Exploiting Tuple Spaces to Provide Fault-Tolerant Scheduling on Computational Grids

Fábio Favarim † Joni da Silva Fraga† Lau Cheuk Lung§
Miguel Correia‡ João Felipe Santos†
† Departamento de Automação e Sistemas, Universidade Federal de Santa Catarina – Brazil
‡ LASIGE, Faculdade de Ciências da Universidade de Lisboa – Portugal
§ Prog. de Pós-Graduação em Informática Aplicada, Pontifícia Universidade Católica do Paraná – Brazil
{fabio, fraga, jfs}@das.ufsc.br *lau@ppgia.pucpr.br* *mpc@di.fc.ul.pt*

Abstract

Scheduling tasks on large-scale computational grids is difficult due to the heterogeneous computational capabilities of the resources, node unavailability and unreliable network connectivity. This work proposes GRIDTS*, a grid infrastructure in which the resources select the tasks they execute, instead of a scheduler finding resources for the tasks. This solution allows scheduling decisions to be made with up-to-date information about the resources. Moreover,* GRIDTS *provides fault-tolerant scheduling by combining a set of fault tolerance techniques to tolerate crash faults in components of the system. The core of the solution is a tuple space, which supports the communication, but also provides support for the fault tolerance mechanisms.*

1. Introduction

A computational **grid infrastructure** is an aggregate of geographically disperse heterogeneous resources on the Internet. Those **resources** can be desktop computers, laptops, supercomputers, clusters of workstations, and storage and database systems [8]. These infrastructures are used by authorized users to run complex **applications**, like weather simulations or filtering of large volumes of data. Computational resources like desktops or laptops are made available by their owners – normally during idle time – to compute those applications. Grids are dynamic in the sense that resources can join and leave them at any time. This dynamic nature allows the grids to contain many machines that would not be available otherwise, but on the other hand, it also means that the infrastructure has to manage the uncertainty about the availability of the resources and their failure.

This paper is about **scheduling** and **fault tolerance** in grids that execute **bag-of-tasks** applications [17]. This is only one among several types of applications that can be executed in grids, but there are many important applications that fit in this category: data mining, massive searches, image processing, etc. These applications are composed by sets of decoupled tasks that can be executed independently, without any kind of synchronization or communication among them. This independence makes these applications specially suited for dynamic grids, since the failure of resources that are executing tasks can easily be handled by rescheduling the interrupted tasks in other resources.

An important objective of any grid infrastructure is to use the resources efficiently, i.e., maximizing the resource utilization while trying to minimize the job's total execution time, also called **makespan** [9]. The performance of the grid depends strongly on the efficiency of the scheduling. A schedule is an assignment of the tasks of a job to a set of resources. Each job consists of a set of **tasks**, and each task has to be executed by one of the grid resources for a certain time. Many schedulers rely on accurate information about resources' attributes (CPU speed and load, memory) and tasks to do the scheduling.

Information about resources is characterized by a set of attributes, like the operating system model/version, the speed and load of the processors, and free memory. This information used by schedulers is usually provided by an **information service** that is responsible for gathering data about all resources that compose the grid.

Gathering information about resources is like taking a snapshot of the grid, i.e., getting the global grid state in a certain instant. This operation is reasonably costly in a large grid, and the snapshot tends to become outdated in a short time when the grid is comprised by a large number of non-dedicated, heterogeneous, widely-dispersed resources. Moreover, getting an accurate snapshot in an asynchronous distributed system (as the Internet) is impossible [5]. The key problem is that information obtained from the information service may be outdated by the time the scheduler needs it to schedule tasks.

This paper proposes GRIDTS, an infrastructure that provides a scheduling solution in which the resources select the tasks they execute, instead of the scheduler finding resources for the tasks. This solution allows scheduling decisions to be made with up-to-date information, since, naturally, each resource has always up-to-date information about itself. Therefore our solution overcomes the problems of getting up-to-date information about resources faced by schedulers rely on this information.

GRIDTS is based on the **generative coordination model**, in which processes (brokers, resources) interact through a shared memory object called **tuple space** [10]. This coordination model supports communication that is decoupled both in time and space, i.e., in which processes do not need to be active at the same time and do not need to know each others locations or addresses [4]. This makes it particularly suited for highly dynamic systems like a grid.

In large-scale grids, the probability of failures happening is high. Many of the current grids have single points of failure, i.e., not all their components are fault-tolerant. GRIDTS is fault-tolerant, in the sense that all components in the system can fail by crashing and the system still behaves as expected. Fault tolerance is enforced using a combination of mechanisms. Transactions are used to guarantee that the failure of a resource or a broker does not cause the loss of a task or leaves the tuple space in an inconsistent state. Checkpointing is used to limit the work lost when a resource fails during the execution of a task, allowing another resource to continue where the first left. Finally, replication is used to enforce the fault-tolerance and availability of the tuple space. We consider only **crash faults**, which in this context can be accidental (some machine really crashes) or forced by a resource owner that wants to remove his/her machine(s) from the grid. We do not consider the possibility of the resources returning results that do not correspond to the execution of the tasks they are supposed to execute, on the contrary to [19].

This work has two main contributions. Firstly, it presents an architecture for a computational grid that allows resources to find tasks suited for their attributes, even if those attributes change with time. This eliminates the complexity of gathering information about the whole grid. Secondly, the infrastructure provides fault-tolerant scheduling by combining a set of fault tolerance techniques to tolerate crash faults in any component of the system.

The remainder of the paper is organized as follows. Section 2 defines the tuple space used to support the scheduling of tasks. Section 3 gives an overview of GRIDTS. Section 4 presents an evaluation of the performance of GRIDTS and compares it with other schedulers. Finally, Section 5 gives some concluding remarks.

2 Tuple Spaces

GridTS is based on a tuple space, a notion first introduced in the Linda programming language [10]. A tuple space can be viewed as a shared memory object that allows distributed processes to interact by inserting tuples in the space. In this space, generic data structures called *tuples* can be inserted, read and removed.

A tuple is an ordered sequence of typed fields. Given a tuple $t = \langle f_1, f_2, ..., f_n \rangle$, each field f_i can be: **actual**, i.e., be a value; **formal**, i.e., a variable name preceded by a question mark ("?"); or a **wildcard**, like "*" meaning any value. A tuple in which all fields are actual is called an **entry** and is denoted by a lowercase letter, e.g., t. A tuple with at least one formal field is called a **template** and is denoted by $\bar{t}$. A tuple space can only store entries, never templates. Templates are used to read tuples in the space.

An important characteristic of this coordination model is the associative nature of the communication: tuples are not accessed through an address or identifier, but rather by their content. An entry t and a template $\bar{t}$ are said to **match** if : (i) both have the same number of fields; (ii) corresponding fields have the same type, and; (iii) corresponding actual fields have the same value. A tuple space provides three basic operations [11, 10]:

- $out(t)$: puts the tuple t in the tuple space (write);
- $in(\bar{t})$: reads and removes a tuple t that matches $\bar{t}$ from the tuple space. If no matching tuple t is available, the process stays blocked until a tuple matching the template $\bar{t}$ is available in the space (destructive read);
- $rd(\bar{t})$: has a behavior similar to in, but leaves the matched tuple t in the tuple space (non-destructive read).

Both the in and rd operations are blocking, but they have also non-blocking versions: $inp()$ and $rdp()$. These operations work in the same way as their blocking versions but return a "tuple not found" value if no matching tuple is available in the tuple space. All these read operations are non-deterministic, because if there is more than one matching tuple available, one of them is chosen arbitrarily. There is one more version of $rd(\bar{t})$ that returns *all* the tuples that match $\bar{t}$: $copy\_collect(\bar{t})$ [15].

2.1 Fault Tolerance

Fault tolerance in the generative coordination model can be considered at two levels:

- *fault tolerance of the tuple space*, i.e., the problem of guaranteeing that the space does not fail if there are faults in the tuple space itself; and
- *application-level fault tolerance*, i.e., to ensure that the applications satisfy certain dependability properties even if some of the applications' processes fail.

In GridTS, the first issue is handled using replication, i.e., by running the tuple space in several servers and ensuring that the tuple space as a whole tolerates the failure of some of the servers [21, 2]. In this paper we consider that the tuple space is indeed implemented by a set of servers and is fault-tolerant but we do not delve into the details of how it is done since the problem is well understood and is solved [21, 2].

The second issue is handled by application-level fault tolerance mechanisms provided by the tuple space, usually **transactions** [12, 2]. This mechanism guarantees essentially that if a process tries to execute a set of operations in the tuple space, either all the operations are executed or none of them is. If the process executes all operations in the set, then the transaction is said to **commit**. If the process fails (crashes) during the execution, then the transaction is **aborted**; if some of the operations have already been executed then the tuple space removes all their effects to guarantee the atomicity. In practice, the detection of a failure works the following way [18]. When a process starts a transaction, it defines a *lease* to do that transaction, i.e., a time interval during which it will execute the transaction's operations. If the process does not commit during that time, the tuple space assumes that the process failed and aborts the transaction. If the process needs more time to execute the transaction, it periodically renews the lease. This lease mechanism involves time assumptions about maximum processing and communication times.

Due to the simplicity of tuple spaces and their language independence, many current programming language provide this communication and interaction model. Among them, the JavaSpaces [18] – part of Java's Jini framework – and the TSpaces [14] are some of the most known. Both support transactions.

3 GridTS

3.1 The Infrastructure

In GRIDTS there is not one but several masters – called **brokers** – that get **jobs** from their **users**, divide the jobs in **tasks** and make available these tasks to the **resources**, which are composing the grid.

An overview of the GridTS infrastructure is shown in Figure 1. The basic functioning is based on the **master-workers** pattern [21, 2]. This pattern has two kinds of entities: one master and several workers. The master gives tasks to the workers that execute them and return the results to the master. In GRIDTS there is not one but several masters – called **brokers** – that get **jobs** from their **users**, divide the jobs in **tasks** and make available these tasks to the **resources**, which are composing the grid. Brokers are usually specific to one class of applications, i.e., they only know how to decompose jobs of this class. For example, if the application deals with processing satellite images, the broker decomposes the image (job) in several parts (tasks), that can be processed by different resources. When a resource/worker finishes executing a task, it gives the result to the broker that assembles all results and return them to the user. All communication between brokers/masters and resources/workers is done exclusively through the tuple space.

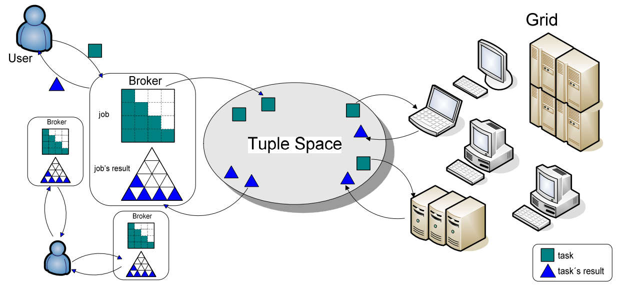


Figure 1. The GridTS infrastructure

In our proposal – GRIDTS – we use a tuple space for supporting task scheduling. Briefly, the idea is the following:

- The user submits a job to a broker that decomposes the job in several small tasks. The broker insert tuples describing these tasks in the tuple space (**task tuples**).
- The resources retrieve from the space tuples that describe tasks they are able to execute, and execute them. After each execution, the result is placed in the tuple space.
- When all job's tasks finish executing, the broker assembles all task results and gives them to the user that submitted the job.

Each task represents one unit of work that may be performed in parallel with other tasks. The tuple that describes a task contains all the necessary information for its execution: the identification of the task, the requirements for its execution (e.g. processor load, processor speed, available memory, operating system), the code to be executed (or the location from where to download it), and the parameters – input data– to the execution of the task (or their location). The users do not need to know which resources will execute the tasks, their location or when these resources will be available.

The architecture of GRIDTS has the immediate benefit of not requiring an information service to give information about the resources (their availability, usage and other parameters), on the contrary of most grid platforms. GridTS enforces a natural form of load balancing since the resources pick tasks adequate to their conditions and get a new one whenever the previous ended. However, there are also some challenges. The first is a problem of fairness since multiple users/brokers can put tasks concurrently in the tuple space. The second is related to fault tolerance: GRIDTS has to tolerate failures in the brokers and, more importantly, to deal efficiently with resources' failures.

3.2 Fairness

To guarantee a **fair scheduling**, the resources have to pick tasks tuples from the tuple space using an appropriate criteria. A solution that ensures fairness is to use a *ticket*. When a broker wants to insert a tuple in the space, it picks a tuple that represents the ticket (**ticket tuple**), inserts the job with the current ticket number, increases the ticket number and inserts it back in the space. A job is represented in the tuple space by a **job tuple**[1] and a set of task tuples. When a resource wants to pick a task, the criteria should be to select the job with lowest ticket to ensure fairness. After the job selection, the resource selects the task to be executed according to some heuristic.

It is important to clarity that the performance of scheduling depends strongly on the efficiency of the heuristic chosen. Thus, any scheduling heuristic that uses any or only local information about resources can be used in GRIDTS. For example, the Workqueue could be used (Section 4.1), i.e., simply getting any task of the chosen job. But when the job's tasks are heterogeneous, slow resources might get large tasks, taking much more time to execute them than a faster resource. This could delay the overall job's execution.

To minimize this mismatch between task size and resource speed, we propose a simple heuristic by classifying both jobs and resources in **classes**. Resources are classified in classes $\mathcal{CR} = \{r_1, \ldots, r_{nc}\}$ according to their speed. For instance, if they are classified in three classes ($nc = 3$), the first class can have resources until 1GHz, the second one resources from 1 to 3GHz, and the third one over 3GHz. Tasks are classified in classes $\mathcal{CT} = \{t_1, \ldots, t_{nc}\}$ according to their size. It is the broker that is responsible for classifying the tasks in classes, putting this information in the task tuple (in the $information$ field). The number of task and resource classes rc is the same and there is a correspondence between classes: class r_1 should include the slower resources and class t_1 the smaller tasks; class r_{nc} should include the faster resources and class t_{nc} the larger tasks.

Resources start getting tasks from the corresponding class of tasks, i.e., if a resource belongs to class r_i it gets a task from class t_i. When there are no more tasks of class t_i, it tries to get a task of class t_{i+1}; if there no tasks from that class, it tries to get from t_{i+2}, etc.; if there are no more from class t_{nc} them it starts trying to get tasks from class t_{i-1}, etc. If there are no more tasks, it means that all job's task are being (or have been already) executed. By enforcing faster resources to execute larger tasks first, and slow resources to execute smaller tasks first, the probability of large tasks being scheduled to slow resources is reduced, and the job execution tends to terminate faster.

[1]Represents all common information of tasks from the same job, as the number of tasks that compose the job, the ticket value associated with the job, information about the attributes for job execution (e.g., the required processor speed, memory, operating system) and the code to be executed (or a reference to its location, e.g, an URL).

3.3 Fault-Tolerance

In a grid environment, with hundreds, thousands, or even tens of thousands of resources, joins, exists and failures of resources are frequent. Thus, we employ fault-tolerant mechanisms to allow the system behave as expected.

Transactions are used by both brokers and resources. Broker uses transactions to ensure that: (1) the job's tasks are insert atomically in the space, i.e., either they are all inserted or none is (in case the broker fails during the insertion); (2) the ticket is not lost if the broker removes it and crashes before inserting it back incremented in the space; (3) to get the results of the tasks atomically from the space. These transactions allow also the broker not to be locked waiting until all the tasks are executed, i.e., the broker can leave the system after having placed the tasks into the space and later run again to get the results.

On the resources-side, transactions are used mainly to guarantee that when a resource fails during the execution of a task, the task tuple is returned to the space to be eventually executed by another resource (or the same if it recovers).

Tasks usually take a long time to execute, e.g., hours or even days, so it is not convenient to restart from scratch the execution of a task whenever the resource that is executing it fails. To minimize this problem, GridTS uses a backward error recovery mechanism that consists in periodically saving the state of the task execution – a **checkpoint** – in the tuple space [13]. If the resource fails, then another resource continues the execution of the task from that checkpoint, thus limiting the work lost when a resource fails during the execution of a task.

The algorithmns and correctness proofs of GRIDTS is described in [7].

4 Evaluation

This section presents a comparison of different grid scheduling algorithms based on simulations. The performance metric used in all simulations was the total time to execute all tasks of a job, also called makespan. The algorithms were evaluated under several different environmental conditions, both without and with faults.

4.1 Scheduling Algorithms

This section introduces some of scheduling algorithms in the literature. The simulations compare GRIDTS with all of them. Some algorithms – knowledge-based – use a centralized service to provide information about the available resources and their characteristics. This solution does not scale well and requires the service to keep track of the resources to maintain the information up-to-date. Proposals that do scheduling without taking into account information about the resources – knowledge-free – have also been proposed.

Workqueue is a knowledge-free scheduler for clusters, not specifically for grids, which does not use any information about resources for task scheduling [6]. The first task waiting to be scheduled is picked and a free resource is assigned arbitrarily to execute it. This procedure is repeated until all tasks are scheduled. After a task is completed, the scheduler assigns a new task to the resource. A problem is that when a large task is assigned to a slow resource, the execution of the complete application may be delayed until the termination of this task.

The basic Workqueue with Replication (WQR) algorithm does the same as Workqueue [16]. However, when there are no more tasks to be executed and there are still idle resources, the tasks that are still running are replicated in these idle resources, i.e., they are also executed in these resources. When a task replica terminates, all its replicas are stopped. The idea is that when a task is replicated there is a chance that a replica is assigned to a faster node, thus augmenting the probability of a faster completion of the task. Simulations have shown that WQR has a good performance if there are free resources, but this is not the case in grids when there are several users constantly requesting the execution of jobs.

Compared with Workqueue and WQR, GRIDTS has the advantage of not wasting resources with replication and letting the resources choose tasks suited for their characteristics, so the probability of a large task being scheduled to a slow resource is minimized.

MFTF (Most Fit Task First) [20] gives more priority to the task that 'fits' better to an available resource. The fitness value is defined as follows: $fitness(i, j) = \frac{100000}{1+|W_i/S_j - E_i|}$. W_i is the workload of the i^{th} task. S_j is the CPU speed of the j^{th} resource according to the information service. E_i is the expected execution time of the i^{th} task. W_i/S_j is the estimated execution time of task i using the resource j. $W_i/S_j - E_i$ is the difference of the estimated execution time and expected task execution time. A small difference indicates greater suitability between task and node. When this difference is zero, it means that the resource is the most suitable to that task. Determining a suitable E_i is quite important in this scheduling method.

MFTF uses dynamic information about the resources in order to be more efficient than a knowledge-free scheduler as WQR. As we stated in the beginning of the paper, getting information about resources is difficult and it can become outdated in a short time due to the dynamism of grid. GRIDTS overcomes this limitation because the resources themselves get the tasks to be executed and then, naturally, know their availability at any time and can get tasks that are more suited to them.

4.2 Simulation Environments

We simulated 2490 scenarios and repeated each of them 10 times to compare GRIDTS with three scheduling algorithms: Workqueue, WQR and MFTF. All algorithms were simulated considering the same set of resources and tasks. GRIDTS was simulated using one, three and five classes (denoted respectively GRIDTS1, GRIDTS3 and GRIDTS5) and WQR using only two replicas (denoted WQR2x). MFTF used perfect information about resources and tasks, something that is difficult to be obtained in the real world.

We used the GridSim toolkit to run our simulations [3]. GridSim supports the modeling and simulation of heterogeneous grid resources, users and application models. It provides the main building blocks for the simulation of applications in grid environments. Using these building blocks to simulate Workqueue, WQR and MFTF was straightforward. However, we had to develop the new scheduling model for GRIDTS.

All simulations used the same value for the grid speed, i.e., for the sum of the resources speeds: 1000. The **resource speed** represents how fast it can execute a task. For instance, a resource with speed 5 can execute a task with size 100 in 20 time units. We also used a fixed value for the job size: 6000000 time units. In a ideal world, the makespan of this job would be 6000 time units, i.e., 100 hours, if the unit was the minute. By fixing the grid speed and the job size, the variation of makespan is due only to the differences of the scheduling algorithms. Related to communication we make assumptions that the transfer times are negligible because we assume the jobs have small input/output data.

In grid computing, the makespan of the jobs depends on several parameters, like the number of resources and tasks, the **task granularity** (task size), the **tasks heterogeneity** (the variation of the tasks' sizes) and the **resources heterogeneity** (the variation of the resources' speeds). We also considered the **fault load**, the number of resources failures, since GRIDTS was designed to be fault-tolerant. The combination of these parameters defines specific execution environments.

The grid's **resources' speed** have a distribution $U(10 - hm/2, 10 + hm/2)$, where $U(a, b)$ represents an uniform distribution from a to b and the values used for hm were 0, 2, 4, 8 and 16. This means that the average speed of all resources is 10. When $hm = 0$, all resources have speed 10, so the grid is homogeneous. The maximum heterogeneity of the resources happens when $hm = 16$ and the speed of the resources varies with distribution $U(2, 18)$.

Concerning the **tasks' granularity**, the experiments considered four groups of task sizes. The mean sizes of the tasks of those groups were 1000, 2500, 10000 and 25000 time units. The higher is the mean size of the tasks, the smaller is the number of tasks per resource. When the mean task size is 1000, there are 6000 tasks and 60 tasks per resource on average, and when the mean task size is 25000, there are 240 tasks and 2.4 task per resource. To simulate the **heterogeneity of tasks**, in each group, the task sizes were varied 0%, 25%, 50%, 75% and 100 %. For instance, a variation of 0% means all tasks have the same size (homogenous job), while a variation of 50% means the tasks' sizes have a uniform distribution $U(5000, 15000)$.

The **fault load** defines the types of faults that are simulated in the system during the execution of an job. In the **failure-free** fault load, there are no failures, i.e., all resources behave correctly. In the **fail-stop** fault load, a percentage of the resources crash during the simulation. We assume that a resource that fails does not recover and rejoin the grid.

4.3 Simulation without Failures

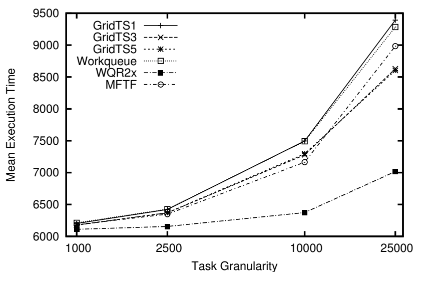
The results presented in this subsection show the performance of the scheduling algorithms in an environment where no faults were injected (i.e., the failure-free fault load). The remain three parameters were varied in this experiment: the task granularity, the task heterogeneity and resource heterogeneity. Figure 2 shows the results of the simulations with varying these parameters.

Tasks' granularity. Varying the tasks' granularity allows to see how each scheduler behaves when there are more or less tasks per resource. Figure 2(a) shows the average makespan with different mean task sizes (1000, 2500, 10000, 25000). Each point was obtained as the average of all levels of tasks' heterogeneity and resources' heterogeneity. It can be observed that when tasks are smaller, the schedulers tend to have similar performances. The reason for this behavior is that there are many tasks per resource, so all resources tend to be busy all the time. However, as the size of the tasks grows, the differences among schedulers' makespan increase.

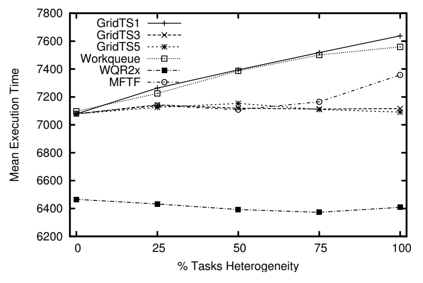
As we expected, GridTS1 had similar performance to Workqueue. With larger tasks, both had the highest makespan, since large tasks can be scheduled to slow resources near the end of the job, taking more time to terminate. The figure shows that the use of classes in GridTS minimizes this effect (GridTS3, GridTS5). Enforcing resources to execute tasks of the most fit class first, the probability of a larger task being scheduled to slow resources becomes smaller. GridTS is better when the number of tasks executed per resource is high.

WQR has better performance than the other schedulers because, at the end of simulation, it replicates the tasks to available resources. This approach however has no impact when successive jobs are being scheduled. Also, when tasks become large – less tasks per resource – the performance of WQR starts to decrease. The reason is that a large task and its replicas can be scheduled to slow resources, harming the job execution time.

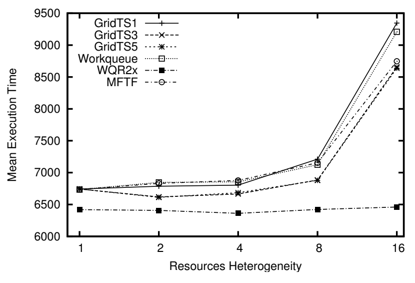
MFTF has good performance only when tasks are small. The justification for this is that MFTF assigns a task to the most suitable resource, but it may not be the fastest resource available. Therefore, the solution chosen by the scheduler may not lead to the best makespan, but it can get stable execution times similar to the expected execution time (E_i) of each task. The fitness of a task to a resource depends on E_i, so calculating E_i is crucial for getting the best makepan possible, but in practice it is hard to obtain. In the simulations, we set E_i to the mean task size divided by the mean resource speed. When tasks are larger, the task sizes heterogeneity leads to a higher distance between the ratio workload/speed and E_i, leading to lower fitness values. Therefore, tasks that are much larger or much smaller than the mean task size get worse fitness values, which harms the scheduling.



(a) Tasks' Granularity



(b) Tasks' Heterogeneity



(c) Resource' Heterogeneity

Figure 2. Average makespan varying the tasks' granularity (a), tasks' heterogeneity (b) and resource' heterogeneity (c) (no failures)

Tasks' Heterogeneity. Figure 2(b) evaluates how each scheduler behaves with different levels of tasks' heterogeneity. Like the previous figure, each point is the average of all levels of resources' heterogeneity and tasks' granularity. Like before, GridTS1 has similar performance to Workqueue. The performance of WQR remains almost unaltered in all cases, due to its replication scheme. Using classes with GridTS (GridTS3 and GridTS5) make its performance stay almost unaltered like WQR. The performance achieved by GridTS3 and GridTS5 can be credited to the ability of a powerful resource to choose a large task to execute. The performance of MFTF becomes worse when the tasks' heterogeneity augments, for the reasons discussed above: the higher the difference among tasks' sizes, the worse the fitness and the higher the job execution time.

Resource' Heterogeneity. Figure 2(c) evaluates how each scheduler behaves with different levels of resources' heterogeneity. As before, each point is the average of all levels of tasks' heterogeneity and tasks' granularity. Again, GridTS1 has similar performance to Workqueue. WQR's performance stays almost unaltered in all cases. The performance of GridTS's classes GridTS3 and GridTS5 stays almost unaltered while the resources' heterogeneity level is less or equal to 8. With level 16, their performance degrades. MFTF presents the same performance as before.

4.4 Simulation with Failures

This section presents the behavior of the algorithms when subject to different fault loads and the influence of using checkpoint mechanism in GridTS.

The experiments were carried out by having a percentage of resources failing, i.e., stopping to execute, at random time during the job execution. For GridTS is shown the results using only three classes. In the experiments, each point is the average for all levels of resources' heterogeneity. The Figure 3 show three different levels of tasks' granularity (2500, 10000, 25000), varying 50% among task sizes in each level.

As can be observed, when there are more than 50% of the resources subject to failures, the performance of GridTS3 becomes better than WQR. The reason for this behavior is that when there are many failed resources, the chance of a resource being available to replicate tasks decreases, so WQR starts behaving like Workqueue.

Similarly to fault-free environments, MFTF does not have good performance in environments subject to failures. Again, this is due to the difficulty in calculating a good value for E_i. It was calculated without considering faults in the system, since it is not clear how this information might be included in the calculation.

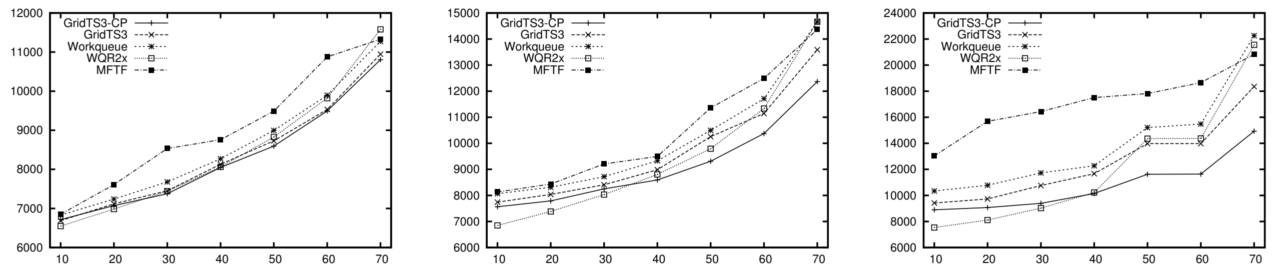
The Figure 3 also allows us to conclude that using checkpointing in GridTS allows it to increase its performance mainly with higher tasks' granularity. The reason is that when tasks are larger, the processing loss that can be avoided is higher. When more than 30% of resources are subject to failures, WQR becomes worse than GridTS3.

4.5 Summary of the Evaluation

The simulations lead us to several interesting conclusions. The first one is that GridTS with 3 or 5 classes of tasks/resources has better makespan than most of the other algorithms, with the exception of WQR when the number of resources failures is not high (in this case GridTS is also better than WQR). However, in the simulations WQR benefited from the fact that each simulation was for a single job, so WQR had the opportunity of using additional resources to replicate tasks and reduce the makespan. However, in grids permanently executing jobs this is not possible.

It is specially interesting that GridTS had better performance than MFTF because MFTF is knowledge-based, needs an information service and the non-trivial definition of a parameter (E_i), while GridTS does not have any of these difficulties.

Another interesting conclusion is that the performance of GridTS improves if there are 3 classes instead of just one, but is similar with 3 and 5 classes, so apparently there is no benefit in having more than 3 classes.



(a) 2500 (b) 10000 (c) 25000

Figure 3. Average makespan considering failures

Finally, the simulations confirm the expected result that the checkpointing mechanism has a positive effect in the makespan when there are resource failures, more when the number of failures is higher.

The simulations performed do not permit to see the benefits of always having fresh information about the resources (in GRIDTS) over having information that may be somewhat old or hard to collect (in the knowledge-based schedulers - MFTF).

5 Conclusion

This paper presented a decentralized and fault-tolerant grid infrastructure (GRIDTS), in which the resources pick the tasks to execute. The communication is made using a tuple space, benefiting from it being decoupled in time and space. GRIDTS combines different fault tolerance techniques – checkpointing, transactions, replication – to provide fault-tolerant scheduling.

We compared the performance of GRIDTS using a large number of simulations in GridSim. The decentralized scheduling mechanism proved to be quite efficient when compared with others in the literature, while not requesting a service to obtain up-to-date information about the resources.

We envisage that GRIDTS can be easily implemented and integrated with current grid systems. We plan to do an implementation in the near future, possibly integrating it with OurGrid [1]. The scalability of the infrastructure will be improved by designing a scalable tuple space.

6 Acknowledgments

This work was supported by the CNPq (Brazilian National Research Council) through processes 140848/2003-7 and 506639/2004-5, and CAPES/GRICES (project TISD).

References

[1] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg. OurGrid: An approach to easily assemble grids with equitable resource sharing. In *Job Scheduling Strategies for Parallel Processing*, pages 61–86. Springer Verlag, 2003.

[2] D. E. Bakken and R. D. Schlichting. Supporting fault-tolerant parallel programming in Linda. *IEEE Transactions on Parallel and Distributed Systems*, 06(3):287–302, 1995.

[3] 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):1175–1220, 2002.

[4] G. Cabri, L. Leonardi, and F. Zambonelli. Mobile agents coordination models for Internet applications. *IEEE Computer*, 33(2):82–89, 2000.

[5] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions Computer Systems*, 3(1):63–75, 1985.

[6] W. Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, J. Sauvé, F. Silva, C. O. Barros, and C. Silveira. Running bag-of-tasks applications on computational grids: The MyGrid approach. In *International Conference on Parallel Processing (ICCP'2003)*, pages 407–416, 2003.

[7] F. Favarim, J. da Silva Fraga, L. C. Lung, M. Correa, and J. F. Santos. Gridts: Tuple spaces to support fault tolerant scheduling on computational grids. http://www.das.ufsc.br/∼fabio/reports/2006-2.pdf, December 2006.

[8] I. Foster and C. Kesselmann, editors. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

[9] N. Fujimoto and K. Hagihara. Near-optimal dynamic task scheduling of independent coarse-grained tasks onto a computational grid. In *International Conference on Parallel Processing (ICPP-03)*, pages 391–398, 2003.

[10] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programing Languages and Systems*, 7(1):80–112, 1985.

[11] D. Gelernter and A. J. Bernstein. Distributed communication via global buffer. In *1st Annual ACM Symposium on Principles of Distributed Computing*, pages 10–18, 1982.

[12] K. Jeong and D. Shasha. Plinda 2.0: A transactional/checkpointing approach to fault tolerant Linda. In *Proceedings of the 13th Symposium on Reliable Distributed Systems*, pages 96–105, 1994.

[13] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, 1987.

[14] T. J. Lehman, S. W. McLaughry, and P. Wycko. T Spaces: The next wave. In *32th Annual Hawaii International Conference on System Sciences (HICSS'99)*, 1999.

[15] A. I. T. Rowstron and A. Wood. Solving the Linda multiple rd problem using the copy-collect primitive. *Science of Computer Programming*, 31(2-3):335–358, 1998.

[16] D. Silva, W. Cirne, and F. V. Brasileiro. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In *9th International Euro-Par Conference (EURO-PAR'03)*, pages 169–180, 2003.

[17] J. A. Smith and S. K. Shrivastava. A system for fault-tolerant execution of data and compute intensive programs over a network of workstations. In *2nd International Euro-Par Conference (EURO-PAR'96)*, pages 487–495, 1996.

[18] Sun Microsystems. Javaspaces service specification. Available in http://www.jini.org/wiki/JavaSpaces_Specification, 2003.

[19] P. Townend, P. Groth, N. Looker, and J. Xu. Ft-grid: A fault-tolerance system for e-Science. In *4th UK e-Science All Hands Meeting (AHM'05)*, 2005.

[20] S.-D. Wang, I.-T. Hsu, and Z. Y. Huang. Dynamic scheduling methods for computational grid environments. In *11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, pages 22–28, 2005.

[21] A. Xu and B. Liskov. A design for a fault-tolerant, distributed implementation of Linda. In *19th Symposium on Fault-Tolerant Computing (FTCS'89)*, pages 199–206, 1989.