Memory Optimization to Build a Schur Complement in an Hybrid Solver

Astrid Casadei, Pierre Ramet

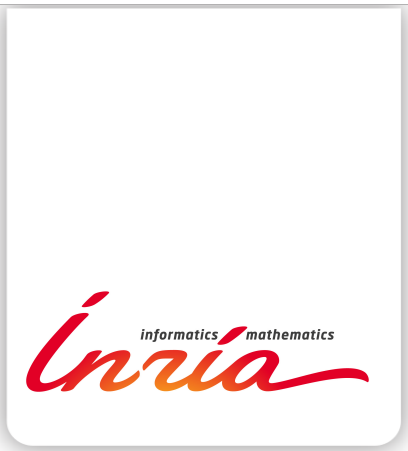
RESEARCH REPORT

N° 7971

21 Mars 2012

Project-Team Bacchus

ISSN 0249-6399 ISRN INRIA/RR--7971--FR+ENG



Memory Optimization to Build a Schur Complement in an Hybrid Solver

Astrid Casadei, Pierre Ramet

Équipe-Projet Bacchus

Rapport de recherche n° 7971 — 21 Mars 2012 — 11 pages

Résumé : La résolution d'un système linéaire $Ax = b$ en parallèle où A est une matrice creuse de grande taille est un problème commun à de nombreuses applications dans le domaine de la simulation numérique. Les approches hybrides, basées sur une décomposition de domaine et un couplage par complément de Schur, sont actuellement très largement étudiées. Dans cette méthode, un solveur direct est utilisé comme une brique de base pour chaque sous-domaine de la matrice, mais les surcoûts mémoires, même si ils sont réduits par rapport à une approche directe complète, restent un problème limitant. Dans cette étude, nous proposons une technique pour réduire les surcoûts mémoires générés lors de la construction du complément de Schur dans un solveur direct. Notre approche permet de réduire de 10% à 30% le pic mémoire sur chaque processus pour des cas tests de référence.

Mots-clés : Algèbre linéaire creuse, Complément de Schur, Solveur hybride, Calcul Haute Performance

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

351, Cours de la Libération
Bâtiment A 29
33405 Talence Cedex

Memory Optimization to Build a Schur Complement in an Hybrid Solver

Abstract: Solving linear system $Ax = b$ in parallel where A is a large sparse matrix is a very recurrent problem in numerical simulations. One of the state-of-the-art most promising algorithm is the hybrid method based on domain decomposition and Schur complement. In this method, a direct solver is used as a subroutine on each subdomain matrix. This approach is subject to serious memory overhead. In this paper, we investigate new techniques to reduce memory consumption during the build of the Schur complement by a direct solver. Our method allows memory peak reduction from 10% to 30% on each processus for typical test cases.

Key-words: Sparse linear algebra, Schur complement, Hybrid solver, High performance computing

1 Introduction

Solving large sparse linear systems $Ax = b$ on distributed memory parallel machines is a topic of great interest for the last ten years. The different methods proposed range from pure direct methods [1], which are robust but time and space expensive, to pure iterative methods, which are cheaper but may suffer from serious convergence problem. Nowadays, most solvers relies on hybrid methods : a preconditioner is first computed, possibly using ideas from direct solvers but consuming less memory and time ressources ; and then, an iterative method taking advantage from the preconditioner is used. In this paper we will focus on the hybrid method known as domain decomposition with Schur complement [2,3,4].

In this method, a domain decomposition is performed to split the matrix into a number of subdomain matrices. Each subdomain matrix is then factorized with a call to a direct solver. One drawback of this approach is that direct solvers use a large amount of memory to make the factorizations. Our goal in this paper is to deal with these memory issues. In the next section, we will present related works about parallel hybrid solvers. Then, we will describe some ideas to reduce memory overcosts corresponding to the factorization of a subdomain matrix with a direct solver. Before concluding and describing prospects for this work, the last section will present experiments we performed with the PaStiX solver [5] to validate our approach on real-world challenging matrices.

2 Related works

The goal of a direct method is to factorize matrix A in two matrices L and U respectively lower and upper triangular such that $A = LU$. This makes the resolution $Ax = b$ trivial. Concerning supernodal approach [6,7] (see [8] for multifrontal method), the factorization is performed as follow. After some preprocessing on A (mainly the build of the final structure of the factorized matrix including fill-in), consecutives columns of the matrix are grouped into column-blocks (supernodes), each containing a number of dense blocks. Two main strategies are then possible to schedule the factorization of a column-block c [9]. The strategy called *right-looking* consists in first factorizing c and then reporting contributions on blocks on the right of c. In the other way, called *left-looking*, contributions from column-blocks on the left of c which impact blocks in c are reported, and then c is factorized. In practice, right-looking is preferred by state-of-the-art solvers since left-looking has been shown to induce time overhead.

Concerning parallel implementation with distributed memory, each processus own several column-blocks, which are allocated at the beginning of the factorization (static mapping). When a contribution from a column-block a to a column-block b has to be reported, two cases occur : b may be owned by the same processus as a or not. If b is local, contribution can be reported immediatly. If b is not local, the contribution is stored in a local extra-memory space called *fan-in*,

as explained in [10]. The sending of contribution is delayed in order to reduce communication time overhead. Unfortunatly, the amount of extra-memory required by this method can be very large. Some previous attempts to reduce this memory consumption have been studied, see [11] for instance.

Several hybrid solvers have already been developped, relying on different strategies. Solvers Gprems [12], Maphys [13] and Hips [14] use a domain decomposition. The Gprems solver use a Schwartz-multiplicative method [15] to compute the global preconditioner from the factorized subdomain matrices. Using a Schur complement method, the Maphys solver additionaly provides an interface which will allow to call a direct solver to make the subdomain matrices factorizations. Our interest of the Hips solver is that the ordering of the interfaces between subdomains allows to build some generic and scalable preconditioners (numerically and in term of memory). Moreover, the part of direct factorization is controlled by the size of the domains and this is independent from the number of processors thanks to a multi-domain-per-processor parallelization scheme [16].

Some other techniques have been studied more recently to enhance the performance of computing an approximate Schur complement [17]. In this work, the authors proposed solutions to take advantage of the sparsity of right-hand-side vectors ; the cost to form update matrices is also reduced thanks to an efficient load balancing technique for sparse matrix-matrix multiplications and communication/computation overlapping.

3 Reduction of extra-memory used during Schur complement computation

The scheme of a domain decomposition method with Schur complement is as follow. First, a domain decomposition is applied. Unknowns in the interior of subdomains are grouped by subdomain and placed on the first rows of the matrix. Unknowns on the interface thus correspond to the last rows of the matrix, as shown on Figure 1 on the left. Submatrix B_i corresponds to edges of the interior of subdomain i, submatrix C_i to edges of the interface and submatrices E_i and F_i to the edges between interior and interface. For each subdomain matrix composed by B_i, E_i, F_i and C_i, a call to a direct solver is done. The direct solver then factorizes B_i, replaces E_i and F_i by $G_i = L_i^{-1}F_i$ and $W_i = E_iU_i^{-1}$ respectively and reports contributions to form the local Schur complement $S_i = C_i - W_iG_i$. G_i and W_i are freed. Next, the hybrid solver makes the final preconditioner, assembling the local Schur complements in a global one S which is incompletly factorized in $\tilde{L}_S\tilde{U}_S$.

Since we are mainly interested by the direct solver calls, indices i will be forgotten in the following. Additionnaly, B will be referred to as the "direct part", and $E/W/F/G$ as the "Schur coupling".

During a call to the direct solver, memory overhead may occur for two reasons. The first one is related to the fan-in implementation, that is to say the local storage of non-local contributions. The second overhead is due to the coupling

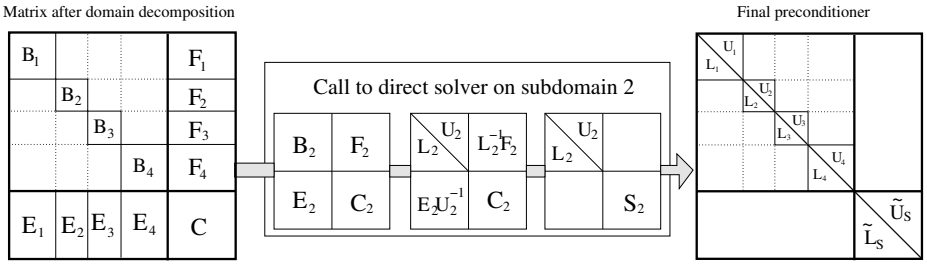


Figure 1. Steps of the preconditionner computation. Matrices $B_i/L_i/U_i$ correspond to the interior of subdomain i (also called *direct part*), matrices C_i/S_i to the interface and matrices $E_i/F_i/W_i/G_i$ to the Schur coupling. This matrix might be factorized with 8 processus, each subdomain beeing factorized with two processus.

matrices W and G, which remain allocated during the whole computation and are freed only at the end.

3.1 Block allocations

A first idea to reduce memory consumption is to postpone the allocation of each block until the moment it is really needed. Actually, the allocation of a block q is required in two cases :

- when the column-block z it belongs to is factorized
- and when a contribution from a column-block on the left of z (which may be local or non-local) has to be reported in q.

The second idea tends to reduce the overhead due to the coupling matrices W and G. Considering the W matrix column-blocks one by one, it can be noticed that a column-block may be freed as soon as it has been treated. This is due to the fact that we use a right-looking algorithm to perform factorization. Thus, when a column-block z has been factorized, we decided to free all the blocks in the coupling part of z.

Those ideas are illustrated on Figure 2 in the case of a single processus. At the beginning, no block is allocated. When column-block 1 is treated, all blocks of this column-block are allocated. Due to contributions on the right, almost all blocks of column-block 2 and 3 required to be allocated as well. Next, coupling blocks of column-blocks 1 - which are now useless - are freed, and factorization of column-block 2 is proceeded. This leads to allocate the last not-yet-allocated block in column-block 2. As it can be seen, contributions from column-block 2 do not require any new block allocation. Then, coupling part of column-block 2 is freed and the factorization is carried on the next column-blocks.

3.2 Left-right-looking version

The problem of the previous algorithm is that a lot of blocks may be allocated very soon. This is due to the right-looking scheme : the factorization of a

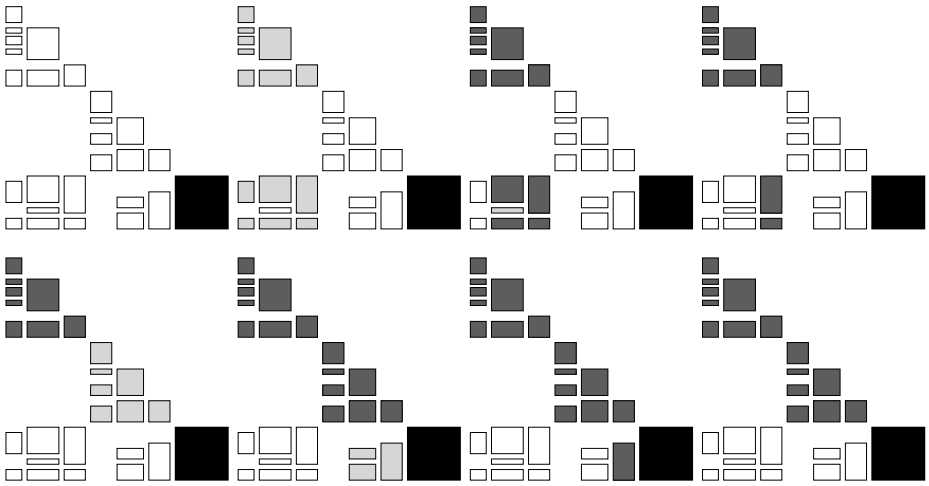


Figure 2. Illustration of the algorithm consisting in allocating memory blocks as late as possible and freeing Schur coupling blocks as soon as possible, on a single processus. Each step correspond to the factorization of a column-block. Non-allocated blocks are shown in white, newly allocated blocks in light grey and other allocated blocks in dark grey. The black square is the Schur complement.

column-block is responsible for allocating a lot of blocks on the right. To handle that, a solution would be to use a left-looking scheme when dealing with local contributions. Unfortunately, since left-looking scheme requires to maintain a lot of blocks on the left allocated, it is not well-suited on the coupling part where we intend to free blocks early. Thus, we decide to introduce a mixed version : a right-looking algorithm is used, except for local contributions in the direct part.

This new strategy is illustrated on Figure 3, on the same matrix as in Figure 2. This time, when column-block 1 is treated, contributions from the direct part are not reported. Nevertheless contributions from the Schur coupling part of the column-block are still reported, which allow these two blocks to be freed. Next, column-block 2 is proceeded. Blocks of the direct part and the remaining non-allocated block of the Schur coupling part are allocated. Contributions from column-block on the left (i.e. column-block 1 only) are reported in the direct part of column-block 2. Contributions from the Schur coupling of column-block 2 are reported on the right, which does not require new allocations on this example. After the Schur coupling part of column-block 2 has been freed, column-block 3 is proceeded, and so on.

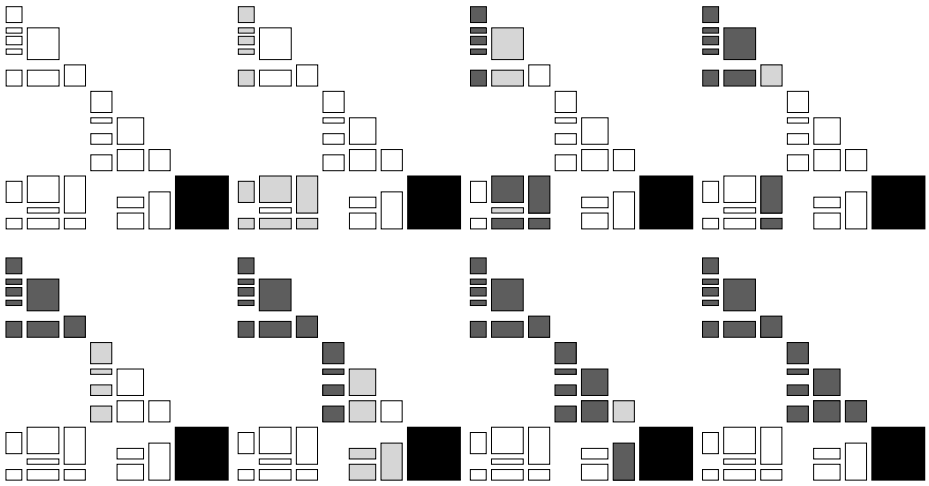


Figure 3. Illustration of a left-right looking algorithm on a single processus. Color conventions are the same as in Figure 2.

4 Numerical experiments

Experiments have been performed to evaluate the improvement allowed by our contributions. We chose two challenging matrices called `AUDI` (943 695 unknowns, 39 297 771 non-zeros) and `HALTERE` (1 288 825 unknowns, 10 476 775 non-zeros). We used the solver Maphys in order to split these matrices in several subdomains. Then, we computed the memory consumption when factorizing one of these subdomain matrices with the solver PaStiX, using strategies described in the previous section.

Figure 4 shows the evolution of memory consumption during the factorization of one subdomain of matrix `AUDI` using four processus and four threads per processus. The curves show memory consumption on processus 1 in three cases : when no optimization is done, when an allocation strategy is applied using a right-looking version (section 3.1), and when a mixed left-right-looking version is used (section 3.2). Comparing the two curves with optimizations, we can see that the left-right looking strategy take a great advantage from the fact that memory grows more gradually at the beginning of factorization. On this test case, the mixed left-right looking algorithm allows to reduce the memory peak from almost 10 % (and 30 % if we take the final memory consumption as reference), which is quite satisfying.

To measure with accuracy the memory saved by our method, we introduced three parameters defined as follow :

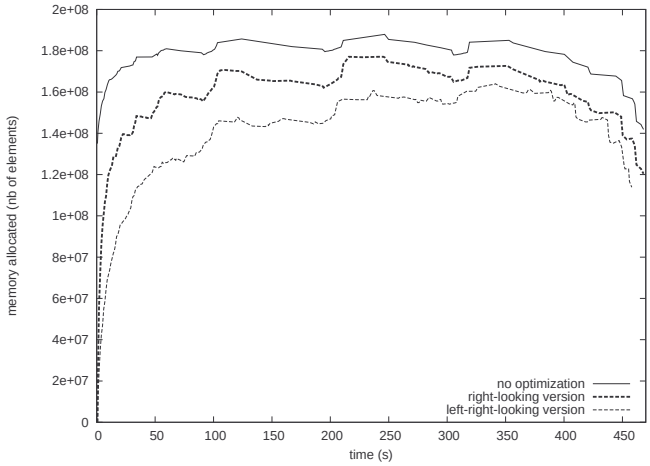


Figure 4. Comparison between memory allocated during preconditioner computation when (a) no optimization is done, (b) allocation strategy is applied using a right-looking algorithm and (c) mixed left-right looking version is used. *Test case : domain 1 of matrix* `AUDI` *cut into two subdomains ; 4 processus and 4 threads per processus were used to factorization a subdomain ; curves are shown for processus 1.*

$$G_{tot} = 1 - \frac{\sum_{p=0}^{nproc-1} m_p^*}{\sum_{p=0}^{nproc-1} m_p} ; G_{max} = 1 - \frac{\max_{p=0}^{nproc-1} m_p^*}{\max_{p=0}^{nproc-1} m_p} ; G_{peak} = 1 - \frac{\sum_{p=0}^{nproc-1} m_p^* - r_p}{\sum_{p=0}^{nproc-1} m_p - r_p}$$

where *nproc* is the number of processus, m_p the memory peak on processus p when no optimization is done, m_p^* the memory peak on p when optimization are allowed and r_p the final memory (*i.e.* the memory required to store L_B, U_B and S for the subdomain considered). G_{tot} represents the average memory reduction on all processus. G_{peak} is the average memory peak reduction on all processus (indeed reference is final memory instead of zero). G_{max} gives the memory reduction on the worst processus.

Matrices `AUDI` and `HALTERE` have been factorized using different cut-out in subdomains, and varying the number of processus and threads used ; see Table 1 for results. When the factorization use a single processus, memory peak is reduced from 58% to 72% on the runs performed. Note that a single processus means there is no extra-memory used for the fan-in, thus the memory peak is only due to the coupling matrices allocated temporarily. Since the amount of memory we potentially save by using our new strategy does not increase when *nproc* does, the peak reduction decrease when *nproc* increase. However, we still found significant peak reduction with 4 and 8 processus.

Table 2 shows the influence of the number of threads used per processus. We can see that there is no significant degradation of the memory reduction when

Test case	domain size	Schur size	*nproc*	G_{tot} (%)	G_{max} (%)	G_{peak} (%)
`AUDI` cut in 8 subdomains, dom. 3	128 955	9 675	1	17,8	17,8	58,6
			4	9,0	8,6	13,6
			8	6,6	4,4	8,4
			16	2,3	0,6	2,7
`AUDI` cut in 8 subdomains, dom. 5	112 656	3 585	1	13,7	13,7	72,3
			4	7,2	9,1	15,6
			8	4,7	0	7,8
			16	2,1	0	3,1
`HALTERE` cut in 8 subdomains, dom. 5	164 372	3 483	1	12,8	12,8	61,9
			4	5,8	4,5	11
			8	5,8	2,3	9,6
			16	1,9	1,5	2,8
`HALTERE` cut in 8 subdomains, dom. 6	165 586	4 038	1	12,6	12,6	59,7
			4	7,7	6,8	15,7
			8	5,4	3,2	8,8
			16	2,5	2,3	3,8

Table 1. Influence of the number of processus per subdomain. *Test case : all tests have been done with 4 threads per processus.*

	G_{tot} (%)			G_{max} (%)			G_{peak} (%)		
nb of threads	1	4	8	1	4	8	1	4	8
memory reduction	8,3	9	7,3	8,1	8,6	6,3	12,2	13,6	11

Table 2. Influence of the number of threads per processus. *Test case : domain 3 of matrix `AUDI` cut into 8 subdomains ; 4 processus were used to factorize the subdomain.*

the number of threads is increased.

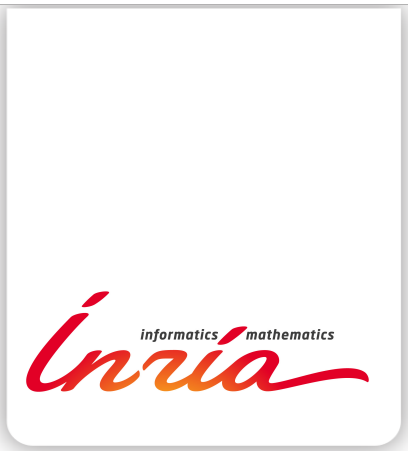
The domain decomposition method allows two levels of processus parallelism. The first one is given by the decomposition of the matrix in multiple subproblems. The second one relies on the factorization of each subdomain in parallel with shared and distributed memory. Since the Hips solver allows already a high degree of parallelism with the domain decomposition, a few number of processus (and possibly lots of threads) will be used to perform the factorization of each subdomain with the PaStiX solver. Thus, in the conditions we are interested in, the optimizations we proposed allow significant memory reduction.

5 Conclusion and future works

Direct methods used by hybrid solvers are memory-consuming during the building step of the Schur complement. In this paper, we presented improvements on factorization algorithm which allow to reduce memory peak on realistic test cases. One of the interest of the Hips solver is that the part of direct factorization is controlled by the size of the domains and this is relatively independent from the number of processors thanks to a multi-domains per processor parallelization scheme. With our improvements, we can now consider bigger sub-problems assigned to a direct factorization with respect of memory constraints by using the parallel implementation of the PaStiX solver. The direct solver can easily scale in terms of performances with several nodes composed of multicore chips and forthcoming GPU accelerators. The memory peak can be reduce by 10% to 30% for such a middle size parallelism. Additionaly, we will work on a way to optimize the parallel schedule during the factorization in order to be more suitable with a mixed left-right-looking implementation.

Références

1. Duff, I.S., Erisman, A.M., Reid, J.K. : Direct methods for sparse matrices. Oxford University Press (London 1986)
2. Saad, Y., Suchomel, B. : Arms : An algebraic recursive multilevel solver for general sparse linear systems. Technical report, Numer. Linear Alg. Appl (1999)
3. Made, M.M.M., van der Vorst, H.A. : Parallel incomplete factorizations with pseudo-overlapped subdomains. Parallel Computing **27**(8) (2001) 989–1008
4. Carvalho, L.M., Giraud, L., Tallec, P.L. : Algebraic two-level preconditioners for the schur complement method. SIAM J. Scientific Computing **22** (1998) 200–1
5. Hénon, P., Ramet, P., Roman, J. : PaStiX : A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. Parallel Computing **28**(2) (2002) 301–321
6. Heath, M.T., Ng, E., Peyton, B.W. : Parallel algorithms for sparse linear systems. SIAM Rev. **33**(3) (1991) 420–460
7. Rothberg, E., Gupta, A. : An efficient block-oriented approach to parallel sparse Cholesky factorization. SIAM J. Sci. Comput. **15**(6) (1994) 1413–1439
8. Amestoy, P.R., Duff, I.S., L'Excellent, J.Y. : Multifrontal parallel distributed symmetric and unsymmetric solvers. Comput. Methods in Appl. Mech. Eng. **184** (2000) 501–520
9. Rothberg, E., Gupta, A. : An evaluation of left-looking, right-looking, and multifrontal approaches to sparse cholesky factorization on hierarchical-memory machines. Int. J. High Speed Comput. **5** (1993) 537–593
10. Ashcraft, A., Eisenstat, S.C., Liu, J.W.H. : A fan-in algorithm for distributed sparse numerical factorization. SIAM Journal on Scientific and Statistical Computing **11** (1990) 593–599
11. Hénon, P., Ramet, P., Roman, J. : Efficient algorithms for direct resolution of large sparse system on clusters of SMP nodes. In : SIAM Conference on Applied Linear Algebra, Williamsburg, USA (2003)
12. Kahou, G.A.A., Nuentsa-Wakam, D. : Parallel GMRES with a multiplicative schwarz preconditioner. (2010)
13. Giraud, L., Haidar, A. : Parallel algebraic hybrid solvers for large 3d convection-diffusion problems. Numerical Algorithms **51** (2009) 151–177 10.1007/s11075-008-9248-x.
14. Gaidamour, J., Hénon, P. : A parallel direct/iterative solver based on a Schur complement approach. In : IEEE 11th International Conference on Computational Science and Engineering, Sao Paulo Brésil (2008) page 98–105
15. Widlund, O. : Some Schwarz methods for symmetric and nonsymmetric elliptic problems. Technical Report TR1991-581, New York University (1991)
16. Hénon, P., Saad, Y. : A parallel multilevel ilu factorization based on a hierarchical graph decomposition. SIAM Journal of Scientific Computing (2006)
17. Yamazaki, I., Li, X.S. : On techniques to improve robustness and scalability of a parallel hybrid linear solver. VECPAR'10, Berlin, Heidelberg, Springer-Verlag (2011) 421–434



RESEARCH CENTRE
BORDEAUX – SUD-OUEST

351, Cours de la Libération
Bâtiment A 29
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399