A Transparent Framework for Hierarchical Master-Slave Grid Computing

Nadia Ranaldo[1] and Eugenio Zimeo[2]

[1] Department of Engineering, University of Sannio,
82100 Benevento, Italy
ranaldo@unisannio.it
[2] Research Centre on Software Technology (RCOST), University of Sannio,
82100 Benevento, Italy
zimeo@unisannio.it

Abstract. The use of grid computing to easily and efficiently execute data and compute-intensive applications strongly depends on new software development approaches able to separate application-domain aspects from non-functional ones, such as task mapping and deployment. In this paper, we present an object-oriented framework that is able to transparently transform non-distributed programs into hierarchical master-slave ones, and to map and schedule them onto a grid computing system. Moreover, the framework is able to leverage services delivered by the underlying middleware platform, such as resource management and communication, to satisfy user requirements. The paper presents the framework architecture, a reflection-based implementation and its evaluation atop of a hierarchical grid middleware.

1 Introduction

Thanks to the increasing amount of resources available across the Internet and to improvements of wide-area network performance, in recent years grid computing is emerging as a viable computing paradigm to execute data and compute-intensive applications.

At the state of the art, two of the main difficulties to wide diffusion of grid technologies are *usability* and *efficiency*: if the computing environment provided by the grid system is seamless, user-friendly and efficient, users will potentially exploit wide-area distributed resources to obtain high performance with a little effort related to the management of the distributed system and the deployment of applications on it. Existing distributed programming approaches based on message-passing (such as MPICH-G2 [1]) adopted for not or limited distributed systems (such as parallel machines or clusters of workstations), or "standard" approaches based on object-oriented technologies (such as Java RMI and CORBA) are hardly applicable to write and execute applications in highly dynamic and geographically distributed computing environments. These approaches, in fact, require to directly deal with problems not encountered for sequential programming, such as non-determinism, synchronization, data partitioning and distribution, load-balancing, fault-tolerance, security, etc.

W. Lehner et al. (Eds.): Euro-Par 2006 Workshops, LNCS 4375, pp. 74–86, 2007.

To overcome the burden of these approaches, new programming models, abstractions, tools and methodologies are required. In this connection, we believe that *object-oriented component frameworks* for high-level distributed programming are strategic to increase the spread of grid computing technologies (even in industrial and enterprise environments) and the productivity of grid programmers. This convincement derives from the analysis of similar technologies, such as Enterprise Java Beans and application servers employed in enterprise environments to separate functional and non-functional aspects in distributed software systems.

To improve efficiency, scalability and adaptability of applications, a framework for grid computing has to: (1) permit the programmer to focus only on domain-dependent aspects of an application, rather than on control and coordination aspects of distribution, which depend on the target environment; (2) be able to reuse the same application logic into different computing environments (such as parallel machines, clusters and Grids).

As concerning distributed computing models, in this work we focus on the master-slave pattern [2], which is a widespread architectural pattern adopted to implement coarse-grained parallel and distributed applications either in local- and wide-area networks. We focus on the hierarchical version of such pattern, since it is particularly effective to be used in intrinsically hierarchical grid computing systems, because of well-defined and limited communication patterns among computing nodes. In these systems, computing nodes are often hosted by heterogeneous resources characterized by limited-bandwidth communication in the levels of the hierarchy close to the user, and high communication performance in the other levels, typically not directly accessible through the Internet because they are often clusters accessible only through a front-end. In future we intend to take into account other widespread patterns currently adopted in the distributed computing, such as divide and conquer and pipeline.

This paper presents a framework to simplify the development of parallel and distributed object-oriented applications for grid systems. The framework, called TMS Framework (*Transparent Master-Slave Framework*), is able to transparently implement hierarchical master-slave applications in a hierarchical grid environment, and to satisfy Quality of Service (QoS) requirements by dynamically exploiting services delivered by underlying middleware platforms. The framework was implemented by leveraging reflection mechanisms provided by a meta-object protocol [3]. We considered, moreover, its customisation for a hierarchical grid middleware [4], which delivers an economy-driven resource broker usable by the TMS Framework to automatically map and schedule distributed tasks satisfying time and cost constraints specified by the user.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 presents the TMS Framework. Section 4 describes a reflection-based framework implementation. Section 5 presents an evaluation of the TMS Framework in writing a distributed application and a preliminary experimental analysis, and finally Section 6 summarizes the paper and presents future work.

2 Related Work

Some frameworks for master-slave applications in dynamic and heterogeneous systems have been proposed in literature. The most significant ones are *AppLeS*

Master-Worker Application Template (AMWAT) [5] and *Condor Master-Worker* (MW) [6]. Also Javelin 3 [7] and Satin [8] are interesting proposals.

AMWAT is a library that provides a software template to implement self-scheduling master-slave applications written in C, C++, and Fortran in distributed memory architectures. The AMWAT programming interface specifies the high-level functionalities that the application developer must minimally supply. Such functionalities are provided in form of portable and reusable modules. In particular, the Application Template module contains fifteen application activity functions, which are provided by developers to implement application-specific functions.

Condor MW is a framework proposed for implementing grid-enabled master-slave applications written in C++. Condor MW provides a "top-level" interface to application software and a "bottom-level" interface, called Infrastructure Programming Interface (IPI). The top-level interface permits to parallelize an application and requires the programmer to re-implement some abstract classes, in particular the MWTask, which is the abstraction of one unit of work, and the MWWorker, which represents a slave process. The IPI interface permits to use existing grid computing toolkits without any changes from the view-point of the application developer.

While the AMWAT approach focuses on application performance in terms of execution time, the Condor MW approach emphasizes the delivery of high throughput computing. It typically deals with many processor faults, since the default Condor behaviour is to vacate a running process on a remote machine when it is no longer in idle status.

Even if the approaches described above permit to simplify writing of master-slave applications by hiding distribution, scheduling and communication aspects, they still require to explicitly write code for the distributed version of the problem, requiring a specific implementation of the application for the master-slave pattern and so limiting the programmer productivity and existing code re-use.

A better separation of functional aspects from non-functional ones can be reached through the new programming approach based on skeletons [9], conceived to design easy-to-use structured parallel programming environments. The idea is to capture recurring patterns in parallel and distributed applications in generic software constructs that can be customized by the programmers to write different applications.

A recent proposal is HOC [10] based on Web services, which requires configuring services through application-specific code, such as, in the master-slave pattern, how to split input parameters among the slaves and how to process them. Such customisation is obtained through the implementation of specific interfaces.

Another proposal that focuses on grid systems is Lithium [11], a library based on Java and RMI, which supports common skeletons, including pipelines, task farms, iterative and data parallel skeletons.

As for the skeleton-oriented approaches, our goal is to simplify writing distributed applications, considering the difficulty in learning new paradigms and programming approaches. For this reason, we propose a framework that permits writing (or re-using) an application in a sequential version, hiding the distributed aspects related to

the pattern/s adopted for its deployment. Our idea is to configure the pattern-related aspects through a preliminary phase that requires writing a configuration file and the classes for the framework customization. Moreover we focus on a framework implementation that hides pattern-related aspects in some configurable components of the system, able to leverage existing grid services, for example resource discovery and load balancing.

Most of the distributed computing environments for master-slave applications, which deliver scheduling functionalities, use mapping algorithms that try to optimise only the execution performance [12] [13] [5]. In a future commercialisation of grid technologies, the resource price will represent a distinctive property to regulate the supply-and-demand for resources. To this end, the work in [14] represents one of the first effort to introduce economy-driven mapping algorithms for generic applications with no control or data dependencies; whereas a previous paper of the authors [15] defines a heuristic for mapping tasks to the slaves of a master-slave application based on deadline and budget constraints.

3 TMS Framework

The TMS Framework design is based on the following principles: (1) *separation of concerns*: the framework has to permit a programmer to concentrate only on the domain-dependent aspects, without dealing with low-level aspects of distributed computation such as the definition of the number of resources, the distribution of tasks among the resources, synchronization, etc.; (2) *code re-use*: the framework has to permit the re-use, in a distributed computing environment, of existing code written to solve the same problem in sequential manner, so permitting to use, in a nearly-seamless way, the same code for execution on a single workstation, or on a homogeneous cluster or on a heterogeneous wide-area distributed system; (3) *adaptability*: the framework has to dynamically leverage services delivered by the underlying computation architecture in order to automatically optimise application execution and fulfil user QoS requirements.

The main objective of the proposed framework is to re-use existing code for sequential execution to automatically produce a parallel and distributed version of it through the adoption of the hierarchical master-slave pattern at run-time. The hierarchical master-slave pattern consists of extending the single master of the canonical pattern to a hierarchy of masters at different levels. The master at the top controls the overall computation and distributes it among the masters at lower levels, and so on, until the computation is sent to the slaves, which directly process the request. The collection of computed results is performed in the reverse order. With respect to the master-slave pattern, it permits to increase scalability by removing the centralized control of a single master, which could easily become a bottleneck for a high number of resources and limited-bandwidth networks.

The TMS Framework provides a run-time distributed environment in which masters and slaves run. To achieve separation of concerns, it defines a generic

architectural skeleton, which can be customized by the user through application-domain code used for sequential version, and some descriptive information for the deployment. The distribution aspects that depend on the underlying computational infrastructure are captured and managed by the framework, without the necessity of application-domain code modification.

The framework is designed so to automatically manage and trigger well-defined coordination activities of the hierarchical master-slave model, which are: (1) splitting of the workload, (2) call to slaves, (3) waiting and gathering of results performed by the master. The idea is to set up such well-defined activities through a configuration phase, which permits to specify the policies to adopt for each activity. In figure 1, the main components of the framework are shown, considering one level of the hierarchy for simplicity.

The framework is used to dynamically parallelize object-oriented applications whose functional aspects are delivered through a method of a class (called in the following *Task* class), which implements a sequential solution to a given problem. To transparently turn the sequential computation of such method into a parallel one, a Task object is used to customize the main framework component, called *TMS Task*. The TMS Task is loaded into the TMS Framework of each computing node and is configured in order to act as master or slave of the computation. For a master node the TMS Task consists of the replication of the original Task object, and a customisable framework component, called *Master Behaviour*, which performs the master functionalities of workload splitting and result gathering.

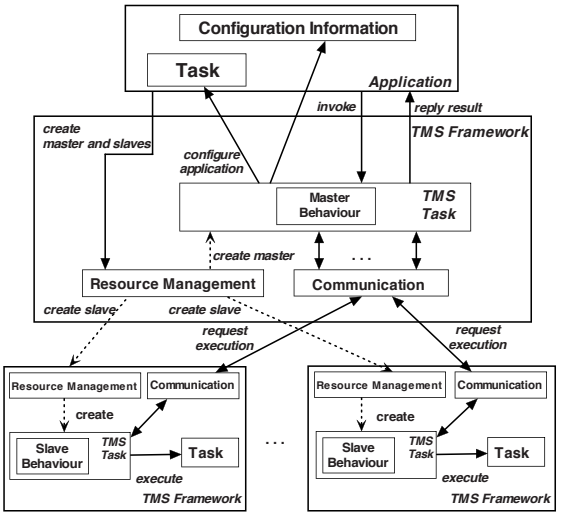

Fig. 1. TMS Framework Architecture

For a slave node the TMS Task consists of the replication of the original Task object, and a framework component, which performs the slave activities, called *Slave Behaviour* component.

The framework contains other two configurable components used to capture and manage the main capabilities required to distribute and manage a master-slave application, that are the *Resource Management* and the *Communication* components. The Resource Management component has the task to schedule and manage the masters and slaves on distributed resources. The communication and synchronization between masters and slaves is managed by the Communication component.

4 Reflection-Based Implementation

The main goal of the TMS Framework is the *implicit* implementation of the hierarchical master-slave pattern, which can be achieved following static or dynamic approaches.

The static approach is based on specialized pre-compilers, which take care of parallelising the application and deploying it on distributed resources. Such approach does not fit in a dynamic distributed environment since it does not permit to perform on-the-fly modifications in order to adapt applications to variations in underlying services and resource availability.

The dynamic approach permits to overcome such limitations and is based on the openness of the system to change, even at run-time, some aspects of its behaviour.

We propose a version of the TMS Framework based on a dynamic approach implemented with reflection mechanisms [2]. A reflection-based framework permits to easily adapt components to changing conditions, and to extend or reconfigure the system to meet new requirements. With this approach, an application is logically divided in two parts: the *meta-level* and the *base-level*. The meta-level is the part of application which provides knowledge of its properties and makes the system self-aware. The system properties available at the meta-level are represented by *Meta Objects*, which encapsulate and represent information about a single system aspect that should be adaptable. The base-level models and implements the application logic and represents the various services the system offers. Its implementation uses information and services provided by the meta-level to remain flexible and independent from those aspects that are likely to be modified.

A reflection-based TMS Framework requires individuating the set of Meta Objects, which capture the incomplete parts of the framework and permit to customize it for the execution of an application. Reflection mechanisms are also used to customize the Resource Management and Communication components so to deliver functionalities exploiting existing basic services of the underlying middleware.

4.1 MOP-Based Implementation

We implemented the dynamic master-slave pattern by exploiting the reflection features provided by Meta-Object Protocol (MOP) implemented in ProActive [3]. It is a proxy-based run-time mechanism, which permits reification of method invocations and constructor calls. It is entirely written in Java and avoids any modification or extension to the JVMs, as opposed to other meta-object protocols.

By using MOP, the TMS Framework permits to employ every existing class to transparently instantiate the set of master and slave active objects (ProActive objects), keeping the application very similar to that used for a sequential computation.

Therefore, the hierarchical master-slave pattern is dynamically implemented and an existing object can be turned in a master able to transparently split the overall task into sub-tasks and in a slave able to perform the assigned part of the overall task.

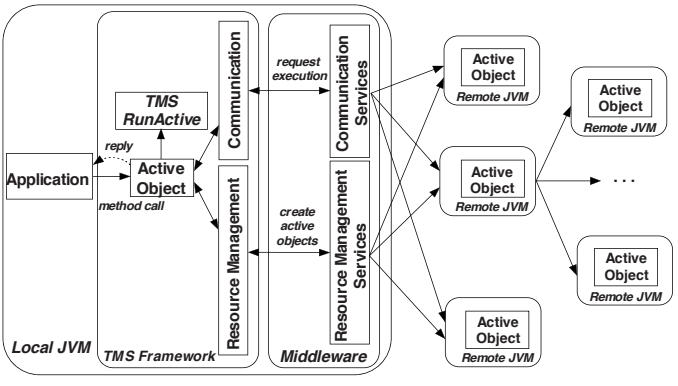

Fig. 2. MOP-based TMS Framework implementation

The behaviour of active objects, with respect to the parallelism exploitation patterns, is specified through the implementation of the `RunActive` interface of ProActive, delivered by the framework and called `TMSRunActive`, which specifies the actions executed by the active object when a method execution request is received (see figure 2). In particular, if the active object is a master, the following actions are performed: (1) to collect information on the performance capabilities of each resource available for computation; (2) to perform a partition of input parameters following the policies indicated in a configuration file; (3) to send method calls to the active objects which are masters or slaves of the lower level, using as input parameters: the original input parameter, if it is a non-partitioned parameter, or the corresponding part of the partition, if it is a partitioned parameter; (4) to wait for the collection of each partial results, which are assembled following the policy specified in a configuration file. If the active object is a slave, it directly executes the method.

4.2 Programming Model

The programming model of the TMS Framework permits the parallelisation of one or more tasks, each represented by a method of an existing class. The parallelisation is initialised through the *Configuration Phase*, performed delivering a configuration file and invoking the static method of the `TMSFramework` class:

```
Object configureDistributedTask(Object original, String configFile);
```

that returns a reified object. The input parameters are `original`, which is an instance of the class used to perform the distributed task and `configFile`, which is the name of the configuration file used to configure the deployment of active objects. It is an

XML-based file, called *Job Description Format* (JDF) in which a part depends on the underlying middleware adopted for active objects deployment, while another is common and is used for the reflection mechanisms. The common part contains the following information: (1) the methods whose invocations have to be distributed over the active objects; (2) for each method, the input parameters that have to be partitioned and the policy to partition each of them; (3) for each method, the assembling policy of the output parameter.

A partition policy is specified by the implementation of the following method of the `SplitHelper` interface:

```
public Object[] split (Object[] data, double[] caps);
```

in which, `data` represents the information used to obtain a partition on an input parameter and `caps` the performance information on each active object, used to eventually obtain a load balanced partition.

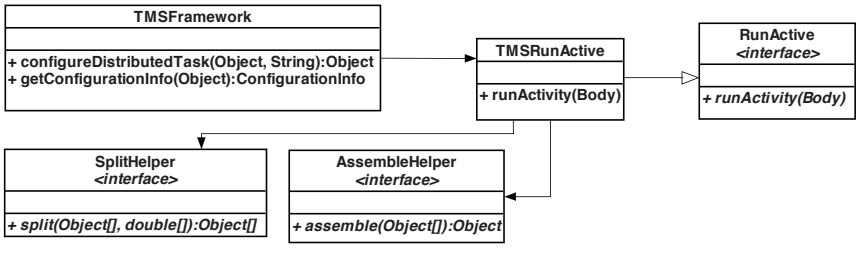

Fig. 3. Class Diagram of MOP-based TMS Framework

An assembling policy is specified by the implementation of the following method of the `AssembleHelper` interface:

```
public Object assemble (Object[] data);
```

in which `data` represents the partial results to assemble into a single object representing the overall result of the distributed computation.

The Configuration Phase is followed by the *Execution Phase*, in which the user performs method invocations in the same way as for standard objects. The method invocation on ProActive active objects is asynchronous, which permits to increase the concurrency among local and remote activities.

4.3 User QoS Requirements

The default version of ProActive leverages Java RMI and, as a consequence, requires the direct handling of scheduling functionalities of resource discovery, selection and task mapping, limiting the capability to fulfil user QoS requirements.

Through the adoption of the ProActive-HiMM adapter [16], the TMS Framework can be configured to transparently leverage HiMM functionalities. HiMM is a Java-based middleware able to exploit hierarchical collections of computers interconnected by heterogeneous networks. Even if HiMM is not a complete grid middleware (it lacks of sophisticated security mechanisms and efficient data access), it delivers all the basic

services of resource discovery, management, scheduling, and efficient communication mechanisms useful to implement master-slave applications into a grid system.

HiMM, in particular, provides an economy-driven broker for master-slave applications which is responsible for automatic resource discovery and task mapping on the basis of availability, performance and cost of resources, and on time and cost parameters specified by the user. It is based on the task mapping heuristic proposed in [15] which permits to minimize the total execution time without exceeding a fixed budget. The HiMM resource broker can be adopted by the TMS Framework to transparently deploy the distributed tasks of resources satisfying time and cost constraints specified by the user in the configuration phase. This is obtained following the programming model described above and using a file (JDF – Job Description Format – file) which contains all the information necessary to exploit broker functionalities of HiMM, which are application information (task dependencies, overall complexity, single task complexity, etc), application code, input data and user requirements. The current version of the HiMM broker does not take into account the mapping problem of master hierarchy because it focuses on a grid system with an intrinsic hierarchical topology, in which the masters are naturally hosted on those machines used as front-end for pools of resources such as clusters.

5 Framework Evaluation

To evaluate the usefulness for programming and to analyse the performance delivered by TMS Framework, a simple application is described. It is the well-known multiplication of square matrices implemented with the master-slave pattern and using the strip partitioning of the left matrix: the master partitions the received left matrix and sends the parts to slaves for processing.

A standard class `Matrix`, eventually already written for sequential applications, delivers a constructor to initialise a bi-dimensional array of `float` values, and the `multiply` method that sequentially executes the multiplication between the current matrix, used as right matrix, and the matrix passed as parameter, used as left matrix.

The following code shows the use of the TMS Framework to turn a standard instance of `Matrix` into a transparent master-slave one:

```
...
Matrix rigMat = new Matrix(...); // initialisation
Matrix leftMat = new Matrix(...);
Matrix result = null;
String configFile = null;
 // Configuration Phase: definition of the XML-based JDF file
 ...
rigMat=(Matrix)TMSFramework.configureDistributedTask(
                                                    rigMat,configFile);
// Execution Phase
result = rigMat.multiply(leftMat);
...
```

The parallelisation of the multiplication of two matrices requires to specify, in a JDF file, the class which contains the method to parallelise, that is `Matrix`, and the classes which implement the `SplitHelper` and `AssembleHelper` interfaces used, respectively, to split a matrix in blocks of rows and to assemble blocks of rows into one block to return a single matrix.

We underline that such classes could be already available in a library included in the framework or delivered by a third-part developer. A section of the JDF file for this application is reported below.

```
<APPLICATION-STRUCTURE>
  <DISTR-PROG-MODEL>Master-Slave</DIST-PROG-MODEL>
  <MIDDLEWARE-SPECIFIC-INFORMATION>
   <USER-REQUIREMENTS>
     <DEADLINE>50000</DEADLINE>
     <BUDGET>100</BUDGET>
     <MAPPING-POLICY>TIME_OPTIMIZATION</MAPPING-POLICY>
   </USER-REQUIREMENTS>
  ...
  </MIDDLEWARE-SPECIFIC-INFORMATION>
  <TASKS>
     <TASK>
      <TASK-CLASS-NAME>Matrix</TASK-CLASS-NAME>
      <METHOD-NAME>multiply</METHOD-NAME>
      <METHOD-PARAMETERS>
        <METHOD-PARAMETER>Matrix</METHOD-PARAMETER>
      </METHOD-PARAMETERS>
      <RETURN-TYPE>Matrix</RETURN-TYPE>
      <DATA>
       <DISTRIBUTED-INPUTS>
        <INPUT>
           <INPUT-TYPE>Matrix</INPUT-TYPE>
           <INPUT-INDEX>0</INPUT-INDEX>
           <PARTITION>
             <PARTITION-CLASS-NAME>TMSFramework.util.MatrixSplitHelper
           </PARTITION-CLASS-NAME>
             <PARAMETERS>
               <PARAMETER>
                 <INPUT-TYPE>Matrix</INPUT-TYPE>
                 <INPUT-INDEX>0</INPUT-INDEX>
               </PARAMETER>
             </PARAMETERS>
           </PARTITION>
        </INPUT>
       </DISTRIBUTED-INPUTS>
       <ASSEMBLING>
          <ASSEMBLING-CLASS-NAME>TMSFramework.util.MatrixAssembleHelper
          </ASSEMBLING-CLASS-NAME>
           <PARAMETERS>
             <PARAMETER>
               <INPUT-TYPE>Matrix</INPUT-TYPE>
               <INPUT-INDEX>-1</INPUT-INDEX>
             </PARAMETER>
           </PARAMETERS>
       </ASSEMBLING>
      </DATA>
     </TASK>
  </TASKS>
</APPLICATION-STRUCTURE>
```

Figure 4 shows the components that must be provided by the developer to configure the framework and Figure 5 shows the deployment of the components on a pool of distributed resources through a broker for resource management. During the configuration phase, the broker is adopted to discover and select a pool of resources able to satisfy user performance and cost requirements specified in the JDF file. Selected resources are adopted to build a hierarchical virtual machine managed through HiMM.

We conducted a preliminary performance analysis on a network of workstations composed of fourteen homogeneous machines, each equipped with Intel Pentium Xeon 2.8 GHz, a RAM of 1GB, running Custer-Linux Rocks ver. 4 operating system, and inter-connected by a Fast Ethernet network. The used software packages are Java 2 SDK 1.4.2, ProActive version 3.0 and HiMM version 1.1. We used the multiplication of two square matrices as benchmark and adopted the time

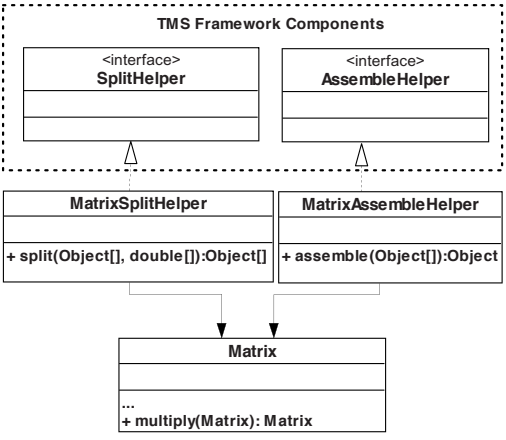

Fig. 4. Configuration of the TMS Framework for the matrix multiplication application

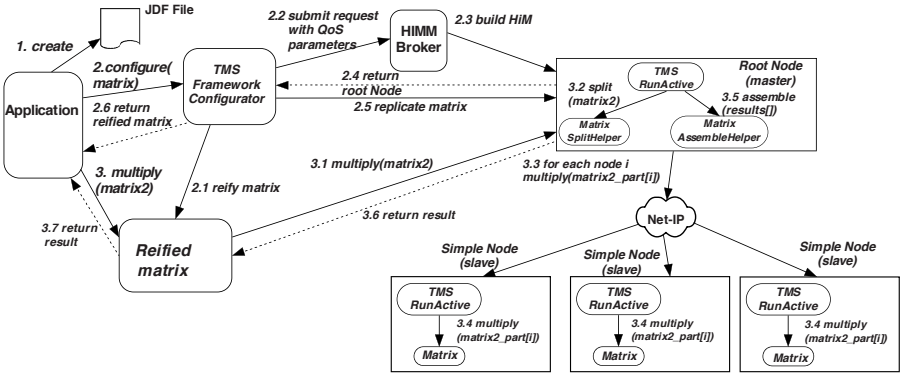

Fig. 5. Dynamics of the TMS Framework components for the execution of matrix multiplication application

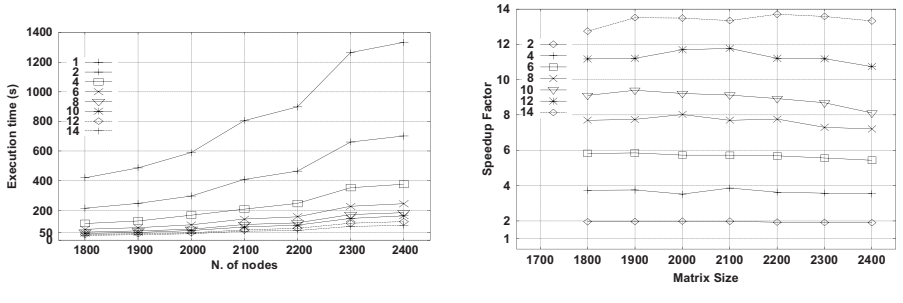

Fig. 6. (a) Execution times (b) Speedup factors

minimization heuristic considering the same performance parameters for each resource so to obtain roughly the same execution time on each of them. We measured the overall execution times and evaluated the speedup factor considering various matrix sizes and various numbers of available resources. The execution times and speedup factors are reported in figure 6 (a) and (b), whose trends show the system efficiency.

6 Conclusion

We defined a component framework able to automatically implement the hierarchical master-slave pattern in a distributed environment leveraging the application code for the sequential solution. We described a reflection-based implementation that exploits reflection to use the services of the underlying grid middleware. The usability of the TMS Framework for writing distributed applications and the results of an experimental analysis to prove the system efficiency were shown. In future, we will test the system scalability of the TMS Framework for a heterogeneous hierarchical environment. Moreover, we intend to customize the TMS Framework so to leverage more efficient communication mechanisms based on IP multicast for master-slave interactions and other middleware services, such as the WSRF-complaint services delivered by Globus [17].

References

1. Karonis, N., Toonen, B., Foster, I.: Mpich-G2: A Grid-enabled Implementation of the Message Passing Interface. Journal of Parallel and Distributed Computing, 63(5) (2003) 551-563
2. Bushmann, F., et al.: Pattern-Oriented Software Architecture: A System of Patterns. J. Wiley and Sons (1996)
3. Caromel, D., Klauser, W., Vayssiere, J.: Towards Seamless Computing and Metacomputing in Java. Concurrency: Practice and Experience, Vol. 10 (11-13) (1998) 1043-1061
4. Di Santo, M., Frattolillo, F., Russo, W., Zimeo, E.: A Component-based Approach to Build a Portable and Flexible Middleware for Metacomputing. Parallel Computing, Elsevier, 28(12) (2002) 1789-1810
5. Berman, F., et al.: Adaptive Computing on the Grid Using AppLeS. IEEE Trans. Parallel and Distributed Systems, 14(4) (2003) 369-382
6. Linderoth, J., Kilkarni, S., Goux, J. P., Yoder, M.: An Enabling Framework for Master-Worker Applications on the Computational Grid. Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing, Pittsburgh, Pennsylvania, (2000) 43-50
7. Neary, M. O., Cappello, P.: Advanced Eager Scheduling for Java-Based Adaptively Parallel Computing. Proceedings of the joint ACM-ISCOPE Conference on Java Grande, (2002)
8. van Nieuwpoort, R. V., Kelmann, T., Bal, H. E.: Efficient Load Balancing for Wide-Area Divide-and-Conquer Applications. Proceedings of the 8-th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, Utah, (2001) 34-43
9. Cole, M. I.: Algorithmic Skeletons: a Structured Approach to the Management of Parallel Computation. MIT Press & Pitman, (1989)

10. Gorlatch, S., Dunnweber, J.: From Grid Middleware to Grid Applications: Bridging the Gap with HOCs. In Future Generation Grids, Springer-Verlag, (2005)
11. Aldinucci, M., Danelutto, M., Teti, P.: An Advanced Environment Supporting Structured Parallel Programming in Java. Future Generation Computer Systems, 19(5) (2003) 611–626
12. Banino, C., Beaumont, O., Carter, L., Ferrante, J., Legrant A., Robert, Y.: Scheduling Strategies for Master-Slave Tasking on Heterogeneous Processor Platforms. IEEE Transaction on Parallel and Distributed Systems, 15(4) (2004) 319-330
13. Martino, V., Mililotti, M.: Scheduling in a Grid Computing Environment using Genetic Algorithms. International Parallel and Distributed Processing Symposium, Florida, USA, (2002)
14. Buyya, R., Murshed, M., Abramson, D.: A Deadline and Budget Constrained Cost-Time Optimization Algorithm for Scheduling Task Farming Applications on Global Grids. In Proceedings of Par. and Distr. Processing Techniques and Applications, USA, (2002)
15. Ranaldo, N., Zimeo, E.: An Economy-driven Mapping Heuristic for Hierarchical Master-Slave Applications in Grid Systems. 15-th Heterogeneous Computing Workshop. In Proceedings of the International Parallel and Distributed Processing Symposium, Greece (2006)
16. Di Santo, M., Frattolillo, F., Ranaldo, N., Russo, W., Zimeo, E.: Programming Metasystems with Active Objects. Proceedings of the International Parallel and Distributed Processing Symposium, France, (2003)
17. WSRF. http://www.globus.org/wsrf