GREMLINS: a large sparse linear solver for grid environment

Raphaël Couturier[1] and Christophe Denis[2] and Fabienne Jézéquel[2]

[1] *Laboratoire d'Informatique de l'Université de Franche-Comté, BP 527, 90016 Belfort CEDEX - France*
email: raphael.couturier@univ-fcomte.fr

[2] *Laboratoire d'Informatique de Paris 6, CNRS UMR 7606, Université Pierre et Marie Curie - Paris 6, 4 place Jussieu, 75252 Paris CEDEX 05 - France*
email: {christophe.denis,fabienne.jezequel}@lip6.fr

Abstract

Traditional large sparse linear solvers are not suited in a grid computing environment as they require a large amount of synchronization and communication penalizing the performance on this architecture. This paper presents some features of the solver designed during the current GREMLINS (GRid Efficient Method for LINear Systems) project. The GREMLINS solver limits the amount of communication as it is based on a coarse grained iterative method called multisplitting method. Moreover, the solver can be executed either in a synchronous or an asynchronous mode. In the latter case, iterations are desynchronized and there is no more synchronization at all. It may result in a faster execution time compared to the synchronous case. Some experiments presented in this paper with the GRID'5000 architecture, a nation wide experimental grid in France, allowed us to highlight interesting features of this solver.

Key words: sparse linear solver, iterative method, asynchronous iterations, multisplitting method, grid computing
PACS:

1 Introduction

Many scientific problems require after, their discretization steps, the solving of large sparse linear systems in order to simulate phenomena close to reality [1]. For decades new techniques have been designed to efficiently solve them. The use of multiple machines could increase the amount of memory available or reduce the execution times required by the solving of very large sparse matrices. In order to benefit from simultaneous machines, one can either use parallel computers or clusters of machines. The former class is composed of homogeneous processors linked by a high speed communication system. Nevertheless, this kind of architecture is very expensive and not available for all scientists. The latter class consists in using a set of traditional computers. In most cases, machines are either used by single users as personal computers or used simultaneously or not by other people to solve large problems. Clusters are more heterogeneous than parallel architecture. The heterogeneity may lie in the processors and more generally in the architecture of the machines (memory, hard drive, operating system) but it can also lie in the network capacities (bandwidth and latency). In order to dispose of a large number of processors, scientists may simultaneously use several clusters. In general, clusters are geographically distant and communications are less efficient than the ones of local clusters or parallel machines. Yet distant clusters computing may be considered as a grid computing platform.

In grid computing contexts, many parallel algorithms may not be efficient, especially with a large number of processors [2]. That is why it is essential to develop new algorithms suited to grid environments. For example, concerning partial differential equations, interested readers may consult [3,4]. Indeed, in distant environments the amount of synchronization and communication has to be limited because the communication network has relatively poor performances. The inefficiency of a standard parallel algorithms, executed in a grid environment, may happen when computations have non negligible dependencies requiring large communications. Furthermore, the parallel algorithm needs to be flexible as the network parameters (bandwidth and latency) and the computing power (due to the load of users' interaction) may drastically evolve during the computation. So, in this context, fine grained algorithms and dynamic load balancing are completely inefficient because of communication costs. Therefore the problem of load balancing with a huge amount of data is difficult to achieve out as efficient static data distribution algorithms are generally NP-complete [5,6].

Solving sparse linear solvers can usually be categorized into two classes: direct methods and iterative ones. The former class consists in finding the exact solution (by neglecting the rounding error) of the linear system after a finite number of steps. Most of efficient algorithms are based on a LU factorization [1,7]. Those algorithms are often used as a black box which can find the solution of a linear system without knowing any property of the matrix. They are quite memory consuming and their parallelization is difficult and present some small sequential parts. The latter class proceeds by successive approximation of the solution until the convergence. Several iterative algorithms have been designed. Dealing with sparse matrices this class of algorithms often requires less computation when the number of iterations is small compared to a direct method. However, the convergence depends on some properties on the matrix. The parallelization of those algorithms is quite easy and the speed up is very good with a parallel architecture.

It is well known that parallel versions of direct solvers require high connection networks because the amount of communication is large. That is why synchronous parallel direct solvers are not suitable for execution in a grid context environment. Standard parallel version of iterative solvers are based on the decomposition of the iterated vector. Each processor is in charge of a part of the vector and all instructions of the sequential algorithms are achieved out in parallel using gather-and-scatter operations. Consequently there are often several synchronizing communications or blocking operations for only one iteration. From a grid point of view, this kind of algorithms would not be appropriate.

Multisplitting algorithms have the characteristic of decomposing the initial system into subsystems and converging to the solution by successive approximation of the solution [8,9]. They differ from standard parallel iterative algorithms by the splitting they provide which may present similarities with decomposition methods. Dealing with sparse linear systems the multisplitting method generalizes the block Jacobi algorithm by allowing the use of asynchronous iterations and by providing the overlapping of some components. In each block, a direct method or an iterative one may be used to solve the subsystem. Multisplitting methods have been widely studied theoretically in literature (see, e.g.,[10]); but few studies report experiments in grid computing context, especially with a large number of processors.

In this paper, we describe the solver we have designed during the GREMLINS project. It is based on the multisplitting method presented in Section 2. Currently it has the particularity of using different sequential solvers to solve subsystems provided by the method: SuperLU [11], MUMPS [12] and the iterative solvers of the SparseLib library [13]. It supports both synchronous and asynchronous execution modes. It should be noticed that the interest for grid computation is to use the asynchronous version. Moreover, to the best of our knowledge, our solver is the only one dedicated to grid computing context. Our solver is not built with a standard communication library for parallel computing because that kind of library does not provide a way of designing efficient parallel asynchronous iterative algorithms. In [14] the use of a multithreaded library is explained and shown in order to obtain this goal. That is why we have chosen to use the CRAC library [15] exposed in Section 3. It is designed to implement both synchronous and asynchronous parallel iterative algorithms. In Section 4, we present and analyze some experiments performed on the GRID'5000 architecture, a nation wide experimental grid in France. Finally we present our concluding remarks and perspectives.

2 The multisplitting method

In the following we consider we have a $n \times n$ sparse linear system

$$Ax = b. \tag{1}$$

We suppose that (1) has a unique solution. The multisplitting method consists in splitting the matrix into horizontal rectangle matrices. Then each processor is responsible for the management of a rectangle matrix. With this distribution, a processor knows the offset of its computation. This offset enables us to define the submatrix, noted $ASub$, which a processor is in charge of managing. The part of the rectangle matrix before the submatrix represents the left dependencies, called $DepLeft$, and the part after the submatrix represents the right dependencies, called $DepRight$. Similarly, $XSub$ represents the unknown part to solve and $BSol$ the right hand side involved in the computation. Figure 1 describes the decomposition of A, b and x into several parts ($DepLeft$, $ASub$, $DepRight$, $Xleft$, $XSub$, $XRight$, $BSub$) required locally by a processor.

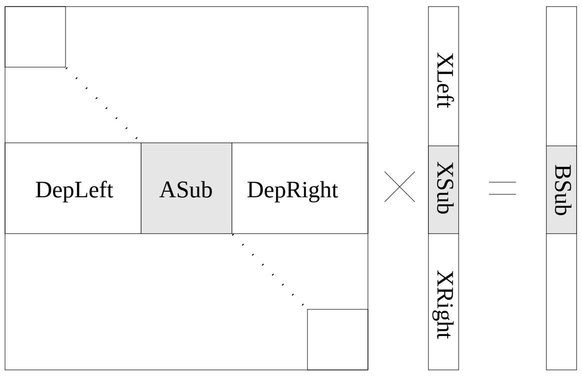

Fig. 1. Decomposition of the matrix

At each step, a processor computes $XSub$ by solving the following subsystem:

$$ASub * XSub = BSub - DepLeft * XLeft - DepRight * XRight$$

Then the solution $XSub$ must be sent to each processor depending on it.

2.1 Multisplitting algorithm

Algorithm 1 summarizes the multisplitting method used to solve a linear system also called the linear multisplitting solver. The four main steps are described as follows:

(1) **Initialization**
The matrix could be loaded or generated. Each processor manages the

Algorithm 1 linear multisplitting method

```
Initialize the communication interface
MyRank = *Rank of the processor*
NbProcs = *Number of processors*
Size = *Size of the matrix*
SizeSub = *Size of the submatrix*
Offset = *Offset of the matrix*
ASub[SizeSub][SizeSub] = *Submatrix*
DepLeft[SizeSub][Offset] = *Submatrix with left dependencies*
DepRight[SizeSub][Size-Offset-SizeSub] = *Submatrix with right dependencies*
DependsOnMe[NbProcs] = *Array with dependent processors*
IDependOn[NbProcs] = *Array with processors this processor depends on*
BSub[SizeSub] = *Array with right hand side of subsystem*
XSub[SizeSub] = *Array with solution of the subsystem*
XLeft[Offset] = *Array with left solution of the system*
XRight[Size-Offset-SizeSub] = *Array with right solution of the system*
BLoc = *Array with local computation of right hand side*
TLoc = *Array used for reception of dependencies*
repeat
  BLoc = BSub
  if MyRank!=0 then
    BLoc = BLoc-DepLeft*XLeft
  end if
  if MyRank!=NbProcs-1 then
    BLoc = BLoc-DepRight*XRight
  end if
  XSub = Solve(ASub,BLoc)
  for i=0 to NbProcs-1 do
    if i!=MyRank and DependsOnMe[i] then
      Send(PartOf(XSub,i),i)
    end if
  end for
  for i=0 to NbProcs-1 do
    if i!=MyRank and DependsOnMe[i] then
      Send(PartOf(XSub,i),i)
    end if
  end for
  for i=0 to NbProcs-1 do
    if i!=MyRank and IDependOn[i] then
      Receive(TLoc,i)
      Update XLeft or Xright with TLoc according to the processor $i$
    end if
  end for
  Convergence detection
until Global convergence is achieved
```

load of the rectangle matrix (in the algorithm the rectangle matrix corresponds to $DepLeft + ASub + DepRight$). Then until convergence, each processor iterates on:

(2) **Computation**
At each iteration, each processor computes $BLoc = BSub - DepLeft * XLeft - DepRight * XRight$. Then, it solves $XSub$ using the $Solve(ASub, BLoc)$ function.

(3) **Data exchange**
Each processor sends its dependencies to its neighbors. As the dependencies can be different from one processor to another, the function $PartOf$ computes the part dedicated to processor i. Nonetheless, when a processor receives a part of the solution vector (noted $XSub$) of one of its neighbors, it should update its part of $XLeft$ or $XRight$ vector according to the rank of the sending processor.

(4) **Convergence detection**
Two methods are possible to detect the convergence. We can either use a centralized algorithm described in [16] or a decentralized version, that is a more general version, as described in [17].

This algorithm may be similar to block Jacobi algorithm. It actually generalizes the block Jacobi method which is only a particular case of the multisplitting method. The two main differences of the multisplitting method are:

- The multisplitting method may use asynchronous iterations. In this case, the execution times may be reduced. For a complete description of asynchronous algorithms, interested readers may consult [18]. In [19,20] authors studied how flexible communications might improve the convergence of asynchronous algorithms. This particularity allows a processor to use results from its neighbors as soon as they arrive. Practically, the flexibility of communications allows to decrease the ellapsed time. We do not use this feature in this work because with the multisplitting method it requires modifications in the inner algorithm (used to solve the linear system) in order to take into account new messages directly in the computation.
- Some components may be overlapped and computed by more than one processor. Overlapping some components of the system may drastically reduce the number of iterations to obtain the convergence whatever the chosen threshold.

From a practical point of view, the use of asynchronous iterations consists in using non blocking receptions, dissociating computations from communications using threads and using an appropriate convergence detection algorithm. Overlapping some components is simple to implement since one only needs to change the variables $Offset$ and $SizeSub$ in Algorithm 1 in order to compute some components by two processors and define how overlapped components are taken into account.

Currently, the matrix distribution is simple: each submatrix is load balanced in terms of number of rows but unfortunately not in terms of computational volume. We are working on an efficient static matrix data distribution based on computational volume estimators in order to decrease the global execution time. Preliminary results presented in [21] show that this data distribution could drastically reduce the global execution time.

The sequential solver used in the algorithm is free. It can be a direct one or an iterative one. For the former class (direct), the most consuming part is the factorization part which is only achieved out at the first iteration. With large matrices, even after the decomposition process, the size of a submatrix that a processor is responsible to solve in sequential, may be quite large. So, the time required to factorize a submatrix may be long. In opposition to a long factorization time, the use of a sequential direct solver allows to solve other iterations very quickly because only the right-hand side changes at each iteration. For the latter class (iterative), iterations of the multisplitting method require a non negligible time, even if this time may, after a given number of iterations, be reduced. So, if in advance, the user knows some properties of the matrix he wants to solve, he can choose the appropriate class of solver.

The important characteristics of a sparse matrix are its pattern and the spectral radius of its iteration matrix. The pattern of a submatrix acts on the time required to factorize it at the first iteration of a solving with a sequential direct solve. Some previous works, already done around the load balancing of the multiple front method [22,23], have successfully been adapted to the linear multisplitting solver. The spectral radius of the iteration matrix is correlated with the number of iterations required to solve the system. The closer the spectral radius is to 1, the more iterations are required to solve the system, as all iterative methods. The convergence condition to make the asynchronous version converge is slightly different and more restrictive than the synchronous one, i.e., in some rare practical cases, the synchronous version would converge whereas the asynchronous one would not. As the explanation for this condition is quite complex, because it lies on several mathematical tools, we invite interested readers to consult [24].

After the presentation of the linear multisplitting algorithm we intend to briefly present the library used for communications as indeed it is quite new.

3 CRAC

CRAC is a library designed to build parallel iterative asynchronous applications. It uses the classical MPI triplet: daemon, application, spawner. The daemon is launched on each machine constituting the Virtual Distributed

Machine (*VDM*). The user develops its application and launches it with the spawner on the desired machines. Meanwhile, the similarity with MPI nearly stops here. Even if the CRAC programming interface uses the message passing paradigm, the communication semantic is completely different and several primitives do not exist in MPI. Furthermore, the internals of CRAC are based on multithreading and even the application is a thread. Finally, the virtual distributed machine relies on a hierarchical view of the network in order to reach machines with private IPs and to limit the bandwidth used on slow links.

In order to define the VDM, we describe the different kinds of machines: master, supermaster, slave, frontal.

- a frontal is a machine that can relay messages from outside the site to the private IP machines of the site. It can also relay messages to another site if a machine cannot send outside the site.
- a slave is a machine with no particular role.
- a master is a machine that collects information from the slaves of the site and relays them to the supermaster, or the opposite.
- a supermaster is a machine that collects/sends information from or to the masters. Obviously, the supermaster is a master but it is unique.

A CRAC daemon is launched on each machine of the VDM. Its main task is to send and receive messages for the application tasks executed on the machine hosting the daemon. For this, two threads are created : a Sender thread and a Receiver thread. Both these threads use a queue. The Sender thread sends long messages in several chunks. The Receiver thread uses a polling mechanism to detect the incoming data. According to the execution mode (synchronous or asynchronous) it stores all messages (in the synchronous mode) or it keeps only the last version of a message (in the asynchronous mode) if it has the same sender and tag.

The application task is a thread which is executed within the daemon context. Thus, the task can directly access the message queues (incoming and outgoing). This is not the case of MPI, in which a task is a process and must communicate (with an Unix socket or shared memory) with the daemon to send/receive data.

From the programmer point of view, CRAC basically proposes three methods to send or receive a message and detect the convergence. The emission of a message is never blocking. The message is copied into the outgoing queue when the Send method is called. The receive method is blocking in the synchronous mode (like in MPI) whereas it is not blocking in the asynchronous mode. In the latter case the method returns the last version of a message if one or more version of the same message arrived and it returns nothing otherwise. The convergence method requires a parameter indicating the local convergence and it returns the global convergence using a centralized or decentralized algorithm on the supermaster node.

4 Experiments

In this section we explain all the experiments we have performed and we analyze obtained results. Experiments have been conducted on the GRID'5000 architecture, a nation wide experimental grid [25]. Currently, the GRID'5000 platform is composed of an average of 1300 bi-processors that are located in 9 sites in France: Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes, Sophia-Antipolis, Toulouse. Most of those sites have a Gigabit Ethernet Network for local machines. Links between the different sites range from 2.5 Gbps up to 10Gbps. Most processors are AMD Opteron. For more details on the GRID'5000 architecture, interested readers are invited to visit the website: www.grid5000.fr.

4.1 Context of experiments

With the CRAC library we have implemented the linear multisplitting solver which has the particularity of using a sequential solver to solve the local linear subsystem issued from the multisplitting method. In the following we call it GREMLINS. Currently, it can be used with the SparseLib, MUMPS or SuperLU solvers. All these solvers are freely available for academic researchers. As CRAC is an object oriented library, GREMLINS uses this paradigm and the sequential solver used to solve the subsystem is an object that is consequently easy to change. Obviously, it is easy to add other sequential sparse linear solvers. GREMLINS has the particularity to be executed either in a synchronous mode or an asynchronous one. In this paper, in order to experiment its efficiency in a grid environment we have used generated square matrices and a real square matrix. Generated matrices are built on-the-fly. This offers the advantage of having matrices of the desired pattern and with the desired form. We have also used a real matrix, which we call in the following advec-diffu; it comes from a 3D advection-diffusion equation discretized with a finite difference scheme (see, for example, [26] for more details on the equation). The size of this matrix is 6,750,000 and the number of non-zero elements is 53,730,000. More precisely the considered system is discretized with a 3D grid composed of 150*150*150 discretized points and the model contains two components per discretized point. For all the experiments we have chosen a 10^{-8} precision using the infinity norm. For more details on the parameters of the simulation, interested reader are invited to consult [27].

The generated matrices we use are built using the following scheme. The diagonal of the matrix is not empty neither are its two neighbor diagonals. Then according to the number of diagonals specified by the user, some of the other diagonals are not empty. Those diagonals are equitably scattered between the diagonal of the matrix and a bandwidth specified by the user. Consequently, in the experiments we report those two parameters (number of diagonals and bandwidth). Off-diagonal non empty elements of a matrix are negative random values with a value between -1 and 0. Diagonal elements are equal to the inverse of the sum of the non empty elements of the same line plus a random value whose interval is defined by the user. This allows us to change the spectral radius of the iteration matrix which acts on the number of iterations required to reach a given threshold during the solving. Such generated matrices are M-matrices [28] for which it is known that multisplitting algorithms converge. Figure 2 illustrates the case where the bandwidth is equal to half the matrix size with 7 non empty diagonals.

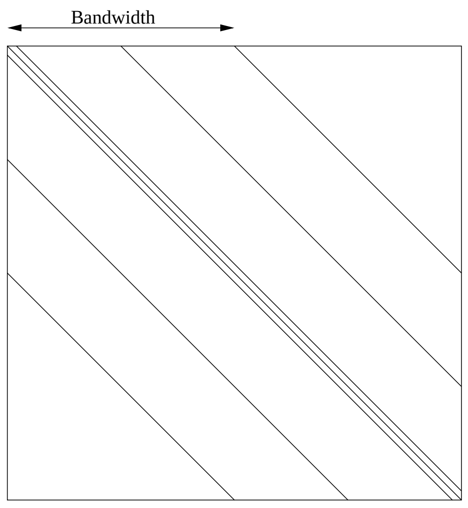

Fig. 2. Example of a generated matrix

In the following experiments we mention the size of the matrix, its parameters, the sequential solver used, the execution mode (synchronous or asynchronous), the execution time in seconds, the number of iterations and the number of processors used with their location and characteristics. It should be noticed that the number of iterations in the synchronous mode is always constant whereas it is not the case in the asynchronous mode since it varies from an execution to another and according to the power of the processor if they are heterogeneous. That is why, in this case, we report an interval with the minimum and the maximum number of iterations according to the execution and the processors. All reported execution times are the mean value of a series of four executions. Finally, it should be noticed that we do not experiment matrices with a too small spectral radius because in this case, the number of iterations would be smaller. So we choose quite complex matrices.

In our experiments, we did not change the default parameters of the MUMPS and SuperLU package. With SparseLib we have chosen the GMRES method [29] with an ILU preconditioner. We remind the fact that the three solvers are used in their sequential version.

Because GRID'5000 is an experiment platform and not a production platform, we cannot reserve nodes during a long time. It is often very difficult to have a large number of processors for more than 4 hours a day, because a large number of scientists use it. Consequently, it requires tremendous endeavors before experimenting. Moreover, in order to have a uniform grid context, users are advised to deploy their environment on the reserved nodes before running a computation. We have always used this method in order to deploy CRAC because without using it, linux distributions are heterogeneous from one site to another. Since it is not possible to reserve the same nodes from one day to another, GRID'5000 represents a very interesting grid computing platform but it has the drawback that experiments are very difficulty reproducible. Moreover, even though other users cannot used reserved nodes, the bandwidth of the network is varying that is why, in the following, some execution times may be slightly surprising.

In the following subsections we highlight some interesting issues that we have experimented.

4.2 Comparison of local and distant executions

In this first series of experiments, we wanted to compare the execution times of our solver using only local nodes and using the same number of nodes with distant sites. To achieve this, we have used a generated matrix of size $10,000,000$ with 70 processors located either in Sophia (AMD 246 2GHz) for the local case or 30 in Orsay (20 AMD 246 2GHz and 10 AMD 250 2.4GHz), 20 in Lille (AMD 248 2.2GHz) and 20 in Sophia (AMD 246 2GHz) for the distant one. Table 1 shows the execution times with the local cluster in Sophia. In this table we can remark that with large bandwidth matrices, GREMLINS is more efficient in the asynchronous mode than in the synchronous one. This can be explained by the fact that the larger the bandwidth is, the more communications are required with the more neighbors. With a smaller bandwidth, the synchronous version is faster. It can also be noticed that the number of iterations required to reach the convergence in the asynchronous mode is always greater than for the synchronous mode. This remark will always be true in the following.

Table 2 shows the execution times of the same matrices as in Table 1 with the same number of nodes but located in three sites. The execution times

Solver	Synchronous		Asynchronous	
	exec. time (s)	nb. iter.	exec. time (s)	nb. iter.
13 diagonals, bandwidth: $5,000,000$				
SparseLib	88.69	142	57.42	[207-296]
MUMPS	98.73	142	70.39	[198-280]
SuperLU	80.23	142	49.00	[241-365]
13 diagonals, bandwidth: $1,000,000$				
SparseLib	79.89	125	57.60	[182-247]
MUMPS	98.33	125	69.75	[174-237]
SuperLU	72.87	125	50.92	[183-255]
13 diagonals, bandwidth: $100,000$				
SparseLib	39.19	51	48.01	[57-75]
MUMPS	15.45	51	19.81	[65-106]
SuperLU	12.40	51	15.21	[71-111]

Table 1
Execution times of the three solvers with a generated matrix of size $10,000,000$ with 70 machines in a local cluster (Sophia).

are higher than in a local context. This is not surprising since distant communications take more time than local ones. With large bandwidth matrices execution times are much longer and the comparison between a local running and a distant one may seem unrelevant. In this case, using a distant cluster is only limited to solve large matrices that cannot be solved using a local cluster.

Nevertheless, the asynchronous version is more robust to distant communications than the synchronous one. This is due to the implicit overlapping of communication by computation inherent to the asynchronous model. However, with smaller bandwidth matrices the behavior of the solver between a local and a distant running is more comparable. The ratio between the local and the distant running is bounded by three although the smallest machines in the distant configuration are the same than in the local one. Another issue is that whatever the sequential solver used for our experiments is, the execution times are relatively similar for this size of matrix.

To sum up this first series of experiments, the ratio between the computation time and the communication time is very important. With those matrices, when the bandwidth increases, the computation time is quite of the same order but the communication time increases drastically. That is why with large bandwidth matrices, GREMLINS is less efficient in a distant context.

Solver	Synchronous		Asynchronous	
	exec. time (s)	nb. iter.	exec. time (s)	nb. iter.
13 diagonals, bandwidth: $5,000,000$				
SparseLib	1,340.12	142	770.62	[1,821-2,354]
MUMPS	1,178.56	142	741.65	[1,582-2,101]
SuperLU	1,109.12	142	736.63	[1,782-2,095]
13 diagonals, bandwidth: $1,000,000$				
SparseLib	1,244.25	125	517.69	[1,876-2,320]
MUMPS	1,318.63	125	512.32	[2,019-2,764]
SuperLU	1,298.71	125	506.76	[2,102-2,908]
13 diagonals, bandwidth: $100,000$				
SparseLib	83.97	51	48.73	[65-86]
MUMPS	60.48	51	42.29	[178-279]
SuperLU	62.35	51	46.56	[283-422]

Table 2
Execution times of the three solvers with a generated matrix of size $10,000,000$ with 70 machines located in 3 sites (30 in Orsay, 20 in Lille and 20 in Sophia).

But it should be noticed that any other solvers would have the same caveat, probably with a stronger effect due to the amount of communications and synchronizations. From a general point of view, the higher this ratio between the computation times and the communication times is, the more the synchronous version is favored compared to the asynchronous one and reciprocally. That is why when executing an algorithm in a local cluster, the synchronous version may be faster than the asynchronous one. Conversely, executing the same algorithm within a grid context, where communication performances are worst, the communication time would be longer and the ratio would often decrease in favor of the asynchronism.

4.3 Impact of the computation amount

In the previous series of experiments we have shown that the communication time is very influential on the execution time especially in a distant context. In this second series of experiments we want to study how the execution time is affected when the computation amount increases. In Table 3, we report the experiments with different sizes of matrices. For each one, the bandwidth is quite small, which ensures that the execution times will not be too long.

For those experiments we have used 120 machines scattered in 4 sites (40 in Rennes (AMD 248 2.2GHz), 40 in Orsay (20 AMD 246 2GHz and 20 AMD 250 2.4GHz), 25 in Nancy (AMD 248 2.2GHz) and 15 in Lille (AMD 248 2.2GHz)). As our goal was not to compare the performances of the different sequential solvers, we have chosen the MUMPS solver for those experiments.

Size of the matrix	Number of diagonals	Bandwidth	Synchronous exec. time (s)	Synchronous nb. iter.	Asynchronous exec. time (s)	Asynchronous nb. iter.
1,000,000	23	1,000	10.05	72	4.55	[303-475]
2,000,000	23	2,000	14.98	69	5.39	[195-229]
4,000,000	23	4,000	19.33	68	12.31	[204-268]
6,000,000	23	6,000	24.19	69	13.34	[146-176]
8,000,000	23	8,000	27.87	68	18.18	[142-143]
10,000,000	23	5,000	28.10	67	22.22	[136-144]

Table 3
Execution times of our solver coupled with MUMPS solver on different generated matrices with 120 machines located in 4 sites (40 in Rennes, 40 in Orsay, 25 in Nancy and 15 in Lille).

Those experiments emphasize that the smaller the size of the matrix is, the more efficient the asynchronous version is, compared to the synchronous one. This is easily understandable since the computation amount increases with the size of the matrix. So the ratio between the computation time and the communication times decreases and communications are less penalizing. The number of iterations to reach the convergence in the asynchronous version clearly shows this point, since this number is larger with matrices of small sizes. When the computation amount becomes more important, the difference between the synchronous and the asynchronous version decreases. It should be noticed that the network bandwidth between the different sites of the GRID'5000 architecture is very important compared to traditional bandwidth networks. So, with the GRID'5000 platform the ratio between the computation time and the communication time for which algorithms are efficient is very different from more traditional grid environment with low bandwidth networks.

4.4 Larger experiments

In a third series of experiments we have tried to reserve a larger number of processors in order to measure the scalability of GREMLINS.

In Table 4 we report three configurations for which we have compared the

Solver	Size of the matrix	Number of diagonals	Bandwidth	Synchronous		Asynchronous	
				exec. time (s)	nb. iter.	exec. time (s)	nb. iter.
MUMPS	10,000,000	13	5,000	16.41	30	10.98	[138-159]
SuperLU	10,000,000	13	5,000	19.83	30	14.77	[85-109]
MUMPS	20,000,000	13	5,000	15.91	22	15.48	[74-79]

Table 4
Execution times of our solver coupled with the MUMPS solver on generated matrices with 190 machines located in 5 sites (30 in Rennes, 30 In Sophia, 70 Orsay, 30 Lyon and 30 Lille).

execution times of the two execution modes of our solver with 190 machines scattered into 5 sites (30 in Rennes (AMD 248 2.2GHz), 30 in Sophia (AMD 246 2GHz), 70 in Orsay (40 AMD 246 2GHz and 30 AMD 250 2.4GHz), 30 in Lyon (AMD 246 2GHz) and 30 in Lille (AMD 248 2.2GHz). It results from those experiments that the two different direct solvers that GREMLINS can use, have a quite similar behavior. With a size equal to $20,000,000$, for this experiment, the synchronous and the asynchronous version require about the same times. This is because the ratio between the computation time and communication time is not in favor of one version or the other. For the matrices of Table 4, we do not choose the same values for the diagonal elements, that is why the number of iterations in the synchronous case is different for the first matrix (of degree $10,000,000$) and the last one (of degree $20,000,000$).

In all our previous experiments, we only studied executions with one computation thread by machines. Let us remind the interested reader that machines in the GRID'5000 architecture are at least bi-processors. As mentioned, CRAC is a multithreaded library, it is possible with the multisplitting algorithm to run more than one computation task by machine. In fact, as soon as the computation task is thread safe, i.e., it supports to be executed by multiple threads, our solver can be executed with multiple computation tasks. Unfortunately, two of the three internal solvers used in GREMLINS are not thread-safe (MUMPS and SuperLU), that is why we only experimented it with SparseLib.

As for this experiment we have only taken two sites and we have chosen matrices with larger bandwidths. Table 5 shows the results. Using only two distant sites, we can remark that the asynchronous version is still faster than the synchronous one. GREMLINS simply allows to use multi-processor machines as soon as the sequential linear solver used is thread safe. Hence, it is easy to increase the computing power without doing anything.

Number of diagonals	Bandwidth	Synchronous		Asynchronous	
		exec. time (s)	nb. iter.	exec. time (s)	nb. iter.
13	300,000	132.51	134	87.14	[634-859]
23	300,000	163.88	141	104.37	[576-809]
13	3,000,000	353.80	142	245.68	[980-1279]

Table 5
Execution times of our solver coupled with the SparseLib solver on generated matrices of size $30,000,000$ with 200 bi-processors located in 2 sites (120 in Orsay, 80 in Sophia), so 400 cpu.

4.5 Experiments with a matrix issued from an advection-diffusion model

Experiments with the advec-diffu matrix are synthetized in Table 6. With this matrix issued from a real application, we can observe that the ratio between the synchronous and the asynchronous version is of the same order as that of the generated matrices. Although SuperLU is slower than the other solvers in the synchronous case, it is equivalent to SparseLib in the asynchronous mode. This can be explained by the fact that the number of iterations is more important than in previous examples, so after the factorization step, a direct solver has less work to do at each iteration than an iterative one.

Solver	Synchronous		Asynchronous	
	exec. time (s)	nb. iter.	exec. time (s)	nb. iter.
MUMPS	54.03	146	39.89	[293-354]
SuperLU	92.01	146	58.95	[259-312]
SparseLib	76.11	146	58.09	[250-291]

Table 6
Execution times of our solver with the advec-diffu matrix with 90 machines located in 3 sites (30 in Rennes, 30 in Sophia and 30 in Nancy).

5 Conclusion

In this paper, we have described the GREMLINS solver. It is based on the multisplitting algorithm which is iterative. It is suited for grid computing context composed of distant sites in which communications have worse performances than in a traditional local cluster. Our solver uses a sequential linear solver on each processor to solve a local linear solver obtained by the decomposition provided by the method. This sequential solver may be either a direct or an iterative one. Moreover, our solver is able to be executed either in the synchronous mode or in the asynchronous mode. In the latter case, communications are overlapped by computation, which may result in a faster execution times.

Our solver is built with CRAC, a multi-threaded library suited for asynchronous iterative computation. This library offers similar performances of MPI in the synchronous case, and allows us, without any modification of the code, to execute either a program in a synchronous or an asynchronous communication mode.

We have experimented our solver with the GRID'5000 architecture in France in order to study its behavior in various contexts of execution. We have used generated matrices in order to have the desired size with the desired form and a real matrix issued from a PDE advection diffusion problem. It follows from these experiments that the synchronous version is quite efficient even in such a context of execution with distant sites. The asynchronous version is often faster. With the SparseLib solver which is a thread-safe library, it is even possible to use multiple computation tasks per machine.

We plan to study how to adapt and use preconditioners to our solver in order to widen the spectral of usable matrices. Using a direct sequential solver, the crucial point lies in the load balancing in the factorization part, because this is often a consuming part. So it would be interesting to study how to use a load balancing algorithm if the partitioning process provides local submatrices for which the factorization process is completely unbalanced.

6 Acknowledgement

The authors sincerely wish to thank the reviewers for their constructive comments.

References

[1] A. Gupta, Recent advances in direct methods for solving unsymmetric sparse systems of linear equations, ACM Transactions on Mathematical Software 28 (3) (2002) 301–324.

[2] C. Engelmann, A. Geist, Super-scalable algorithms for computing on 100, 000 processors, in: International Conference on Computational Science, Vol. 3514

of LNCS, Springer, 2005, pp. 313–321.

[3] M. Garbey, D. Tromeur-Dervout, On some Aitken-like acceleration of the Schwarz method, International journal for numerial methods in fluids 40 (1) (2002) 1493–1513.

[4] D. Tromeur-Dervout, Y. Vassilevski, Choice of initial guess in iterative solution of series of systems arising in fluid flow simulations, J. Comput. Phys. 219 (1) (2006) 210–227.

[5] F. Manne, T. Sørevik, Partitioning an array onto a mesh of processors, in: PARA, 1996, pp. 467–477.

[6] O. Beaumont, A. Legrand, F. Rastello, Y. Robert, Static LU decomposition on heterogeneous platforms, The International Journal of High Performance Computing Applications 15 (3) (2001) 310–323.

[7] L. Grigori, X. S. Li, A New Scheduling Algorithm for Parallel Sparse LU Fac torization with Static Pivoting, in: Super Computing 2002, IEEE computer society press and ACM sigarch, 2002, paper 139 on CD.

[8] D. P. O'Leary, R. E. White, Multi-splittings of matrices and parallel solution of linear systems, Journal on Algebraic and Discrete Mathematic 6 (1985) 630–640.

[9] R. E. White, Multisplitting of a symmetric positive definite matrix, SIAM Journal on Matrix Analysis and Applications 11 (1990) 69–82.

[10] P. Spitéri, J.-C. Miellou, D. El Baz, Parallel asynchronous Schwarz and multisplitting methods for a nonlinear diffusion problem, Numerical Algorithms 33 (1–4) (2003) 461–474.

[11] X. S. Li, J. W. Demmel, SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems, ACM Transactions on Mathematical Software 29 (2) (2003) 110–140.

[12] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, S. Pralet, Hybrid scheduling for the parallel solution of linear systems, Parallel Computing 32 (2) (2006) 136–156.

[13] J. Dongarra, A. Lumsdaine, R. Pozo, K. Remington, A sparse matrix library in c++ for high performance architectures, in: Second Object Oriented Numerics Conference, 1994, pp. 214–218.

[14] J. M. Bahi, S. Contassot-Vivier, R. Couturier, Performance comparison of parallel programming environments for implementing AIAC algorithms, Journal of Supercomputing 35 (3) (2006) 227–244.

[15] R. Couturier, S. Domas, Crac: a grid environment to solve scientific applications with asynchronous iterative algorithms, Tech. rep., LIFC - Université de Franche-Comté, http://info.iut-bm.univ-fcomte.fr/staff/couturie/crac.pdf (2006).

[16] J. M. Bahi, S. Contassot-Vivier, R. Couturier, Evaluation of the asynchronous iterative algorithms in the context of distant heterogeneous clusters, Parallel Computing 31 (5) (2005) 439–461.

[17] J. M. Bahi, S. Contassot-Vivier, R. Couturier, F. Vernier, A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms, IEEE Transactions on Parallel and Distributed Systems 1 (2005) 4–13.

[18] D. P. Bertsekas, J. N. Tsitsiklis, Parallel and Distributed Computation: Numerical Methods, Prentice Hall, Englewood Cliffs NJ, 1989.

[19] D. El Baz, P. Spitéri, J.-C. Miellou, D. Gazen, Asynchronous iterative algorithms with flexible communication for non linear network problems, Journal of Parallel and Distributed Computing 38 (1996) 1–15.

[20] D. El Baz, A. Frommer, P. Spitéri, Asynchronous iterations with flexible communication: contracting operators, J. Comput. Appl. Math. 176 (1) (2005) 91–103.

[21] S. Contassot-Vivier, R. Couturier, C. Denis, F. Jézéquel, Efficiently solving large sparse linear systems on a distributed and heterogeneous grid by using the multisplitting-direct method, in: 4th International Workshop on Parallel Matrix Algorithms and Applications, PMAA'06, 2006, pp. 21–22.

[22] C. Denis, J.-P. Boufflet, P. Breitkopf, M. Vayssade, B. Glut, Load balancing issues for a multiple front method, in: International Conference on Computational Science, LNCS 3037, 2004, pp. 163–170.

[23] C. Denis, J.-P. Boufflet, P. Breitkopf, A load balancing method for a parallel application based on a domain decomposition, in: 19th IEEE and ACM Int. Parallel and Distributed Processing Symposium, IPDPS 2005, 2005, p. 8 pages.

[24] J. Bahi, R. Couturier, Parallelization of direct algorithms using multisplitting methods in grid environm ents, in: IPDPS 2005, IEEE Computer Society Press, 2005, pp. 254b, 8 pages.

[25] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, I. Touche, Grid'5000: A large scale and highly reconfigurable experimental grid testbed, International Journal of High Performance Computing Applications 20 (4) (2006) 481–494.

[26] J. G. Verwer, J. G. Blom, W. Hundsdorfer, An implicit-explicit approach for atmospheric transport-chemistry problems, Appl. Numer. Math. 20 (1-2) (1996) 191–209.

[27] J. Bahi, R. Couturier, K. Mazouzi, M. Salomon, Synchronous and asynchronous solution of a 3d transport model in a grid computing environment, Applied Mathematical Modelling 30 (7) (2006) 616–628.
URL `http://dx.doi.org/10.1016/j.apm.2005.06.017`

[28] J. M. Bahi, J.-C. Miellou, K. Rhofir, Asynchronous multisplitting methods for nonlinear fixed point problems, Numerical Algorithms 15 (3,4) (1997) 315–345.

[29] Y. Saad, Iterative Methods for Sparse Linear Systems, PWS Publishing, New York, 1996.