A matrix inversion method with YML/OmniRPC on a large scale platform

Maxime Hugues and Serge G. Petiton

LIFL, University of Science and Technology of Lille,
59655 Villeneuve d'Ascq, France
{maxime.hugues,serge.petiton}@lifl.fr

Abstract. YML is a dedicated framework to develop and run parallel applications over a large scale middleware. This framework makes easier the use of a grid and provides a high level programming tool. It is independent from middlewares and users are not in charge to manage communications. In consequence, it introduces a new level of communications and it generates an overhead. In this paper, we proposed to show the overhead of YML is tolerable in comparison to a direct use of a middleware. This is based on a matrix inversion method and a large scale platform, Grid'5000.

1 Introduction

Intensive numerical applications like simulation, DNA decoding, climate prediction are parallelised and distributed. Most of those experiments are made by expert scientists of various domains on grids. The main difficulty for them is that each grid has its own properties and different middlewares are deployed on them. Therefore, scientist users must adapted the code to each middleware and this induces a waste of time. The Grid complexity requires a high level programming tool to hide all process to users.

YML [1] is one workflow solution. It is developed at the University of Versailles by the Nahid Emad's team. This framework is dedicated to develop and run parallel applications over large scale middleware. A workflow language named *YvetteML* is used by YML to describe parallelism of each application. YML provides a compiler and a just-in-time scheduler which allows to manage the execution of parallel applications. This transparent management allows to hide numerous communications and code coupling for complex applications. However, to make YML independent from middlewares, an additional communication level is necessary to link the just-in-time scheduler and the selected middleware. The fallout of this layer is that it creates an overhead contrary to a direct use of a client middleware program.

We proposed to evaluate the overhead of YML in different cases. The second section presents motivations of this paper. The third section introduces the Grid'5000 experimental platform, gives an overview of YML Framework and the grid middleware OmniRPC. Then, the Block-based Gauss-Jordan application

will be quickly defined and explained. YML experiments and results are presented and analysed in the fifth section. Finally, we conclude and present our future work in the sixth section.

2 Motivations

Companies and laboratories of various domains are more and more interested in grid computing. But in most of case, they have not the technical knowledge to program a grid. YML offers the possibility to develop and run a parallel application without managing communications and is independent from middlewares. The distributed computing is to speed up computations. So, the performance of a workflow framework is an important point and it has not to introduce a significant overhead. In this paper, we propose to estimate and compare the YML overhead with OmniRPC, a cluster/grid middleware. Some experiments are proposed and based on a classical algorithm of matrix inversion, the block-based Gauss-Jordan method. This algorithm offers task dependencies and a lot of communications which are keys of performances of a grid computation. Experiments are done on Grid'5000, a French large scale infrastructure for grid research and experiments. A cluster of 101 heterogeneous nodes is emulated on Grid'5000 to begin. Secondly, we want to observe the management of the computation resources when they are not enough numerous for the number of computation tasks. A cluster emulation of 10 nodes is done with the same applications. However, most of computing resources are distributed across a city, a country or the world like our platform distributed over three different sites (Lille and Orsay in France, Tsukuba in Japan). We proposed to emulate this case on Grid'5000 with heterogeneous networks and heterogeneous computing nodes over five sites geographically. Nevertheless, complex applications have a huge amount of data. These applications use out-of-core techniques. The last experiment is done with an OmniRPC program which uses out-of-core. This program is taken as referent to evaluate the overhead generated by YML.

3 Platform and environment

In a first step, the GRID'5000 platform is presented, followed by the YML framework, OmniRPC middleware, and to finish by the block-based Gauss-Jordan matrix application.

3.1 GRID'5000 Platform

Grid'5000 [2] is a large scale infrastructure for grid research. It is composed of nine geographically distributed clusters and each one has between 100 to 1000 heterogeneous nodes. This cluster of clusters is interconnected by the French national research network RENATER. Grid'5000 provides reconfiguration and monitoring tools to find out grid issues. This platform allows users to make

reservation, reconfiguration, run preparation and run experiment by using OAR and Kadeploy for nodes reservation and deployment of specific environment which built by user. Grid'5000 is used to investigate issues at different levels of the grid. This includes network protocols, middleware, fault tolerance, parallel/distributed programming, scheduling and issues in performance.

3.2 YML Framework

YML [3] is a framework dedicated to develop and run parallel applications on grids and peer to peer middleware. It is composed of a compiler and a just-in-time scheduler which manages tasks and data dependencies between components [4]. This framework implies a lot of data exchange through the network. To provide and take over data to each component on demand, there is the Data Repository server dedicated. Moreover, YML is independent from the middleware by using an adaptation layer called back-end, see the figure 1.

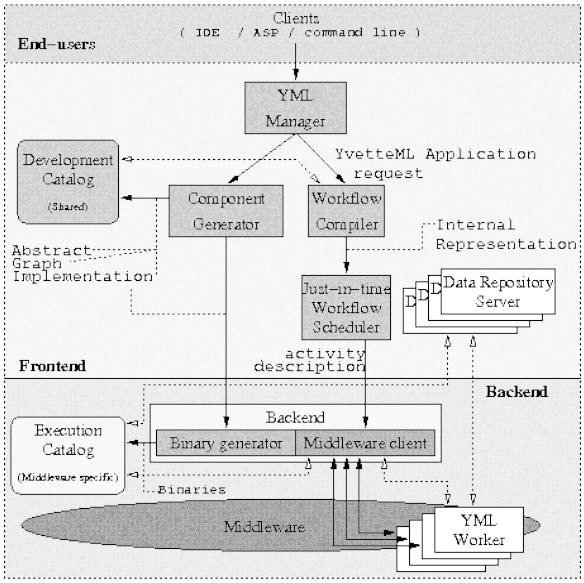

Fig. 1. YML design

To describe applications and their executions, YML includes a workflow language called *YvetteML*. The development of an YML application is made using components approach. *YvetteML* components are described using XML and they are three of kinds:

– Abstract component: an abstract component defines the communication interface with the other components. This definition gives the name and the

communication channels with other components. Each channel corresponds to a data in input, in output or both and is typed. This component is used in the code generation step and to create the graph.
- Implementation component: an implementation component is the implementation of an abstract component. It provides the description of computations The implementation is done by using common language like C or C++. They can have several implementations for a same abstract component.
- Graph component: a graph component carries a graph expressed in *YvetteML* instead of a description of computation. It provides the parallel and sequential parts of an application and the synchronize events between dependent components.

Moreover, those three components are independent of middlewares. So, to use an application on another grid with a different middleware, the scientist user has just to compile each component for the middleware of his choice.

3.3 OmniRPC

OmniRPC [5] is a thread-safe remote procedure call (RPC) system, based on Ninf [6], for cluster and grid environment. It supports typical master/worker grid applications. Workers are listed in a XML file named as the host file. For each host, the maximum number of job, the path of OmniRPC, the connection protocol (ssh, rsh) and the user can be defined. An OmniRPC application contains a client program which calls remote procedures through the OmniRPC agent. Remote libraries which contain the remote procedures are executed by the remote computation hosts. Remote libraries are implemented like a executable program which contains a network stub routine as its main routine. The declaration of a remote function of remote library is defined by an interface in the Ninf interface definition language (IDL). The implementation can be written in familiar scientific computation language like FORTRAN, C or C++.

3.4 Block-based Gauss-Jordan matrix inversion

One of the most classical methods for dense matrix inversion is the block-based Gauss-Jordan algorithm [7]. Let A and B be two squares matrices of dimension N, partitioned into $(p \times p)$ blocks of dimension n. Let B be the inverted matrix of A, progressively built. Each of the p steps has three parts (see (1), (2), (3) in the corresponding algorithm 1).

The first part is to invert the pivot block. In the second part, $2(p-1)$ blocks product and finally $(p-1)^2$ blocks triadic are computed. $(p-1)^2$ processors are necessary for computation and each loop **'For'** is executed in parallel.

It is necessary to take into account task dependencies. The figure 2 shows the intra-step and inter-steps parallelism. At each step, the loops (1) and (2) depend on the computation of the inverse block B_{kk} and the loop (3) partially depends on (1) and (2). Then, all matrix products in the loops (1) and (2) are independent tasks and are executed in parallel. The loop (3) is executed in

Algorithm 1 The block-based Gauss-Jordan matrix inversion

Input: A (partitioned into $p \times p$ blocks)
Output: $B = A^{-1}$

```
For k = 0 to p − 1
  B_kk = A^{-1}_kk
  For i = k + 1 to p − 1 (1)
    A_ki = B_kk × A_ki
  End For
  For i = 0 to p − 1 (2)
    If (i ≠ k)
      B_ik = −A_ik × B_kk
    End If
    If (i < k)
      B_ki = B_kk × B_ki
    End If
  End For
  For i = 0 to p − 1 (3)
    If (i ≠ k)
      For j = k + 1 to p − 1
        A_ij = A_ij − A_ik × A_kj
      End For
      For j = 0 to k − 1
        B_ij = B_ij − A_ik × B_kj
      End For
    End If
  End For
End For
```

parallel too, because it can start without the complete end of (1) and (2). At the step 2, the computation of the blocks represented by dark squares is not finished. The small numbers in squares represent the computation of the step 3. Our implementation of the Gauss-Jordan algorithm only use the intra-step parallelism. This method is implemented in a client OmniRPC program and in a YML program

4 Experiments

The experiments are done with different architectures of clusters. The table 1 gives the description of the resources that we used. The sites are interconnected by a heterogeneous gigabit ethernet. Before experiments, a node is reserved and a minimal debian is deployed on it to build a dedicated environment. OmniRPC, YML framework and the necessary libraries are installed on it. Then, the environment is recorded on each necessary site for next deployments. The dedicated environment is universal, so it is not necessary to rebuild it on each site to take into account their particularities. The next step is to reserve the required nodes on the grid for the experiment. A shell script is specified to deploy the dedicated

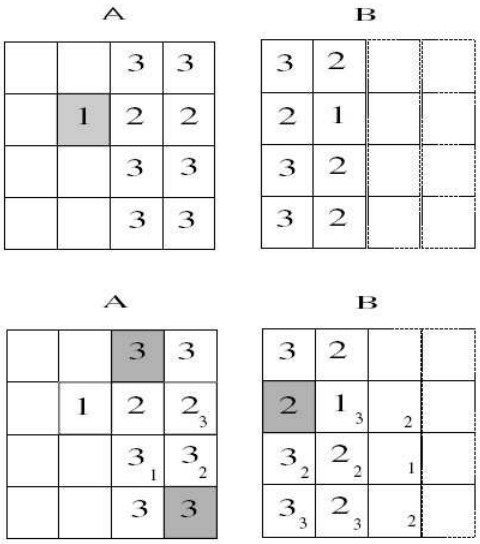

Fig. 2. Intra-step and inter-steps dependencies

environment, prepare the host file which contains the list of computation nodes, launch experiments and get results.

Site	Nodes	CPU/Memory
Nancy	120	2x DC INTEL xeon, 1.6GHz/2GB
Nancy	47	2x AMD64 opteron, 2GHz/2GB
Orsay	216	2 x AMD64 Opteron, 2.0GHz/2GB
Lyon	70	2 x AMD64 Opteron, 2.4GHz/2GB
Sophia	56	2 x DC AMD64 Opteron, 2.2GHz/4GB
Rennes	99	2 x AMD64 Opteron, 2.0GHz/2GB
Rennes	64	2 x AMD64 Opteron, 2.2GHz/2GB

Table 1. Computational nodes of Grid'5000

Moreover, one more node is deployed for each experiment. It is not taken into account in the number of deployed nodes, only computation nodes are taken into account. This node plays the role of the client/server and contains the host file required by YML and OmniRPC. The secure shell (ssh) is defined in the host file for communications between the master and workers. In the host file, the maximum number of jobs is set to two because all nodes are dual-processors. The multiplicity of cores are not taken into account in this number. To have a correct estimation of the YML overhead the same parallelism and operations are

described in OmniRPC and YML. Four components are defined to implement the block-based Gauss-Jordan algorithm:

1. inversion: to inverse one matrix block
2. prodMat: to compute the two blocks product
3. mProdMat: to compute the negative of two blocks product
4. ProdDiff: to compute the difference between one block and a block matrix product

For OmniRPC experiments, the remote libraries with four components are registered on all computation nodes after the deployment. Then, the executable program takes an argument the number of blocks and it is launched for each value. For YML experiments, abstract and implementation components are compiled and saved in the dedicated environment. A graph component is defined for each number of blocks. After the deployment, all graph components are copied and compiled on the client node. Then, the Data Repository Server is started and the scheduler is launched for each compiled graph component. The first experiment is to emulate a cluster on Grid'5000. 101 nodes are reserved on two clusters of Nancy with mainly dual core Intel Xeon. For a fixed block size (n = 1500) the number of blocks is varied. The condition of $(p - 1)^2$ processors is respected. This increasing variation of the number of blocks goes up with the number of tasks. Thereby, the amount of data dependencies and communications increase too. The execution time of OmniRPC and YML should be made a cubic variation with a constant gap of execution.

5 Results and analysis

After, the presentation of the dedicated environment and the condition of experiments, a description and an analysis of each experiment is done in this part. The first experiment is an evaluation of the YML overhead in the situation of one cluster composed of heteregenous nodes and networks. The second experiment is similar to the first, but the computation resources are insufficient for the algorithm of Gauss-Jordan. The third experiment is in case of a cluster of clusters distributed geographically. The last section is experiments with an OmniRPC program which uses out-of-core taken as referent to evaluate the overhead of YML.

5.1 Cluster of 101 nodes

YML overhead is first studied on a single cluster : one geographic site, with heterogeneity of nodes and networks. The first experiment is an emulation on Grid'5000 of a cluster composed of 101 nodes. The Xeon processors are the most numerous nodes. So, the cluster offers 202 processors that satisfy the condition of $(p-1)^2$ processors required by the block-based Gauss-Jordan method. But this

p	Number of tasks	OmniRPC	YML + Backend OmniRPC	Overhead
2	8	281 s	344 s	22.41 %
3	27	487 s	559 s	14.78 %
4	64	712 s	914 s	28.37 %
5	125	965 s	1359 s	40.82 %
6	216	1250 s	2070 s	65.60 %
7	343	1575 s	3103 s	97.01 %
8	512	2102 s	5008 s	138.24 %

Table 2. Time of execution for a cluster of 101 nodes, with block size = 1500

condition is available until p equal 15, beyond the computation resources are insufficient. The block size is fixed and for several number of blocks the experiment is done.

The table 2 shows that the overhead is in relation with the number of blocks. More the number of blocks is important, thereof the number of tasks because there are correlated, more the overhead increases. Firstly, this evolution of the overhead comes from in part of the resolution method of dependencies. The YML scheduler resolves task dependencies at run-time, in opposition to OmniRPC which knows dependencies at compile-time. In an OmniRPC program, the user defines the tasks which have to be waited and launched. In a YML program, the program is translated into a workflow, then the scheduler reads the workflow and launches the tasks without knowing which will end first. The main difficulty for YML is to settle dependencies when it comes at the same time. Moreover, YML has a centralized approach and manages dependencies and data exchange.

Secondly, in this experiment the client/server node has globally the same configuration than the computing nodes. YML has a workload more important than an OmniRPC application. The fallout is the time of execution of an YML program is longer. Although, the overhead stays tolerable until p equal 6, with an overhead of 65 % for 216 tasks.

The execution time of a job on a node play an important role in the overhead of YML and the workload of the scheduler. To reduce the execution on a node and to show this phenomenon, the block size is successively fixed at 1000 and 500. In consequence, a computing node is going to be less loaded and the YML scheduler will be more requested to resolve dependencies at a same time. Furthermore, the data repository server will be more requested to deliver and receive data. So, in this case the overhead should be more important than the first experiment with the block size fixed at 1500.

The first observation is the overhead for a block size of 1000 is less important than a block size of 1500, see table 3. This comes from YML which imports and exports data on the hard disk. If the block size is decreased, the time to read and write the data is shorter. The amount of data to treat between a block size of 1000 and 1500 has a ratio of 1/2. The computation time for a block is approximately the same, but the time to write/read data on hard disk is different. So, the access time to the disk gained by YML decreases the overhead.

p	Number of tasks	OmniRPC	YML + Backend OmniRPC	Overhead
2	8	108 s	132 s	22.22 %
3	27	165 s	236 s	43.03 %
4	64	242 s	323 s	33.47 %
5	125	328 s	461 s	40.54 %
6	216	425 s	653 s	53.64 %
7	343	541 s	905 s	67.28 %
8	512	676 s	1427 s	111.09 %

Table 3. Time of execution for a cluster of 101 nodes, with block size = 1000

p	Number of tasks	OmniRPC	YML + Backend OmniRPC	Overhead
2	8	41 s	48 s	17.07 %
3	27	50 s	63 s	26.00 %
4	64	58 s	82 s	41.37 %
5	125	75 s	125 s	66.66 %
6	216	83 s	207 s	150.39 %
7	343	103 s	277 s	168.93 %
8	512	130 s	379 s	191.53 %

Table 4. Time of execution for a cluster of 101 nodes, with block size = 500

But for a block size of 500, the overhead is more important, see table 4 and figure 3. Because the computation time of a block on a node is shorter, then the scheduler of YML is more requested to resolve dependencies. Furthermore, the data repository server is more requested too for the data exchanges.

5.2 Cluster of 10 nodes

To decrease the use of the YML scheduler and the data repository server, a second experiment is done with few computation resources. The use case is the same than the first experiment: one cluster, one geographic site, with heterogeneity of nodes and networks. The second experiment is an emulation of a cluster composed of 10 nodes. So, the cluster offers 20 processors that satisfy the condition of $(p - 1)^2$ processors required by the block-based Gauss-Jordan method. But this condition is available until p equal 5, beyond the computation resources are insufficient. The block size is fixed and for several number of blocks the experiment is done.

First observation, the execution times for a cluster of 10 nodes are less important than a cluster of 101 nodes until p equal 5 for a block size of 1500, see table 5. When OmniRPC starts, it builds a database which contains the computing nodes. In this experiment, the nodes are less than the experiment with 101 nodes. So, the database is built faster. For p from 6 to 8, the execution times have an additional delay. Because the computing resources are not sufficient beyond p equal 5. To execute the next task, it is required to expect a free node.

Second observation, the overhead is not very important. It is between 10 and 20 percent for p from 2 to 5. The overhead is lower than the results of the

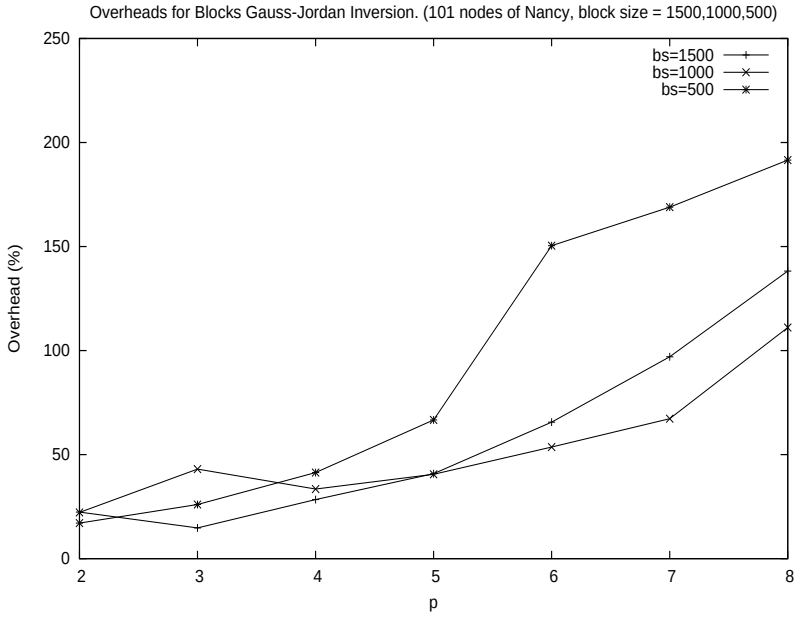

Fig. 3. Overhead for different blocks size on a 101 nodes cluster on Nancy

experiment of 101 nodes on one cluster. The number of tasks is more important than the computer resources available. The fallout is the scheduler of YML is less requested to solve the data dependencies at a same moment. The dependencies are solved faster. The data repository server is less used to deliver and receive the data. This explains that the execution times are approximately the same as the experiment of 101 nodes for $p > 5$.

5.3 Cluster of clusters

After the evaluation of the YML overhead in the case of one cluster. We want to evaluate the overhead in the case of a cluster of clusters like our platform distributed between Lille, Orsay in France and Tsukuba in Japan. This experiment is an emulation on Grid'5000 of a cluster of 101 nodes distributed over 5 geographic site (Nancy, Orsay, Sophia, Lyon, Rennes) and over 6 clusters. The nodes are distributed in an homogeneous way, 17 nodes per site, excepted Rennes which has 34 nodes distributed over two clusters. The site of Nancy counts 16 nodes and the node which plays the role of client/server. So, the cluster offers 202 processors that satisfy the condition of $(p - 1)^2$ processors required by the block-based Gauss-Jordan method. But this condition is available until p equal 15, beyond the computation resources are insufficient.

The heterogeneity of networks adds time for communications between the computing nodes and the server. The difference of execution times for OmniRPC

p	Number of tasks	OmniRPC	YML + Backend OmniRPC	Overhead
2	8	203 s	247 s	21.67 %
3	27	406 s	453 s	11.57 %
4	64	660 s	740 s	12.12 %
5	125	982 s	1204 s	22.60 %
6	216	1454 s	1945 s	50.00 %
7	343	1874 s	3073 s	63.98 %
8	512	2600 s	6076 s	133.69 %

Table 5. Time of execution for a cluster of 10 nodes, with block size = 1500

p	Number of tasks	OmniRPC	YML + Backend OmniRPC	Overhead
2	8	307 s	361 s	17.48 %
3	27	506 s	578 s	14.22 %
4	64	727 s	910 s	25.17 %
5	125	1193 s	1487 s	24.64 %
6	216	1659 s	2702 s	62.86 %
7	343	2258 s	3164 s	40.12 %
8	512	2921 s	5836 s	99.79 %

Table 6. Time of execution for a cluster of clusters of 101 nodes, with block size = 1500

between the table 6 and the table 2 are between 20 and 800s. In the case of YML, the difference of execution times are less significative, between 20 and 630s. The overhead of YML in this configuration of cluster of clusters are acceptable, between 14 and 63 percent. Because the communication times are important, so the scheduler and the data repository server of YML are less requested.

5.4 Out-of-core Gauss-Jordan

The previous evaluations of the overhead are made in the worst case. After each component call, YML reads and writes data on the hard disk. However, our OmniRPC program stores data in live memory. So, the access time to the data for YML is more important than OmniRPC. In this section, the evaluation of the overhead is made with the same YML program and an OmniRPC program which uses out-of-core. It is very important to notice that the OmniRPC program with out-of-core is not a version of Gauss-Jordan with out-of-core. The program only reads and writes data at each step of the Gauss-Jordan method, like [8]. This evaluation is firstly done with the same condition than the first experiment: one cluster, with heteregenous nodes and networks. This cluster is based on Nancy and has 101 nodes. So, the cluster offers 202 processors that satisfy the condition of $(p - 1)^2$ processors required by the block-based Gauss-Jordan method. But this condition is available until p equal 15, beyond the computation resources are insufficient. To compare these new results with the first experiment, the block size is fixed at 1500 and for several number of blocks the experiment is done.

p	Number of tasks	OmniRPC out-of-core	YML + Backend OmniRPC	Overhead
2	8	321 s	344 s	7.16 %
3	27	539 s	559 s	3.71 %
4	64	851 s	914 s	7.40 %
5	125	1193 s	1359 s	13.91 %
6	216	1625 s	2070 s	27.38 %
7	343	2049 s	3103 s	51.44 %
8	512	2564 s	5008 s	95.31 %

Table 7. Time of execution of Gauss-Jordan out-of-core on a cluster of 101 nodes, with block size = 1500

In the table 7, the execution times of OmniRPC are higher than the results in the table 2. This increase comes from the location of data. In the case of our Gauss-Jordan out-of-core, data are located on the hard disk. They are loaded in the main memory when they are going to be used in the step k. The access time to data located on the hard disk is more important than the access time to data located in main memory. The figure 4 shows the overhead is smaller with an OmniRPC program which uses out-of-core. The difference between the overheads varies from 11.07% for p=3 to 45.47% for p=7.

The overhead of YML is less in the case where the Gauss-Jordan algorithm stores data on the hard disk. Moreover, we could see the cluster of clusters configuration decreases the overhead. The next experiment is also to combine this configuration with our OmniRPC out-of-core program. The goal of this experiment is to see if the overhead of YML can be decreased in the case of a large scale distributed application.

p	Number of tasks	OmniRPC Out-of-Core	YML + Backend OmniRPC	Overhead
2	8	329 s	361 s	9.72 %
3	27	590 s	578 s	-2.03 %
4	64	947 s	910 s	-3.90 %
5	125	1412 s	1487 s	5.31 %
6	216	2080 s	2702 s	29.90 %
7	343	2796 s	3164 s	13.16 %
8	512	3605 s	5836 s	61.88 %

Table 8. Time of execution of Gauss-Jordan out-of-core on a cluster of clusters, with block size = 1500

There is a gain for p equal 3 and 4. YML is faster than our OmniRPC program for this two cases. The scheduler of YML solves dependencies at the runtime. In opposition, the dependencies of our OmniRPC programme are specified by the coder when he writes the application. So, the OmniRPC programme have not to

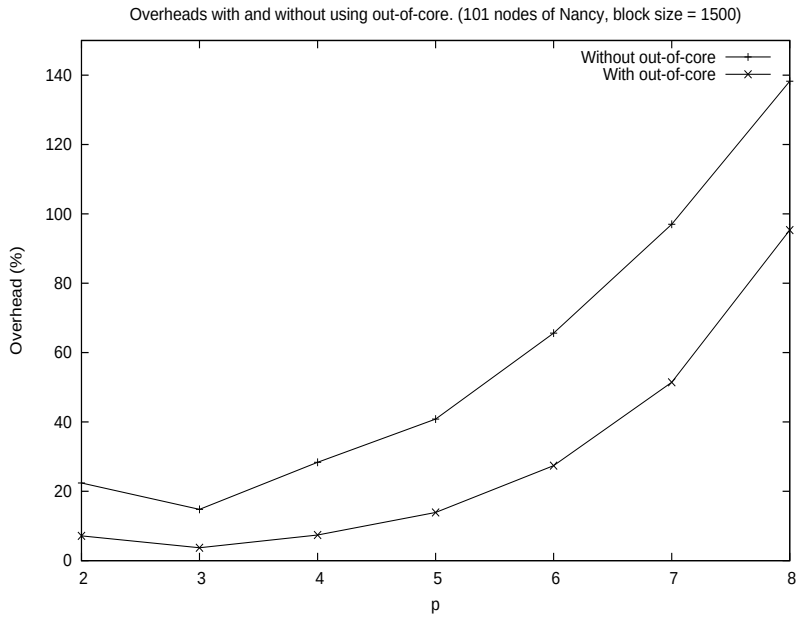

Fig. 4. Overheads with and without using out-of-core

solve dependencies but only to synchronize the dependent tasks with some wait and call functions.

6 Conclusion

In this work, we have evaluated the overhead of YML compared to an OmniRPC programme with different architecture of cluster and in two cases of data storage. In the first case, the OmniRPC program of reference stored data in the main memory. So, the OmniRPC program accessed faster to data than YML which had to load and unload data from hard disk at each component call. The experiments have showed that the overhead of YML was not important when the scheduler is not overloaded to solve the data dependencies, between 10 to 60% for a maximum of 512 tasks. In the second case, we have taken as referent an OmniRPC program which stores data on the hard disk. Because a lot of complex programs which have an important amount of data use out-of-core techniques. In this context, the overhead of YML is under 95% for the same experiments of the first case. Furthermore, with an architecture of cluster of clusters, YML can be faster than our OmniRPC program. Even though, 100 or 150% of overhead are high values in some cases. It is important to notice that YML allows to reuse the code for different middlewares. An YML application has not to be adapt for an other platform which uses a different middleware. Moreover, it is not necessary to

manage communications like an MPI program. And the end user can improve parts of code to decrease the overhead.

To complete this work, we plan to evaluate the overhead with huge matrix and clusters which have more processors. YML does not integrate a data persistence system and multicore management at the moment. But we could evaluate the overhead with an emulation of the data persistence. The matrix blocks would be regenerated on the nodes. And the scheduler of YML will be improve to have less overhead. In the future, YML could use many middlewares at the same moment with a multi-backend support. This feature is in prevision to program hybrid clusters with a high level programming tool. For the moment, the framework YML is a performing tool for average application and has a tolerable overhead. It allows to program easily a grid without manage communications, it supports two middlewares and soon three with the arrival of the backend for condor.

7 Acknowledgment

In addition, we thank a lot Olivier Delannoy and Nahid Emad from PRiSM, Laurent Choy and Mitsuhisa Sato from the University of Tsukuba for their help and collaboration on YML and OmniRPC. We thank a lot too the INRIA for their help on Grid'5000.

References

1. YML Project Page. http://yml.prism.uvsq.fr.
2. Franck Cappello, Eddy Caron, Michel J. Daydé, Frédéric Desprez, Yvon Jégou, Pascale Vicat-Blanc Primet, Emmanuel Jeannot, Stéphane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Benjamin Quétier, and Olivier Richard. Grid'5000: a large scale and highly reconfigurable grid experimental testbed. In *The 6th IEEE/ACM International Conference on Grid Computing*, pages 99–106, 2005.
3. Olivier Delannoy, Nahid Emad, and Serge G. Petiton. Workflow Global Computing with YML. In *The 7th IEEE/ACM International Conference on Grid Computing*, pages 25–32, 2006.
4. S. Petiton. Parallelization on an MIMD computer with real-time Scheduler. *Aspects of Computation on Asynchronous Parallel Processors, North Holland*, 1989.
5. Mitsuhisa Sato, Taisuke Boku, and Daisuke Takahashi. OmniRPC: a Grid RPC ystem for Parallel Programming in Cluster and Grid Environment. In *The 3rd IEEE International Symposium on Cluster Computing and the Grid*, pages 206–, 2003.
6. Mitsuhisa Sato, Hidemoto Nakada, Satoshi Sekiguchi, Satoshi Matsuoka, Umpei Nagashima, and Hiromitsu Takagi. Ninf: A Network Based Information Library for Global World-Wide Computing Infrastructure. In *HPCN Europe*, pages 491–502, 1997.
7. L. M. Aouad, S. Petiton, and M. Sato. Grid and Cluster Matrix Computation with Persistent Storage and Out-of-Core Programming. In *The 2005 IEEE International Conference on Cluster Computing, September 2005. Boston, Massachusetts*, 2005.
8. Nordine Melab, El-Ghazali Talbi, and Serge G. Petiton. A Parallel Adaptive version of the Block-based Gauss-Jordan Algorithm. In *IPPS/SPDP*, pages 350–354, 1999.