Infrastructure-Agnostic Programming and Interoperable Execution in Heterogeneous Grids

Enric Tejedor[1], Javier Álvarez[1], Rosa M. Badia[1,2]

[1] Barcelona Supercomputing Center (BSC-CNS)
Jordi Girona 29, 08034 Barcelona (Spain)
`enric.tejedor@bsc.es, javier.alvarez@bsc.es, rosa.m.badia@bsc.es`
[2] Artificial Intelligence Research Institute (IIIA),
Spanish Council for Scientific Research (CSIC)
E-08193 Bellaterra, Barcelona (Spain)

Abstract. In distributed environments, no matter the type of infrastructure (cluster, grid, cloud), portability of applications and interoperability are always a major concern. Such infrastructures have a high variety of characteristics, which brings a need for systems that abstract the application from the particular details of each infrastructure. In addition, managing parallelisation and distribution also complicates the work of the programmer.
In that sense, this paper demonstrates how an e-Science application can be easily developed with the COMPSs programming model and then parallelised in heterogeneous grids with the COMPSs runtime. With COMPSs, programs are developed in a totally-sequential way, while the user is only responsible for specifying their tasks, i.e. computations to be spawned asynchronously to the available resources. The COMPSs runtime deals with parallelisation and infrastructure management, so that the application is portable and agnostic of the underlying infrastructure.

Keywords: Grid programming models, Workflow managers, Parallelism exploitation

1 Introduction

In distributed environments, no matter the type of infrastructure (cluster, grid, cloud), portability of applications and interoperability are always a major concern [3, 2]. Different infrastructures can have very diverse characteristics. Besides, even in the scope of a given infrastructure, there is typically a plethora of alternatives to implement and execute an application, and often several vendors compete to dominate the market. Choosing one of the alternatives usually ties the application to it, e.g. due to the use of a certain API. As a result, it may be hard to port the application, not only to another kind of infrastructure, but also to an equivalent platform provided by another vendor or managed by different software.

Standards do appear, either 'de facto' or produced by collaborative organisations that develop them, as in the case of the Open Grid Forum [6], but it is often complicated for them to be widely accepted. This situation, which is likely to keep happening in future scenarios, increases the importance of systems that free the user from porting the same application over different platforms.

On the other hand, some of the difficulties of programming applications for distributed infrastructures are not related to their particular characteristics, but to the duty of parallelisation and distribution itself [16]. This includes aspects like thread creation and synchronisation, messaging, data partitioning and transfer, etc. Having to deal with such aspects can significantly complicate the work of the programmer as well.

In that sense, this paper demonstrates how the COMPSs programming model and runtime system can be used to easily develop and parallelise applications in distributed infrastructures. More precisely, we discuss an example of an e-Science application that was programmed with the COMPSs model. Such application does not include any API call, deployment or resource management detail that could tie it to a certain platform. In addition, the application is programmed in a fully-sequential fashion, freeing the programmer from having to explicitly manage parallelisation and distribution.

Furthermore, we present some experiments that execute that application in large-scale heterogeneous grids controlled by different types of middleware. A runtime is responsible for hiding that heterogeneity to the programmer, interacting with the grids and making them interoperable to execute the application. Consequently, the application remains agnostic of the underlying infrastructure, which favours portability.

The paper is structured as follows. Section 2 provides an overview of the COMPSs programming model and runtime system. Section 3 introduces the use-case e-Science application. Section 4 describes the Grid testbed used in the experiments. Section 5 presents the results of the experiments. Finally, Section 6 discusses some related work and Section 7 concludes the paper.

2 Overview of COMP Superscalar

This section introduces the COMP Superscalar (COMPSs) programming model, as well as the runtime system that supports the model's features. COMPSs is tailored for Java applications running on distributed platforms like clusters, grids and clouds. For a more detailed description of COMPSs, please see [20, 22, 21].

2.1 Programming Model

The COMPSs programming model can be defined as task-based and dependency-aware. In COMPSs, the programmer is only required to select a set of methods and/or services called from a sequential Java application, for them to be run as *tasks* - asynchronous computations - on the available distributed resources.

The task selection is done by providing a Task Selection Interface (TSI), a Java interface which declares those methods/services, along with some metadata. Part of these metadata specifies the direction (input, output or in-out) of each task parameter; this is used to discover, at execution time, the data dependencies between tasks. The TSI is not a part of the application: it is completely separated from the application code and it is not implemented by any of the user's classes; its purpose is merely specifying the tasks.

With COMPSs, sequential Java applications can be parallelised with no modifications: the application code does not contain any parallel construct, API call or pragma. All the information needed for parallelization is contained in the TSI. Besides, the application is not tied to a particular infrastructure: it does not include any resource management or deployment information.

2.2 Runtime System

The runtime system receives as input the class files corresponding to the sequential code of the application and the TSI. Before executing the application, the runtime transforms it into a modified bytecode that can be parallelised. In particular, the invocations of the user-selected methods/services are automatically replaced by an invocation to the runtime: such invocation will create an asynchronous task and let the main program continue its execution right away.

The created tasks are processed by the runtime, which dynamically discovers the dependencies between them, building a task dependency graph. The parallelism exhibited by the graph is exploited as much as possible, scheduling the dependency-free tasks on the available resources. The scheduling is locality-aware: nodes can cache task data for later use, and a node that already has some or all the input data for a task gets more chances to run it.

The interaction of the runtime with the infrastructure is done through JavaGAT [11], which offers a uniform API to access different kinds of Grid middleware. COMPSs uses JavaGAT for two main purposes: submitting tasks and transferring files to Grid resources. Thus, the runtime is responsible for transferring task data and managing task execution through JavaGAT, while the application is totally unaware of such details.

3 The SimDynamics Application

The SimDynamics application, which will be used in the experiments presented in Section 5, is a sequential Java program that makes use of DISCRETE [9], a package devised to simulate the dynamics of proteins using the Discrete Molecular Dynamics (DMD) methods.

Starting from a set of protein structures, the objective of SimDynamics is to find the values of three parameters that minimise the overall energy obtained when simulating their molecular dynamics with DISCRETE. Hence, SimDynamics is an example of a parameter-sweeping application: for each parameter, a fixed number of values within a range is considered and a set of simulations (one per structure) is performed for each combination of these values (configuration). Once all the simulations for a specific configuration have completed, the configuration's score is calculated and later compared to the others in order to find the best one.

In order to run SimDynamics with COMPSs, a total of six methods invoked from the application were chosen as tasks. This was done by defining a TSI that declares those methods. Figure 1 contains a fragment of this TSI, more precisely the selection of method simulate as a task. The parameters of simulate are three input files, an input string and an output file. The declarations of the other five methods are analogous to this one.

```
public interface SimDynamicsItf {

    @Method(declaringClass = "simdynamics.SimDynamicsImpl")
    void simulate(
        @Parameter(type = FILE) String paramFile,
        @Parameter(type = FILE) String topFile,
        @Parameter(type = FILE) String crdFile,
        String natom,
        @Parameter(type = FILE, direction = OUT) String average
    );

    ...

}
```

Fig. 1. Code snippet of the Task Selection Interface for the SimDynamics application, where the simulate method is selected as a task. The @Method annotation specifies the class that implements simulate, and the @Parameter annotation contains parameter-related metadata (type, direction).

4 Testbed Infrastructure

The SimDynamics application was executed with COMPSs on real large-scale scientific grids. The whole infrastructure used in the tests is depicted in Figure 2, and it includes three grids: the Open Science Grid, Ibergrid and a small grid owned by the Barcelona Supercomputing Center [1].

Such infrastructure represents an heterogeneous testbed, comprised by three grids belonging to different administrative domains and managed by different middleware. The next subsections briefly describe the topology of these grids and explain how the COMPSs runtime was able to hide the complexity of their heterogeneity, keeping the Grid-related details transparent to the application.

4.1 Grids

Open Science Grid Each of the Open Science Grid (OSG) [8] sites is configured to deploy a set of Grid services, like user authorisation, job submission and storage management. Basically, a site is organised in a *Compute Element* (CE), running in a front-end node known as the *gatekeeper*, plus several *worker nodes* (or execution nodes). The CE allows users to run jobs on a site by means of the Globus GRAM

(Grid Resource Allocation Manager) [14] interface; at the back-end of this GRAM gatekeeper, each site features one or more local batch systems - like Condor [23] or PBS [7] - that process a queue of jobs and schedule them on the worker nodes. Besides, the standard CE installation includes a GridFTP server; typically, the files uploaded to this server are accessible from all the nodes of the site via a distributed file system like NFS (Network File System [5]).

Ibergrid Similarly to OSG, the Ibergrid infrastructure [10, 4] is composed by different sites, each one with a gatekeeper node interfacing to the cluster, a local resource management system (batch) and a set of worker nodes. However, in Ibergrid the middleware installed is gLite [15] and job management is a bit different: instead of submitting the jobs to a given CE directly, the user proceeds by interacting with a *Workload Management Server* (WMS), which acts as a meta-scheduling server. Therefore, matchmaking is performed at a higher level: the WMS interrogates the Information Supermarket (an internal cache of information) to determine the status of computational and storage resources, and the File Catalogue to find the location of any required input files; based on that information, the WMS selects a CE where to execute the job.

BSC Grid The BSC Grid is a cluster located in the BSC premises and formed by five nodes. Three of them have a single-core processor at 3.60GHz, 1 GB of RAM and 60 GB of storage. The other two have a quad-core processor at 2.50GHz each core, 4 GB of RAM and 260 GB of storage. The BSC Grid supports interactive execution: the user can connect to any of the nodes separately via SSH and launch computations on them. Moreover, files can be transferred to/from the local disk of each node through SSH as well.

4.2 Configuration and Operation Details

In order to run the SimDynamics application in the described testbed, the testing environment was configured as shown in Figure 2.

The access point to the Grid was a laptop equipped with a dual-core 2.8 GHz processor and 8 GB RAM. This machine hosted the main program of the application, and therefore it had the COMPSs runtime and the JavaGAT library and adaptors installed. In addition, prior to the execution, the credentials for each grid were obtained and installed as well.

Concerning the Grid middleware, the points below list the GAT adaptors and the corresponding grids where they were used:

- *Globus GRAM and OSG*: a total of six OSG sites that support our virtual organisation (VO), Engage, were used in the tests, each with its own CE. The gatekeeper of every CE was contacted by means of the Globus GRAM adaptor, used for task submission and monitoring in OSG.
- *gLite and Ibergrid*: the gLite adaptor was used to submit and monitor tasks by connecting to an Ibergrid WMS, which is in charge of selecting the execution site in Ibergrid. Among all the WMS at the disposal of our VO (ICT), the one with most availability was chosen.

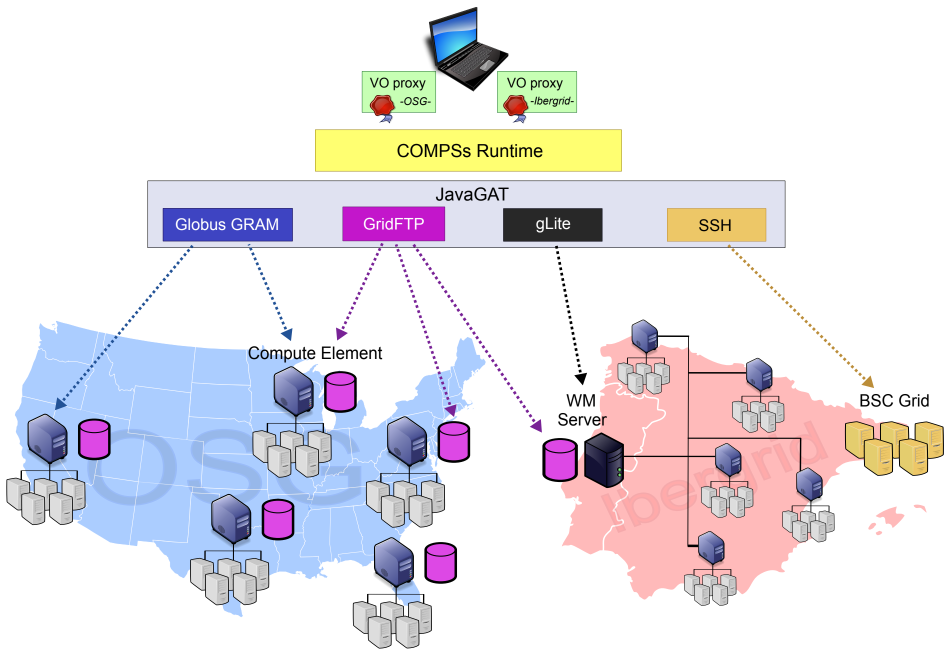

Fig. 2. Testbed comprising two large-scale scientific grids (Open Science Grid, Ibergrid) and a BSC-owned grid. The SimDynamics application, running on a machine with COMPSs, interacts with the grids through JavaGAT and its middleware adaptors.

- *GridFTP (OSG and Ibergrid)*: the SOG CEs and the Ibergrid WMS offer each a GridFTP server. The GAT GridFTP adaptor was used to transfer files to those servers during execution.
- *SSH and BSC Grid*: two nodes of BSC Grid were used in the tests, being accessed through the GAT SSH adaptors for task submission and file transfer.

Before execution, there was a previous phase of deployment where some required files were installed in the grids: the worker runtime and the classes and executables of the application tasks. In OSG, the files to be deployed were copied to the GridFTP server of each CE, so they could be accessed from the worker nodes. In Ibergrid, the files were transferred to the GridFTP server of the WMS, since the final execution site is not known in advance in this scenario; each time a job is created in Ibergrid, those files are copied by the worker runtime from the GridFTP server to the site where the job will run. Finally, in BSC Grid the files were placed in the local disk of the nodes.

At execution time, the master runtime of COMPSs sends the SimDynamics tasks and transfers files to the three grids by means of JavaGAT. In OSG, the input files of each task are first pre-staged to the GridFTP server of the target CE, thus being accessible through the NFS server of that CE too; after that, when the job is created in the CE to execute the task, the worker runtime copies the input files from NFS to the local disk of the target worker node; similarly, the output files are copied from local to NFS at the end of the task, thus being available in the GridFTP server as well. In Ibergrid, the task input files are transferred to the GridFTP server of the WMS; the pre and post-staging of those files to/from the final worker node is taken care by gLite: the WMS chooses the execution site, sends the job to the head node of that site, then the task is locally scheduled and the input files are copied from the GridFTP server to the local disk of the worker node (the process is inverse for the output files). Lastly, the BSC Grid scenario is simpler since the files can be directly transferred to/from the local disk of the final execution node.

When scheduling tasks on the grids, the COMPSs runtime takes into account locality: a task will be assigned, if possible, to a resource that already possesses one or more of the task's input files (in its GridFTP server or local disk). Whenever a resource is freed (a task finishes), the scheduler chooses the task with the best score among the pending ones, the score being the number of task input files in the resource. Note that Ibergrid counts as a single entity for locality, because the final destination of the job is not decided by COMPSs. If some input file is missing in the chosen resource, such file is replicated to that resource. If the source and destination resources share the same credentials (e.g. two OSG sites) such transfer happens directly between them; otherwise, the file is first copied to the laptop and then to the destination resource.

5 Evaluation

This section presents the results of executing the SimDynamics application (Section 3) in the described testbed (Section 4). These tests will show how the tasks of an e-Science application are executed in three different grids with COMPSs.

From the point of view of the application, *all the Grid management* discussed in Section 4 *is transparent*. The application deals with its parameters, like number of structures and coefficients. For these experiments, 27 different configurations were considered in the parameter sweeping. This leads to a total of 586 tasks, including 270 simulation tasks - the most computationally-intensive with about two minutes of execution time each. The rest of the tasks are lightweight, with a duration of less than 10 seconds.

Figure 3(a) shows how tasks were distributed among the three grids during an execution of SimDynamics with COMPSs. The six OSG resources were the ones that consumed more tasks; indeed, among all the OSG sites that support our VO, the ones with most availability were chosen. The two BSC Grid nodes also executed a considerable number of tasks because they are directly accessible and therefore those tasks did not suffer from queue waiting times. Ibergrid received less load because of three factors. First, the Ibergrid queue times in these tests were high, which caused tasks scheduled in Ibergrid to wait. Second, regarding the internal scheduling policies of the Ibergrid sites, several sites offer to our VO only opportunistic access to their resources; some other sites reserve a certain number of slots with priority but they are shared by all the Ibergrid VOs. Finally, the

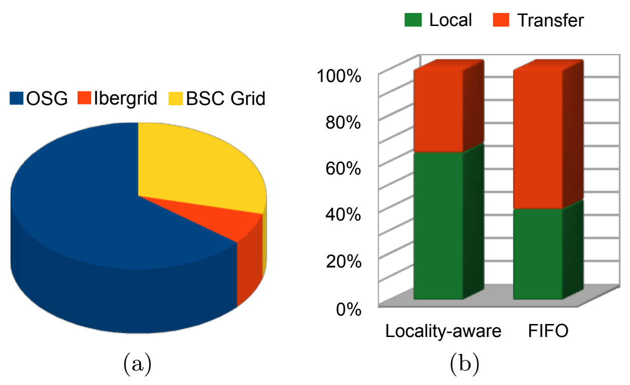

Fig. 3. Test results for the SimDynamics application when run with COMPSs in the Grid testbed: (a) distribution of the SimDynamics tasks among the three grids; (b) comparison of percentage of transfers between the locality-aware and FIFO scheduling algorithms.

Table 1. Task submission and file transfer statistics for SimDynamics.

		# Task sub.		**# File tra.**	
Grid	**Resource**	*OK*	*Failed*	*OK*	*Failed*
OSG	brgw1.renci.org	72	4	102	1
	gridgk01.racf.bnl.gov	43	0	70	1
	rossmann-osg.rcac.purdue.edu	57	14	89	11
	smufarm.physics.smu.edu	69	1	92	1
	stargrid02.rcf.bnl.gov	55	0	90	1
	u2-grid.ccr.buffalo.edu	62	1	96	0
	TOTAL	358	20	539	15
Ibergrid	wms01.ific.uv.es	33	209	58	0
	TOTAL	33	209	58	0
BSC Grid	bscgrid05.bsc.es	122	0	116	0
	bscgrid06.bsc.es	73	0	79	0
	TOTAL	195	0	195	0
	TOTAL	586	229	792	15

errors when submitting tasks to the WMS were quite frequent, which made tasks go through a (sometimes long) resubmission process.

In that sense, Table 1 contains the statistics of errors in task submissions and file transfers for the different grids and a particularly faulty execution of SimDynamics, in order to demonstrate the fault tolerance mechanisms of the COMPSs runtime. In general, the OSG sites presented only occasional failures in task submissions and file transfers, which were easily solved with resubmissions and retransfers with no need for task rescheduling. On the contrary, the errors when connecting to the Ibergrid WMS were common, possibly because of a bug in the JavaGAT gLite adaptor or because of the WMS itself; in order to face that issue,

```
public interface SimDynamicsItf {
    @Constraints(operatingSystem = "Scientific Linux")
    @Method(...)
    void genReceptorLigand(...);

    @Constraints(appSoftware = "DISCRETE")
    @Method(...)
    void simulate(...);

    @Constraints(memory = 4)
    @Method(...)
    void evaluate(...);

    ...
}
```

Fig. 4. Detail of the task constraint specification in the TSI of SimDynamics.

several retries were attempted when necessary for a task (6 per task on average), progressively increasing the time between two resubmissions. The most reliable combination of grid/adaptor was BSC Grid/SSH, for which no errors of any kind were registered.

Regarding data locality, Figure 3(b) illustrates the benefits of using a locality-aware task scheduling algorithm. Such algorithm is especially important in a highly-distributed testbed like the one in Figure 2, where data transfers are costly. Figure 3(b) compares two executions of SimDynamics, one using locality-aware scheduling and another one applying a FIFO (First In First Out) strategy, and it shows the percentage of transfers actually performed versus the percentage of locality (the transfer was not necessary because the input file was already on the target execution resource), the total being the number of input files of all tasks. The locality-aware algorithm achieved remarkable results, preventing almost 2 out of every 3 transfers.

A final series of tests intended to demonstrate how to use constraints to force the scheduling of tasks on certain resources, in case those tasks have some hardware/software requirements. Let us assume that each kind of task in SimDynamics has some resource requirements; Figure 4 shows how they can be specified in COMPSs by means of the @Constraints annotation, at method level, in the TSI. In this example, genReceptorLigand must be executed in nodes running Scientific Linux, which is the operating system installed in Ibergrid. Second, simulate is supposed to run in resources where the DISCRETE software is present; here, such capability was assigned only to OSG sites. Finally, evaluate has a hardware constraint attached - more precisely, the amount of physical memory - which was only known and specified in the resources file for the BSC Grid nodes. The three other kinds of task not shown in Figure 4 have analogous constraints.

As a result of the constraints, at execution time the scheduling of tasks on resources was the one depicted in Figure 5. This graph is a smaller version (only 8 configurations) just for illustration purposes. In conclusion, the programmer can use task constraints to make sure that a given group of tasks will be executed in one or more resources that conform to a set of requirements.

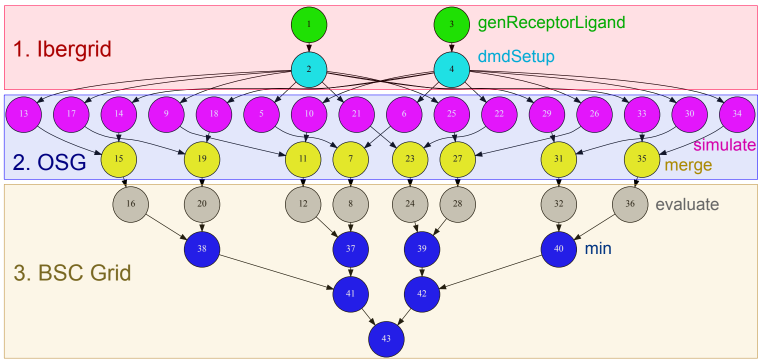

Fig. 5. Reduced version of the SimDynamics graph (the real one contains 586 tasks). The constraints in Figure 4 lead to the task scheduling on the grids represented by this figure.

6 Related Work

Apart from COMPSs, there exist other programming models for Grid applications. Ninf-G [18] offers a programming model where client programs can call libraries on remote resources using a client API that is built on top of the Globus Toolkit. Ninf-G's model is more complex than COMPSs', since the programmer has to substantially modify the original application code by including the invocations to the GridRPC API. Furthermore, COMPSs can submit tasks using different kinds of Grid middleware. Satin [24] permits to express divide-and-conquer parallelism in Java applications, marking method invocations for asynchronous spawning. Nevertheless, the programmer must explicitly use a synchronisation primitive to wait for the spawned tasks; unlike Satin, COMPSs takes care of task and data synchronisation automatically, and it is not restricted to the divide-and-conquer paradigm. OpenWP [12] is a Grid programming and runtime environment with a set of directives that have to be included in the application code to express parallelism and distribution. The main difference between COMPSs and OpenWP is that the latter requires to indicate the dependencies between tasks in the application code, whereas the former finds them automatically at execution time.

With respect to workflow managers, some systems have been proposed to specify the elements of a workflow and the connections between them, either graphically or by means of a high-level workflow description language; in this sense they differ from COMPSs, where the workflow graph is implicitly defined by a concrete execution of an application and built automatically and dynamically at runtime. Taverna [17] is a well-known graphical tool for designing and executing Grid workflows. A Taverna workflow is specified by a directed acyclic graph where nodes represent software components. Each edge in the graph denotes a data dependency from an output port of the source node to an input port of the destination node. The nodes of a Taverna workflow can be computations executed in the Grid and also Web Services, similarly to COMPSs. Triana [19] also permits to describe applications by dragging and dropping their components and connecting them together to build a workflow graph; like in COMPSs, Triana workflows can access the Grid through JavaGAT. Pegasus [13] is a workflow management system that takes high-level workflow descriptions and automatically maps them to Grid resources; Pegasus performs execution site selection, manages the input data and provides directives for data transfer and registration.

7 Conclusions and Future Work

This paper has shown how an e-Science application can be easily developed with the COMPSs programming model and then parallelised in heterogeneous grids with the COMPSs runtime. Such application is programmed sequentially, while the user is only responsible for specifying its tasks. No API call or resource management details appears in the application, so that it is portable and agnostic of the underlying infrastructure. All the burden of parallelisation and infrastructure management is left to the COMPSs runtime; this paper has demonstrated how this runtime can deal with grids managed by different middleware, making them interoperable while keeping the application unaware of Grid details.

The future work includes supporting the use of logical files in COMPSs executions, possibly by creating a JavaGAT adaptor that manages them; such files are referenced with logical names that can be associated to several physical locations. Furthermore, we plan to extend the locality-aware algorithm to take into account not only the number of input files but also their size when deciding the target resource of a task.

Acknowledgements

This work has been supported by the following institutions: the Universitat Politècnica de Catalunya with a UPC Recerca predoctoral grant; the projects of Computación de Altas Prestaciones V and VI (TIN2007-60625, TIN2012-34557); the Spanish Government with grant SEV-2011-00067 of Severo Ochoa Program. On the other hand, the Ibergrid and the Open Science Grid organisations have granted us access to their infrastructures.

References

1. Barcelona Supercomputing Center. http://www.bsc.es.
2. Cloud interoperability and portability remain science fiction. http://searchcloudcomputing.techtarget.com/feature/Cloud-interoperability-and-portability-remain-science-fiction.
3. Grid Interoperation Now Community Group (GIN-CG). http://www.ogf.org/gf/group_info/view.php?group=gin-cg.
4. Iniciativa Nacional Grid. http://www.gridcomputing.pt.
5. Network File System. http://www.ietf.org/rfc/rfc3010.

6. Open Grid Forum. http://www.gridforum.org/.
7. Open Portable Batch System. http://www.openpbs.org/.
8. Open Science Grid. http://www.opensciencegrid.org.
9. ScalaLife Pilot Applications - DISCRETE. http://www.scalalife.eu/applications.
10. Spanish National Grid Initiative. http://www.es-ngi.es/.
11. G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schütt, E. Seidel, and B. Ullmer. The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. In *Proceedings of the IEEE*, volume 93, pages 534–550, 2005.
12. M. Cargnelli, G. Alleon, and F. Cappello. OpenWP: Combining annotation language and workflow environments for porting existing applications on grids. In *Proceedings of the 2008 9th IEEE/ACM International Conference on Grid Computing*, GRID '08, pages 176–183, Washington, DC, USA, 2008. IEEE Computer Society.
13. E. Deelman, G. Singh, M. hui Su, J. Blythe, A. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal*, 13:219–237, 2005.
14. I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Int. Journal of Supercomputer Applications*, 11(2):115–128, 1997.
15. E. Laure, C. Gr, S. Fisher, A. Frohner, P. Kunszt, A. Krenek, O. Mulmo, F. Pacini, F. Prelz, J. White, M. Barroso, P. Buncic, R. Byrom, L. Cornwall, M. Craig, A. D. Meglio, A. Djaoui, F. Giacomini, J. Hahkala, F. Hemmer, S. Hicks, A. Edlund, A. Maraschini, R. Middleton, M. Sgaravatto, M. Steenbakkers, J. Walk, and A. Wilson. Programming the Grid with gLite. In *Computational Methods in Science and Technology*, page 2006, 2006.
16. P. E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org, Corvallis, OR, USA, 2012. Available: http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html.
17. P. Missier, S. Soiland-Reyes, S. Owen, W. Tan, A. Nenadic, I. Dunlop, A. Williams, T. Oinn, and C. Goble. Taverna, reloaded. In M. Gertz, T. Hey, and B. Ludaescher, editors, *SSDBM 2010*, Heidelberg, Germany, June 2010.
18. Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *Journal of Grid Computing*, 1(1):41–51, 2003.
19. I. Taylor, M. Shields, I. Wang, and A. Harrison. Visual Grid Workflow in Triana. *Journal of Grid Computing*, 3(3-4):153–169, September 2005.
20. E. Tejedor and R. M. Badia. COMP Superscalar: Bringing GRID Superscalar and GCM Together. In *Eighth IEEE International Symposium on Cluster Computing and the Grid*, CCGrid '08, Lyon, France, pages 185–193, May 2008.
21. E. Tejedor, J. Ejarque, F. Lordan, R. Rafanell, J. Álvarez, D. Lezzi, R. Sirvent, and R. M. Badia. A Cloud-unaware Programming Model for Easy Development of Composite Services. In *3rd IEEE International Conference on Cloud Computing Technology and Science*, CloudCom '11, Athens, Greece, November 2011.
22. E. Tejedor, M. Farreras, D. Grove, R. M. Badia, G. Almasi, and J. Labarta. A high-productivity task-based programming model for clusters. *Concurrency and Computation: Practice and Experience*, 24(18):2421–2448, 2012.
23. D. Thain, T. Tannenbaum, and M. Livny. Condor and the Grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002.
24. R. V. van Nieuwpoort, G. Wrzesińska, C. J. Jacobs, and H. E. Bal. Satin: A high-level and efficient grid programming model. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(3):1–39, 2010.