J Supercomput
DOI 10.1007/s11227-011-0563-y

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

Fabienne Jezequel · Raphaël Couturier · Christophe Denis

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 performance 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 paper, the performance of the GREMLINS code obtained with several libraries for solving the linear subsystems is analyzed. Its performance is also compared with that 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.

Keywords Asynchronous iterations · Grid computing · Iterative method · Multisplitting method · Sparse linear solver

F. Jezequel
UPMC Univ Paris 06, UMR 7606, Laboratoire d'Informatique de Paris 6, 4 place Jussieu, 75252 Paris CEDEX 05, France

R. Couturier (✉)
LIFC, University of Franche Comte, IUT Belfort-Montbéliard, BP 527, 90016 Belfort Cedex, France
e-mail: raphael.couturier@univ-fcomte.fr

C. Denis
EDF Research and Development, SINETICS Department, 1 avenue du Général de Gaulle, 92141 Clamart CEDEX, France

Published online: 04 February 2011

1 Introduction

Numerous scientific applications must solve large sparse linear systems. Because of considerable requirements in terms of memory allocation and execution time, it may happen that these computations cannot be carried out on a single-processor computer. Several multiprocessor environments exist, such as parallel machines or clusters of computers. A grid may be defined as a set of interconnected local clusters. The large number of processors it offers may be a relatively inexpensive answer to growing computational needs. Because of the variety of machines and interconnection networks it is usually composed of, a grid is a heterogeneous environment. Since the performance 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[1] code has been developed to solve efficiently large sparse linear systems on a grid [13]. It implements the multisplitting method [26, 29] which is based on a decomposition of the matrix into rectangular submatrices. Each processor belonging to the grid solves linear subsystems using either 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[2] library [1] or the SuperLU[3] library [15] or using iterative methods from the SparseLib[4] library [16]. The PETSc[5] library [6] is a popular suite of data structures and routines for scientific computing. Applications developed with PETSc are portable: a common code can be run on a sequential machine or on various parallel architectures. PETSc employs the MPI[6] standard for all message-passing communication. By paying particular attention to memory allocation, PETSc takes full advantage of parallel machines. For solving linear systems with PETSc, various iterative methods and also direct methods from external libraries can be used.

The originality of this paper lies in the two different types of work it describes. First, the GREMLINS code has been improved to allow each processor in a grid to use PETSc for solving its linear subsystems. Second, the performance of the PETSc library for solving large linear systems has been compared with that of the GREMLINS code, both on a local cluster and on a grid consisting of processors from several geographical sites.

In [13], the initial version of this work is described. In particular, the complete multisplitting algorithm with many implementation details is presented. The CRAC environment which enabled the implementation of asynchronous iterative algorithms is described. However, as this work was less advanced, the systems solved in [13]

[1] GREMLINS (GRid Efficient Methods for LINear Systems): http://info.iut-bm.univ-fcomte.fr/gremlins.

[2] MUMPS (MUltifrontal Massively Parallel Sparse direct Solver): http://graal.ens-lyon.fr/MUMPS.

[3] SuperLU: http://crd.lbl.gov/ xiaoye/SuperLU.

[4] SparseLib: http://math.nist.gov/sparselib++.

[5] PETSc (Portable, Extensible Toolkit for Scientific computation): http://www-unix.mcs.anl.gov/petsc.

[6] MPI (Message Passing Interface): http://www-unix.mcs.anl.gov/mpi.

could not be as large as now. In the present paper, important features of this previous work are reminded, so that it can be self-contained. The focus is put on experiments showing the relevancy of this present work. In particular, the GMRES method implemented in PETSc is compared, as an inner solver, to other solvers (SparseLib, MUMPS and SuperLU). Furthermore the GREMLINS code is compared with PETSc, the standard sparse matrix solver. This comparison highlights that with geographically distant sites, this standard solver is not so efficient.

This paper is organized as follows. In Sect. 2 some related works are presented and discussed. The principles of the multisplitting method and the architecture of the GREMLINS code are presented in Sect. 3. Numerical experiments are described in Sect. 4. First, the performance of the GREMLINS code has been analyzed, several possible libraries being used to solve serially the linear subsystems generated by the multisplitting method. Then the performance of the PETSc library and that of the GREMLINS code have been compared. Both numerical experiments have been carried out in a local and in a distant context. Section 5 presents concluding remarks and planed perspectives.

2 Related works

Many scientists are interested in solving large sparse linear systems. Solvers can be classified into direct or iterative methods. Concerning direct methods, the most efficient ones are based on the LU decomposition [18, 22]. Because of the complexity of the elimination process in direct methods, iterative methods are usually preferred for very large systems. A wide range of iterative methods is available [28]: for example, Jacobi, Gauss-Seidel and Krylov subspace methods (such as conjugate gradient, GMRES, BICGSTAB) can be cited. The convergence of iterative methods can be improved by preconditioners such as the Successive Over Relaxation (SOR) preconditioner [23] and sparse approximate inverse preconditioners that are based on factorized sparse approximate inverses or on the minimization of some convenient norm [12, 21]. Recently, explicit approximate inverse preconditioners have been introduced for solving sparse linear systems [17, 19, 20]. In [7], interested readers will find issues for implementing iterative methods in a sequential manner. Most solvers have also been designed in parallel to leverage computation power of clusters. Nevertheless, few methods have been adapted in the context of grid computing with geographically distant clusters.

One concern in the parallelization of solvers is the identification of synchronization points. In Krylov methods, which are based on projections into Krylov subspaces, the computation of a vector (by a matrix-vector product) is usually followed by its orthogonalization against a set of vectors. Inner products in the orthogonalization act as synchronization points. Designed to obtain more parallelism and data locality, the s-step variants [10, 11] consist in generating a basis for the Krylov subspace first, and to orthogonalize this set afterward. They showed satisfactory performances on homogeneous multiprocessor machines. However in a heterogeneous environment, algorithms must be both asynchronous and coarse-grained.

A hybrid version of GMRES is presented in [30]. This method combines a parallel GMRES method with the least square method that requires some eigenvalues

obtained from a parallel Arnoldi algorithm. In the paper, only small matrices are considered: the largest one has only 3,600 unknowns.

The implementation of a parallel 3D solver based on Navier–Stokes system is described in [24]. The solver is built with Globus. Experiments are also quite small since the number of unknowns is less than 20,000.

In [13], large sparse linear systems are solved in a grid computing context using GREMLINS. In this work, only direct solvers are experimented inside the multisplitting method.

A parallel hybrid solver based on both direct methods and iterative methods is presented in [25]. It allows one to solve large matrices but it is only dedicated to homogeneous clusters, not to grid environments.

Except previous works described in [13], all the solvers cited in this section are based on synchronous iterative methods. One of the originality of the GREMLINS solver is to be able to run either in synchronous or asynchronous iteration mode.

3 The multisplitting method

3.1 Principles of the multisplitting method

For solving a linear system, the multisplitting method generalizes the block Jacobi method. Moreover, 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 $n \times n$ nonsymmetric 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 rectangle matrices. For the sake of simplicity, let us consider the decomposition generates as many rectangle matrices as processors. Thus, each processor is in charge of managing one submatrix, denoted by $ASub$. 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 $DepRight$. 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.

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

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

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.

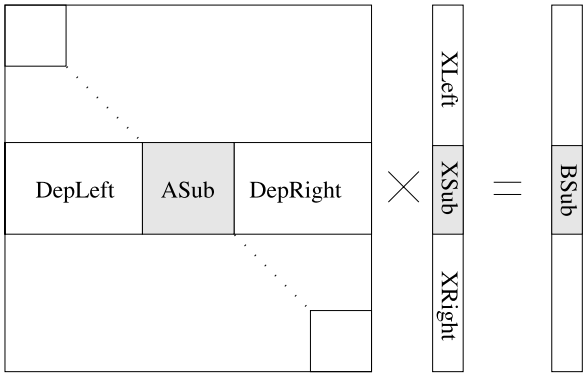

Fig. 1 Decomposition of the matrix A, the solution vector X and the right-hand side vector B into several parts required locally by a processor

1. Initialization:
   The matrix can be loaded from a data file or generated at run time. Each processor manages the load of the rectangle matrix $DepLeft + ASub + DepRight$. Then until convergence, each processor iterates on:
2. Computation:
   At each iteration, each processor computes $BLoc = BSub - DepLeft \times XLeft - DepRight \times XRight$. Then it solves the linear system $ASub \times XSub = BLoc$.
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 [2] or a decentralized one, that is a more general version, as described in [3].

In the multisplitting method, the model of asynchronous iterations may reduce the run time. In this case, receptions are nonblocking, computations are dissociated from communications using threads, and an appropriate convergence algorithm is used. Additional references on theoretical aspects of asynchronous iterative algorithms can be found in [8].

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. In the synchronous version, the spectral radius of the iteration matrix associated with each submatrix must be strictly less than 1; in the asynchronous version, the spectral radius of the absolute value of each iteration matrix must be strictly less than 1 [4]. 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. The impact of overlapping over the speed of convergence is exemplified in [4].

3.2 Comparison of the multisplitting and the GMRES method

The multisplitting method is compared with the GMRES method, which is widely used for solving sparse nonsymmetric linear systems. This comparison is exemplified in Sect. 4.3 by numerical experiments performed using the GREMLINS code and the GMRES solver from the PETSc library.

There is a fundamental difference between the multisplitting and the GMRES method. The former can be executed with the asynchronous model which may overlap communications by computation. In this case, the number of iterations to reach the convergence is often larger, but there is no more synchronization between processors. In the latter, at each iteration, there are commonly three synchronizations. There is a huge synchronization step that allows processors to exchange their dependencies with all their neighbors before computing the sparse matrix vector product. Then there is another synchronization which allows processors to reduce their scalar products. Finally, there is a last synchronization for computing the norm and deciding whether the convergence criterion is reached or not.

Comparing data transfers involved in each method, the multisplitting algorithm only requires data transfer for the matrix vector product. But the main feature of this method, in the asynchronous mode, is that even if some neighbors cannot send their data dependencies because they are not ready to transfer them, then a processor can compute its local matrix vector product even if some entries have not been updated. In opposition, the GMRES method requires that all processors send their data dependencies before computing the matrix vector product. Moreover, there are two other synchronizations. So, in a grid environment context with geographically distant sites and variable network parameters, synchronizations are really penalizing and occur by definition simultaneously. In opposition, asynchronous iterative algorithms offer the advantage of better scheduled communications. Moreover, the overlap of communications by computation allows the overall system to converge faster even if some communications links are slower than other ones or processors quite heterogeneous.

In fact, the previous explanation is also true for other iterative methods which cannot be executed in asynchronous mode. All synchronous methods require to have a synchronization step before computing the matrix vector product. Then depending on the method, the number of reductions is variable but at least one is required to compute the convergence test.

3.3 The GREMLINS code

The GREMLINS code implements in C++ the multisplitting method for solving nonsymmetric sparse linear systems. It uses the CRAC[7] library [14] 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: sending a message, receiving a message, and detecting 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 its assigned processor. In the previous version of the GREMLINS code [13], three scientific libraries could be chosen for solving the subsystems: MUMPS [1], SparseLib [16], and SuperLU [15]. The GREMLINS code has been improved to also allow the use of the PETSc library [6].

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 to communicate.
- 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 Compressed Sparse Row (CSR) format that consists of 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 representation 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, previously described, into 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.

[7] CRAC (Communication Routines for Asynchronous Computations).

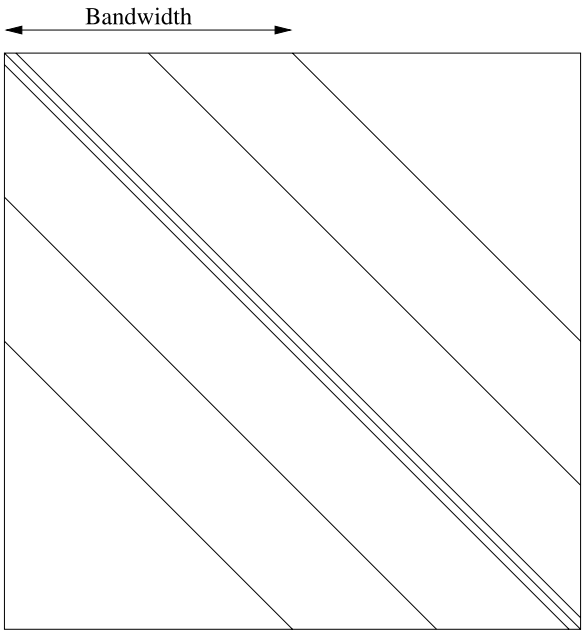

Fig. 2 A generated matrix with 7 nonempty diagonals and a bandwidth equal to half the matrix size

4 Numerical experiments

4.1 Context of the experiments

The multisplitting method has been used with processors located in the same cluster or in different sites to solve linear systems arising from real life problems [4]. In this case, a 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 [4], the size of the largest matrix assigned from a file is 130,228. The associated linear system is solved on at most 20 processors. In order to solve larger problems without long data file transfers, the linear systems studied in this paper are generated at run time.

Each processor computes specific matrix rows, so that each matrix is automatically distributed on the processors. Each generated matrix consists of several nonempty diagonals: the main diagonal, the two nearest neighbor diagonals and other diagonals are equally scattered between the main diagonal and the desired bandwidth. As an example, a matrix with 7 non-empty diagonals and a bandwidth equals to half the matrix size is represented in Fig. 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 fixed value and a random value from an interval specified by the user (in practice the fixed value equals 1.1 and the random value lies between 0 and 1.2). Such generated matrices are M-matrices [5] (defined as Z-matrices with eigenvalues whose real parts are positive). Z-matrices, i.e., matrices whose diagonal entries are strictly positive and off-diagonal entries nonpositive, and also diagonally dominant matrices satisfy the convergence condition of both the synchronous and the asynchronous version of the multisplitting method [4].

Numerical experiments have been carried out on GRID'5000,[8] an experimental grid platform featuring 5,000 processors which are geographically distributed across nine sites in France [9]. Recently, this network has been augmented to include one site in Brazil. 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 depend on the resources available 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.

Two types of experiments are described. First, the performance obtained using different libraries to solve the subsystems in the multisplitting method is analyzed. Second, the multisplitting method is compared with the restarted version of the GMRES (General Minimal RESidual) method [28] implemented in the PETSc library. This second point provides a comparison of the GREMLINS solver with a standard parallel one.

4.2 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 [28].

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. That means that if several instances of the same solvers are run, a nonthread-safe program will crash.

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 fewer iterations and, therefore, lead to a faster convergence than the corresponding one from Table 1 in the same context (local or distant processors).

[8] GRID'5000: http://www.grid5000.fr.

Table 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)	Stand. dev.	Nb. iter.	Time (s)	Stand. dev.	Nb. iter.
13 diagonals						
MUMPS	98.79	0.31	83	93.79	0.51	[240-249]
SuperLU	84.09	0.24	83	98.07	0.50	[417-441]
SparseLib	87.21	0.26	83	91.68	0.48	[388-426]
PETSc	84.14	0.29	83	95.70	0.43	[424-457]
23 diagonals						
MUMPS	278.43	0.23	148	258.98	0.37	[421-439]
SuperLU	253.71	0.24	148	248.57	0.49	[506-532]
SparseLib	272.39	0.28	148	259.32	0.41	[441-451]
PETSc	270.04	0.36	148	255.46	0.48	[411-414]
33 diagonals						
MUMPS	407.06	0.29	205	376.94	0.41	[556-574]
SuperLU	367.49	0.31	205	351.04	0.37	[714-747]
SparseLib	394.02	0.27	205	364.86	0.34	[604-608]
PETSc	398.23	0.32	205	369.91	0.45	[527-566]

The run time and the number of iterations performed by the outer solver in the multisplitting method, in both the synchronous and the asynchronous mode, are reported in Tables 1 and 2. In each case, the computation has been performed ten times with the time reported being the mean value. As the run time varies from one execution to another, the standard deviation from the mean execution time is also reported. 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 reported into square brackets. In the asynchronous mode, within one execution, the number of iterations also varies from one processor to another. At each execution, it is the number of iterations performed by the supermaster, a processor that has a specific function for communications with the CRAC library [14] that has been measured.

With the matrices considered, both in a local context and in a distant context, no inner solver performs clearly better than the others. 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, so do the computational volume, the number of iterations in the synchronous mode and the run time both in the synchronous mode and in the asynchronous one. Run times in the asynchronous mode are slightly better than in the synchronous one from a certain number of diagonals.

Table 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

Solver	Synchronous			Asynchronous		
	Time (s)	Stand. dev.	Nb. iter.	Time (s)	Stand. dev.	Nb. iter.
13 diagonals						
MUMPS	25.15	0.68	12	42.09	1.01	[199-215]
SuperLU	23.42	0.81	12	44.15	1.29	[510-526]
SparseLib	23.00	0.77	12	32.67	1.12	[272-310]
PETSc	23.57	0.82	12	40.89	0.91	[322-453]
23 diagonals						
MUMPS	57.00	0.92	17	54.35	1.01	[170-188]
SuperLU	55.45	0.67	17	51.40	1.27	[333-389]
SparseLib	54.88	0.98	17	54.82	0.99	[302-330]
PETSc	55.33	0.85	17	53.87	1.16	[322-369]
33 diagonals						
MUMPS	83.88	1.01	21	75.16	1.06	[191-199]
SuperLU	78.54	0.89	21	66.58	0.95	[344-359]
SparseLib	79.83	0.96	21	70.44	1.59	[230-255]
PETSc	79.12	0.78	21	71.70	1.27	[203-218]

4.3 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 already mentioned, no inner solver performs clearly better in the experiments reported in Sect. 4.2. The MUMPS library has been chosen since it consumes less memory than SuperLU. Because no preconditioner has been implemented yet in the GREMLINS code, the GMRES method has also been used without any preconditioner. The run time and the number of iterations of the outer solver in the GREMLINS code have been compared with those of the restarted 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 to achieve good performances during matrix assembly. By specifying the number of nonzeros per row (before actually setting the matrix values), the total execution time has been significantly reduced.

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 asynchronous one decreases. This difference depends on the

Table 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)						PETSc		
	Synchronous			Asynchronous					
	Time (s)	Stand. dev.	Nb. iter.	Time (s)	Stand. dev.	Nb. iter.	Time (s)	Stand. dev.	Nb. iter.
13	15.50	0.21	12	20.59	0.45	[54-56]	17.56	0.25	12
23	32.04	0.24	17	39.56	0.47	[66-72]	24.28	0.28	14
33	42.11	0.34	21	48.90	0.56	[81-82]	27.69	0.19	15
43	54.95	0.27	25	58.65	0.39	[78-81]	30.58	0.22	16
53	62.49	0.18	28	66.48	0.45	[97-100]	34.13	0.27	17
63	73.40	0.27	32	76.33	0.41	[104-110]	37.78	0.32	18

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. The local context of this experiment is favorable to the PETSc library. 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 run times 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 Sect. 4.2, the performance of the multisplitting method is 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. As 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).

All the matrices studied for Table 5 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, the communication time also increases and, therefore, the run time increases as well. 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, the GREMLINS code performance is 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 performance of the GREMLINS code is better than that of the PETSc code, except when the GREMLINS code is run in the asynchronous mode with the matrix of size 3×10^7.

Table 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

Nb. diagonals	Multisplitting (MUMPS)						PETSc		
	Synchronous			Asynchronous					
	Time (s)	Stand. dev.	Nb. iter.	Time (s)	Stand. dev.	Nb. iter.	Time (s)	Stand. dev.	Nb. iter.
13	20.76	0.92	37	23.14	1.09	[172-198]	42.10	1.12	47
23	27.56	1.01	46	33.02	1.02	[215-245]	49.09	0.87	54
33	38.71	0.98	57	33.58	1.52	[169-186]	55.41	1.32	60
43	51.48	1.05	68	43.50	0.98	[173-189]	57.75	1.09	68
53	65.03	1.09	78	53.58	1.23	[187-204]	69.20	1.1	71
63	75.04	0.98	91	72.44	1.54	[243-286]	76.92	1.34	80

Table 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

Size	Multisplitting (MUMPS)						PETSc		
	Synchronous			Asynchronous					
	Time (s)	Stand. dev.	Nb. iter.	Time (s)	Stand. dev.	Nb. iter.	Time (s)	Stand. dev.	Nb. iter.
2×10^7	20.76	0.76	37	23.14	0.89	[172-198]	42.10	0.99	47
3×10^7	29.53	1.08	39	56.66	1.02	[281-314]	52.53	0.67	47
4×10^7	36.53	1.01	41	43.17	0.78	[160-179]	59.68	0.87	47
5×10^7	39.16	1.61	36	53.18	1.21	[158-175]	59.10	0.99	46
6×10^7	51.64	0.68	42	77.02	0.19	[195-213]	77.94	0.67	55
7×10^7	98.47	1.23	36	93.71	1.91	[189-215]	120.09	1.87	46

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 the other hand, the conditioning of the matrix for the GMRES method. The size of the matrices studied in this article is too high for their conditioning to be exactly evaluated. However, a satisfactory conditioning of the matrices can deduced from the convergence observed with the GMRES method.

5 Conclusion and perspectives

For solving a linear system, the multisplitting method is an iterative method that consists in splitting the matrix into rectangle 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. Several libraries can be used for solving the subsystems: MUMPS, SparseLib, SuperLU, and also PETSc in the current version of the GREMLINS code.

The GREMLINS code performance has been analyzed 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, the choice of the solver for the subsystems has no significant impact in terms of performance, neither in a local nor in a distant context. From a certain number of diagonals in the matrix, the asynchronous mode may lead to better performances than the synchronous one. This performance difference, that is slight on GRID'5000, is more marked if the network bandwidth is degraded [4].

The multisplitting method implemented in the GREMLINS code has been compared with the GMRES method implemented in the PETSc library. Because the GREMLINS code has been designed to run efficiently in a grid environment, its performance is particularly satisfactory in a distant context. In the numerical experiments carried out in a distant context, the run time of PETSc is up to twice the one of the GREMLINS code. On a local cluster, the performance of PETSc is usually better. Again, a ratio between the run times that is at most 2 has been noticed.

Several perspectives to this work are planed. Each matrix involved in the 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 [4]. 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 without long data file transfers, the GREMLINS code may be linked with a finite element method software, such as the ParaFEM free library [27]. After the finite element computation, the large sparse linear system resulting from the modeling 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 grid, even if some processors are on the same local parallel cluster. The GREMLINS code could be improved to make a better use of 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 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.

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

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

Experiments presented in this paper 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).

References

1. Amestoy PR, Guermouche A, L'Excellent J-Y, Pralet S (2006) Hybrid scheduling for the parallel solution of linear systems. Parallel Comput 32(2):136–156

2. Bahi JM, Contassot-Vivier S, Couturier R (2005) Evaluation of the asynchronous iterative algorithms in the context of distant heterogeneous clusters. Parallel Comput 31(5):439–461
3. Bahi JM, Contassot-Vivier S, Couturier R, Vernier F (2005) A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms. IEEE Trans Parallel Distrib Syst 1:4–13
4. Bahi JM, Couturier R (2005) Parallelization of direct algorithms using multisplitting methods in grid environments. In: IPDPS'2005, 19th international parallel and distributed processing symposium, Denver, Colorado, USA, April 2005. IEEE Comput Soc, Los Alamitos, pp. 254b, 8 pages
5. Bahi JM, Miellou J-C, Rhofir K (1997) Asynchronous multisplitting methods for nonlinear fixed point problems. Numer Algorithms 15(3–4):315–345
6. Balay S, Buschelman K, Eijkhout V, Gropp WD, Kaushik D, Knepley MG, Curfman McInnes L, Smith BF, Zhang H (2004) PETSc users manual. Technical report ANL-95/11, Revision 2.1.5, Argonne National Laboratory
7. Barrett R, Berry M, Chan TF, Demmel J, Donato J, Dongarra J, Eijkhout V, Pozo R, Romine C, Van der Vorst H (1994) Templates for the solution of linear systems: building blocks for iterative methods, 2nd edn. SIAM, Philadelphia
8. Bertsekas DP, Tsitsiklis JN (1989) Parallel and distributed computation: numerical methods. Prentice-Hall, Englewood Cliffs
9. Bolze R, Cappello F, Caron E, Daydé M, Desprez F, Jeannot E, Jégou Y, Lanteri S, Leduc J, Melab N, Mornet G, Namyst R, Primet P, Quetier B, Richard O, Talbi E-G, Touche I (2006) Grid'5000: a large scale and highly reconfigurable experimental grid testbed. Int J High Perform Comput Appl 20(4):481–494
10. Chronopoulos AT (1991) s-step iterative methods for (non)symmetric (in)definite linear systems. SIAM J Numer Anal 28(6):1776–1789
11. Chronopoulos AT, Gear CW (1989) s-step iterative methods for symmetric linear systems. J Comput Appl Math 25(2):153–168
12. Cosgrove JDF, Dias JC, Griewank A (1992) Approximate inverse preconditioning for sparse linear systems. Int J Comput Math 44:91–110
13. Couturier R, Denis C, Jézéquel F (2008) GREMLINS: a large sparse linear solver for grid environment. Parallel Comput 34(6–8):380–391
14. Couturier R, Domas S (2007) CRAC: a grid environment to solve scientific applications with asynchronous iterative algorithms. In: IPDPS'2007, 21st international parallel and distributed processing symposium, Long Beach, California, USA, March 2007. IEEE Comput Soc, Los Alamitos, pp 289–296
15. Demmel JW, Eisenstat SC, Gilbert JR, Li XS, Liu JWH (1999) A supernodal approach to sparse partial pivoting. SIAM J Matrix Anal Appl 20(3):720–755
16. Dongarra J, Lumsdaine A, Pozo R, Remington K (1994) A sparse matrix library in C++ for high performance architectures. In: Second object oriented numerics conference, pp 214–218
17. Gravvanis GA, Giannoutakis KM (2006) On the performance of parallel normalized explicit preconditioned conjugate gradient—type methods. In: IPDPS'2006, 20th international parallel and distributed processing symposium, Rhodes Island, Greece, April 2006. IEEE Comput Soc, Los Alamitos
18. Golub GH, van Loan C (1996) Matrix computations. The Johns Hopkins University Press, Baltimore
19. Gravvanis GA (2002) Explicit approximate inverse preconditioning techniques. Arch Comput Methods Eng 9(4):371–402
20. Gravvanis GA (2009) High performance inverse preconditioning. Arch Comput Methods Eng 16(1):77–108
21. Grote MJ, Huckle T (1997) Parallel preconditioning with sparse approximate inverses. SIAM J Sci Comput 18:838–853
22. Duff I, Erisman M, Reid J (1986) Direct methods for sparse matrices. Oxford University Press, London
23. Hageman LA, Young DM (1981) Applied iterative methods. Academic Press, San Diego
24. Langer U, Zulehner W, Yang H, Baumgartner M (2007) GStokes: a grid-enabled solver for the 3D Stokes/Navier–Stokes system on hybrid meshes. In: ISPDC'2007, 6th international symposium on parallel and distributed computing, Hagenberg, Austria, July 2007. IEEE Comput Soc, Los Alamitos, pp 377–382
25. Manguoglu M, Sameh AH, Schenk O (2009) PSPIKE: a parallel hybrid sparse linear system solver. In: Sips HJ, Epema D, Lin H-X (eds) Euro-par. Lecture notes in computer science, vol 5704. Springer, Berlin, pp 797–808
26. O'Leary DP, White RE (1985) Multi-splittings of matrices and parallel solution of linear systems. SIAM J Algebr Discrete Methods 6:630–640

27. ParaFEM: A general parallel finite element message passing library. The University of Manchester. http://www.rcs.manchester.ac.uk/research/parafem
28. Saad Y (1996) Iterative methods for sparse linear systems. PWS Publishing, New York
29. White RE (1990) Multisplitting of a symmetric positive definite matrix. SIAM J Matrix Anal Appl 11:69–82
30. Zhang Y, Bergère G, Petiton S (2008) Large scale parallel hybrid GMRES method for the linear system on grid system. In: ISPDC'2008, 7th international symposium on parallel and distributed computing, Krakow, Poland, July 2008. IEEE Computer Society, Los Alamitos, pp 244–249