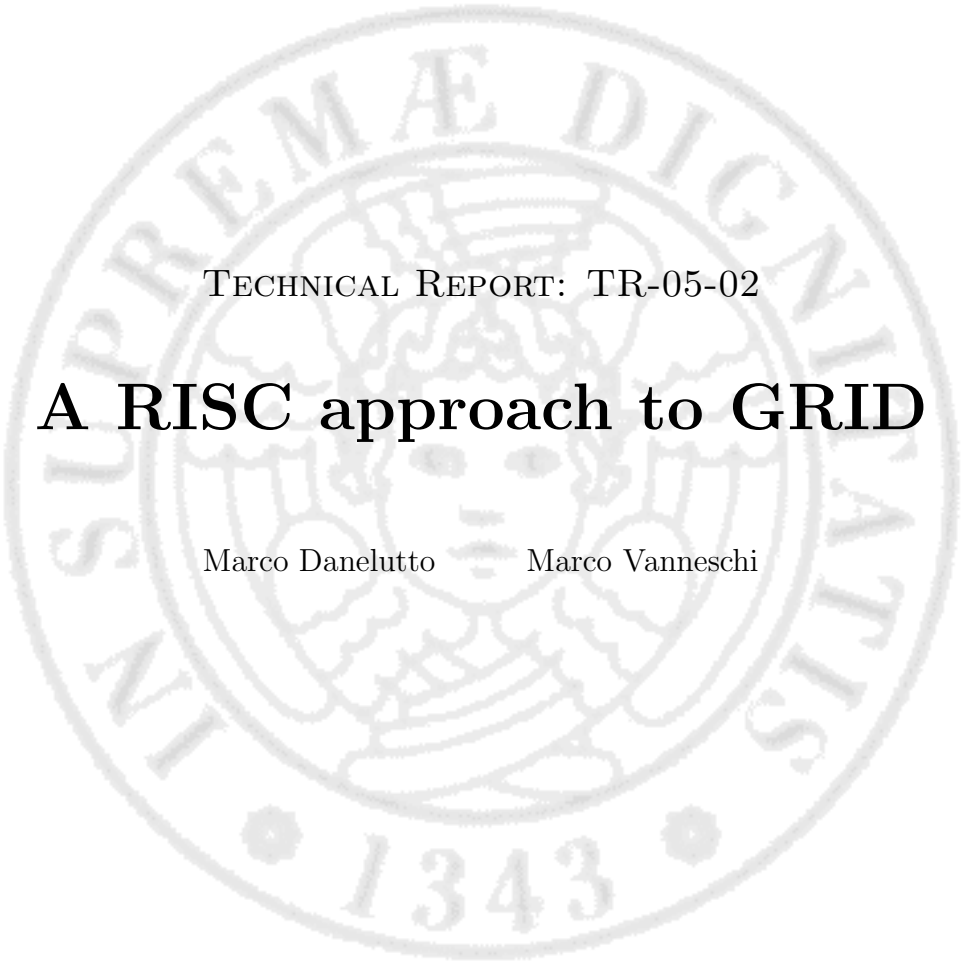
Università di Pisa
Dipartimento di Informatica



Technical Report: TR-05-02

A RISC approach to GRID

Marco Danelutto Marco Vanneschi

January 24, 2005

Address: via F. Buonarroti 2, 56127 Pisa, Italy. Tel: +39 050 2212700 Fax: +39 050 2212726

A RISC approach to GRID

Marco Danelutto Marco Vanneschi

January 24, 2005

Abstract

Current GRID technology provides users/programmers with extended and comprehensive middleware tools covering all the basic aspects a programmer must deal with when writing GRID aware applications. Programmers of GRID aware applications use such middleware systems to gather the needed resources, stage code and data to the selected computing resources, schedule computations on them and so on. Overall, a huge programming effort is required in order to come to a working, efficient GRID aware application code. Here we propose a different approach. By recognizing that most GRID applications share a common parallel/distributed structure, we propose to use application managers that take care of all the details involved in the implementation of well known GRID aware application schemas. Such application managers use the functionalities of a RISC GRID core run time system (RGC). The RGC may be built on top of existing GRID middleware. Programmers of GRID aware applications, therefore, do not directly use GRID middleware. Rather, they structure the application using the available application managers. The application managers, in turn, invoke the basic GRID services provided by RGC to accomplish both functional and performance user requirementq.

Keywords GRID, structured parallel programming models, components, RISC

1 Introduction

GRID is currently being considered as "the" target architecture for a wide range of applications. In particular, GRID is a very promising architecture when higher performance is required than those available at your own machine(s). Basically, a GRID system is a geographically distributed collection of possibly parallel, interconnected processing elements that all run some kind of common GRID middleware[10]. Globus [1] is probably the more widely used of such middleware systems. Globus provides a full set of services that can be used to program GRID applications. Such services include, as an example, services that can be queried to know which resource are currently available on the GRID, services that allow program and data to be staged at remote nodes as well as

services that can be used to schedule tasks onto the GRID processing elements (nodes) according to different scheduling policies. Currently, a typical GRID application requires a consistent amount of code (script code, XML configuration files, "middleware" specific code, etc.) to exploit the middleware features in such a way the application can be efficiently executed on the GRID. The programmer of the GRID application is charged to write all this code, at the moment. This means that the programmer of the GRID application, besides being an expert (programmer) of the application field, must also be expert in GRID middleware programming/handling.

To evaluate the amount of effort required to write a GRID application, take into account two simple cases: code staging and resource lookup in case of failures.

- When GRID applications are to be executed, resources are looked up in the GRID using the appropriate middleware services and then processes are scheduled for the execution on such resources (i.e. on the discovered available processing elements). This process has nothing complicate inside. The programmer must learn the API of the middleware used to build the GRID and then he/she must issue the proper calls for discover the available resources, "stage" (that is transfer) the code to be executed as well as the data to be processed to remote nodes and start the execution of such code. Eventually, he/she must "stage back" the computed results from the remote node to the proper place where they can be stored/consumed. Since Condor times [22], the responsibility of all these steps has been moved from programmers to the GRID run time system. Using Condor, a programmer can simply declare which code has to be computed, which input data are to be supplied to the code execution, which constrains on the target architecture have to be satisfied when executing the code (e.g. memory size, processor class, etc.) and eventually the Condor run time takes care of all the steps need to perform the computation.

- When applications are run on a GRID, the programmer must also face faulty situations arising because of the geographic distribution of the processing elements (network faults, that is nodes becoming unreachable, network slowdowns, etc.) or because of heterogenous and non-dedicated nature of these processing elements (higher priority tasks to be executed on (part of) the processing elements currently used, as an example). In order to solve this kind problems, the programmer of GRID application must organize the code in such a way that checkpoints are established within the application. These checkpoints can then be used to move the processes belonging to the application to other available GRID nodes when a fault is detected involving the GRID node originally executing that code. This process, as the staging activity formerly described, has nothing complicated inside, in principle. However, the checkpoint localization in the code is usually better devised by the programmer, which is the expert in the application field. In addition, the events that may trigger a checkpoint migration can be evaluated differently depending on the application.

Therefore, the whole process has always been and it is currently in charge of the programmer, and no middleware or GRID support tool we know is able to automatically take care of this process, at the moment.

The problems just discussed above are even more sensible when high performance computing (HPC) on GRID is taken into account. In this case, not only they have to be solved, and therefore the programmer must write proper code to handle them, but they also must be solved as efficiently as possible. And this poses further burden on the programmer, actually.

In this work we propose a new approach to GRID programming, aimed at moving most of the GRID specific efforts needed while developing HPC GRID applications from programmers to GRID tools and run time systems. We wish to clearly separate responsibilities between programmer and programming tools. That is, we want to leave to the programmer the responsibility of organizing the application specific code and to the programming tools (i.e. the compiling tools and/or the run time system) the responsibility of properly interacting with the GRID. By moving the decisions concerning GRID usage into the programming tools, we can arrange things in such a way that only a core subset of common GRID features are exploited in the programming tools. This because the programming tools can be designed to take decisions and implement strategies that are normally more elaborated with respect to the ones a generic GRID application programmer can take. In turn, having just a small core set of GRID mechanisms to implement, we can focus on their implementation and overall achieve a much higher level of optimization with respect to the optimization levels achieved in the implementation of (current) large GRID middleware. Overall, this mimics what happened in the sequential compiler/Von Neumann processor case: after designing more and more elaborated instruction sets, we understood that more efficiency can be achieved by optimizing a compact, essential instruction set and by moving the burden of properly using its instructions in the compilers. And this is why we decided to name this approach a *RISC approach* to the GRID.

The rest of the paper is organized as follows: Section 2 describes our idea of *application manager*, that is the media used to separate programmer and system responsibilities. Section 3 outlines the features that must be supported by the RISC GRID implementation. Eventually Section 4 discusses some preliminary results we achieved that support our proposal and the most significant related work.

2 Application managers

Most of the current HPC GRID applications show a common "parallel (or distributed) structure". Our idea is to encapsulate each one of these distinct common parallel structures within a parametric *application manager* module. Each application manager module should completely and efficiently handle all the details relative to the implementation of the modeled GRID parallel structure,

by properly interacting with the underlying GRID middleware system. Furthermore, each application manager should make available to the programmers methods that allow to provide the application specific code and data types in such a way that a complete, specific application can be automatically derived from the application manager.

This approach inherits from our previous experience in structured parallel programming [6, 7, 5]. In particular, the algorithmic skeleton [9, 14, 18], the design pattern [15] and coordination language [17] approaches to parallel programming exploit this idea of encapsulating the parallelism/distribution exploitation patterns within predefined, possibly extensible, programming environment features that the users may use to write their applications.

2.1 The test case: task farm computations

In order to introduce our concept of application manager consider the structure shared by embarrassingly parallel and parameter sweeping applications (embarrassingly parallel computations are often referred to as *task farm* computations). Task farm parallel applications are those applications where a set of input "tasks" (i.e. data sets) must all be independently processed executing the same code (i.e. they are *farmed out* to a set of worker processes). Each input task eventually generates a result, which can be used independently of the others. Parameter sweeping applications closely resemble embarrassingly parallel ones. Actually, taking into account their parallel structure they *are* embarrassingly parallel. The only difference is that the tasks in the input set are related each other as each one represents a set of parameters that can be used as input for the same application, usually some kind of modeling application. The analysis of the results generated from different parameter sets allows the user to individuate the "best" candidate parameter set.

The amount and the kind of programmer work required to implement an embarrassingly parallel GRID application is well known. The programmer must query the GRID middleware to look for GRID resources (i.e. processing nodes), then he/she must use middleware primitives to stage the code (once and forall) to a set of GRID nodes (possibly, the *best* nodes have to be chosen among those available) and eventually he/she must arrange a loop whose body stages input tasks to one of the remote nodes, starts computation at the remote node and eventually gathers the computed results from the remote node.

In the following sections, we will explain as all these steps can be hidden inside an application manager in such a way that task farm computations can be seamlessly developed and efficiently executed on the GRID.

2.2 Structure of the application manager

An application manager must provide three distinct kind of functionalities: basic functionalities, management functionalities and application specific functionalities. Basic functionalities implement all the steps needed to run a task farm

computation on the GRID exploiting the RGC features. Management functionalities provide the user (or different managers, see Section 2.4) the control over the task farm computation and, in particular, provide handy ways to setup the input tasks, start the computation, setup the performance requirements, etc. Last but not least, application specific functionalities provide methods to customize the manager in such a way he can implement a particular application, that is, methods that allow to specify the exact code to be executed at the remote nodes, the data types involved (tasks, results), and so on. In case the application manager is implemented in the Java framework, for instance, it will be implemented as an abstract class, basic functionalities will be implemented as protected methods, application specific functionalities as abstract methods that must be supplied by the user and management functionalities as public methods that use all the other methods.

The general structure of an application manager is depicted in Figure 1, left part. The management interface is the one used to handle management functionalities. The application specific interface is used to provide the manager the code actually specializing the manager to execute a given application. Eventually, the RGC interface allows the manager to interact with the RGC actually running on the GRID. In the following paragraphs we describe several features proper of the task farm application manager in detail.

Resource discovery Before starting any computation on the GRID, the application manager must discover the resources that will eventually be used to implement the computation. Therefore, the application manager running on the local machine must query the RGC in order to understand if such resources are available, where they are located and which are their features. This task is performed invoking the discovery service of the RGC. Such service must provide methods allowing to specify the number and the kind of computing resources the managers want to look for. The task manager must be able to figure out which are the "best" resources among those located by the RGC, in case the RGC answers more resource than those needed to execute the task farm application. Resource discovery is obviously part of the basic functionalities of this application manager. Actually, this activity is not peculiar of the task farm manager: every manager should start gathering available resource info from the GRID. The peculiar part of the task farm manager is the usage made of the discovered resources (that is their usage as task farm workers) and the way it chooses the resources to use among all those discovered (that is picking up always the more powerful resources is the maximum bandwidth is required or picking up even less powerful resource accordingly to the performance contract specified (see next paragraph below)). Both this steps can be different in case of application managers modeling a different kind of GRID application.

Performance contracts The task farm manger should achieve the maximum performance possible. The best it can do is to have a remote processing element ready to compute a new task as soon as the new task is available (e.g. it has

been read from the disk, received by another application, etc.). The manager should possibly try to discover a number of remote processing nodes sufficient to accomplish this goal before actually starting the computation. In case the RGC is not able to discover a suitable number of resources as well as in the case it has been able to discover them but then one of the resource discovered becomes slower due to some temporary overload or to some network problem, the manger should try to discover and utilize more or better GRID resources to complete the computation. In a sense, this is an activity needed to satisfy the *performance contract* of the task farm manager. This performance contract is either the one derived observing how the computation proceeds (that is considering if when a new task is available we actually also have an idle remote node) or a contract explicitly provided by the programmer using the management functionalities of our manager. The programmer may know that a bandwidth $\mathcal{B}$ has to be achieved (possibly smaller than the maximum bandwidth) because the task farm application is inserted in a framework that only requires that bandwidth in task execution, as an example. Again, this activity is not peculiar of the task farm, *per se*. Every application manager should behave in such a way the user provided performance contract is satisfied, if any, or the maximum performance allowed by the available resources is granted. The thing that is peculiar to the task farm paradigm is the way this behavior is achieved: every time the task farm application manager understands it is below the "optimal" performance it starts searching either better or new remote nodes to increase the worker string bandwidth.

Remote computation control and fault tolerance The task farm manager should implement remote computation control. In particular, it must take care of controlling that the remote nodes actually compute the tasks assigned to them. This means that the manager must be able to understand whether a remote processing node is still active. In case the manager understands the node is no more active (that is in case of remote node failure as well as of network failure) the task(s) assigned to that remote node should be reassigned to other remote nodes. Possibly, new remote nodes should be discovered to this purpose, using the RGC discovery service. This kind of control actually implements fault tolerance in the task farm manager. A different kind of "soft" fault tolerance can be considered in the task farm manager, that has already been outlined above: in all those cases where the measured performance does not match the expected performance, some kind of corrective action must be planned and implemented. We already mentioned the case when a remote processing node cannot guarantee the expected performance because of a temporary overload or because of a network slowdown. We should also mention the case where the tasks do not require a constant amount of time to be computed. In case of "hot spots", i.e. of sequences of heavy tasks, the application manager should arrange to temporarily discover and use further GRID resources to compute this hot spot set of tasks. Although fault tolerance must be ensured by every kind of application manager, the way it is achieved here is peculiar of the task farm

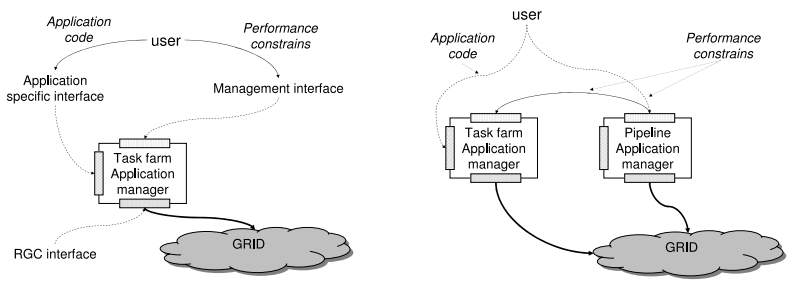


Figure 1: Manager usage outline: single manager (left) multiple, combined managers (right)

manager. More precisely, it is peculiar of the task farm paradigm. Being all the tasks independent, once the manager understands a task computation is failed, it can simply reschedule that computation to another node. Furthermore, the whole process computing a single task is actually performed in three steps: task staging to the remote node, task computation at the remote node and result staging from the remote node. These three steps roughly correspond to the an RPC/RMI mechanism. Fault tolerance can those be ensured exploiting already existing and known fault tolerant mechanisms for RPC/RMI. As we'll see below, this is not possible in case of pipeline task managers, due to the different logical network of processes used to implement the pipeline.

2.3 Other managers

To illustrate our application manager concept we used the task farm paradigm, which is a common, well understood GRID application paradigm. Here we want to discuss a couple of other managers that raise further problems and therefore can be used to better understand the manger concept.

A pipeline mananager In case our application is naturally organized in stages, a pipeline manager should be used to run the application on the GRID. The pipeline manager should address at least two further points with respect to the task farm manager. First, the tasks delivered to a non final pipeline stage produce results which are to be consumed by another (the following) pipeline stage. Therefore, the manager should "instruct" the remote nodes computing the stage to properly redirect the results to other remote nodes rather than sending them back to the manager, as it happened in the task farm application manager, instead. Second, the application manager should take care of keeping the performance of the stages balanced, to avoid that the overall pipeline performance turns out to be limited by the slower stage. This is a kind

of different performance contract the manager must guarantee, with respect to the one of the task farm manager. In the task farm manager, the behavior of the single remote node executing tasks does not affected the activity and the performance contracts of the other nodes. In this case, a slow node possibly impairs the power of the remote nodes executing subsequent pipeline stages, and therefore the manager should arrange things in such a way this does not happen. In both cases, the application manager can be implemented to efficiently address these topics once and forall, independently of the features of the application that will be eventually run using the pipeline manager.

An independent forall manager To illustrate further problems that have to be addressed and solved in the application manager, consider an independent forall application manager, that is a manager that runs on the GRID an application that is structured as a loop whose iterations are all independents. In this case the further problem to be addressed is that possibly each iteration requires a subset of a given input data structure to complete. And the subsets relative to different iterations may possibly overlap (any computation computing a new A_{ij} out of the old A_{ij} and of its neighbor values falls in this case). In this case the application manager should provide the proper subsets of the data structure to the remote nodes computing each iteration. However, in order to minimize the amount of data to be passed through the network, the application manager should either apply some kind of affinity scheduling [19] or some kind of static scheduling of the iterations to the remote nodes. In the former case, once a remote node has computed an iteration of the independent forall, the application mananager should try to schedule to that node iterations that require (part of) the same data it already received to compute previous iterations. In the latter case, the application manager should try to statically split the input data structure into overlapping sets in such a way that these sets are just sent once to the remote nodes and this communication is the only one needed to allow the remote nodes to compute the full set of iterations. The application manager can also be implemented in such a way that heuristics are used to decide if the first or the second case have to be chosen, depending on the feature of the application, those features specified by the programmer through the application specific features of the manager.

2.4 Combining the managers

Once a number of managers have been developed, each taking care of efficient execution of a particular GRID application structure, some of these managers may be combined to implement more complex GRID application structures. As an example, consider an application that can be naturally modeled by a pipeline manager, but having a stage which is considerably heavier than the other ones. In this case, the pipeline manager can simply instantiate a task farm manager to implement that pipeline stage. Therefore the performance contracts of the task farm manager will be provided by the pipeline manager (see Figure 1, right part). In a sense, the pipeline manager becomes the "programmer" of the

task farm application manager. The only constrain imposed on the manager structure by this possibility of nesting the managers, is that the management interface of the manager is uniform across all the managers. In this case, for instance, it is clear that each manager should provide a method to set up the performance contract of the controlled application.

3 A RISC GRID core

In this Section, we want to outline features that must be implemented in the RISC GRID core to support the manager concept outlined in Section 2. We eventually discuss the structure of a RISC GRID code we are currently experimenting that implements all these features.

A RGC supporting the managers described in Section 2 must provide at least the following set of core functionalities:

- a (distributed) discovery service, to be used to gather the resources needed to complete the HPC comuputation
- a set of (code and data) staging features, to be used to move data and code to and from the remote nodes
- remote control facilities, i.e. the ability to start, stop and checkpoint a computation on the remote node, to be used to control the computations on the remote nodes gathered for the application execution
- a monitoring/introspection service, i.e. the ability to monitor the execution of a computation at the remote nodes as well as to look at the different features of the remote processing nodes, to be used to control distributed execution of the application and to gather the performance parameters needed to tune the application execution.
- a communication infrastructure, i.e. the ability to set up communication channels (e.g. TCP/IP sockets) between the remote nodes involved in the computation, to be used to allow the remote nodes to exchange both data and control signals

All these features must be provided in a *secure way*. Discovery and staging services must be implemented in such a way that only authorized managers (e.g. those belonging to one of the recognized virtual organizations) receive the information concerning the available resources and are actually allowed to stage data and code. Proper mechanisms (e.g. based on sandboxes) must ensure that hosted code does not damage the local infrastructure. The same property must be ensured for monitoring and introspection. Something special instead is needed for the communication infrastructure. In this case, not only we must guarantee security, but we also need to arrange the RGC run time support in such a way that GRID application programmer can use communications (sockets, for instance) without being concerned with all the usual problems

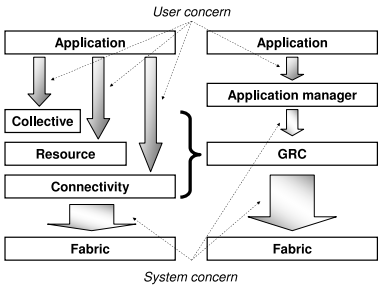


Figure 2: RGC (right) vs. "classical" GRID layers (left)

related to fire-walls, protocols etc. To this purpose we advocate the adoption of a well known port much in the sense of port 80 in the WEB framework. That is, we assume to be able to use a well know port number for the overall ensemble of the RGC features, including user communications, and we also assume that this port is usually open on firewalls. Taking into account the grain of GRID applications (i.e. the amount of time spent communicating data with respect to the amount of time spent computing at the remote nodes) the usage of a single, multiplexed port can be reasonable and applications providing nearly optimal performance can be anyway designed and implemented.

At the moment, our vision of the RGC run time support is the following: a single, daemon process installed on each one of the machines participating in the GRID. This process provides methods (usually as RPC/RMI) than can be called to retrieve node information through introspection, setup (stage) code and data, start/stop/checkpoint computations and setup communication infrastructure (i.e. declare sockets). In Section 4 we will discuss how a prototype of such daemon has been implemented in Java exploiting RMI. The daemon process can be interpreted in two distinct ways: on the one side, as *server* providing all the methods necessary to set up GRID computations and, on the other side, as an *interpreter* able to execute any kind of (legal) code, provided by some application manager according to the RGC protocols.

The general outline of the RGC just discussed and the application managers discussed in Section 2 can be interpreted with respect to usual "anatomy" GRID picture as depicted in Figure 2. The RGC is actually the layer just abstracting the "fabric" hardware and software layer. It provides the basic mechanisms that in the original figure are provided by the three layers named connectivity, resource and collective. Differently from the original figure, however, the application layer does not directly refer to the mechanism layers. Instead, the access is completely mediated by the application manager layer, which is built on top of the RGC layer. This represents the major strength point of our approach. Being the application manager layer designed by expert, GRID aware

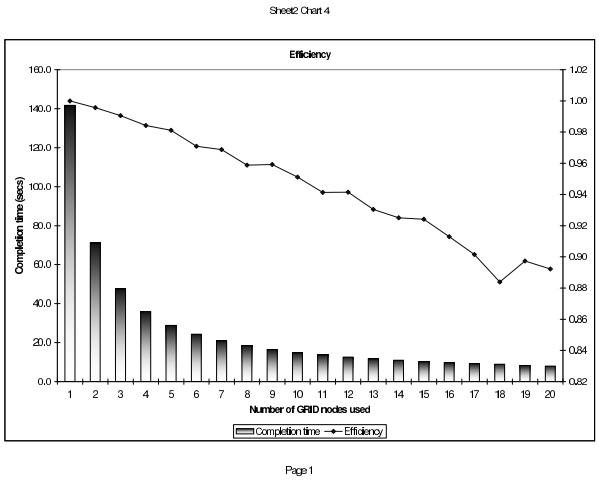


Figure 3: Scalability (task farm application manager)

programmers, and as the user only needs to call the (very) high level application manager services to implement his/her GRID application, the overall procedure leading to the production code of the GRID application turns out to be sensibly simplified and made effective due to the clear separation of responsibilities and roles.

4 RGC experiments and related work

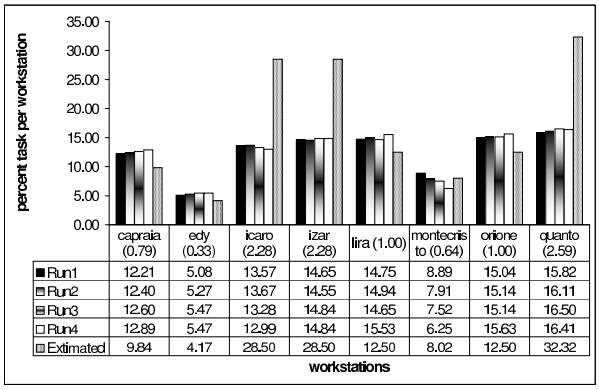
In this Section, we will outline both the experiments currently validating the proposed approach and the most notably related work. The whole work concerning RGC is mainly being developed in the framework of the Italian national FIRB project "*GRID.it*". In particular, that projects aims at designing a structured, component based, GRID parallel programming environment named ASSIST [25, 3, 2]. ASSIST has been originally developed to target heterogeneous cluster/networks of workstations. This ASSIST version runs on top of POSIX-TCP/IP workstations equipped with ACE [21]. Currently, a modified version of ASSIST is available, that can be used to run ASSIST programs on Globus 2.4 GRID [8]. Next year, the definitive ASSIST GRID version will be available that will be based on the RGC/application managers approach [4]. ASSIST is a complete parallel programming environment that provides the programmer with handy ways of expressing parallelism exploitation patterns at a very high level. To develop a parallel cluster/GRID application, the programmer must only setup a set of parameters (sequential portions of code, written in either C, C++ or FORTRAN) to specialize the available parallel programming patterns

(or skeletons). In our group, we also have a small prototype entirely written in Java, Lithium [5] that was also aimed to be a testbed for the implementation solutions to be adopted in ASSIST, being quite simple with respect to ASSIST and more easily modifiable to make experiments. We therefore modified the Lithium prototype and implemented `mulithium`, a smaller version of the original full skeleton programming environment, just including pipelines and farms. And we included in `mulithium` the basic features discussed in this paper. The RGC was implemented exploiting Java RMI, according to the principle that RGC run time is basically a daemon server running on a (open) well knonw port (1099 in this case). The basic features inserted in the RGC run time support include limited introspection features (e.g. ways to know the kind of machine the run time is running on), code and data staging capabilities as well as remote execution control features (e.g. start task, query task completion). We also developed a *quasi*-full featured task farm manager. The manager is actually capable of discovering nodes belonging to the GRID, staging the needed code on a subset (possibly all of them) of the discovered nodes, and computing a full set of tasks using those nodes. It is also able to repair damages caused by faulty nodes (e.g. nodes that took part in the computation and that are no more responding) and it has all the capabilities needed to satisfy performance contracts, either specified by the user or derived at run time, as explained in Section 2. We are currently experimenting with `mulithium` using several interconnected clusters located in Pisa. The typical results we are currently achieving are those of Figures 3 to 5.

Figure 3 plots the execution times of a task farm application processing 1K tasks when application manger discovers an increasing number of homogeneous workstations. Good scalability is achieved as efficiency is always larger than (or close to) 90%.

Figure 4, plots the total percentage of tasks executed on each one of 8 workstations, during the computation of another task farm application. The 8 workstation have different processors (different clock, different kind of Pentium, ranging from Celeron to P IV, different amounts of main store), and they also have different loads. Beside the name of the workstation an ideal "power index" is shown, which is derived from the Linux BogoMIPS measure. This Figure shows how more powerful machines actually execute more tasks than the other machines participating in the computation. The difference between the expected number of tasks per machine (i.e. the total number of tasks divided by the number of workstation and weighted w.r.t. to the workstation power index) and the measured number is mainly due to the fact the workstation all presented a different load factor when the experiments have been performed. In fact, icaro, izar and quanto are the most powerful and therefore must utilized machines in the pool; their load (as derived from the Linux command `uptime`) was near 2, whereas the load of the other machines was below 1.

Last but not least, Figure 5 plots the cost of node fault recovery. We considered three different application runs and we measured the time spent in completing the application in two cases: first when all the discovered GRID resources where available all the time, and then in case one or two of the GRID



	capraia (0.79)	edy (0.33)	icaro (2.28)	izar (2.28)	lira (1.00)	montecristo (0.64)	orione (1.00)	quanto (2.59)
Run1	12.21	5.08	13.57	14.65	14.75	8.89	15.04	15.82
Run2	12.40	5.27	13.67	14.55	14.94	7.91	15.14	16.11
Run3	12.60	5.47	13.28	14.84	14.65	7.52	15.14	16.50
Run4	12.89	5.47	12.99	14.84	15.53	6.25	15.63	16.41
Extimated	9.84	4.17	28.50	28.50	12.50	8.02	12.50	32.32

Figure 4: Load balancing (task farm application manager)

nodes used was shutdown during application execution. The figure plots the percent increase in the execution time of the application and shows that this increase is always lower that 10%.

All the experiments have been performed using synthetic applications, i.e. applications that only have the structure of real applications. In particular, we used a parametric task computation code allowing to vary both the time spent in computing the task and the size of both the input task and the output result. In this way, we have been able to perform different experiments with different computation grains. The numbers shown in the Figures of this work refer to values typical of a medical image processing application we previously used to validate the Lithium prototype [5].

Despite the fact that these experiments were run on simple networks of workstations, without any kind of other GRID related software but the RGC run time and the `mulithium` application manager, it is worth pointing out that the application written by the user is very simple. All the effort needed to achieve efficiency on the GRID is hidden in the application manager and in the RGC run time support. Basically, what the programmer must write in `mulithium` to prepare the task application is something like the code shown in Figure 6. The upper part of the code the skeleton code of the sequential portions of application specific code the programmer must supply to the manager. The lower part of the code, instead, represents almost all the code needed to run the GRID aware task farm application.

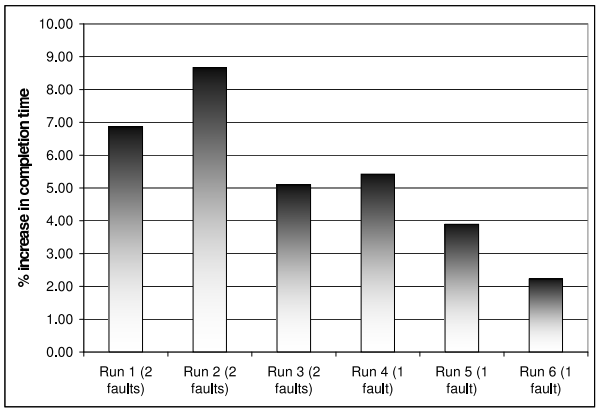


Figure 5: Fault tolerance cost: percent increase of completion time in case of two node faults repaired

4.1 Related work

Different ongoing projects currently aim at finding suitable ways of programming the high performance applications on GRID. The Condor system was originally being developed to manage cluster of workstations [20] and subsequently it has been moved to the GRID scenario [24]. Condor allows to handle task farm computations, although such computations are simply handled as a set of batch jobs, each with its own code to be executed and data to be processed.

The GrADS project [13, 23] is a large project aiming at providing handy, efficient programming tools for the GRID. It heavily uses the performance contract concept and we actually started considering the GrADS contract concept when designing the performance handling part of the managers.

Many other projects try to reduce the amount of effort required to the programmer to build a full GRID aware application. The Polder project, as an example, is aimed at providing support for the development of interactive distributed simulations, which represent a sensible GRID application [12]. However, such projects only cover some peculiar aspects of GRID programming that are those aspects more relevant to the application field they want to address.

Other projects are aimed at providing portal based facilities to access the GRID and therefore to simplify the development of GRID aware applications. The Legion grid portal is an example of this kind of projects [16]. Such kind of projects provide a lower level support to GRID application development with respect to our approach. They build an additional layer on top of the available GRID middleware but still expose the GRID application programmer to most of the cumbersome details of GRID programming.

```
/**** this is the code used to specify a sequential portion of code */
public class WorkerSeqCode implements Compute {
/* this is the method actually computing a result out of a task */
        public Object compute(Object task) {
                Object result = null;
                ...
                result = ...
                return result;
        }
}
/**** This is the code in the main, modeling the GRID application */
/* declare the code processing each one of the tasks */
Compute farmWorker = (Compute) new WorkerSeqCode(...) ;
/* declare a task farm application manager */
Farm main = new Farm(farmWorker);
Manager manager = new Manager( performanceConstrain ,
main, ...);
/* prepare the tasks to be computed */
/* for all convenient t */
        ... manager.addTask(t);
manager.noMoreTasks();
/* now compute on the GRID */
manager.eval();
/* that's all; can get results */
while(manager.moreResults()) {
        result = manager.getResult();
        ...
}
```

Figure 6: Skeleton code of a `mulithium` task farm application

Last but not least, the whole research effort aimed at integrating GRID programming and WEB services [11] presents many features that are aimed at simplifying the programming of GRID applications. Despite the fact that many of the services implemented in this framework are quite higher level with respect to normal GRID middleware service, the approach is still an approach that provides lots of *mechanism* to the programmer leaving, in the meanwhile, the complete responsibility of the correct usage of such services to the programmer.

In particular, no one of the approaches to GRID programming just mentioned tries to reduce the amount of features implemented in the GRID middleware (that is of mechanisms) in favor of better, possibly more complex capabilities or policies implemented in the programming tools.

5 Conclusions

In this work we outlined a novel approach to GRID programming. The approach is based on the adoption of a reduced instruction set GRID middleware whose features are exploited within application managers. Each application manager implements all the GRID related code needed to run applications with a given GRID structure. To develop a new GRID application sharing a structure of a given manager, the programmer must simply instantiate the manager and provide the application specific portions of code. The manager takes care of all the details needed to run the application on the available GRID resources. We also discussed some preliminary experiments aimed at verifying the feasibility of the approach. We are currently implementing a new structured GRID programming environment exploiting the features described in this work in the framework of a three year national research project.

Acknowledgements

We wish to thank Marco Aldinucci, Sonia Campa, Massimo Coppola and Corrado Zoccolo for their useful comments on this work. This work has been partially supported by Italian national FIRB project no. RBNE01KNFP "*GRID.it*" and by Italian national strategical projects "legge 449/97" No. 02.00470.ST97 and 02.00640.ST97.

References

References

[1] The Globus Project Home Page. www.globus.org, 2003. pointer to manuals, techreps and papers inside.

[2] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, M. Danelutto, P. Pesciullesi, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo. A framework for experimenting with structured parallel programming environment design. In *Proceedings of Parco'2003*, 2003. to appear.

[3] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo. The Implementation of ASSIST, an evironment for Parallel and Distributed Programming. In H. Kosh, L. Boszormenyi, and H. Hellwagner, editors, *Proceedings of Euro-Par 2003 Parallel Processing*, number 2790 in LNCS, pages 712–721. Springer Verlag, 2003.

[4] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a Research Framework for High-performance Grid Programming Environments. Technical Report TR-04-09, Dept. Computer Science, University of Pisa, 2004.

[5] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, 2003.

[6] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A Structured High level programming language and its structured support. *Concurrency Practice and Experience*, 7(3):225–255, May 1995.

[7] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: a heterogeneous environment for HPC applications. *Parallel Computing*, 25:1827–1852, December 1999.

[8] R. Baraglia, M. Danelutto, D. Laforenza, S. Orlando, P. Palmerini, R. Perego, P. Pesciullesi, and M. Vanneschi. AssistConf: A Grid Configuration Tool for the ASSIST Parallel Programming Environment. In *Proceedings of the Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 193–200. IEEE, 2003. ISBN 0-7695-1875-3.

[9] M. Cole. Bringing skeletons out of the closet, 2004. to appear in Parallel Computing.

[10] I. Foster and C. Kesselman (Editors). *The Grid 2 Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, December 2003.

[11] I. Foster, C. Kesselmann, J. M. Nick, and S. Tuecke. The Physiology of the Grid - An Open Grid Service Architecture for Distributed Systems Integration. available at http://www.globus.org/research/papers/ogsa.pdf, 2003.

[12] K. A. Iskra, R. G. Belleman, G. D. van Albada, J. Santoso, P. M. Sloot, H. E. Bal, H. J. W. Spoelder, and M. Bubak. The Polder Computing Environment: a system for interactive distributed simulation. *Concurrency and Computation: Practice and Experience*, 14(13–15):1313–1335, November-December 2002.

[13] Ken Kennedy, Mark Mazina, John Mellor-Crummey, Keith Cooper, Linda Torczon, Fran Berman, Andrew Chien, Holly Dail, Otto Sievert, Dave Angulo, Ian Foster, Dennis Gannon, Lennart Johnsson, Carl Kesselman, Ruth Aydt, Daniel Reed, Jack Dongarra, Sathish Vadhiyar, and Rich Wolski. Toward a Framework for Preparing and Executing Adaptive Grid Programs. In *Proceedings of NSF Next Generation Systems Program Workshop (International Parallel and Distributed Processing Symposium 2002)*, April 2002. Fort Lauderdale, FL.

[14] H. Kuchen. A skeleton library. In *Euro-Par 2002, Parallel Processing*, number 2400 in LNCS, pages 620–629. Springer Verlag, August 2002.

[15] S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, and K. Tan. From patterns to frameworks to parallel programs. *Parallel Computing*, 28(12):1663–1684, December 2002.

[16] A. Natrajan, A. Nguyen-Tuong, M. A. Humphrey, M. Herrick, B. Clarke, and A. S. Grimshaw. The Legion Grid Portal. *Concurrency and Computation: Practice and Experience*, 14(13–15):1313–1335, November-December 2002.

[17] G. A. Papadopoulus and F. Arbab. Control-driven coordination programming in shared dataspace. In *" "*, volume 1277 of *LNCS*. Springer Verlag, 1997.

[18] J. Serot and D. Ginhac. Skeletons for parallel image processing: the SKIPPER approach. *Parallel Computing*, 28(12):1685–1708, December 2002.

[19] Srikant Subramaniam and Derek Eager. Affinity scheduling of unbalanced workloads. In *Supercomputing '94*. IEEE Computer Society, 1994.

[20] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Condor – a distributed job scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.

[21] The ACE team. The Adaptive Communication Environment home page. http://www.cs.wustl.edu/ schmidt/ACE.html, 2004.

[22] The Condor team. The CONDOR project homepage. http://www.cs.wisc.edu/condor/, 2004.

[23] The GrADS team. The grads home page. 2004.

[24] Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002.

[25] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, December 2002.