Replica management in GRID superscalar

Jesús Malo Poyatos, Jonathan Martí Fraiz, Toni Cortes*

Storage-system Research Group

Barcelona Supercomputing Center

{jesus.malo,jonathan.marti,toni.cortes}@bsc.es

*Universitat Politècnica de Catalunya

Abstract

File replication is one of the best known techniques used to improve the performance of file access in the GRID. Replication allows applications to reduce their execution time because jobs can profit locality and greater availability of data.

We will describe how a transparent replica management mechanism has been implemented into GRID superscalar, a workflow tool for grid enabling applications. This development adds to GRID superscalar the basic infrastructure for future replica management researches.

We will also show, as expected, that profiting every unavoidable data transfers and avoiding unnecessary transfers of output files with replicas are efficient ways to improve performance of grid applications. Both improvements have been achieved without transparency-lost to users and applications.

1 Introduction

GRID computing is becoming a suitable environment to run large applications that normally have high I/O needs. Many solutions have been proposed to allow these applications to access data that is not located in the same nodes where they are running. One of the most well used techniques is to copy the needed files to the node where the application will be executed. This allows applications to access files at local speed.

In order to take advantage of these time-consuming copies, many systems keep track of these replicas and do not remove them hoping that future executions that need these files will run in the locations where the files are replicated [9]. As this is a common solution in GRID execution environments, we planned to add this feature into GRID superscalar [8]. Actually most current system take partial approaches that work on given situations but are not general or transparent enough.

1.1 Replica access nowadays

Current GRID execution environments and work-flow tools already handle file replication [5]. The idea is that somehow, and this is not relevant at this time, the system maintains a set of replicas of a file at different hosts. When the application wants to access a file, the system queries a service (such as SRB [1] or RLS [4]) about the location of the replicas for the file. This query is done using an ID that should identify the file uniquely. With the information of all available replicas, the system decides which one to use, or whether a new replica is needed. The main problem found in this process consists on how to uniquely identify files.

The most common solution proposed is to create some kind of unique ID per file, often called logical file name (LFN). LFNs are included in the application code or in a job description file, thus making this replica mechanism not transparent neither to applications [7] nor to users. The application needs to know LFNs in order to query the replica catalog for getting real locations of files and users have to specify what files are in the replica catalog or in local file systems. Real locations are often known as physical file names (PFN). Therefore, replica catalogs are specialized databases storing metadata and mappings of LFNs and PFNs [3].

2 Related work

Usually grid work-flow tools relay the responsibility of assigning LFNs to users. Reptor [7] is the basic component of projects using replication and follows this pattern. It is not really a work-flow tool but a replica metadata service. It automatically assigns a globally unique identifier (GUID) for replica identification but makes users assign aliases, which must be unique in the whole system. Associations among users, LFNs and GUIDs are stored in the Replica Metadata Catalog. On the contrary, GRID superscalar allows users to assign LFNs like they typically do with local files, so the level of achieved transparency is greater.

Lightweight Data Replicator (LDR), shown in [10], is a data replication tool, which uses the Globus RLS [4] for replica location and performs the generation of new replicas where they are required. However, LDR does not handle any consistency among replicas. GRID superscalar is able to keep a strong consistency among replicas and job scheduling based on replica locality, whereas LDR does not have those features.

Pegasus [5] is probably the work-flow tool with characteristics most similar to GRID superscalar ones. Pegasus registers every intermediate data generated in a replica catalog service, being able to reuse file transfers. However, it lacks data access transparency because it requires job description files. Users indicate in these descriptions which files are local ones and which must be searched in the replica catalog, because they are LFNs. On the contrary, the developed replica management mechanism for GRID superscalar is totally transparent to users and grid applications programmers.

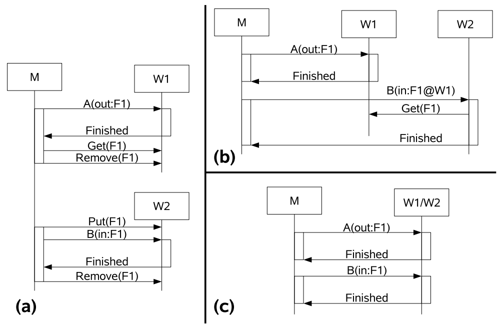

Figure 1: Original approach and new ones

3 Motivation

3.1 Avoiding unnecessary transfers

Grid applications usually store their results in files after execution. These output files are temporally stored in remote nodes until the execution of the job has finished. At this point files are transferred to user's host, where the execution started, and they are deleted, freeing their storage space in the remote host. This is the usual behavior of applications in the GRID, performing data moving to local disks once the execution has finished.

Although traditional approach is totally valid, when a grid application requires a previously generated file as input, it is adding an avoidable execution time increase. Let's suppose an application A which generates file F1 and another application B which reads file F1, as shown in figure 1(a). Application A is submitted to node W1 from the master node M. Afterwards, application B is also submitted to node W2 from node M.

- If W1 is a different node to W2 (figure 1(b)), one transfer could be avoided, because F1 would be transfer to W2 from W1 directly, instead of the two required transfers of the traditional behavior (one from W1 to M and a second one from M to W2).

- If W1 is the same node than W2 (figure 1(c)), transfers of F1 will not be required,

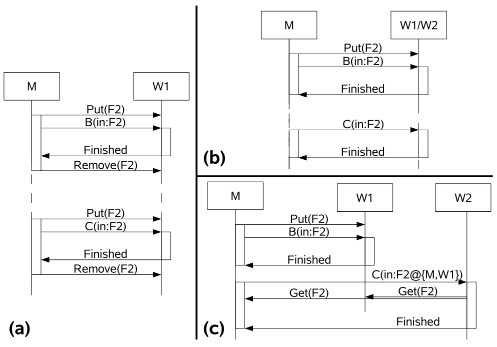

Figure 2: Profiting unavoidable transfers

because F1 will be in the right place for execution of B.

Avoiding this kind of transfers allows to reduce execution time of jobs in an efficient way for applications which require previously generated files.

3.2 Taking advantage of unavoidable transfers

During the execution of applications in grids, some transfers are always required. This kind of transfers happen when hosts storing the data are busy, but some computations about these data are required. In this case, jobs are submitted to free nodes and transfers are performed to them. Files resulting from these transfers are usually deleted once the job which needed them finished. However, if storage space is not a problematic issue, this behavior can be improved taking advantage of those unavoidable transfers.

Every transfer can be well-spent if transferred files are handled as replicas. In this case, when a file is transferred to a host, it must be added to the list of existing replicas of that file, adding a new entry in the replica catalog. This transfer will be useful if, at least, two applications need the same input file. Taking account of this new replica, new transfers of the same input file can be avoided. Moreover, the existence of several replicas of a file can be used for performing parallel transfers when other new replica is required. Both advantages are shown in figure 2(b) and figure 2(c), respectively, whereas figure 2(a) shows the traditional approach.

3.3 How replica access should be

From our point of view, LFNs should be something completely transparent to users and applications. Our point of view is that developers should write applications as if they were going to be executed in a single host, but they should still run on the GRID. In this scenario, programmers only know about file names and directories, not about some other kind of IDs that identify the file globally in the whole GRID.

Regarding how applications become GRID executable, we rely on GRID superscalar [8], which is a work-flow tool that is able to convert sequential applications into GRID ones. Nevertheless, it is important to notice that the discussion presented in this paper, although implemented as part of GRID superscalar, can also be applied to any other GRID execution environment or work-flow tool.

On the file system side, the idea is that the files seen by applications, while running in the GRID, should be the same as the ones seen in the host launching the application. This means that workflow managers should be able to translate names from this file system view to LFNs without the interaction of users or programmers.

4 Adding replica management to GRID superscalar

4.1 GRID superscalar

GRID superscalar is a new programming paradigm for GRID enabling applications, composed of an interface and a run-time. With GRID superscalar, a sequential application composed of tasks of a certain granularity is automatically converted into a parallel application where the tasks are executed in different servers of a computational GRID.

The behavior of the application when run on GRID superscalar is the following: for each task candidate to be run in the GRID, the GRID superscalar run-time inserts a node in a task graph. Then, the GRID superscalar run-time system seeks for data dependencies among the different tasks of the graph. If a task does not have any dependency with previous tasks which have not been finished yet or which are still running, it can be submitted for execution to the GRID. In that case, the GRID superscalar run-time requests a GRID server to the broker and submits the task.

Those tasks that do not have any data dependency between them can be run on parallel on the grid. This process is automatically controlled by the GRID superscalar runtime, without any additional effort for the user. GRID superscalar is notified when a task finishes. Afterwards, structures are updated and any task than now have its data dependencies resolved, can be submitted for execution.

4.2 Transparent replica management

Because the characteristics and ease of use of GRID superscalar, every new improvement must keep the same level of abstraction. GRID superscalar allows people to perform distributed computing without having to deal with complex paradigms. Because this reason, taking advantage of replicas must be as transparent as possible to application programmers and users.

When an application using GRID superscalar is running, every open operation of a file is processed by the run-time. File names are searched across the replica catalog and if the file is a replicated one, its replicas are taken into account. After this search, internal components, such as scheduler or transfer mechanism, are able to decide where new jobs will be placed based on locality policies and where replicas can be retrieved from.

Once the execution of a job generates a new replica of a file, this is added to the corresponding replica catalog entry and, from now on, it will be available for every new job, indeed after the execution finishes. Every output file will generate a remote file available for every posterior execution of applications.

As every replica management system, the writing access to a replicated file adds consistency troubles among replicas [6]. In our current implementation we have adopted a strong consistency approach: when a task writes in a replica, the remaining replicas are invalidated. At this point, the metadata contained in the related entry is updated and the invalidated ones are removed from it. This invalidation is also stored in the structures of the GRID superscalar runtime that will erase useless replicas once the execution of the application finishes.

Users do not care about replicas because access is totally transparent to them. A set of developed tools allows users to list, read and remove replicated files as if they were local files. They can also get a local file and convert the replicated file into a local one, removing every replicated data of the file.

4.3 Replica Catalog

Instead of using a replica catalog like Globus RLS [4] or P-RLS [2], based on Giggle framework [3], we have developed a lighter approach. Metadata information relative to a replicated file is contained in a file, shadow file, in the file system where the original file was generated. This approach satisfies our current replica requirements. It has also been developed having in mind the ease to change it for using others catalogs, if needed.

Shadow files contain information about the size of file content and replica information. This one stores the identification of hosts where replicas are located, access paths to them, and some performance metadata used for further optimizations. In order to reduce the visual effect of these shadow files, we have grouped all of them in a shadow directory. Each directory has its shadow directory (as one of its children) and contains the shadow files of all files in it.

Access to metadata is as fast as accessing to local files (shadow files are local files containing metadata), so the performance penalty should not be significant compared to file accesses.

4.4 Changes to GRID superscalar

In order to incorporate the replica management mechanism to GRID superscalar, we have to change some components of it and deal with some issues of its internal design.

As we have introduced previously, we modified the component which handles the jobs submissions. The changes added a query to the replica catalog for every file used by a job. From this query, the real locations of replicas are got and the job scheduler is warned of them, being able to take file locality into account. This change also modifies the *stage-in* job submission parameters, since files can already exist in the worker where job will be submitted.

To avoid the deletion of output files we changed the component responsible of run-time finalization. This component performs the transfers of output files to the local file systems, removes every generated or transferred data to workers and cleans up all the auxiliary files generated. We changed its functionality to only cleaning up auxiliary files, removing every *stage-out* and the corresponding *clean-up* parameters of the cleaning job.

5 Evaluation

5.1 Environment

For our experiments we used a grid of three hosts, each one with two Intel Pentium 4 CPUs at 3.6GHz with 1 GB of RAM. All the machines were running OpenSUSE 10.1 with kernel 2.6.16.13-4-smp and GLOBUS toolkit 4.0.3. The network connecting these machines was a switched 100Mb Ethernet network.

5.2 Synthetic test

In order to evaluate the maximum benefit that you can achieve with replicated files we have run a simple writer/reader test in the previously mentioned grid environment. The test is composed by two synthetic applications:

- A writer which writes a file as big as the user specifies.

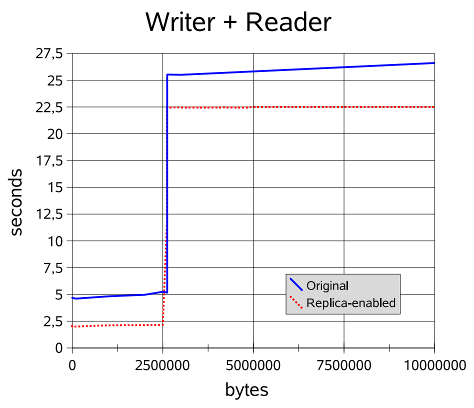

Figure 3: Execution of writer followed by reader

- A reader which reads the file previously generated by the writer.

Both applications were run with the replica-enabled version of GRID superscalar and the original one. Figure 3 summarizes the results of the experiments. It shows the execution time of both applications with increasing file sizes, from 1 byte to 10 megabytes.

As you can see, there are two different behaviors in the results although the replica-enabled version is always faster than the original one. The change of behaviors happen when data size is 2.5 MB. This is due to the buffer cache of the worker machine running this application. We determined exactly that it is 2.5 MB.

On one hand, with data sizes smaller than 2.5 MB both versions achieve a nearly constant execution time. This is due to data size is small enough to be stored in the buffer cache. Since the replica-enabled version does not perform any call to GLOBUS for retrieving the output file and sending input ones, is faster than the original version, which does it. So the improvement is due to reduction of unnecessary calls.

On the other hand, data sizes greater than 2.5 MB can not be stored into the buffer cache assigned to the application and execution time scales linearly with the size. These sizes show the benefits of leaving output files in the generator machine. The replica-enabled version has a lower execution time because there are no transfers of data whereas in the original version execution time is increased with the data transfer time.

5.3 Matrix operation test

In order to evaluate the impact of using replicas in applications, we have run one which performs user-specified operations with matrixes based on GRID superscalar. This test is composed by a set of three different executions:

- The application is run three times. In the first execution it computes the addition of matrixes A and B, getting the matrix E as output. The second one adds matrixes C and D and matrix F is the result. Finally the third execution computes the addition of E and F, showing how the use of remote files affect the global execution time.
- The application is run with the four previously used matrixes (A, B, C and D) and computes the addition of them. It produced the same final result than the previous test, but in this case, input matrixes had some replicas and execution time was reduced because some of them were stored in hosts where computation was submitted. In this case, the application generates jobs of pair of matrixes, so it finally generated three tasks, as the previous test, but in the same execution, whereas the previous test had been in three different ones.
- The application is run with different levels of replication in its input files. It computes the addition of the four matrixes A, B, C and D too, but, in this case, these matrixes are not replicated, replicated only in a worker machine or totally replicated in both possible workers. This test shows how both the number of replicas and their placement can have influence on the execution time.

Results of the first test are summarized in figure 4. As happened with the synthetic test,

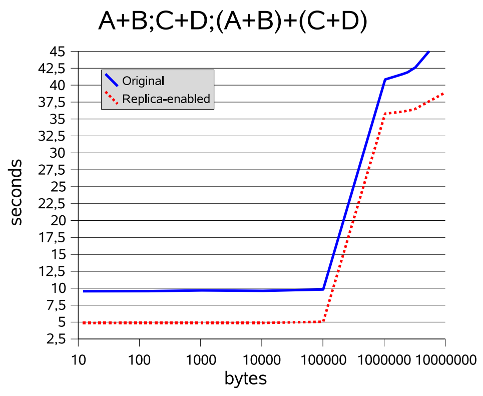

Figure 4: Reuse of output generated replicas

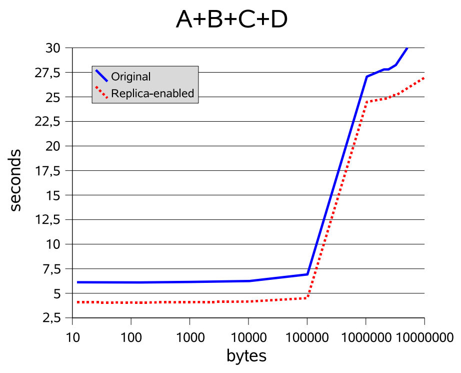

Figure 5: Reuse of input generated replicas

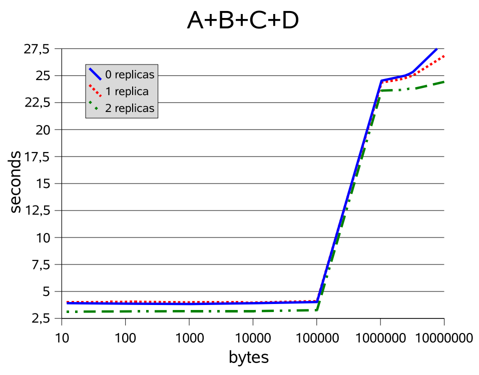

Figure 6: Influence of the number of replicas

the replica-enabled version of GRID superscalar is faster than the original one. Results are pretty similar, getting a close to constant execution time for matrix sizes below the size of the buffer cache of worker machines, and a linearly scaled execution time for bigger sizes. As we remarked before, this behavior is due to the assigned buffer caches of workers to the application. In this case, interval measurements were wider than in the synthetic test and the edge has been attenuated by the representation of figures 4, 5 and 6. In the case of the first test, you can also notice that differences are greater. This is due to the increase of the number of calls to Globus, for data sizes fitting inside the buffer caches, and the data transfer time added by the original version, when sizes are over the existing buffer-cache size.

Figure 5 shows the results of the second test. In this case, results have a similar behavior to the previously explained test but the execution is faster. This improvement is achieved because the whole operation is performed in a execution instead of three ones. This way, overhead is reduced for both versions because the GRID superscalar run-time is started only once. Regarding the benefits of reusing replicated inputs, the replica-enabled version takes advantage of them, achieving a shorter execution time for all data sizes.

Finally, the influence of replica placement and replica selection is shown in figure 6. As you can see, with different number of replicas, execution time spent by applications can vary. Results show how using a totally replicated file is always the fastest approach, although for different environments to the used one it can be inadmissible because the required storage space. However you can also see that replication can affect negatively to execution time. This is the case where only one replica exists in one worker. This happens because the first task (A+B) is sent to the worker with the replicas but the second one is sent to a worker without any replicated data. This causes an unavoidable transfer from the first one to the second one, transfer which happens concurrently to the execution of the task in the first worker. Because this overhead, this replication scheme is not suitable in this case.

6 Future work

The work presented in this paper is just the basic infrastructure to manage replicas in GRID superscalar. From this point, we plant to optimize this replica management and to add new functionality in many different ways:

- Predict when and where a file will be used to create a replica before the application is executed and thus remove the overhead of replica creation from the execution of the application
- Implement smart mechanisms for replica deletion based on past usage, predicted usage and economic models.
- Allow partial replicas (not replicate the whole file, only the needed parts).
- Integrate the mechanisms to create replicas in advance and the cleanup of replicas with the partial replicas.
- Improve the replica mechanisms to allow streaming.

7 Conclusions

We have presented a replica management mechanism developed for GRID superscalar. This mechanism allows GRID superscalar to access remote files with transparency to users and grid applications programmers.

We have also shown that replica management is able to reduce the execution time of grid-enabled applications by means of profiting every performed transfer and avoiding deletion of potentially reusable files. Every new output file is kept in its generating host avoiding transfers of reused files. Every transfer of an input file is kept in the target machine and logged as a new replica for that input file. Influence of replicas over the execution time was shown and improvements were achieved, both for reusing input replicated files and new remote output files, with a synthetic test and a realistic application.

This development gives us the necessary infrastructure for future replica management researches.

8 Acknowledgments

We thank the development team for GRID superscalar for their support while modifying the parts needed to include the replica management presented in this paper.

This work has been partially supported the by the Spanish Ministry of Science and Technology under the TIN2004-07739-C02-01 grant.

References

[1] Chaitanya K. Baru, Reagan W. Moore, Arcot Rajasekar and Michael Wan, *The SDSC storage resource broker*, Proc. of the 1998 conference of the Centre for Advanced Studies on Collaborative Research, December 1998.

[2] Min Cai, Ann L. Chervenak and Martin Frank, *A Peer-to-Peer Replica Location Service Based on a Distributed Hash Table*, Proceedings of the 2004 ACM/IEEE conference on Supercomputing, 2004.

[3] Ann L. Chervenak, Ewa Deelman, Ian T. Foster, Leanne Guy, Wolfgang Hoschek, Adriana Iamnitchi, Carl Kesselman, Peter Z. Kunszt, Matei Ripeanu, Bob Schwartzkopf, Heinz Kurt Stockinger and Brian Tierney, *Giggle: A Framework for Constructing Scalable Replica Location Services*, Proc. of the IEEE Supercomputing 2002, 2002.

[4] Ann L. Chervenak, Naveen Palavalli, Shishir Bharathi, Carl Kesselman and Robert Schwartzkopf, *Performance and Scalability of a Replica Location Service*, Proc. 13th International Symposium on High-Performance Distributed Computing, June 2004, pp 182-191.

[5] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob and Daniel S. Katz, *Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems*, Scientific Programming Vol. 13 No. 3, 2005, pp 219-237.

[6] Dirk Dullmann and Ben Segal, *Models for Replica Synchronization and Consistency in a Data Grid*, Proc. of the 10th IEEE International Symposium on High Performance Distributed Computing, 2001.

[7] Peter Z. Kunszt, Erwin Laure, Heinz Stockinger and Kurt Stockinger, *Advanced Replica Management with Reptor*, PPAM, Lecture Notes in Computer Science, 2003, pp 848-855.

[8] Raul Sirvent, Josep M. Perez, Rosa M. Badia and Jesus Labarta, *Automatic Grid workflow based on imperative programming languages*, Concurrency and Computation: Practice and experience Vol. 18 No. 10, December 2005, pp 1169-1186.

[9] S. Venugopal, R. Buyya and K. Ramamohanarao, *A Taxonomy of Data Grids for Distributed Data Sharing, Management, and Processing*, ACM Computing Surveys Vol. 38 No. 1, 2006, pp 1-53.

[10] LIGO Project, *Lightweight Data Replicator*, http://www.lsc-group.phys.uwm.edu/LDR/, 2004.