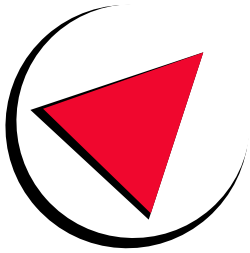

L I F C

LABORATOIRE D'INFORMATIQUE DE L'UNIVERSITE DE FRANCHE-COMTE

EA 4157

Solving large sparse linear systems in a grid environment : the GREMLINS code versus the PETSc library

Fabienne Jézéquel — Raphaël Couturier — Christophe Denis

Rapport de Recherche n° RR 2008–06

THÈME 3 – Juillet 2008

Solving large sparse linear systems in a grid environment : the GREMLINS code versus the PETSc library

Fabienne Jézéquel , Raphaël Couturier , Christophe Denis

Thème 3

Algorithmique Numérique Distribuée

Rapport de Recherche

Juillet 2008

Abstract: Solving large sparse linear systems is essential in numerous scientific domains. Several algorithms, based on direct or iterative methods, have been developed for parallel architectures. On distributed grids consisting of processors located in distant geographical sites, their performances may be unsatisfactory because they suffer from too many synchronizations and communications. The GREMLINS code has been developed for solving large sparse linear systems on distributed grids. It implements the multisplitting method that consists in splitting the original linear system into several subsystems that can be solved independently. In this article, the performances of the GREMLINS code obtained with several libraries for solving the linear subsystems are analysed. Its performances are also compared with those of the widely used PETSc library, that enables one to develop portable parallel applications. Numerical experiments have been carried out both on local clusters and on distributed grids.

Key-words: sparse linear solver, iterative method, asynchronous iteration, multisplitting method, grid computing

Laboratoire d'Informatique de l'Université de Franche-Comté,
Antenne de Belfort — IUT Belfort-Montbéliard,
rue Engel Gros, BP 527, 90016 Belfort Cedex (France)
Téléphone : +33 (0)3 84 58 77 86 — Télécopie : +33 (0)3 84 58 77 32

Résolution de systèmes linéaires creux de grande taille dans un environnement de grilles de calcul : comparaison des bibliothèques GREMLINS et PETSc

Résumé :

La résolution de systèmes linéaires creux de grande taille est essentielle à de nombreux domaines scientifiques. Plusieurs algorithmes, basés sur des méthodes directes ou itératives, ont été développées pour des architectures parallèles. Sur les grilles distribuées, constituées de processeurs localisés dans des sites géographiques distants, leur performances peuvent être décevantes en raison des synchronisations et des communications trop fréquentes. Le logiciel GREMLINS a été conçu pour résoudre des systèmes linéaires creux de grande taille sur des grilles distribuées. Il est basé sur la méthode de multidécomposition qui consiste à découper le système linéaire original en plusieurs sous systèmes qui peuvent être résolu de manière indépendante. Dans cet article, nous analysons les performances du code GREMLINS avec plusieurs bibliothèques pour résoudre les sous-systèmes linéaires. Nous comparons également les performances de GREMLINS avec la bibliothèque PETSc qui permet de développer des applications parallèles portables. De nombreuses expérimentations ont été menées sur des clusteurs locaux et des grilles distribuées.

Mots-clés : résolution de systèmes linéaires creux, méthode itérative, itération asynchrone, méthode de multidécomposition, calcul sur grille

Laboratoire d'Informatique de l'Université de Franche-Comté,
Antenne de Belfort — IUT Belfort-Montbéliard,
rue Engel Gros, BP 527, 90016 Belfort Cedex (France)
Téléphone : +33 (0)3 84 58 77 86 — Télécopie : +33 (0)3 84 58 77 32

Solving large sparse linear systems in a grid environment : the GREMLINS code versus the PETSc library

Fabienne Jézéquel[a] , Raphaël Couturier[b] , Christophe Denis[a,c]
[a] Laboratoire d'Informatique de Paris 6
Université Pierre et Marie Curie - Paris 6
4 place Jussieu, 75252 Paris CEDEX 05 - France
email : {Christophe.Denis,Fabienne.Jezequel}@lip6.fr
[b] Laboratoire d'Informatique de l'Université de Franche-Comté
BP 527, 90016 Belfort CEDEX - France
email : Raphael.Couturier@univ-fcomte.fr
[c] School of Electronics, Electrical Engineering & Computer Science
The Queen's University of Belfast, Belfast BT7 1NN, U.K.

7 septembre 2008

Solving large sparse linear systems is essential in numerous scientific domains. Several algorithms, based on direct or iterative methods, have been developed for parallel architectures. On distributed grids consisting of processors located in distant geographical sites, their performances may be unsatisfactory because they suffer from too many synchronizations and communications. The GREMLINS code has been developed for solving large sparse linear systems on distributed grids. It implements the multisplitting method that consists in splitting the original linear system into several subsystems that can be solved independently. In this article, the performances of the GREMLINS code obtained with several libraries for solving the linear subsystems are analysed. Its performances are also compared with those of the widely used PETSc library, that enables one to develop portable parallel applications. Numerical experiments have been carried out both on local clusters and on distributed grids.

1 Introduction

Numerous scientific applications imply to solve a large sparse linear system. Because of large requirements in terms of memory allocation and execution time, it may happen that this computation cannot be carried out on a single-processor computer. Several multi-processor environments exist,

such as parallel machines or clusters of computers. A distributed grid may be defined as a set of connected local clusters. The large number of processors it offers may be a relatively cheap answer to growing computational needs. Because of the variety of machines and interconnection networks it is usually composed of, a distributed grid is a heterogeneous environment. Because the performances of numerical algorithms designed to run on parallel homogeneous computers may be unsatisfactory on such a grid, new coarse grained and asynchronous efficient parallel algorithms must be proposed.

The GREMLINS code has been developed for efficiently solving large sparse linear systems on a distributed grid [9]. It implements the multisplitting method [13, 16], based on a decomposition of the matrix into band submatrices. Each processor belonging to the grid solves linear subsystems using a direct or an iterative method. Successive approximations to the global solution are computed. These iterations can be performed in a synchronous or in an asynchronous mode. With the first version of the GREMLINS code, the linear subsystems could be solved using direct methods from the MUMPS [1] or the SuperLU [12] library or using iterative methods from the SparseLib [11] library. The PETSc library is a popular suite of data structures and routines for scientific computing [6]. Applications developed with PETSc are portable : a common code can be run an a sequential machine or on various parallel architectures. PETSc employs the MPI standard for all message-passing communication. By paying particular attention to memory allocation, PETSc takes fully advantage of parallel machines. For solving linear systems, it enables to use various iterative methods and also direct methods from external libraries.

The originality of this article lies in two types of works. Firstly, the GREMLINS code has been improved in order to allow each processor in a distributed grid to use PETSc for solving its linear subsystems. Secondly, the performances of the PETSc library for solving large linear systems have been compared with those of the GREMLINS code, both on a local cluster and on a grid consisting of processors from several geographical sites. The article is organized as follows. The principles of the multisplitting method and the architecture of the GREMLINS code are presented in Section 2. Numerical experiments are described in Section 3. Firstly, the performances of the GREMLINS code have been analysed, several possible libraries being used to solve serially the linear subsystems generated by the multisplitting method. Then the performances of the PETSc library and the GREMLINS code have been compared. Both numerical experiments have been carried out in a local and in a distant context. Section 4 presents concluding remarks and planed perspectives.

2 The multisplitting method

2.1 Principles of the multisplitting method

For solving a linear system, the multisplitting method generalizes the block Jacobi method. Nevertheless the multisplitting method supports the asynchronous iteration model, it can be used with direct and/or iterative inner solvers (even simultaneously) and it allows processors to compute common components by mixing freely overlapped components between processors. Its main principles are described here.

Let us consider the $n \times n$ non-symmetric sparse linear system

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

and let us assume it has a unique solution. The multisplitting method consists in splitting the matrix into horizontal band matrices. For the sake of simplicity, let us consider the decomposition generates as many band matrices as processors. Thus each processor is in charge of managing a submatrix, denoted by $ASub$. The part of the band matrix before the submatrix represents the left dependencies, called $DepLeft$, and the part after the submatrix represents the right dependencies, called $DepRight$. Let us denote by $XSub$ the part of the solution vector and $BSub$ the part of the right-hand-side vector involved in the computation. Figure 1 describes the decomposition of A, x and b into several parts ($DepLeft$, $ASub$, $DepRight$, $Xleft$, $XSub$, $XRight$, $BSub$) required locally by a processor.

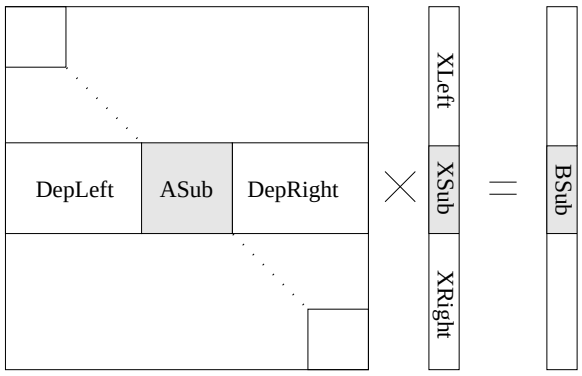

FIG. 1 – Decomposition of the matrix A, the solution vector x and the right-hand-side vector b into several parts required locally by a processor

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

$$ASub * XSub = BSub - DepLeft * XLeft - DepRight * XRight. \tag{2}$$

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

Solving a linear system using the multisplitting method requires several steps described below.

1. **Initialization**
   The matrix can be loaded from a data file or generated at run time. Each processor manages the load of the band matrix $DepLeft+ASub+DepRight$. Then until convergence, each processor iterates on :
2. **Computation**
   At each iteration, each processor computes $BLoc = BSub - DepLeft * XLeft - DepRight * XRight$. Then, it solves the linear system $ASub * XSub = BLoc$.
3. **Data exchange**
   Each processor sends $XSub$, the part of the solution vector it has computed, to the other processors. When a processor receives a part of the solution vector from another processor, it should update the appropriate part of $XLeft$ or $XRight$ according to the rank of the sending processor.
4. **Convergence detection**
   Convergence can be detected using a centralized algorithm described in [3] or a decentralized one, that is a more general version, as described in [4].

In the multisplitting method, asynchronous iterations may reduce the run time. In this case, receptions are non blocking, computations are dissociated from communications using threads and an appropriate convergence algorithm is used. For more references on theoretical works concerning asynchronous iterative algorithms, interested readers are invited to read [7].

The serial solver used for the linear subsystems can be a direct one or an iterative one. With a direct solver, the most consuming part is the factorization of the submatrix, that is performed at the first iteration only. Then other iterations are faster, because only the right-hand-side changes. With an iterative solver, all the iterations require approximatively the same time.

The number of iterations required to solve the system is related to the spectral radius of the iteration matrix : the closer the spectral radius is to 1, the more iterations are required, as for all iterative methods. The convergence condition in the asynchronous version is more restrictive than in the synchronous one [2]. In some rare practical cases, the synchronous version would converge whereas the asynchronous one would not.

As a remark, some elements of the solution vector may be computed by several processors. This overlapping may reduce the number of iterations required to obtain the convergence. In [2], the authors have shown an example of the impact of overlapping over the speed of convergence.

2.2 The GREMLINS code

The GREMLINS code implements in C++ the multisplitting method for solving non-symmetric sparse linear systems. It uses the CRAC library [10] for communication. Depending on a flag set by the user in the GREMLINS code, communications with CRAC can be synchronous or asynchronous. Although the internals of CRAC are based on multithreading, the CRAC programming interface uses the message passing paradigm. CRAC basically has three functionalities : send a message, 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 sending method is called. The receiving method is blocking in the synchronous mode, whereas it is not in the asynchronous mode. In the latter case, if one or several versions of a message arrived, the method returns its last version, otherwise it returns nothing. The convergence method requires a boolean argument indicating if local convergence has been achieved and determines if global convergence has been reached using a centralized algorithm.

With the multisplitting method, the initial linear system is split into subsystems. Each subsystem is solved on one processor. In the previous version of the GREMLINS code [9], three scientific libraries could be chosen for solving the subsystems : MUMPS [1], SparseLib [11] and SuperLU [12]. The GREMLINS code has been improved to allow the use of the PETSc library [6] also.

The GREMLINS code consists of :

– C++ methods that first ensure the distribution of the matrix and then, in an iterative process, compute the right-hand-sides, send them to the different processors, receive the solutions from the different processors and detect the convergence. These methods implement an iterative so-called *outer* solver and use the CRAC library for communication.

– C++ methods using a scientific library that solve the subsystems in a serial way. These methods implement a sequential so-called *inner* solver, that can be direct or iterative, depending on the library chosen. For each library, the inner solver consists of at most two methods : a constructor (that classically performs initializations in object oriented programming models and is not necessarily present) and a method called *solve* that actually computes the solution of the linear system.

With the outer solver, the initial matrix and the submatrices are represented in a CSR (Compressed Sparse Row) format that consists in three arrays : one for the column indices, one for the numerical values and one for the positions in the previous arrays of the first entry in each row. The right-hand-side and the solution of each subsystem are represented by classical numerical arrays.

With the inner solver, the respresentation for the matrix, the right-hand-side and the solution depends on the library used. If necessary, the matrix

is converted from the CSR format previoulsy described to another format required by the library. This conversion is performed once, in the constructor of the inner solver. In the *solve* method, the right-hand-side and the solution may also be converted if particular types are required for these two arrays.

3 Numerical experiments

3.1 Context of the experiments

With the GREMLINS code, the linear system to be solved can be either generated at run time or assigned from a file. In the latter case, the file is stored on a processor which is in charge of distributing the data to the others. Therefore the memory size of this processor limits the size of the file to be processed. In the numerical experiments described in this section, for the matrices to be relatively large, they are generated at run time.

Each processor computes specific matrix rows, such that each matrix is automatically distributed on the processors. Each generated matrix consists of several non-empty diagonals : the main diagonal, the two nearest neighbor diagonals and other diagonals equitably scattered between the main diagonal and the desired bandwidth. As an example, a matrix with 7 non-empty diagonals and a bandwidth equal to half the matrix size is represented in Figure 2. Off-diagonal entries are random values between -1 and 0. Each diagonal entry is the inverse of the sum of the entries of the same row plus a random value from an interval specified by the user. Such generated matrices are M-matrices [5] (defined as Z-matrices with eigenvalues whose real parts are positive). Z-matrices, *i.e.* matrices whose off-diagonal entries are non-positive, and also diagonally dominant matrices satisfy the convergence condition of both the synchronous and the asynchronous version of the multisplitting method [2].

Numerical experiments have been carried out on GRID'5000[1], an experimental grid platform that aims at featuring a total of 5000 processors and gathers 9 sites geographically distributed in France [8]. Most of those sites have a Gigabit Ethernet network for local machines. Links between the different sites range from 2.5 Gb/s to 10 Gb/s. Processors in the platform are mostly AMD Opteron, but also Intel Xeon and Intel Itanium.

To run a code on the GRID'5000 platform, processors have to be reserved. The choice of the sites and the number of processors used depends on the ressources availability in the grid. Because clusters in the GRID'5000 architecture use different operating systems and libraries, a common Linux image has been deployed on the nodes reserved for the experiments described in this section. Thus the same operating system, libraries and compilers could be available on any site.

[1]URL address : http ://www.grid5000.fr

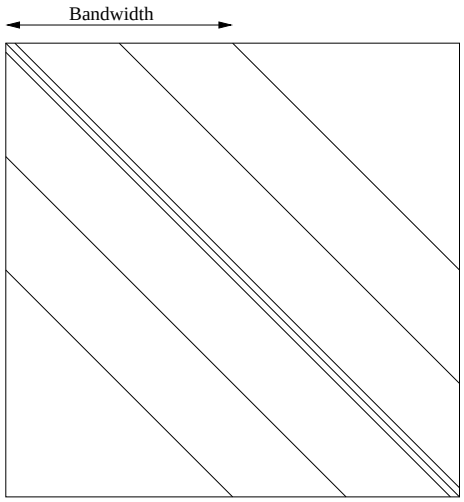

FIG. 2 – A generated matrix with 7 non-empty diagonals and a bandwidth equal to half the matrix size

Two types of experiments are described. Firstly, the performances obtained using different libraries to solve the subsystems in the multisplitting method are compared. Secondly, the multisplitting method is compared with the GMRES method [15] implemented in the PETSc library. This second point provides a comparison of our solver with a standard parallel one.

3.2 Comparison of different inner solvers in the GREMLINS code

Different inner solvers for the subsystems in the multisplitting method have been compared : direct solvers from the MUMPS or the SuperLU library and iterative solvers from the PETSc or the SparseLib library. With the latter libraries, the GMRES method has been used with an ILU preconditioner.

Table 1 presents results measured in a local context : 100 processors with a frequency of 2.4 GHz in Orsay. The results presented in Table 2 have been measured with 155 processors in a distant context : 59 processors in Rennes, 50 processors in Sophia and 46 processors in Toulouse, having a frequency of respectively 2.0 GHz, 2.0 GHz and 2.6 GHz. With multicore processors, one core per processor has been used, because inner solvers are not thread safe except SparseLib.

The matrices involved in Tables 1 and 2 have the same size (2.10^7), the same bandwidth (2.10^6) and the same number of diagonals (13, 23 or 33) but their elements have different values. These values result from a combination of random values and parameters that are set by the user and have an impact

on the convergence speed of the multisplitting method. Indeed the number of iterations required to achieve convergence and therefore the execution time of the GREMLINS code is related to the spectral radius of the iteration matrix in the multisplitting method. Although they have the same pattern, the matrices from Tables 1 and 2 have been generated using different parameters. In the synchronous mode, a matrix from Table 2 would require less iterations and therefore lead to a faster convergence than the corresponding one from Table 1 in the same context (local or distant processors).

The run time and the number of iterations performed by the outer solver in the multisplitting method, in the synchronous mode and in the asynchronous one, are reported in both Tables. In each case, the code has been run several times. The run time that is reported is actually the mean value of the different run times measured. In the synchronous mode, the number of iterations is constant from one execution to another. It is not the case in the asynchronous mode, for which the number of iterations performed depends on the network traffic ; the minimum and the maximum number of iterations measured have been indicated. In the asynchronous mode, within one execution, the number of iterations also varies from one processor to another. It is the number of iterations performed by the supermaster, a processor that has a specific function for communications with the CRAC library [10], that has been measured.

Solver	Synchronous		Asynchronous	
	time (s)	nb. iter.	time (s)	nb. iter.
13 diagonals				
MUMPS	98.79	83	93.79	[240-249]
SuperLU	84.09	83	98.07	[417-441]
SparseLib	87.21	83	91.68	[388-426]
PETSc	84.14	83	95.70	[424-457]
23 diagonals				
MUMPS	278.43	148	258.98	[421-439]
SuperLU	253.71	148	248.57	[506-532]
SparseLib	272.39	148	259.32	[441-451]
PETSc	270.04	148	255.46	[411-414]
33 diagonals				
MUMPS	407.06	205	376.94	[556-574]
SuperLU	367.49	205	351.04	[714-747]
SparseLib	394.02	205	364.86	[604-608]
PETSc	398.23	205	369.91	[527-566]

Tab. 1 – Execution times with the four solvers for generated matrices of size 2.10^7 and bandwidth 2.10^6 on 100 processors in a local cluster in Orsay.

Solver	Synchronous		Asynchronous	
	time (s)	nb. iter.	time (s)	nb. iter.
13 diagonals				
MUMPS	25.15	12	42.09	[199-215]
SuperLU	23.42	12	44.15	[510-526]
SparseLib	23.00	12	32.67	[272-310]
PETSc	23.57	12	40.89	[322-453]
23 diagonals				
MUMPS	57.00	17	54.35	[170-188]
SuperLU	55.45	17	51.40	[333-389]
SparseLib	54.88	17	54.82	[302-330]
PETSc	55.33	17	53.87	[322-369]
33 diagonals				
MUMPS	83.88	21	75.16	[191-199]
SuperLU	78.54	21	66.58	[344-359]
SparseLib	79.83	21	70.44	[230-255]
PETSc	79.12	21	71.70	[203-218]

TAB. 2 – Execution times with the four solvers for generated matrices of size 2.10^7 and bandwidth 2.10^6 on 155 processors : 59 in Rennes, 50 in Sophia and 46 in Toulouse.

With the matrices considered, both in the local context and in the distant one, one can notice that the performances obtained with the four inner solvers are similar. In the synchronous mode, the number of iterations performed by the outer solver is the same whatever the inner solver is. As the number of diagonals increases, the computational volume increases, the number of iterations in the synchronous mode increases and so does the run time both in the synchronous mode and in the asynchronous one. Performances in the asynchronous mode are slightly better than in the synchronous one from a certain number of diagonals.

3.3 Comparison of the GREMLINS code and the PETSc library

The multisplitting method implemented in the GREMLINS code has been compared with the GMRES method implemented in the PETSc library, both in a local and in a distant context. The inner solver used in the multisplitting method is a direct one from the MUMPS library. As mentioned in 3.2, this choice has no significant impact on the run time of the outer solver. Because no preconditioner has been implemented yet in the GREMLINS code, the GMRES method has been used with no preconditio-

ner also. The run time and the number of iterations of the outer solver in the GREMLINS code have been compared with those of the GMRES method. Although the run time required by the generation of the matrix has not been reported, particular attention has been paid to memory allocation. Indeed with PETSc preallocation of memory is critical for achieving good performances during matrix assembly. A correct estimation of the number of nonzeros per row (before actually setting the matrix values) has significantly reduced the total execution time.

Table 3 presents results measured in a local context (100 processors with a frequency of 2.4 GHz in Orsay) with matrices of size 2.10^7 and bandwidth 2.10^6. The matrices studied for Table 3 with 13, 23 or 33 diagonals had also been used for Table 2. As the number of diagonals increases, the run time logically increases. It is noticeable that, in this experiment, with the multisplitting method the run time is slightly higher in the asynchronous mode than in the synchronous one. When the number of diagonals increases, the relative difference between the synchronous execution time and the asynchronouse one decreases. This difference depends on the matrix, the processors and the interconnection network involved. Indeed the run time is lower in the asynchronous mode than in the synchronous one, on the one hand for the same matrix with 23 or 33 diagonals in a distant context (see Table 2) and on the other hand in the same context for a matrix with 23 or 33 diagonals having the same pattern but element values that lead to a slower convergence (see Table 1).

Except with the matrix having 13 diagonals, the number of iterations and the run time are lower with the GMRES method implemented in PETSc than with the multisplitting method. This may be explained by the optimizations inherent to the PETSc library in a local context. As the number of diagonals increases, the ratio of the run time of the GREMLINS code over the one of the PETSc code increases. In this experiment, this ratio is at most 2.

Tables 4 and 5 present performances measured in a distant context, on 198 processors : 68 in Orsay (2.4 GHz), 70 in Rennes (2.0 GHz) and 60 in Sophia (2.0 GHz).

The results reported in Table 4 refer to matrices of size 2.10^7 and bandwidth 2.10^4. As already noticed in 3.2, the performances of the multisplitting method are better in the asynchronous mode than in the synchronous one from a certain number of diagonals. In this experiment, the run time of the PETSc code is higher than the one of the GREMLINS code. Like in Table 3, as the number of diagonals increases, the ratio of the run time of the GREMLINS code over the one of the PETSc code also increases. In Table 4, this ratio, that remains less than 1, is at least 0.5 (this value refers to the matrix with 13 diagonals).

The matrices studied for Table 5 all have 13 diagonals. Their size S varies from 2.10^7 to 7.10^7 and their bandwidth is $10^{-3}S$. As their size increases,

Nb. diagonals	Multisplitting (MUMPS) Synchronous time (s)	Synchronous nb. iter.	Asynchronous time (s)	Asynchronous nb. iter.	PETSc time (s)	PETSc nb. iter.
13	15.50	12	20.59	[54-56]	17.56	12
23	32.04	17	39.56	[66-72]	24.28	14
33	42.11	21	48.90	[81-82]	27.69	15
43	54.95	25	58.65	[78-81]	30.58	16
53	62.49	28	66.48	[97-100]	34.13	17
63	73.40	32	76.33	[104-110]	37.78	18

TAB. 3 – Execution times of the GREMLINS code and the PETSc code for generated matrices of size 2.10^7 and bandwidth 2.10^6 on 100 processors in a local cluster in Orsay.

Nb. diagonals	Multisplitting (MUMPS) Synchronous time (s)	Synchronous nb. iter.	Asynchronous time (s)	Asynchronous nb. iter.	PETSc time (s)	PETSc nb. iter.
13	20.76	37	23.14	[172-198]	42.10	47
23	27.56	46	33.02	[215-245]	49.09	54
33	38.71	57	33.58	[169-186]	55.41	60
43	51.48	68	43.50	[173-189]	57.75	68
53	65.03	78	53.58	[187-204]	69.20	71
63	75.04	91	72.44	[243-286]	76.92	80

TAB. 4 – Execution times of the GREMLINS code and the PETSc code for generated matrices of size 2.10^7 and bandwidth 2.10^4 on 198 processors : 68 in Orsay, 70 in Rennes and 60 in Sophia.

the communication time increases and therefore the run time increases. As their size varies, the number of iterations both with the GREMLINS code in the synchronous mode and with the PETSc code does not differ much. As usually noticed in Tables 1 to 4 for matrices with 13 diagonals, performances with the multisplitting method are better in the synchronous mode than in the asynchronous one, except for the matrix of size 7.10^7. It is noticeable that the number of iterations with the PETSc code is slightly higher than the one with the GREMLINS code in the synchronous mode. The performances of the GREMLINS code are better than those of the PETSc code, except when the GREMLINS code is run in the asynchronous mode with the matrix of size 3.10^7.

Remark 1 *The number of iterations is related to, on the one hand, the spectral radius of the iteration matrix for the multisplitting method, and on*

Size	Multisplitting (MUMPS)				PETSc	
	Synchronous		Asynchronous			
	time (s)	nb. iter.	time (s)	nb. iter.	time (s)	nb. iter.
2.10^7	20.76	37	23.14	[172-198]	42.10	47
3.10^7	29.53	39	56.66	[281-314]	52.53	47
4.10^7	36.53	41	43.17	[160-179]	59.68	47
5.10^7	39.16	36	53.18	[158-175]	59.10	46
6.10^7	51.64	42	77.02	[195-213]	77.94	55
7.10^7	98.47	36	93.71	[189-215]	120.09	46

TAB. 5 – Execution times of the GREMLINS code and the PETSc code for generated matrices having 13 diagonals on 198 processors : 68 in Orsay, 70 in Rennes and 60 in Sophia.

the other hand, the conditioning of the matrix for the GMRES method. The size of the matrices studied in our experiments was too high for their conditioning to be exactly evaluated. However, from the convergence observed with the GMRES method, we can deduce a satisfactory conditioning of the matrices.

4 Conclusion and perspectives

For solving a linear system, the multisplitting method is an iterative method that consists in splitting the matrix into band submatrices. In a distributed environment, each processor may be in charge of managing a submatrix. The GREMLINS code enables one to use several variants of the multisplitting method in a grid environment. First iterations can be performed in a synchronous or in an asynchronous mode. Then the linear subsystems that arise from the matrix decomposition can be solved using a direct or an iterative method. For solving the subsystems, direct and iterative solvers from several libraries have been compared in a local context (*i.e.* on processors from the same cluster) and also in a distant one (*i.e.* on processors from clusters located in different geographical sites). With the matrices studied, no significant difference in terms of performance has been noticed, both in a local or in a distant context. From a certain number of diagonals in the matrix, the asynchronous mode may lead to slightly better performances than the synchronous one. The multisplitting method implemented in the GREMLINS code has been compared with the GMRES method implemented in the PETSc library. On a local cluster, Petsc takes advantage of optimizations, its performances are usually better in such an environment. In our numerical experiments carried out in a local context, the run time of the GREMLINS code is at most twice the one of PETSc.

In a distant context, the performances of the GREMLINS code are usually better. Again a ratio that is at most 2 has been noticed.

Several perspectives to this work are planed. Each matrix involved in our numerical experiments is not entirely managed by one processor. A part of the matrix is generated by each processor belonging to the grid. Matrices arising from real life problems have also been studied [2]. In this case, a file is stored on one processor that sends parts of the matrix to the others. But this limits the size of the matrix. In order to solve large real life problems, the GREMLINS code may be linked with a finite element method software, such as the ParaFEM free library [14]. After the finite element computation, the large sparse linear system resulting from the modelling would be solved using the GREMLINS code, without being explicitly built. Each processor would build and solve a local sparse linear system.

The GREMLINS code can be run in a synchronous or in an asynchronous mode, thanks to the CRAC library. But CRAC does not make any difference between processors belonging to the distributed grid, even if some processors are on the same local parallel cluster. The GREMLINS code could be improved to better use the local parallel clusters in a grid. Because the PETSc library is designed to fully take advantage of parallel computers and local clusters, it could be used over local clusters to solve in parallel linear systems generated by the multisplitting method. Communications would be performed, on the one hand, by the MPI library used by PETSc on local clusters and, on the other hand, by the CRAC library on distant clusters. This implies adaptations in the CRAC library, that should become compatible with MPI.

Acknowledgement

The GREMLINS project is supported by the French National Research Agency (ANR) under grant ANR-JC05-41999.

Experiments presented in this article have been carried out using the Grid'5000 experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see http ://www.grid5000.fr).

Références

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

[2] J. Bahi and R. Couturier. Parallelization of direct algorithms using multisplitting methods in grid environments. In *IPDPS 2005*, pages 254b, 8 pages. IEEE Computer Society Press, 2005.

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

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

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

[6] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. Curfman McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

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

[8] 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, and I. Touche. Grid'5000 : A large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4) :481–494, 2006.

[9] R. Couturier, C. Denis, and F. Jézéquel. GREMLINS : a large sparse linear solver for grid environment. *Parallel Computing*, 34(6–8) :380–391, 2008.

[10] R. Couturier and S. Domas. CRAC : a grid environment to solve scientific applications with asynchronous iterative algorithms. Technical report, LIFC - Université de Franche-Comté, 2006. http ://info.iut-bm.univ-fcomte.fr/staff/couturie/crac.pdf.

[11] J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington. A sparse matrix library in C++ for high performance architectures. In *Second Object Oriented Numerics Conference*, pages 214–218, 1994.

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

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

[14] ParaFEM : A general parallel finite element message passing library. The University of Manchester. http ://www.rcs.manchester.ac.uk/research/parafem.

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

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

Laboratoire d'Informatique de l'université de Franche-Comté
UFR Sciences et Techniques, 16, route de Gray - 25030 Besançon Cedex (France)

LIFC - Antenne de Belfort : IUT Belfort-Montbéliard, rue Engel Gros, BP 527 - 90016 Belfort Cedex (France)
LIFC - Antenne de Montbéliard : UFR STGI, Pôle universitaire du Pays de Montbéliard - 25200 Montbéliard Cedex (France)

http://lifc.univ-fcomte.fr