An Hybrid Approach for the Parallelization of a Block Iterative Algorithm

Carlos Balsa[1], Ronan Guivarch[2], Daniel Ruiz[2], and Mohamed Zenadi[2]

[1] CEsA–FEUP, Porto, Portugal
balsa@ipb.pt
[2] Université de Toulouse ; INP (ENSEEIHT) ; IRIT
{guivarch, ruiz, mzenadi}@enseeiht.fr

Abstract. The Cimmino method is a row projection method in which the original linear system is divided into subsystems. At every iteration, it computes one projection per subsystem and uses these projections to construct an approximation to the solution of the linear system.
The usual parallelization strategy in block algorithms is to distribute the different blocks on the available processors. In this paper, we follow another approach where we do not perform explicitly this block distribution to processors within the code, but let the multi-frontal sparse solver MUMPS handle the data distribution and parallelism. The data coming from the subsystems defined by the block partition in the Block Cimmino method are gathered in an unique block diagonal sparse matrix which is analysed, distributed and factorized in parallel by MUMPS. Our target is to define a methodology for parallelism based only on the functionalities provided by general sparse solver libraries and how efficient this way of doing can be.

1 Introduction

The Cimmino method is a row projection method in which the original linear system is divided into subsystems. At each iteration, it computes one projection per subsystem and uses these projections to construct an approximation to the solution of the linear system. The Block-CG method can also be used within the Block Cimmino Iteration to accelerate its convergence. Therefore, we present an implementation of a parallel distributed Block Cimmino method where the Cimmino iteration matrix is used as a preconditioner for the Block-CG.

In this work we propose to investigate a non usual methodology for parallelization of the Block Cimmino method where the direct solver package MUMPS (MUltifrontal Massively Parallel sparse direct Solver [1,2]) is incorporated in order to schedule and perform most of the parallel tasks. This library offers to the user the facilities to call the different phases of the solver (analysis, factorization, solution) while taking care of the distribution of data and processes.

The outline of the paper is the following: the Block Cimmino Algorithm is described in section 2 and parallelization strategies are exposed in section 3. More details about our strategy are given in section 4. We finish by a presentation of some preliminary numerical results in section 5 that give us some hints for future improvements and developments.

2 Block Cimmino Algorithm

The Block Cimmino method is a generalization of the Cimmino method [3]. Basically, we partition the linear system of equations:

$$\mathbf{A}x = b, \tag{1}$$

where $\mathbf{A}$ is a $m \times n$ matrix, into l subsystems, with $l \leq m$, such that:

$$\begin{pmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \vdots \\ \mathbf{A}_l \end{pmatrix} x = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_l \end{pmatrix} \tag{2}$$

The block method [4] computes a set of l row projections, and a combination of these projections is used to build the next approximation to the solution of the linear system. Now, we formulate the Block Cimmino iteration as:

$$\begin{aligned} \delta_i^{(k)} &= \mathbf{A}_i^+ b_i - \mathbf{P}_{\mathcal{R}(A_i^T)} x^{(k)} \qquad (3) \\ &= \mathbf{A}_i^+ \left(b_i - \mathbf{A}_i x^{(k)} \right) \\ x^{(k+1)} &= x^{(k)} + \nu \sum_{i=1}^{l} \delta_i^{(k)} \qquad (4) \end{aligned}$$

In (3), the matrix $\mathbf{A}_i^+$ refers to the classical pseudo-inverse of $\mathbf{A}_i$ defined as: $\mathbf{A}_i^+ = \mathbf{A}_i^T \left(\mathbf{A}_i \mathbf{A}_i^T \right)^{-1}$. However, the Block Cimmino method will converge for any other pseudo-inverse of $\mathbf{A}_i$ and in our parallel implementation we use a generalized pseudo-inverse [5], $\mathbf{A}_{i\ \mathbf{G}^{-1}}^{-} = \mathbf{G}^{-1} \mathbf{A}_i^T \left(\mathbf{A}_i \mathbf{G}^{-1} \mathbf{A}_i^T \right)^{-1}$, were $\mathbf{G}$ is some symmetric and positive definite matrix. The $\mathbf{P}_{\mathcal{R}(A_i^T)}$ is an orthogonal projector onto the range of $\mathbf{A}_i^T$.
We use the augmented systems approach, as in [6] and [7], for solving the subsystems (3)

$$\begin{bmatrix} \mathbf{G} & \mathbf{A}_i^T \\ \mathbf{A}_i & 0 \end{bmatrix} \begin{bmatrix} u_i \\ v_i \end{bmatrix} = \begin{bmatrix} 0 \\ b_i - \mathbf{A}_i x \end{bmatrix} \tag{5}$$

with solution:

$$\begin{aligned} v_i &= - \left(\mathbf{A}_i \mathbf{G}^{-1} \mathbf{A}_i^T \right)^{-1} r_i \\ u_i &= \mathbf{A}_{i\ \mathbf{G}^{-1}}^{-} (b_i - \mathbf{A}_i x) \qquad (6) \\ &= \delta_i \end{aligned}$$

The Block Cimmino method is a linear stationary iterative method with a symmetrizable iteration matrix [8]. When A has full rank, the symmetrized iteration matrix is symmetric positive definite (SPD). We can accelerate its rate of convergence with the use of Block Conjugate Gradient method (Block-CG) [9,10].

3 Parallelization strategy

The Block-Cimmino algorithm generates a set of fully independent blocks that are the basis for a first level of natural parallelism. As opposed to usual parallelization strategies that we will recall shortly in the next subsection, and that consists in distributing in some way each block onto processors, while having more blocks in general than processors to manage better load balancing, we introduce an other strategy (detailed in subsection 3.2) that resumes in gathering the blocks into a larger but single sparse linear system with a block diagonal structure, and handle all the levels of parallelism at the same time.

3.1 Manual parallelism description

As described in [11], a first, and natural strategy would be to distribute the blocks evenly on the available processors.

Blocks' distribution It could be difficult to achieve a good load balancing since the distribution of the augmented systems blocks must be closely related to the number of processors. From a technical point we can consider the following issues:

– if there is a high number of processors, we can create the same number of blocks, but the speed of convergence of such block iterative algorithms is often slowed down when the number of blocks increases;
– if the blocks are too small, the cost of communication can become prohibitive relatively to the computations.

In general, it's better to have more blocks than processors to have more degree of freedom for better load-balancing.

Parallelism exploitation In a problem with different physical properties having blocks with different dimensions is a common thing, which may overload the processors supposed to handle them. Moreover the other processors will stall once they finish their work as there is a synchronization after the solve part for the projectors sum.

Furthermore, we may not have a fully and unrestricted choice in the block partitioning of our matrix. Some problems involving different physical properties require a particular partitioning that have to be respected otherwise it may disturb the convergence rate.

3.2 Automatic parallelism with MUMPS

In order to get rid of the problems noticed in the manual block distribution, we chose to use the direct sparse solver MUMPS (**MU**ltifrontal **M**assively **P**arallel sparse direct **S**olver)[1], which will handle the block distribution and the parallelism.

In our approach we create the block diagonal sparse matrix from the data coming from the subsystems defined by the block partition (2) in the Block Cimmino method. The matrix is then given to MUMPS for an in-depth analysis which will permit a fine grain parallelism handling while respecting the matrix structure and augmented systems' properties. The matrix is then distributed following the mapping generated by this analysis, and afterward factorized in parallel. At each Block-CG iteration, MUMPS solves in parallel the system involved during the preconditioning step.

The added advantage of this way of doing, is that the sparse linear solver will handle (without any extra development for us) all the levels of parallelism available, and in particular those coming from sparsity structure and BLAS3 kernels on top of the block partitioning. This also gives us more degrees of freedom when partitioning the matrix, with the possibility to define less blocks than processors but larger ones. This may help to increase the speed of convergence of the method while still keeping a good parallelism since the three levels of parallelism are managed together.

The analysis and the factorization –part of the preprocessing step– are handled by MUMPS. However, during the solve step –as the solve is done in parallel– we have to gather the distributed results onto the master to perform the remaining Block-CG operations (daxpy, ddot, residual computation) in sequential. The next achievement to reach our goal is to perform most of these operations in parallel by keeping the results distributed and performing the Block-CG operations in parallel. To achieve this, we need to exploit MUMPS-embedded functionalities for data management and communications (distributed solution) and implement others (distributed RHS).

4 Strategy details

Our approach inherits from the direct methods the three main steps which are the *Analysis*, the *Factorization* and the *Solution*. For the sake of simplicity we consider two main steps by merging the Analysis and the Factorization steps into the *Preprocessing* step. Meanwhile, Block-Cimmino and Block-CG will be assimilated into the *Solve* step.

4.1 Preprocessing

During this step most of the Block-Cimmino preprocessing is done. As described in Fig.1, we go through the following processes:

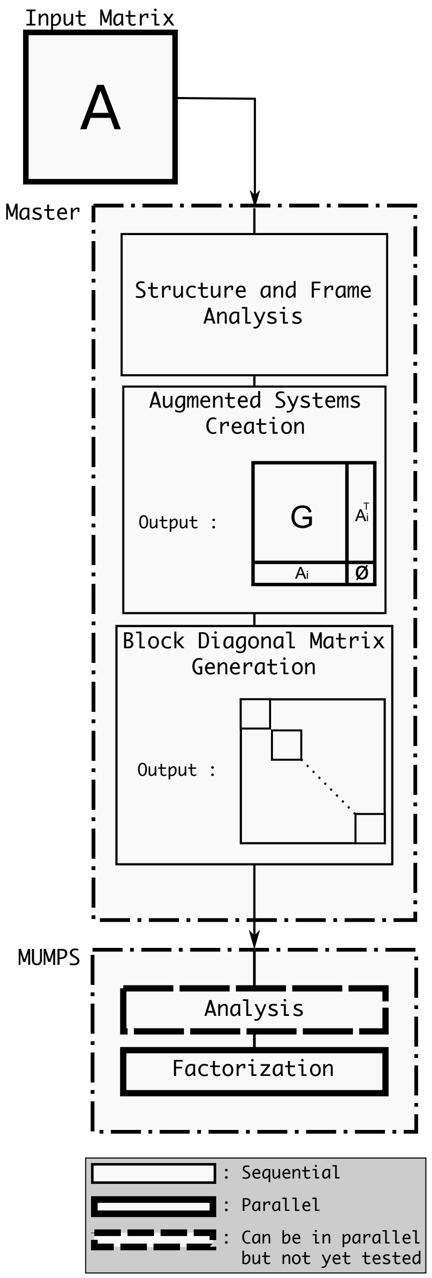
Input Matrix
A
Master
Structure and Frame Analysis
Augmented Systems Creation
Output :
G
A_i^T
A_i
Ø
Block Diagonal Matrix Generation
Output :
MUMPS
Analysis
Factorization
: Sequential
: Parallel
: Can be in parallel but not yet tested

Fig. 1. Preprocessing

Structure and Frame analysis : Using the matrix structure and the block decomposition information we start by analyzing the dependence between the blocks by finding the non-zero values on the common columns. This information helps us to find the parts of the projections to sum during the Bloc-CG iterations.

Augmented Systems and Block Diagonal Matrix creation : The new blocks' structure is used to generated the augmented systems as described in (5). We then combine these augmented systems into a block diagonal matrix

The Analysis : MUMPS handles during this step the following steps:

- Structure analysis and Scaling
- Graph ordering using METIS.
- Mapping the nodes over the available processors by respecting matrix block diagonal structure. MUMPS will recognize it as a forest and thus maps the nodes over the processors according to the number of nodes and their weight.
- Distribute the matrix over the processors.

The Factorization : The distributed matrix is LDL^T factorized in parallel. The factors of the block diagonal matrix are kept in place and distributed onto the processors for future access during the solves in the Block-CG iterations.

4.2 Solve: The Block-CG acceleration

The Block Cimmino method is a linear stationary iterative method with a symmetrizable iteration matrix [8].

$$E_{RJ}\ x \ = \ h_{RJ} \tag{7}$$

$$\begin{aligned} \text{With : } E_{RJ} &= \textstyle\sum_{i=1}^{l} A_i^T (A_i A_i^T)^{-1} A_i \\ \text{and : } h_{RJ} &= \textstyle\sum_{i=1}^{l} A_i^T (A_i A_i^T)^{-1} b_i \end{aligned}$$

One way to solve (7) in parallel is to use a distributed QR factorization which will easily help to calculate E_{RJ}, an implementation is under current development and released under the name of QR-MUMPS: `http://mumps.enseeiht.fr/doc/ud_2010/qr_talk.pdf`. The other approach –our case– is using augmented systems described in Section 2.

This system has two main properties:

- When A is consistent our system is consistent and symmetric positive semi-definite.
- When A has full rank our system is symmetric positive definite

Algorithm 1 Block-CG acceleration algorithm

```
1: $X^{(0)} \leftarrow random$
2: $\tilde{R}^{(0)} \leftarrow H - EX^{(0)}$
3: $\tilde{P}^{(0)} \leftarrow \tilde{R}^{(0)}$
4: $k \leftarrow 0$
5: loop
6:     $\Omega^{(k)} \leftarrow EP^{(k)}$
7:     $\beta^{(k)} \leftarrow (\tilde{P}^{(k)T} G \Omega^{(k)})^{-1} (\tilde{R}^{(k)T} G \tilde{R}^{(k)})$
8:     $X^{(k+1)} \leftarrow X^{(k)} + \beta^{(k)} \tilde{P}^{(k)}$
9:     if Converged then
10:        exit loop
11:    end if
12:    $\tilde{R}^{(k+1)} \leftarrow \tilde{R}^{(k)} - \beta^{(k)} \Omega^{(k)}$
13:    $\alpha^{(k)} \leftarrow (\tilde{R}^{(k)T} G \tilde{R}^{(k)})^{-1} (\tilde{R}^{(k+1)T} G \tilde{R}^{(k+1)})$
14:    $\tilde{P}^{(k+1)} \leftarrow \tilde{R}^{(k+1)} + \alpha^{(k+1)} \tilde{R}^{(k)}$
15:    $k \leftarrow k + 1$
16: end loop
```

The second property made our choice to use Block-CG as acceleration. This method [9,10] simultaneously searches for the next approximation to the system's solution in a given number of Krylov subspaces, and this number is given by the *block size* of the Block-CG method. The Block-CG method converges in a finite number of iterations in absense of roundoff errors. The Block-CG loop is described in Algorithm 1.

We use MUMPS during the operations involved in the lines 2 and 6 of Algorithm 1. These operations are equivalent to solve the subsystems (8) in parallel, and then centralize the solutions to combine the projectors. The solve process can be presented as in the Fig.2.

$$\begin{bmatrix} \mathbf{G} & \mathbf{A}_i^T \\ \mathbf{A}_i & 0 \end{bmatrix} \begin{bmatrix} u_i \\ v_i \end{bmatrix} = \begin{bmatrix} 0 \\ b_i - \mathbf{A}_i x \end{bmatrix} \tag{8}$$

The main Block-CG loop (run on the master) prepares the Right Hand Sides for each augmented system and generates a single RHS –to accelerate the convergence rate of Block-CG we can generate multiple RHS– corresponding to the system to be solved with the block diagonal matrix generated during the preprocessing. The RHS is then sent to MUMPS to solve the system.

MUMPS distributes the RHS according to the way the augmented systems were mapped on the available processors during the *Preprocessing* step. Then each subsystem (8) is solved in parallel with no association between the number of subsystems and the number of processors. Once solved the solutions are sent back to the master to be combined and used in the rest of Block-CG iteration.

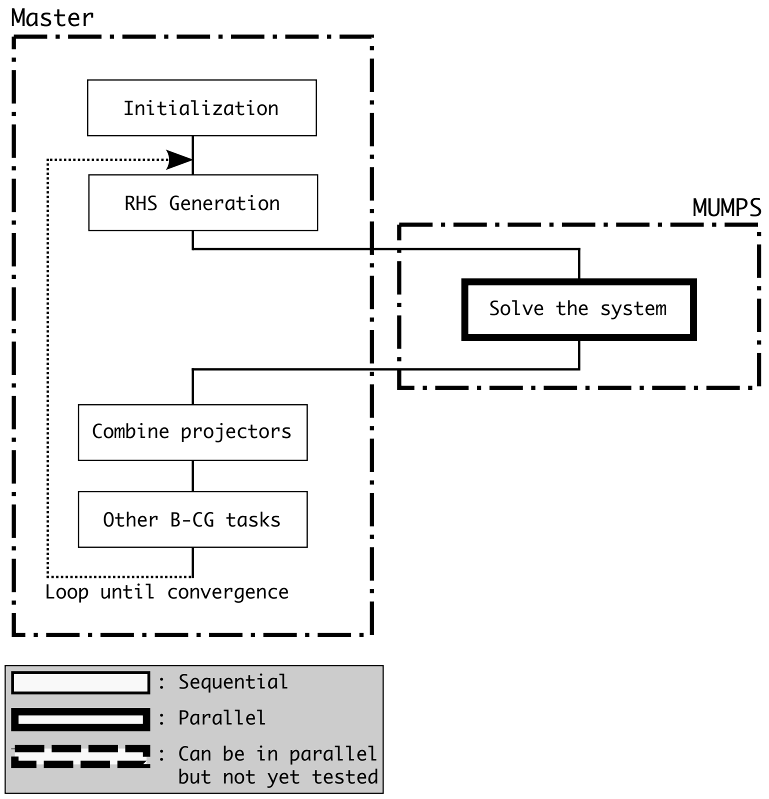

Fig. 2. Block-CG acceleration loop

5 Preliminary numerical results

The numerical experiments were carried out on the Hyperion supercomputer (`http://www.calmip.cict.fr/spip/spip.php?rubrique90`). Hyperion is the latest supercomputer of the CICT (Centre Interuniversitaire de Calcul de Toulouse) and was ranked 223rd in November's Top500.org ranking. With its 352 bi-Intel "Nehalem" EP quad-core nodes it can develop a peak of 33TFlops. Each node has 4.5GB memory dedicated for each of the cores with an overall of 32GB fully available memory on the node.

Our testing method respects the following:

– Each MPI-Process is mapped on a single Quad-Core CPU with a dedicated memory access.
– We run each test case –relative to the number of processors– with three different numbers of blocks.
– All blocks have same number of rows.

The results we present are for a single matrix, Oberwolfach/boneS10 with a dimension of 914,898, which is an Oberwolfach 3D trabecular bone. It is a symmetric real matrix and of full rank (`www.cise.ufl.edu/research/sparse/matrices/Oberwolfach/boneS10.html`).

5.1 Factorization and Solve steps

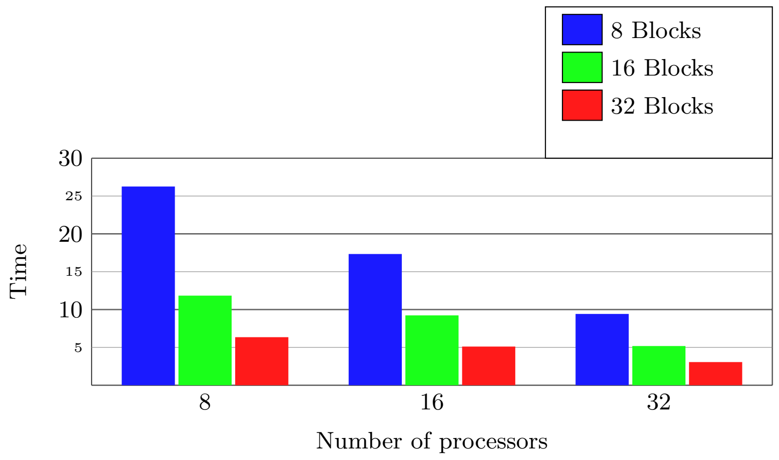

Fig. 3. Factorization Step

8 Blocks	16 Blocks	32 Blocks
3.432D+11	1.954D+11	1.028D+11

Table 1. Number of operations during the node elimination

Usually the first problematic part is the factorization during which the structure of the matrix determines how fast it will be done. From the results we got in the Fig.3, we benefit directly from the fact that MUMPS handles the three levels of parallelism.

	8 Processors	16 Processors	32 Processors
8 Blocks	799	437	256
16 Blocks	670	376	226
32 Blocks	554	323	205

Table 2. Average memory usage per processor during the factorization (in MBytes)

From the factorization results we retain the following information:

- The more blocks we have the faster the factorization goes. This is mainly due to the narrow band matrix given to MUMPS resulting from the small diagonal blocks size. We see this effect also from the number of operations to be done during the factorization (Table 1)
- The good locality of data –result of a good mapping during the analysis– gives a fast distributed computing.
- The memory usage (Table 2) decreases when the number of blocks or processor increases. This is a desirable property to expect for very large 3D PDE problems in particular.

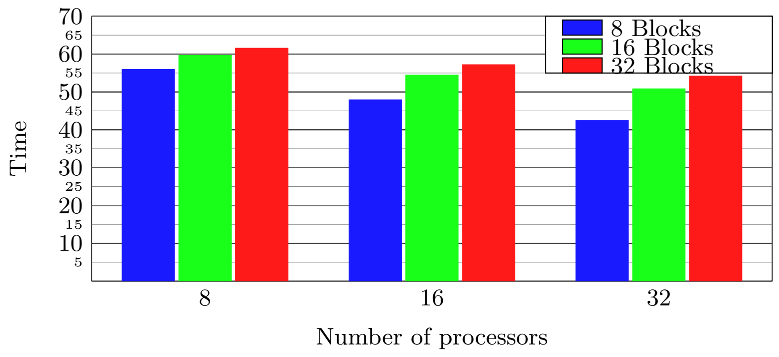

Fig. 4. Solve Step

For the solution phase we have:

- As during the factorization, the solution phase (Fig.4), is faster when we increase the number of processors. Even if the volume of communications increases, the computing phase becomes shorter (Table 3) at a faster rate.
- However when increasing the number of blocks the solution takes more time this is due to the fact that when we increase the number of blocks the size of the block diagonal matrix becomes bigger and so does the RHS, making the communication process a bit longer (Table 4). This process includes gathering the solution onto the master and sending the RHS to the processors.

The results in Table 3 shows the details about the time spent on each step in a solve call. We notice that the communication time takes a big part of the solving process when we increase the number of working processors and blocks. Implementing in parallel the rest of Block-CG's operations (daxpy, ddot, etc.) will help to get rid of the communication part.

		8 Processors	16 Processors	32 Processors
8 Blocks	Scatter	0.4	0.43	0.47
	Gather	0.62	0.67	0.69
	Computing	1.24	0.80	0.5
16 Blocks	Scatter	0.42	0.48	0.50
	Gather	0.65	0.70	0.74
	Computing	0.95	0.64	0.46
32 Blocks	Scatter	0.44	0.51	0.53
	Gather	0.62	0.79	0.83
	Computing	1.0	0.58	0.43

Table 3. Time spent on each part of one solve call (seconds)

		8 Processors	16 Processors	32 Processors
8 Blocks	Number of Iterations	23	23	23
	Total Time (sec)	56.03	48.02	42.53
	Computing Time (sec)	28.45	18.49	11.58
	Communication	49.22%	62%	72.77%
16 Blocks	Number of Iterations	27	27	27
	Total Time (sec)	59.03	54.46	50.92
	Computing Time (sec)	25.84	17.37	12.43
	Communication	56.22%	68.11%	75.59%
32 Blocks	Number of Iterations	27	27	27
	Total Time (sec)	61.65	57.29	54.30
	Computing Time (sec)	27.24	15.82	11.51
	Communication	55.82%	72.39%	78.80%

Table 4. Time spent in the solution phase

In our previous work results [13] the solution time increases when the number of processor increases, and it's heavily penalized with 16 blocks. It was due to the fact that the mapping was not taking in account the forest structure in the dependency tree. With the current fine tuning, MUMPS notices the structure of the matrix and maps the data according to that and we gain when increasing the number of processors.

6 Ongoing work

The current results show the good parallelism performance and scalability. It helped us to identify the bottlenecks and problems.

To implement the final targeted parallel version, we have to use the basic embedded functionalities already available in the MUMPS package, such as residual computation used in iterative refinement for instance, and design new ones whenever necessary.

We have also to define an user interface to address directly these functionalites within the parallel iterative solver in order to appropriately exploit the data distribution already established and handled by MUMPS. This will help us especially during the parallelisation of Block-CG's sequential parts and get rid of the actual but useless gather and scatter communications.

We plan also to experiment the potential of the final solver on matrices coming from real problem modelised on an industrial software for the simulation of atmospheric circulation above mountain in the field of wind energy production [12].

Acknowledgment *This particular work has strongly benefited from sustained interactions with the members of the MUMPS project development team. Indeed, the strategy for parallelism and the particular sparse structure exploited in this approach was needing fine tuning as well as some tiny adaptations of the MUMPS package to achieve encouraging results. We are therefore very grateful to all the MUMPS project members, and in particular to P. Amestoy, J-Y. L'Excellent, F-H. Rouet, and B. Uçar for the various fruitful exchanges. We also wish to thank K. Kaya from CERFACS for the very useful suggestions concerning pre-ordering strategies.*

References

1. Amestoy, P., Buttari, A., Combes, P., Guermouche, A., L'Excellent, J.Y., Pralet, S., Ucar, B.: Multifrontal massively parallel solver - user's guide of the version 4.9.4. (2009)
2. Amestoy, P., Duff, I., L'Excellent, J.Y., Koster, J.: A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM Journal on Matrix Analysis and Applications **23** (2001) 15–41
3. Cimmino, G.: Calcolo approssimato per le soluzioni dei sistemi di equaziono lineari. In: Ricerca Sci. II. Volume 9, I. (1938) 326–333
4. Arioli, M., Duff, I.S., Noailles, J., Ruiz, D.: A block projection method for sparse matrices. SIAM Journal on Scientific and Statistical Computing (1992) 47–70
5. Campbell, S.L., Meyer, J.C.D.: Generalized inverses of linear transformations. Pitman, London (1979)
6. Bartels, R.H., Golub, G.H., Saunders, M.A.: Numerical techniques in mathematical programming. In J.B. Rosen, O. L. Mangasarian, K.R., ed.: Nonlinear Programming. Academic Press, New York (1970)
7. Hachtel, G.D.: Extended applications of the sparse tableau approach - finite elements and least squares. In Spillers, W.R., ed.: Basic question of design theory. North Holland, Amsterdam (1974)

8. Hageman, L.A., Young, D.M.: Applied Iterative Methods. Academic Press, London (1981)
9. O'Leary, D.P.: The block conjugate gradient algorithm and related methods. Linear Algebra Appl. **29** (1980) 293–322
10. Arioli, M., Ruiz, D.: Block conjugate gradient with subspace iteration for solving linear systems. In: Second IMACS International Symposium on Iterative Methods in Linear Algebra, Blagoevgrad, Bulgaria (1995) 64–79
11. Arioli, M., Drummond, A., Duff, I.S., Ruiz, D.: A parallel scheduler for block iterative solvers in heterogeneous computing environments. In: Proceedings of the seventh SIAM conference on Parallel Processing for Scientific Computing, Philadelphia, USA, SIAM (1995) 460–465
12. Silva Lopes, A., Palma, J.M.L.M., Castro, F.A.: Simulation of the Askervein flow. Part 2: Large-eddy simulations. Boundary-Layer Meteorology **125** (2007) 85–108
13. Balsa, C., Guivarch, R., Raimundo, J., Ruiz, D.: MUMPS Based Approach to Parallelize the Block Cimmino Algorithm. In: International Meeting High Performance Computing for Computational Science (VECPAR), Toulouse, 2008. http://vecpar.fe.up.pt/2008/papers/45.php