Increasing Parallelism for Workflows in the Grid

Jonathan Martí[1], Jesús Malo[1], and Toni Cortes[1,2]

[1] Barcelona Supercomputing Center
{jonathan.marti,jesus.malo,toni.cortes}@bsc.es
[2] Universitat Politècnica de Catalunya *
http://www.ac.upc.edu

Abstract. Workflow applications executed in Grid environments are not able to take advantage of all the potential parallelism they might have. This limitation in the usage of parallelism comes from the fact that when there is a producer/consumer situation communicating using files, the consumer does not start its execution till the producer has finished creating the file to be consumed, and the file has been copied to the consumer (if needed).
In this paper, we propose a publish/subscribe mechanism that allows consumers to read the file at the same time it is being produced. In addition, this mechanism is implemented in a transparent way to the application, so does not require any special feature from the local filesystems.
Finally, we show that our mechanisms can speedup applications significantly. In our best test we divided by two the execution time of some applications, but other applications may have even higher benefits.

Key words: Grid Workflow Parallelism Storage

1 Introduction

Parallel applications are one of the main targets for the Grid, since they are built by several tasks that perform work mostly independent of the other ones. Sometimes data has to flow from one task to another and some mechanisms, such as message passing or shared files, can be used to achieve this. The most clear example of Grid applications that communicate using files can be found in workflows, where a workflow engine divides the application into tasks and decides what dependencies these tasks have among them. In this environment, most of the dependencies are due to shared files, and the general case is that a given task cannot be executed till its input file has been written. This will only happen when predecessor task has finished.

The way current Grid and workflow systems handle these dependencies is very simple. Let's assume a workflow with only two tasks where *task2* depends on the output of *task1*. First *task1* is executed and once it finishes, the system decides the node where *task2* will be executed and copies the file (if needed) to

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

this node. This copy is normally done using GridFTP or a similar mechanism. Finally, *task2* is executed and can use the file produced by *task1*.

We can see that this mechanism does not exploit all potential parallelism we could achieve. If *task2* could start processing the file before it is completed, then we could start both tasks in parallel and allow the second one to process the file as it is being generated. Unfortunately, this kind of mechanisms are not available in Grid environments, and this is the problem we address.

In this paper we propose a publish/subscribe mechanism that allows a file to be generated by one task and consumed (while it is being generated) by as many tasks as needed in the Grid. This solution will be integrated in GRID superscalar [1], a well-known workflow engine.

2 Design and implementation

We have implemented a publish/subscribe middleware that allows subscribers to read remote files, while they are being generated by producers. Figure 1 shows a simple scenario where our approach is running. As you can see, the daemon on the publisher side is reading the file contents while it is being generated by *task1*. In parallel, the publisher daemon is sending these data to the subscribed daemons which write the contents to a named pipe and optionally to a temporary file, so that *task2* could read the data concurrently.

It is also important to notice that all this mechanism is completely transparent to the application and that no modification is done on the application code due to the proposed data forwarding.

Using named pipes enables an application running on the subscriber side to read the contents while they are being written. The problem in using regular files would be that appending data to a file produces the consequent *EOF*, and a typical application based on regular reads would fail trying to read the contents from that file. On the other hand, if the daemon places the named pipe where the application hopes to find the file causes the application to block till data it needs is read.

Using temporary files is optional, but on one hand it allows eventual transparency, since if a file copy is left in the subscriber side, the middleware is finally acting as a traditional file-transfer protocol; in the other hand it allows dealing with replica management mechanisms.

Certainly, our mechanism could introduce CPU overhead on nodes while monitoring the published files. This is the reason behind our decision to monitor files using a dynamic polling mechanism that enables the adaption of the event-based communication protocol to file generation rate (this is detailed later in the Implementation issues). However, our approach is designed for large applications that produce large files (e.g. hundreds of MB or GB), so the overhead introduced by monitoring files is negligible compared with benefits obtained.

Furthermore, taking into account the middleware design, it would be easy to integrate it as a Globus web-service. Doing it, would enable to start the daemons/service as soon as Globus does.

Operations provided by the daemon.

- Publish file: asks the daemon to monitor a specific local file. The daemon is then prepared to accept incoming subscriptions for that file.
- Subscribe to remote file: asks the daemon to subscribe to a file located at a remote host. The daemon generates a new named pipe to attach the incoming remote contents (i.e. incoming update events - event-based protocol is explained later). Furthermore, these incoming events may be written to a temporary file to eventually keep a replica.
- Process subscription: asks the daemon to process an incoming subscription, coming from a remote host, to a local file.
- Process event: asks the daemon to process an incoming event related to a remote file which is subscribed to.
- Worker is over: informs the daemon that worker has finished generating some published file. This is broadcast to each subscriber to that file which proceed closing the named pipe and placing the temporary file instead.

Event-based communication protocol.

Files are transferred using a communication protocol based on two types of events: *UPDATE* and *END*.

While the published file is being generated, each subscriber to this file receives *UPDATE* events coming from the daemon responsible of that file. These events contain the data corresponding to pieces of the file that each subscriber has not received yet. So that, these data is processed by subscribed daemons appending them to the associated named pipe and temporary file.

Eventually, when the remote file is closed, so the generation is over, it is produced the *END* event from remote daemon to the subscribers. When subscribers receive it, they proceed closing the named pipe and placing the temporary file instead of it, so the file is saved on subscriber too as if transferred in a classical way.

Implementation of the daemon.

Figure 2 shows the interaction between the main daemon components. The box in the middle shows these components: *EventProducer*, *EventDispatcher*, and *EventProcessor*. *EventDispatcher* and *EventProcessor* manage an event queue to dispatch and process events respectively. The daemon interacts on the one hand with remote daemons, shown with the upper boxes, and on the other hand with local files and named pipes.

The components introduced before are implemented as the following threads:

- EventProducer: thread that monitors a local file declared to be published, and generates events for each subscriber to that file. These events are enqueued at the EventDispatcher event queue.
- EventDispatcher: thread that sends the events enqueued by *FIFO* policy.
- EventProcessor: thread that process the events received from a daemon to which it is subscribed (*UPDATE* or *END*).

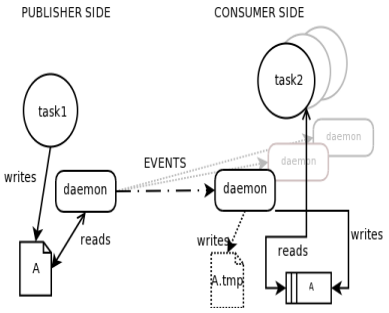

Fig. 1. Essential scenario.

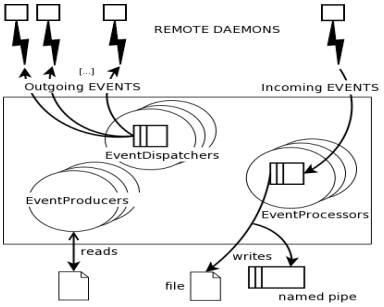

Fig. 2. Daemon components.

Adaptive file monitoring.

As we have introduced in section 2, we implemented an adaptive monitoring algorithm to reduce the CPU overhead produced by the *EventProducer*. This thread is in charge of monitoring one published file. To know whether the file has changed, so new events must be produced, *EventProducer* polls the file size periodically. Polling the file size too much often, obviously increase the CPU overhead, and even more if there are many *EventProducers* monitoring several files. So to deal with this situation, we decided to correct the polling time dynamically with a minimum of 10 ms, i.e. the minimum elapsed time between each polling request must be 10 ms. The period time increases when *EventProducer* polls the file size and it has not changed, otherwise the period time decreases.

3 Integration in GRID superscalar

To validate our approach, we integrated it with GRID superscalar [1]. GRID superscalar is a programming framework that enables the easy development of applications to be run in a computational Grid.

GRID superscalar uses a master/worker schema to map the application workflow. That is, the master is responsible to control the application course by means of delegating its corresponding tasks to several workers. Specifically, the master evaluates task dependencies in terms of the input/output data arguments they use. Then, based on this evaluation, it calculates how to parallelize the application in the Grid, i.e. how to map the workflow among the nodes.

When the master detects a dependency between two tasks *task1(out fileA)* and *task2(in fileA)*, it solves it by executing first *task1*, then transferring *fileA* to worker responsible of *task2* and eventually executing *task2*.

Here it is where our approach comes into play. The idea is that using our middleware enables the master to delegate the execution of *task2* while *fileA* is being generated by *task1*. Figure 3 shows an abstract comparison between both behaviours in terms of wasted time. With the original GRID superscalar, tasks with data dependencies are executed consecutively, whereas using our middleware they could be executed concurrently.

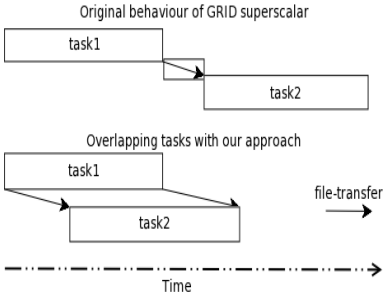

Fig. 3. Task overlapping.

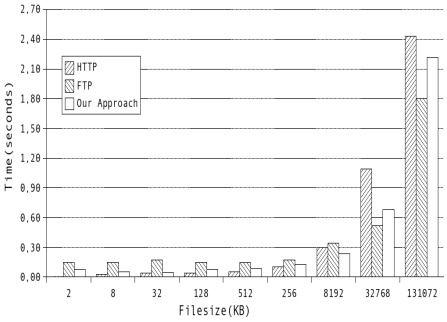

Fig. 4. FTP vs HTTP vs Our protocol

4 Performance evaluation

Case 1: The protocol.

First of all, we compared our communication protocol based on events versus HTTP and FTP. We did it in terms of file-transfer bandwidth, since we wanted to test if the performance of our approach is acceptable according to protocols specifically designed for point-to-point file-transfer.

Our middleware is actually designed for support N to M file content distribution, but as it must be implemented at application level since we are working in Grid environments, you can see our middleware as a derivative from several point-to-point file-transfers. Therefore, doing this test we compare the performance between the point-to-point basis of our protocol versus the existing ones.

The environment where we performed the test consists of two Pentium M 1.73Gh, with 1GB of memory, and Gigabit Ethernet cards.

To obtain the HTTP results, we developed a shell-script based on the *wget* command to transfer files from 1KB to 128MB from one laptop to the other. To obtain the FTP results we developed a shell-script based on *ftpupload* to transfer the same files. And finally to obtain the results about our approach we subscribed the daemon from one of the laptops to the other and transferred files already generated using our event-based communication protocol. Furthermore, in order to be on equals terms as far as possible, we disabled the temporary file creation for our approach, which obviously carries out an overhead while it supposes two writes per actual write.

To deal with interferences in our LAN, we executed the tests several times, but actually we did not observed any significant difference among executions.

Figure 4 shows the results obtained, from what we conclude that on one hand FTP is between 30-40% faster than the others for large files, what is expected due to it is designed for point-to-point communications and optimized for transferring large files. However, our approach is close to HTTP, which is a good result considering that *wget* is designed for point-to-point communications, although it is slower than FTP because it is not optimized for transferring large files as FTP does.

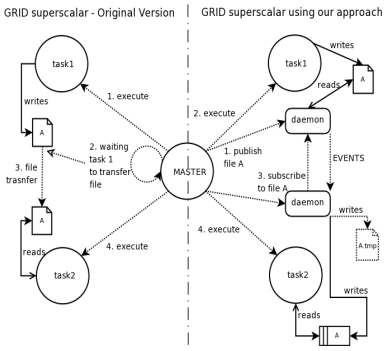

Fig. 5. GRID superscalar integration.

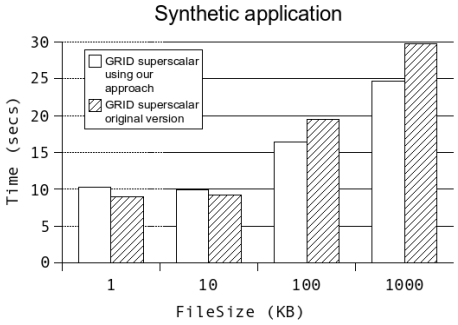

Fig. 6. Synthetic case results.

As a global conclusion,our approach is acceptable for file-transfer communications in terms of bandwidth because taking account of our approach allows N to M communications among daemons, as a publish/subscribe approach, it carries out an inevitable overhead, but even so we obtained results comparable to well-known protocols that do not deal with goals like our approach does.

Case 2: Synthetic load.

We checked the benefits of our approach integrating it with GRID superscalar. Figure 5 shows the comparison between GRID superscalar using our approach, and the original version. Depending on the version used, the behaviour for running the basic scenario of two data dependent tasks is different. In terms of time, we already introduced the difference in figure 3 commented before.

We have checked what happened when GRID superscalar manages an application based on two functions *task1(out fileA)* as the producer and *task2(in fileA)* as the consumer. So *task1* is executed as a worker in one of the nodes, and *task2* is executed as a worker on the other node. These nodes are: Khafre (PIII 700MHz,2.5GB memory) Khepri (Power4 1000MHz, 2GB memory).

We have executed one hundred tests for this experiment to deal with possible interferences with another users either in *Khafre* or *Khepri*. For each test, the *fileA* used in the application varies from 1KB to 10MB. We decided to stop at 10MB because in fact, the benefits appear since file is larger than 100KB and they could be obviously extrapolated to larger applications which generate larger files. Figure 6 shows the results obtained.

Case 3.1: Matrix multiply.

The application called *MatMul* is based on matrix multiplications, specifically it does (AxB)x(CxD) where A,B,C,D are input matrices. These matrices are passed as files. So first task, *task1*, is responsible for computing AxB, the second task, *task2*, computes CxD, and finally the third task, *task3*, computes (AxB)x(CxD) with the results obtained from *task1* and *task2*. So there is an obvious data dependency between *task3* and its predecessors *task1* and *task2*.

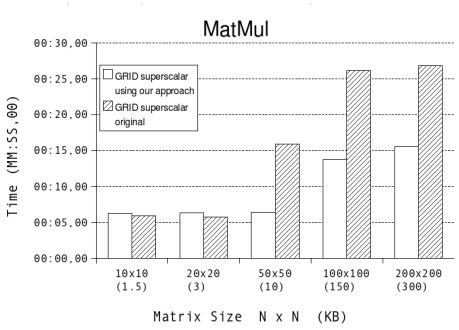

Fig. 7. MatMul results.

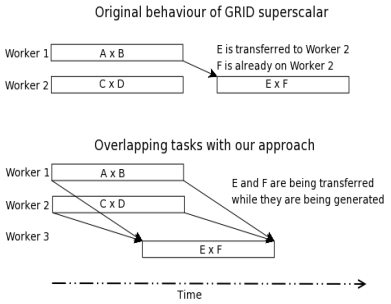

Fig. 8. MatMul behaviour through time.

We have executed the application ten times for both cases, with and without using our approach. Furthermore, we performed the tests with different matrix sizes, from 10x10 (1.5KB) to 200x200 (300KB). Results are shown in figure 7.

The significance of this test is the behaviour obtained with both cases commented before, since the master scheduling, and consequent assignments of task to workers, are different depending on whether GRID superscalar is using our approach or not. Figure 8 shows the utilization of each worker for both cases.

Specifically, the original version does not take advantage of a third node, because file F is already generated and accessible from node 2, whereas using our middleware does it, overlapping execution of tasks. Then, benefits appear with input matrices from 20x20 on. The improvement increases from 100x100 up to 200x200 matrices, reaching a 42% of time reduction.

Case 3.2: Double cryptography.

In this test we performed the evaluation of our approach using an application based on cryptology we called GRIDCypher. This is a basic application where a file must be encrypted in two steps from sender A to the receiver B, so that it is possible to ensure two things: first, the file will be only read by the receiver B since it has been encrypted with its public key and could only be decrypted with B private key; second, the receiver could trust the file was generated by the sender A, since it could be only decrypted with A's public key.

We have executed the application ten times for both cases, with and without using our approach. Furthermore, we performed the tests with different file sizes, from 100 bytes to 100KB. Results are shown in figure 9.

Figure 10 shows the utilization of each worker while this execution is performed. Using our approach, workers run concurrently while files are transferred step by step. On the other hand, the original GRID superscalar does not take advantage of more than one worker, since as soon as each step is finalized the same worker is able to do the next one.

The original version saves all the file transfers, but for each step the current task must wait the entire file from previous one, whereas using our approach,

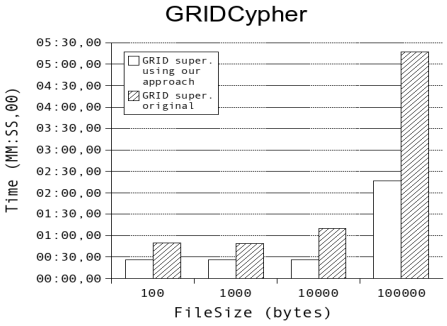

Fig. 9. GRIDCypher results.

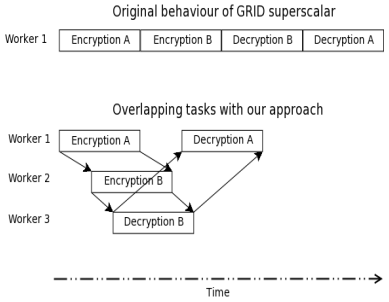

Fig. 10. GRIDCypher through time.

GRID superscalar could deal with parallelism between more than one worker. This behaviour is reflected in the results which shown that using our middleware benefits increase since the first case, reaching a 57% of time reduction.

Case 3.3: Multitrace processing.

In this test we performed the evaluation of our approach using an application based on processing PARAVER [2] traces. The scenario consists of two tasks *task1* and *task2* which decompress two PARAVER traces in parallel. Afterwards, the decompressed traces are merged with the Unix *merge* command executed by a third task *task3*. Finally, the merged trace is filtered via *grep* command.

There are interesting data-dependencies between the first two decompression tasks and the responsible for merging them. Eventually, there is another data-dependency between this merging task and the final one about post-processing.

We have executed the application ten times for both cases, with the original GRID superscalar and using our approach. Furthermore, we ran the tests with different file sizes, from 1MB to 10MB. Results are shown in figure 11.

Benefits are shown in all the cases, but significantly ones come from 7,5MB on. Figure 12 shows the utilization of each worker while the application is running. The original GRID superscalar version does not take advantage of more than two workers due to data dependencies among defined tasks, whereas using our approach, the benefits coming from overlapping tasks are important enough to reach a time reduction from 7% to 41%.

5 Related Work

Mainly, the are two types of publish/subscribe systems: topic-based and content-based. Topic-based systems are much like newsgroups. The idea is that subscribers express their interest by joining a group (related to a topic), so that all messages/events related to that topic are broadcast to all users subscribed to that group. Content-based systems are designed to allow subscriptions to specific contents, i.e. users express their interests specifying predicates over the values of

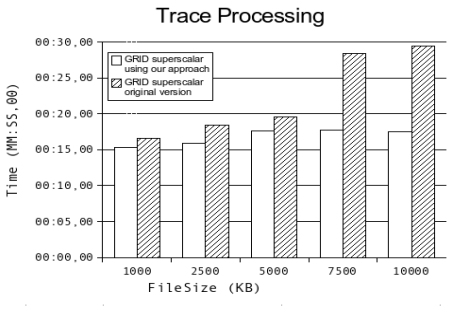

Fig. 11. Trace case results.

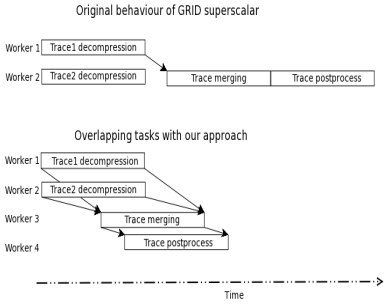

Fig. 12. Traces case through time.

a number of well-defined attributes. Therefore the matching of published events to subscriptions is done based on the content, i.e. the values of the attributes.

Our approach could be all part of the topic-based publish/subscribe systems taking into account that published files could be seen as topics. Currently it exists several topic-based approaches, like Scribe [3] or Bayeux [4], but mostly of them are not designed for file distribution.

There are other topic-based publish/subscribe systems commonly used in the world wide web, like RSS [5] which is based on broadcasting XML messages. But they are implemented using client-side polling to ask for the contents.

Current research lines on file-sharing distributed systems are mostly focused on using P2P [6] techniques. Peers transfer files using protocols that allow file chunks to be retrieved from remote peers in parallel. In a sense, this could address the problem that our middleware does, since peers could download each file from just one remote peer. But P2P systems are not designed to enable peers to read the contents sequentially and while they are being generated.

To address the entire issue this paper is focused on, we know of a system called Distributed File System (DFS) [7]. DFS is designed for enable file distribution in publish/subscribe fashion as our approach does. DFS allows the files to be read while they are being remotely generated thanks to a special file communication protocol based on events as our middleware does.

But the main problem in DFS is that when a file has to be broadcast to several subscribers, everyone has to receive the data before going on. This restriction introduces two basic problems. First, if a subscriber host is down, DFS enters into a loop trying to send the event to this host and making the other subscribers to wait for it. Second, regarding the integration of the middleware with a system like GRID superscalar, if the middleware used was DFS, an application based on task schema as shown in figure 13 would suffer an unnecessary delay in *task3* causing an unnecessary virtual dependency between *task2* and *task4*. On the other hand, using our middleware, the *EventDispatcher* thread responsible of sending events to the daemons subscribed to a specific local file, do not wait for any subscribers' response to continue sending other possible enqueued events. So that using our approach, the application commented could be executed faster.

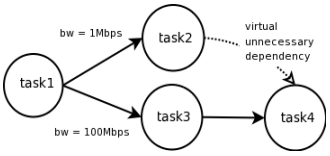

Fig. 13. Virtual unnecessary dependency using DFS instead of our approach

Furthermore, DFS is a framework with its own I/O API, so that it is necessary to change the application code to use DFS *open, close, read, write* I/O calls, whereas with our approach the applications keep their original code.

6 Conclusions

In this paper we have presented a publish/subscribe mechanism that can be used to improve parallelism in workflow applications by allowing the processing of a file at the same time as it is being generated in a different node in the Grid.

We have also integrated this mechanism into GRID superscalar to show that workflow engines can use this mechanism very easily and improve parallelism without requesting any special feature from the file system used.

Finally, we have measured the performance benefit of such a mechanism and have conclude that even for small files (less than 100KB), a significant performance (up to a speedup of 2) can be achieved.

References

1. Badia, R., Labarta, J., Sirvent, R., Prez, J., Cela, J., Grima, R.: Programming Grid Applications with GRID superscalar. Journal of Grid Computing **1**(2) (June 2003) 151–70
2. : Obtain detailed information from raw performance traces http://www.cepba.upc.es/paraver/.
3. Castro, M., Druschel, P., Kermarrec, A., Rowstron, A.: SCRIBE: A large-scale and decentralized application-level multicast infrastructure. IEEE Journal on Selected Areas in communications (JSAC) **20**(8) (2002) 1489–1499
4. Zhuang, S.Q., Zhao, B.Y., Joseph, A.D., Katz, R.H., Kubiatowicz, J.D.: Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In: Proceedings of NOSSDAV. (June 2001)
5. : Rss advisory board announcements and really simple syndication news http://www.rssboard.org/.
6. Lua, K., Crowcroft, J., Pias, M., Sharma, R., Lim, S.: A survey and comparison of peer-to-peer overlay network schemes. Communications Surveys & Tutorials, IEEE (2005) 72–93
7. Chen, K., Huang, Z., Li, B., Huang, E., Rajic, H., Kuhn, R., Chen, W.: Distributed File Streamer: A Framework for Distributed Application Data Coupling. In: 7th IEEE/ACM Grid. (September 2006) 168–175