessary to couple the management of MPI applications with InteGrade's Execution Manager (EM), adding functions for process location and synchronization.

Using MPICH-IG, native MPI applications can be transparently deployed on an InteGrade grid, without the need to modify their source code or even to recompile them. This enables all the benefits of inter-cluster execution (in order to scale up to a greater number of machines), as well as the opportunistic use of idle resources, which are not normally feasible with cluster-based MPI platforms. It is also worth noting that, in MPICH-IG, the tasks that comprise an MPI application are run on machines that are dynamically chosen by the InteGrade grid scheduler (GRM). This is in contrast with conventional MPI platforms, where such choice needs to be made *a priori* and hard-coded in a configuration file.

Another feature that is present in MPICH-IG, in contrast with conventional MPI platforms, refers to the recovery of individual tasks of an application. This prevents the whole application from being restarted from scratch, further contributing to raise the chances of successful completion of application execution. Application recovery (after the occurrence of failures) is supported by monitoring the execution state of application tasks, as described in Section 5.1. Faulty tasks are then restarted on a different grid node, as scheduled by the GRM. Task recovery is implemented as described in Section 5.2. While other MPI platforms focus specifically on fault-tolerance and recovery, notably MPICH-V (Bosilca et al , 2002), they usually rely on homogeneous clusters. MPICH-IG instead removes this limitation and enables the dynamic scheduling of non-dedicated machines, as well as dedicated clusters, further contributing to scale up the number of available computing resources.

Interestingly, these features, most notably fault-tolerance and the opportunistic use of idle resources, favor MPICH-IG when compared to other approaches to integrate MPI into grid computing environments, such as MPICH-G2 (Karonis, Toonen and Foster , 2003). In common to MPICH-G2 is the ability to run MPI applications in large scale heterogeneous environments, as well as the ability to switch from one communications protocol to another depending on the relative location of the application tasks.

Regarding performance, Figure 4 shows a comparison of MPICH-IG (without checkpointing) and MPICH2, considering the execution time for a parallel matrix multiplication application. To put the results in perspective, the figure also shows a hypothetical case (called "ideal") when the execution time does not include the overheads for managing application execution.

As can be seen, MPICH-IG imposes a small overhead to application execution that is mainly due to its application execution protocol, which requires the transfer of application binaries to the compute nodes. Therefore, the overhead will be proportionally lower for long-running applications that

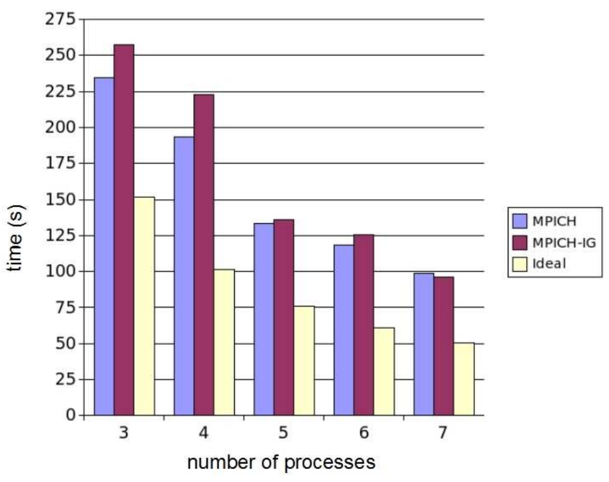

Figure 4: Comparing the execution time for matrix multiplication on MPICH2 and MPICH-IG.

can take hours or even days to execute instead of a few minutes, such as the example illustrated on Figure 4.

4.3. Managing Application Data

Users can register an application in InteGrade using either the Application Submition and Control Tool (ASCT) or the InteGrade Web portal, providing the application description and one or more execution files to enable its execution on multiple platforms. InteGrade stores the application executables in the Application Repository (AR) module, together with metadata describing the application and the platforms for which executables are available.

Managing and storing application input and output files is more difficult than managing executables, since the former can be much larger. To deal with those files, we developed a distributed data repository called Opp-

Store (Camargo and Kon , 2007). Access to this distributed repository is performed through a library called *access broker*, which interacts with OppStore.

OppStore is a middleware that provides reliable distributed data storage using free disk space from shared grid machines. The goal is to use this free disk space in an opportunistic way, i.e., only during the idle periods of the machines. The system is structured as a federation of clusters and is connected by a Pastry peer-to-peer network (Rowstron, Druschel , 2001) in a scalable and fault-tolerant way. This federation structure allows the system to disperse application data throughout the grid. During storage, the system slices the data into several redundant, encoded fragments and stores them in different grid clusters. This distribution improves data availability and fault-tolerance, since fragments are located in geographically dispersed clusters. When performing data retrieval, applications can simultaneously download file fragments stored in the highest bandwidth clusters, enabling efficient data retrieval.

When an InteGrade user submits an application for execution, its input files are first stored in distributed grid machines using the access broker. The execution request is then sent to the GRM, which selects the LRMs that will execute the application and forwards the request to those nodes. When a LRM receives an application execution request, it obtains the application binary for its particular platform from the Application Repository and the input file from OppStore. When the execution finishes, the output files are stored in OppStore.

Using OppStore, application input and output files can be obtained from any node in the system. Consequently, after a failure, restarting of an application execution in another machine can be easily performed. Also, when an application execution finishes, the output files uploaded to the distributed repositories can be accessed by the user from any machine connected to the grid.

To determine the availability of data stored in a large-scale grid composed of non-dedicated machines, we simulated a grid composed of 100 clusters, with the number of machines on each cluster randomly chosen as 10, 20, 50, 100, and 200. We defined three different usage patterns, based on real measurements, and which are randomly assigned to each cluster. In the first pattern, the mean idle time is 60% during the day and 80% during the night and weekends. The second pattern has idle times of 25% and 40%, and the third 40% and 70%, respectively. We distributed the clusters uniformly across 24 time zones.

We simulated the storage of ten thousand files and attempted to later retrieve those files, checking the number of fragments that could be recovered for each file. The file retrieval requests were repeated during a simulated period of one month. Figure 5 shows the percentage of successful retrieval requests for the file retrieval attempts. Using only the idle periods of the shared machines to retrieve data in realistic situations, we could recover enough fragments to reconstruct the files in 99.9% of the requests, for files encoded in 24 fragments, from which 8 are required for reconstruction, a replication factor of 3. When all the fragments necessary to reconstruct a file are not available immediately, the broker can download the available fragments and wait until the remaining fragments become available.

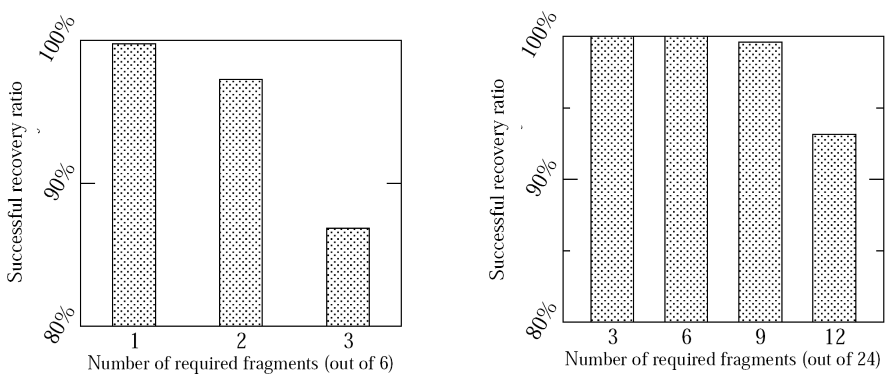

Figure 5: Percentage of successful file retrieval requests.

We also performed experimental evaluations (Camargo and Kon , 2007) that showed that the time required to store and retrieve files in OppStore is mostly dependent on the bandwidth of cluster connections. The broker downloads the file fragments in parallel and, since the number of available fragments is normally larger than the number of required fragments for file reconstruction, the broker can choose the fragments located in the clusters with the fastest connections. Consequently, file retrieval can be performed very efficiently in OppStore.

5. Fault Tolerance in Grid Environments

On opportunistic grids, application execution can fail due to several reasons. System failures can result not only from an error on a single component but also from the usually complex interactions between the several grid components that comprise a range of different services. In addition to that, grid environments are extremely dynamic, with components joining and leaving the system at all times. Also, the likelihood of errors occurring during the execution of an application is exacerbated by the fact that many grid applications will perform long tasks that may require several days of computation.

To provide the necessary fault tolerance functionality for grid environments, several services must be available, such as: (a) **failure detection:** grid nodes and applications must be constantly monitored by a failure detection service; (b) **application failure handling:** various failure handling strategies can be employed in grid environments to ensure the continuity of application execution; and (c) **stable storage:** execution states that allow recovering the pre-failure state of applications must be saved in a data repository that can survive grid node failures. This section describes Integrade failure detection and handling techniques. Integrade stable storage is based on Oppstore, already presented on Section 4.3.

5.1. InteGrade Failure Detection

Failure detection is a very important service for large-scale opportunistic grids. The very high rate of churn makes failures a frequent event and the capability of the grid infrastructure to efficiently deal with them has a direct impact on its ability to make progress. Hence, failed nodes should be detected quickly and the monitoring network should itself be reliable, so as to ensure that a node failure does not go undetected. At the same time, due to the scale and geographic dispersion of grid nodes, failure detectors should be capable of disseminating information about failed nodes as fast and reliably as possible and work correctly even when no process has a globally consistent view of the system. Moreover, the non-dedicated nature of opportunistic grids requires that solutions for failure detection be very lightweight in terms of network bandwidth consumption and usage of memory and CPU cycles of resource provider machines. Besides all of these requirements pertaining to the functioning of failure detectors, they must also be easy to set-up and use; otherwise they might be a source of design and configuration errors. It is well-known that configuration errors are a common cause of Grid failures (Medeiros et al. , 2003).

The aforementioned requirements are hard to meet as a whole and, to the best of our knowledge, no existing work in the literature addresses all of them. This is not surprising, as some goals, e.g., a reliable monitoring network and low network bandwidth consumption, are inherently conflicting. Nevertheless, they are all real issues that appear in large-scale opportunistic grids, and reliable grid applications are expected to deal with them in a realistic setting. The Integrade failure detection service (Castor Filho et al. , 2008) is an attempt to achieve this ambitious goal. It includes a number of features that, when combined and appropriately tuned, address all these challenges while adopting reasonable compromises for the ones that conflict. The most noteworthy features of the proposed failure detector are the following: (i) a gossip- or infection-style approach (Renesse, Minsky and Hayden , 1998; Gupta, Chandra and Goldszmidt , 2001), meaning that the network load imposed by the failure detector scales well with the number of processes in the network and that the monitoring network is highly reliable and descentralized; (ii) separation between failure detection and dissemination of information about failures, considering that the two services are related to different requirements; (iii) a crash-recover failure model, instead of simple crash; (iv) compatibility with firewalls and NATs, due to the use of TCP as the transport protocol; (v) ease of use and configuration; (vi) low resource consumption (memory, CPU cycles, and network bandwidth); and (vii) automatic balancing of the monitoring load.

The failure detector runs on each grid node. A distinguishing feature of InteGrade's failure detector is that it acknowledges that the reliability of the monitoring network and the speed of dissemination of information about node failures are dictated by different requirements. Therefore, it makes sense to set up these two characteristics of the failure detector in terms of two distinct parameters:

- The number of failure detectors monitoring a node is established by the k parameter, an integer constant defined by the grid administrator. Periodically, each process sends heartbeats to every one of the k nodes monitoring it. If a predefined amount of time elapses without a failure detector receiving heartbeats from a process it monitors, it assumes that the monitored process has failed. In this case, it notifies some failure detectors about this. The k parameter is chosen to be a small integer that still guarantees a reasonable amount of monitoring redundancy, usually 4 or 5. It has been shown experimentally (Horita, Taura and Chikayama , 2005) that this number of monitoring relations guarantees that the probability of having a disconnected monitoring network (i.e., one with at least one node that is not monitored by any other node) is extremely low.
- The number of notified failure detectors is dictated by the j parameter (with $j \leq k$), also set by the grid administrator. When a process P receives from a process Q a notification that process R has failed, it sends a notification to j failure detectors randomly chosen amongst

the k ones that monitor it, excluding Q and R, if necessary. If P already knew that R had failed, it simply ignores the notification. We use a reactive approach to send notifications about failures and do not attempt to reduce the number of notifications by, for example, piggybacking them in heartbeat messages (Das, Gupta and Motivala , 2002). Piggybacking is very cheap in terms of number of messages sent, but it requires a number of protocol rounds before the notification reaches all the nodes in the grid. A reactive approach, however, might result in a large number of messages being sent in a small amount of time. To avoid this scenario, we choose j to be as small as possible, so that dissemination speed still grows exponentially. Hence, we usually set j to 2, the lowest integer number greater than 1. Since a reasonable k makes it commonplace for more than one node to detect a failure, even with a low value of j it is still highly probable that every failure detector in the network eventually gets to know about it.

The InteGrade failure detector attempts to avoid overloading parts of the network by periodically balancing the monitoring load. If a monitored node perceives that it is being monitored by more than k nodes, it randomly choses one of them and asks this node to stop monitoring it. Furthermore, the failure detection protocol attempts to avoid the situation where a process ends up monitoring much more than a pre-established limit of l processes (usually quantos). The failure detector of every process X periodically checks whether it monitors more than l nodes. If so, it randomly choses a process S amongst the ones it monitors and asks S to find another process to monitor it. S is then responsible for selecting the process L it knows about that has the lowest monitoring load and asking L to monitor it. If this procedure is successful and S ends up being monitored by L, it tells X to stop monitoring it, thus reducing X's monitoring load. This simple protocol requires a small amount of information about known monitoring loads to be periodically piggybacked in heartbeat messages. The protocol has the desirable property of never allowing the number of processes that monitor a given process to go below k. Furthermore, it requires a very small number of extra message exchanges and the definition of only one new message type. Finally, the periodic approach adopted in the load balancing protocol, albeit slower to converge, avoids the heavy load imposed by a reactive approach.

Our failure detector is also very easy to use and portable. It is implemented in Lua (Ierusalimschy , 1996), an extensible and lightweight programming language. Lua makes it easy to use the proposed failure detector from programs written in other programming languages, such as Java, C, and C++. Moreover, it executes in several platforms. Currently, we have successfully run the failure detector in Windows XP, MacOS X, FreeBSD, and several flavors of Linux (Debian, Ubuntu, VectorLinux, and Slackware). The entire implementation of the failure detector comprises only 55kb of Lua source code, including comments.

5.2. InteGrade Application Recovery

In order to overcome application execution failures, Integrade provides support for the most used failure handling strategies: (1) **retrying**: when an application execution fails, it is restarted from scratch; (2) **replication**: the same application is submitted for execution multiple times, generating various application replicas; all replicas are active and execute the same code with the same input parameters at different nodes; and (3) **checkpointing**: periodically saves the process' state in stable storage during the failure-free execution time. Upon a failure, the process restarts from the latest available saved checkpoint, thereby reducing the amount of lost computation. As part of the application submission process, users can select the desired technique to be applied in case of failure. These techniques can also be combined resulting in four more elaborate failure handling techniques: *retrying* (without checkpoint and replication), *checkpointing* (without replication), *replication* (without checkpointing), and *replication with checkpointing*.

The InteGrade **checkpointing** mechanism adds to the basic application execution protocol steps responsible for gathering application state into a checkpoint and its storage in machines executing the Autonomous Data Repository (ADRs) module of OppStore. ADRs are located at resource provider machines in the grid and are responsible for managing grid data stored on these machines.

To gather application state, InteGrade includes a portable application-level checkpointing mechanism for sequential, bag-of-tasks, MPI and BSP parallel applications (Camargo, Kon and Goldman , 2005). This portability allows an application's stored state to be recovered on a machine with a different architecture from the one where the checkpoint was generated. In application-level checkpointing, the application is responsible for providing the data that will be put in the checkpoint. To automate this process, we implemented a precompiler that inserts, into application code, the statements responsible for gathering and restoring the application state from the checkpoint.

When the checkpointing library needs to store a checkpoint, it queries the Cluster Data Repository Manager (CDRM) from OppStore, which manages all the ADRs from the cluster, about available ADRs. Checkpoint data recovery also involves a query to the CDRM, requesting the list of ADRs where the application checkpoints were stored. Checkpoint storage and retrieval is driven by a chosen storage strategy. The available strategies are:

- *Data replication*: stores full copies of checkpoint data on different machines. This approach uses more storage space and network bandwidth, but less processing power, since no coding is performed;

- *Rabin's classic Information Dispersal Algorithm* (IDA) (Rabin , 1989): encodes the checkpoint into redundant fragments, such that regenerating the original checkpoint is possible using only a subset of them. This approach uses less network bandwidth and storage space, but more processing power.

These two strategies allow the application library to select a trade-off between the use of storage space and network bandwidth and the CPU time necessary to encode the checkpoint. Checkpoint encoding and transfer are performed by a separate application thread, allowing the application to concurrently continue its execution. Finally, to improve performance, the library transfers the several checkpoint fragments or replicas to the repositories in parallel.

We evaluated the overhead of the checkpointing mechanism using the replication and IDA strategies using a matrix multiplication Hayashida et al. (2005) application. Figure 6 shows the results for the executions of the application using two matrix sizes (3200x3200 and 4800x4800), configured to run without checkpointing and with checkpointing using replication and two IDA configurations. The first, IDA(7,1), generates 8 fragments from which 7 are required to reconstruct the original file, and the second, IDA(6,2), generates 8 fragments and require 6 for file reconstruction. We executed each experiment 16 times, with the vertical bars showing the mean measured overhead and the error bars representing the standard deviation. For a matrix of size 4800x4800, which generates global checkpoints of 791MB, and a minimum interval between checkpoints of 60s, the overhead is 13% when using replication, 18% when using IDA(7,1) and 20% for IDA(6,2). This overhead can be reduced by increasing the checkpointing interval. For example, with a checkpointing interval of 5 minutes, the overhead would be of approximately 2.6% using replication.

These results show that the use of checkpointing has low overhead even when running coupled parallel applications that generate large checkpoints. Using the IDA strategy causes a higher overhead, as it is necessary to encode the checkpoints using processor cycles from the machines running the parallel application. Using replication causes a smaller overhead, but uses more network and storage resources. Since parallel application normally run on a single cluster connected by a local LAN and storage space is not a scarce resource, the default strategy used is replication of the checkpoint data.

Replication of the execution of application tasks is another failure handling technique commonly applied on grid environments. While checkpointing imposes an overhead to application execution (for gathering and storing the execution state), replication requires a larger amount of grid resources

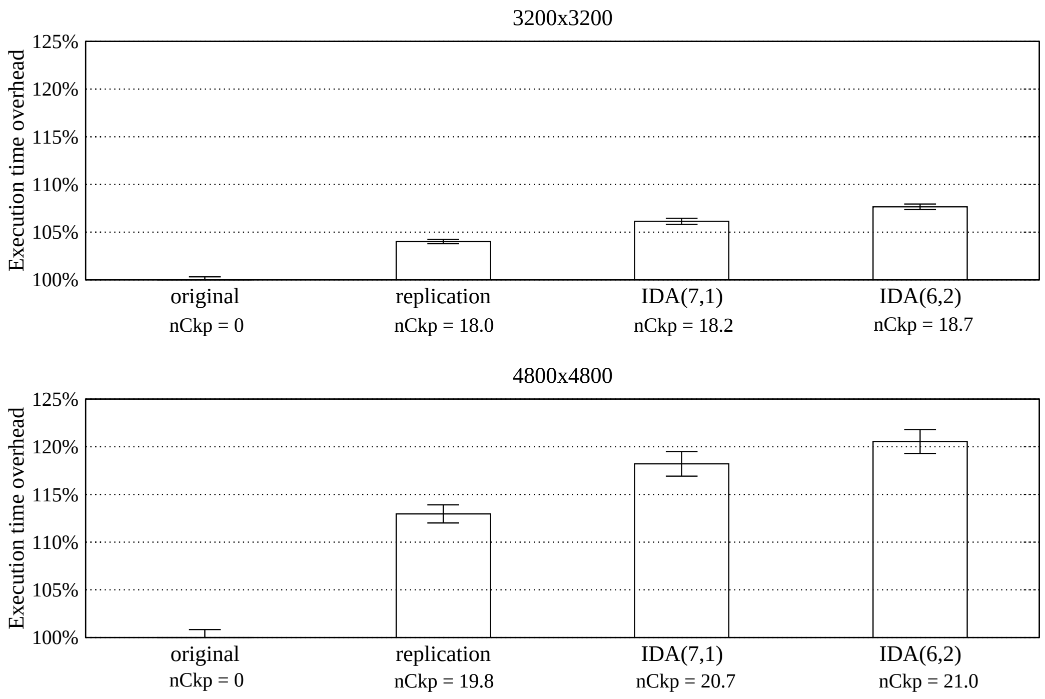

Figure 6: Checkpointing execution overhead using different checkpointing strategies.

for executing the same task.

InteGrade allows replication for sequential, bag-of-tasks, MPI and BSP applications. The amount of generated replicas is currently defined during the application execution request, issued through the Application Submission and Control Tool (ASCT). The request is forwarded to the Global Resource Manager (GRM), which runs a scheduling algorithm that guarantees that all replicas will be assigned to different nodes. Another InteGrade component, called Application Replication Manager (ARM), concentrates most of the code responsible for managing replication. The GRM instantiates a new ARM for every application execution request that demands replication. The ARM gathers the application input files from the ASCT and forwards a copy of the execution request to the LRM of every node involved with the application execution. Each LRM instantiates an application process. The ARM also registers each application process with the Execution Manager (EM) and, in case of a replica failure, starts its recovery process. When the first application replica concludes its job, the ARM kills the remaining ones, releasing the allocated grid resources. In this way, the complexity of replication is confined to the ARM code, minimizing the required changes on InteGrade components to support replication.

6. Summary, Conclusions, and Future Directions

Opportunistic grid middleware enables the use of the existing computing infrastructure available in laboratories and offices in universities, research institutes, and companies to execute computationally intensive parallel applications. Nevertheless, executing this class of applications on such a dynamic and heterogeneous environment is a daunting task, especially when non-dedicated resources are used, as in the case of opportunistic computing.

This article presented recent advances on the InteGrade opportunistic grid middleware concerning the support for application execution, covering the related fields of application scheduling, execution management, and fault tolerance.

The InteGrade module called Local Usage Pattern Analyzer (LUPA) predicts each machine's idle periods by locally performing Usage Pattern Analysis of machine resources using an unsupervised machine learning approach. Experimental results demonstrate that the proposed strategy is very successful in predicting idleness, with success rates above 90% at all times, leading to better scheduling decisions and minimizing task migrations.

InteGrade is very flexible with respect to the supported application classes. It currently allows the execution of sequential, parametric (bag-of-tasks), and parallel applications following either the BSP or MPI models. Native MPI applications can be transparently deployed on an InteGrade grid, without the need to modify their source code or even to recompile them. InteGrade's MPI support emphasizes scalability as well as the opportunistic use of idle resources, which are not normally feasible with cluster-based MPI platforms. It also allows the recovery of individual tasks of an application, which prevents the whole application from being restarted from scratch. InteGrade also allows the execution of BSP applications through the implementation of several functions from the Oxford BSPlib, including both DRMA (direct remote memory access) and BSMP (bulk synchronous message passing).

Concerning the management of application data, which includes the application binaries, input and output data, InteGrade's OppStore component provides reliable distributed data storage using the free disk space from shared grid machines in an opportunistic way, i.e., only during the idle periods of the machines. Experimental results demonstrate that OppStore can recover enough fragments to reconstruct files in 99.9% of the tested requests and that file retrieval can be performed very efficiently.

Since opportunistic grid environments are highly prone to failures, special care was taken on InteGrade to circumvent application execution disruptions. The failure detection infrastructure separates failure detection from the dissemination of information about failures, allowing the grid administrator to adjust two distinct parameters: the number of failure detectors monitoring a node and the number of notified failure detectors. A gossip- or infection-style approach is also used, leading to a scalable, reliable, and decentralized monitoring approach. InteGrade provides support for replication, checkpointing, and retry recovery techniques that can also be combined together in a flexible way. Checkpointing and replication are available for sequential, bag-of-tasks, MPI and BSP parallel applications and experimental results demonstrate that checkpoint usage has a low overhead even when running coupled applications that generate large checkpoints.

All those presented features, when combined, provide a robust and flexible environment that allows the efficient execution of large scale, computationally-intensive parallel applications, even on highly dynamic environments, such as opportunistic grids. As future work we intend to deploy the InteGrade middleware in a large opportunistic grid and evaluate its performance in a real scenario with hundreds of machines and users. The InteGrade source code and documentation is available as free software distributed under the LGPL license at `http://www.integrade.org.br`.

References

Marcelo Finger, Germano C. Bezerra, and Danilo M. R. Conde. Resource Use Pattern Analysis for Opportunistic Grids. In *6th International Workshop on Middleware for Grid Computing (MGC 2008)*, Leuven, Belgium, December, 2008.

Renato Maia, Renato Cerqueira, and Ricardo Cosme. OiL: An Object Request Broker in The Lua. In *Proc. 5th Tools Session of the Brazilian*

Simposium on Computer Networks (SBRC2006), Curitiba, Brazil, June 2006.

H. B. Barlow. Unsupervised learning. In G. Hinton and T. J. Sejnowski, editors, *Unsupervised Learning: Foundations of Neural Computation*, pages 1–17. MIT Press, 1999.

George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vincent Neri, and Anton Selikhov. MPICH-V: toward a scalable fault tolerant MPI for volatile nodes In Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, pages 1–18, Baltimore, Maryland, 2002. IEEE Computer Society Press.

Raphael Y. de Camargo, Fabio Kon, and Alfredo Goldman. Portable checkpointing and communication for BSP applications on dynamic heterogeneous Grid environments. In *SBAC-PAD'05: The 17th International Symposium on Computer Architecture and High Performance Computing*, Rio de Janeiro, Brazil, October 2005.

Raphael Y. de Camargo and Fabio Kon. Design and implementation of a middleware for data storage in opportunistic grids. In *CCGrid '07: Proceedings of the 7th IEEE/ACM International Symposium on Cluster Computing and the Grid*, Washington, DC, USA, 2007. IEEE Computer Society.

Marcelo C. Cardozo, Fábio M. Costa. MPI Support on Opportunistic Grids based on the InteGrade Middleware In *Proceedings of the 2nd Latin American Grid International Workshop (LAGrid)*, Campo Grande, Brazil, 2008.

Fernando Castor Filho, Augusta Marques, Raphael Y. de Camargo and Fabio Kon. A group membership service for large-scale grids. In *Proceedings of the 6th International Workshop on Middleware for Grid Computing*, Leuven, Belgium, 2008.

Abhinandan Das, Indranil Gupta, and Ashish Motivala. Swim: Scalable weakly-consistent infection-style process group membership protocol. In *International Conference on Dependable Systems and Networks (DSN 2002)*, pages 303–312, Bethesda, USA, June 2002.

Brian Everitt, Sabine Landau, and Morven Leese. *Cluster Analysis.* A Hodder Arnold Publication, 4th edition edition, 2001.

Ian Foster and Carl Kesselman. *Grid 2: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, 2003.

Andrei Goldchleger, Fabio Kon, Alfredo Goldman, Marcelo Finger, and Germano Capistrano Bezerra. InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. *Concurrency and Computation: Practice and Experience*, 16(5):449–459, March 2004.

Andrei Goldchleger, Alfredo Goldman, Ulisses Hayashida, Fabio Kon. The implementation of the BSP parallel computing model on the InteGrade Grid middleware In *Proceedings of the 3rd international workshop on Middleware for grid computing*, Grenoble, France, 2005. ACM Press.

Indranil Gupta, Tushar Deepak Chandra, and Germán S. Goldszmidt. On scalable and efficient distributed failure detectors. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing*, pages 170–179, August 2001.

Ulisses Kendi Hayashida, Kunio Okuda, Jairo Panetta and Siand Wun Song. Generating Parallel Algorithms for Cluster and Grid Computing. In *Proceedings of the the 2005 International Conference on Computational Science and its Applications*, pages 509–516, May 2005.

Yuuki Horita, Kenjiro Taura, and Takashi Chikayama. A scalable and efficient self-organizing failure detector for grid applications. In *Proceedings of the 6th ACM/IEEE International Workshop on Grid Computing (GRID 2005)*, Seattle, USA, November 2005.

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua - an extensible extension language. *Software: Practice Experience*, 26(6):635–652, 1996.

N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(3):551–563, May 2003.

Raissa Medeiros, Walfredo Cirne, Francisco Vilar Brasileiro, and Jacques Philippe Sauvé. Faults in grids: Why are they so bad and what can be done about it? In *Proceedings of the 4th IEEE International Workshop on Grid Computing (GRID 2003)*, pages 18–24, Phoenix, USA, November 2003.

MPI Forum. MPI-2: Extensions to the Message-Passing Interface. MPI Forum. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html (accessed on 02/17/2009), July 1997.

Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems In *Proceedings of the Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms*, pages 329–350, isbn 3-540-42800-3, Heidelberg, Germany, 2001

M. O. Rabin. Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance. *Journal of ACM*, 36(2):335–348, 1989.

Robbert van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In *Proceedings of Middleware'1998*, Lake District, England, September 1998.

R. R. Sokal. Clustering and classification: Background and current directions. In *In Proceedings of the Advanced Seminar on Classification and Clustering*, 1996.

Sergio Theodoridis and Konstantinos Koutroumba. *Pattern Recognition.* Academic Press, 2003.

Object Management Group (OMG). Common Object Request Broker Architecture (CORBA) Specification, Version 3.1. Available at `http://www.omg.org/spec/CORBA/3.1/`. 2008.