

Exploiting Thread-Level Parallelism in the Iterative Solution of Sparse Linear Systems

José I. Aliaga^{a,1}, Matthias Bollhöfer^{b,2}, Alberto F. Martín^{a,1}, Enrique S. Quintana-Ort^{a,1}

^a*Dpto. de Ingeniería y Ciencia de Computadores, Universidad Jaime I, 12.071-Castellón (Spain)*

^b*Institut Computational Mathematics, TU Braunschweig, 38106 Braunschweig (Germany)*

Abstract

We investigate the efficient iterative solution of large-scale sparse linear systems on shared-memory multiprocessors. Our parallel approach is based on a multilevel ILU preconditioner which preserves the mathematical semantics of the sequential method in ILUPACK. We exploit the parallelism exposed by the task tree corresponding to the nested dissection hierarchy (task parallelism), employ dynamic scheduling of tasks to processors to improve load balance, and formulate all stages of the parallel PCG method conformal with the computation of the preconditioner to increase data reuse. Results on a CC-NUMA platform with 16 processors reveal the parallel efficiency of this solution.

Key words: Large sparse linear systems, factorization-based preconditioning, preconditioned conjugate gradients, task-level parallelism, shared-memory multiprocessors

1. Introduction

One of the numerical problems which arises most frequently in modern linear algebra is the solution of large sparse systems of equations. In applications involving the discretization of partial differential equations (PDEs), the efficient solution of sparse linear systems is one major computational task. The same key subproblem appears in many other application areas as, e.g., in quantum physics or circuit and device simulation. This is also the case for nonlinear equations as well as large-scale eigenvalue computations since the iterative methods employed in those problems usually require the solution of multiple linear systems. As the size of the underlying applications increases (e.g., three spatial dimensions for PDEs or increasing number of devices in integrated circuits), the development of fast and efficient numerical solution techniques becomes crucial.

Sparse direct solvers have proven to be extremely efficient for a large class of application problems, but they are also known to perform poorly for others. For example, direct methods are highly efficient when applied to two-dimensional (2D) PDEs, but they dramatically slow down for three-dimensional (3D) problems because of considerable fill-in [1]. Approximate factorization techniques combined with Krylov subspace methods are an appealing alternative for these kind of application problems where fill-in becomes an issue [2].

In this paper we present a parallel variant of a highly efficient multilevel incomplete LU factorization (ILU) method which has been successfully applied in sparse large scale application problems with up to millions of equations [3, 4, 5]. This multilevel method is based on the so-called inverse-based approach which lays the foundations for the software package ILUPACK³. The fundamental difference between ILUPACK and other common ILU preconditioners resides in controlling the growth of the inverse triangular factors; see [3, 6] for a theoretical justification of this approach and [7] for an extensive empirical comparative study of the sequential variant of ILUPACK and other preconditioners.

Email addresses: aliaga@icc.uji.es (José I. Aliaga), m.bollhoefer@tu-bs.de (Matthias Bollhöfer), martina@icc.uji.es (Alberto F. Martín), quintana@icc.uji.es (Enrique S. Quintana-Ort)

¹Supported by A.I. Hispano-Alemanas HA2007-0071 and CICYT project TIN2005-09037-C02-02.

²Supported by DAAD grant D/07/13360.

³<http://ilupack.tu-bs.de>.

Parallel computation of ILU-type preconditioners and the iterative solution of sparse linear systems on distributed-memory architectures has been extensively studied in the past. Jones and Plassman’s approach [8] identifies a high degree of parallelism in the computation of ILU(0) preconditioners by means of a multicoloring of the undirected graph representing the nonzero adjacency structure of the coefficient matrix. The codes are available in BlockSolve95, a parallel library which can be installed as an external package of PETSc⁴. Karypis and Kumar [10] introduced parallel threshold (PILUT) preconditioners, and Hysom and Pothen [9] developed level-of-fill (PILU(k)) preconditioners; both approaches are based on graph partitionings with balanced subdomains and small edge cuts. A similar method is presented in [11] by Made and Van der Vorst. Within each subdomain, these approaches order the interior nodes prior to the boundary nodes, ensuring that, during the elimination of the interior nodes, no fill-edges (and thus new dependencies) arise between interior vertices belonging to different subdomains. However, during the elimination of the boundary nodes, fill-edges arise among nodes belonging to different subdomains, which have to be managed for efficient parallel computation. While PILU(k) is based on a multicoloring of the subdomain intersection graph [9], PILUT uses a multistage algorithm [10] which starts by explicitly forming the graph resulting from the elimination of the interior nodes. At each stage of the algorithm, an independent subset of the boundary nodes is computed and eliminated, and the new graph which results from this elimination is explicitly formed for the next stage. The PILUT and PILU(k) codes are distributed as part of the Hypre⁵ library. A somewhat different approach is the one of pARMS [12], the parallel version of the ARMS (Algebraic Recursive Multilevel Solvers) [13] library. pARMS is based on a domain decomposition approach which uses ARMS for the subdomain solves, followed by a preconditioned iterative scheme for the Schur complement corresponding to the boundary nodes.

The main contribution of the parallelization approach presented in this paper consists of the interplay between the algebraic levels generated by the multilevel ILU method and the concurrency which is induced by the nested dissection hierarchy. The resulting method is multilevel in two directions, accommodating the semantics of the algebraic approach while exploiting a high degree of parallelism simultaneously. Another important novelty of our parallelization approach is that we target the recent uprise of multi-core processors and, in preparation for the future many-core systems, we address the iterative solution of linear systems using *techniques* which may yield higher performance for shared-memory architectures than message-passing. In particular, our method exhibits the following features:

- Algebraic parallelization based on a task tree (built from nested dissection) to match the degree of hardware parallelism of the target platform.
- Parallelization driven by the dependencies in the task tree (*task-parallelism*), not limited by the control structures of a code (*control-parallelism*).
- Dynamic scheduling to improve load-balancing during execution managed by a run-time that can be tuned for a particular target platform and a specific application problem.
- Application of the preconditioner conformal with the logical structure built during its construction to reduce the number of data transferences (cache misses). The same requirement is also set for other major operations in PCG: matrix-vector products, inner products, and *axpy* updates.
- Use of OpenMP to parallelize all stages of the parallel solution of the linear system: computation and application of the preconditioner as well as the other matrix operations in the preconditioned conjugate gradient (PCG) iterative method [14].

We believe that the analysis of these techniques in combination with the focus on a platform like a shared-memory multiprocessor are two significant original contributions of this paper. While in this paper we restrict our discussion to symmetric positive definite (SPD) systems of equations, we expect our parallelization approach to be also suitable for more general cases.

The paper is organized as follows. In Section 2 we present the general framework for the parallelization approach: the iterative solution process is first reviewed, and the serial preconditioner and its parallel version are presented next.

⁴<http://www.mcs.anl.gov/petsc/petsc-as>.

⁵https://computation.llnl.gov/casc/linear_solvers/sls_hypre.html.

Section 3 addresses the parallelization of the application of the parallel preconditioner and other matrix and vector operations inside the PCG method. In Section 4 we demonstrate the effectiveness of our method for several large-scale examples which confirm that speed-up close to linear can be attained using a moderate number of processors on a shared-memory machine. A few concluding remarks close the paper in Section 5.

2. Iterative solution of symmetric positive definite linear systems

Consider the linear system of equations

$$Ax = b, \quad (1)$$

where $A \in \mathbb{R}^{n,n}$ is a large sparse SPD matrix, $b \in \mathbb{R}^n$, and $x \in \mathbb{R}^n$ is the sought-after solution. We propose to solve (1) iteratively using the PCG method, with the preconditioner based on a *multilevel incomplete Cholesky (MIC) decomposition* [3]. We start by reviewing the foundations of the PCG method in subsection 2.1. The framework of the serial preconditioner and its parallel variant are presented in subsections 2.2 and 2.3, respectively. The computational aspects related with the efficient computation of the parallel preconditioner are presented in subsection 2.4 and, finally, subsection 2.5 briefly discusses the numerical relations between the sequential and the parallel preconditioners, offering evidence that the semantics of our approach to parallelization is close to its sequential counterpart.

2.1. Preconditioned conjugate gradients

The PCG method is among the best iterative approaches to solve SPD systems. When applied to (1), the convergence rate of the method is strongly related to the condition number of the preconditioned system

$$\kappa = \lambda_{\max}(M^{-1}A)/\lambda_{\min}(M^{-1}A), \quad (2)$$

where an efficient preconditioner $M \approx A$ will cause the iteration to converge in a moderate number of steps. In practice, the PCG method consists of a sequence of matrix-vector multiplications, scalar products, *axpy* updates ($y := y + \alpha \cdot x$), and the application of the preconditioner M . Thus, the bulk of the computational cost of the method is in the matrix-vector products and the computation $M^{-1}A$. While the distribution of the cost among these two operations depends on the application problem, numerical experience with ILUPACK indicates that applying the preconditioner is often the most expensive operation.

2.2. The multilevel incomplete Cholesky preconditioner

The Cholesky decomposition of $A \in \mathbb{R}^{n,n}$ is defined as

$$A = \bar{L}\bar{L}^T = (LD^{1/2})(D^{1/2}L)^T = LDL^T, \quad (3)$$

where $L \in \mathbb{R}^{n,n}$ is unit lower triangular and $D \in \mathbb{R}^{n,n}$ is diagonal. In the incomplete Cholesky decomposition we compute an approximate factorization LDL^T such that $A = LDL^T + E$. Compared with (3), small perturbations are introduced in the form of a matrix E which consists of those entries dropped during the factorization process. However a small $\|E\|$ does not always imply that $M = LDL^T$ is a good preconditioner for A as, for the PCG method, the actual requisite is that the condition number of the preconditioned system $M^{-1}A = (LDL^T)^{-1}A$ is small. This is certainly fulfilled if $\|D^{-1/2}L^{-1}EL^{-T}D^{-1/2}\|$ is small.

Inverse-based incomplete factorization techniques [3] pursue the computation of a preconditioner LDL^T with $\|L^{-1}\| \leq \nu$ for some prescribed small $\nu > 1$. Typically, $\nu = 3$, $\nu = 5$ or $\nu = 10$ are good choices. The usage of inverse-based factorization techniques is grounded in two major theoretical properties. To review these, consider a partial incomplete factorization

$$A \equiv \begin{pmatrix} B & F^T \\ F & C \end{pmatrix} = LDL^T + E, \quad (4)$$

with $C \in \mathbb{R}^{m,m}$, L a unit triangular matrix satisfying $\|L^{-1}\| \leq \nu$, and matrices L and D partitioned as

$$L = \begin{pmatrix} L_B & 0 \\ L_F & I_m \end{pmatrix}, \quad D = \begin{pmatrix} D_B & 0 \\ 0 & S_C \end{pmatrix}, \quad (5)$$

where I_m refers to the identity square matrix of order m , D_B is a diagonal matrix and $S_C \in \mathbb{R}^{m,m}$ is the approximate Schur complement of B in A . A first argument in favor of inverse-based incomplete factorizations addresses the error that is introduced by dropping small entries. If $A = LDL^T + E$, then $L^{-1}AL^{-T} = D + L^{-1}EL^{-T} \equiv D + \bar{E}$ so that $\|\bar{E}\| \leq \nu^2\|E\|$. Depending on the method employed to obtain the approximate Schur complement S_C , the bound can be further improved to $\|\bar{E}\| \leq \nu\|E\|$; for details, refer to [3]. This addresses, at least partially, the objective of keeping $\|D^{-1/2}L^{-1}EL^{-T}D^{-1/2}\|$ small. The second and more important argument relates inverse-based decompositions and algebraic *multilevel methods*. To review this, consider the case when no dropping is applied in (4), i.e., $E = 0$. Denote the eigenvalues of A as $0 < \lambda_1 \leq \dots \leq \lambda_n$, and the eigenvalues of S_C in (5) as $0 < \mu_1 \leq \dots \leq \mu_m$. If L from (5) satisfies $\|L^{-1}\|_2 \leq \nu$, then the eigenvalues of S_C are in the range of the small eigenvalues of A ; that is,

$$\lambda_i \leq \mu_i \leq \frac{\nu^2}{1 - \|D_B^{-1}\|_2 \nu} \lambda_i \approx \nu^2 \lambda_i,$$

for all i such that $\lambda_i \ll 1/(\|D_B^{-1}\|_2 \nu)$; for the proof, refer to [6]. In terms of PDEs this states that the ‘‘coarse grid system’’ S_C reveals the low eigenmodes of the original system A . It also justifies the use of a small bound ν since otherwise the eigenvalue inclusion is meaningless. The objective of keeping $\|L^{-1}\|$ below a prescribed moderate bound ν is thus disclosed as an algebraic coarsening strategy. Typically the system at hand does not initially satisfy $\|L^{-1}\| \leq \nu$, except if some strict diagonal dominance

$$\sum_{\substack{j=1 \\ j \neq i}}^n |a_{ii}^{-1} a_{ij}| \leq 1 - \frac{1}{\nu},$$

for $i = 1, \dots, n - m$, is satisfied (cf. [6]). This is often an unrealistic case and, therefore, additional pivoting in (4) is necessary in practice to achieve a small $\|L^{-1}\|$. As a consequence, pivoting for inverse-based incomplete factorization techniques is fairly different from those employed in any other incomplete factorizations. Inverse-based pivoting requires to skip a large portion of unknowns (such as 30%) during the partial factorization.

Example 1. We illustrate the pivoting effect for the case of the 2D Laplacian problem based on a five-point-star difference stencil (see Figure 1, left) on a uniform square grid in two spatial dimensions having $n = 127$ grid nodes in each direction. The initial system is also reordered using nested dissection which can be seen observed the similarity between Figure 1 and the graph G_A in Figure 3. The ILUPACK pivoting strategy is repeated for S_C and introduces several levels of approximate partial factorizations of type (4) in order to successively force $\|L^{-1}\| \leq \nu$ on every level. We also illustrate the total multilevel preconditioner (see Figure 1, right) composed over seven levels along with the size of the system matrices A, S_C and further coarser level matrices. I.e., we state the matrices $L + L^T$ in the diagonal blocks and F and F^T in the sub and super block diagonal part and repeat this representation for S_C and all subsequent levels.

Example 1 illustrates the necessity of a multilevel approach to obtain an inverse-based incomplete factorization. Moreover, the parallelization of this approach is significantly more challenging than that of a standard incomplete factorization, where pivoting is used occasionally, as a safeguard, and where the multilevel formulation is rare. The efficient parallelization of an inverse-based decomposition is complicated by the need of interlacing algebraic levels with the concurrency constraints.

The pivoting strategy in ILUPACK employs an estimation of $\|L^{-1}\|$ obtained along with the computation of the incomplete Cholesky decomposition. Specifically, if at some step k of the factorization, the norm of the k -th row of L^{-1} exceeds the bound ν , then the k -th row and column of the matrix are moved, respectively, to the last row and column of the matrix; otherwise the factorization continues (see Figure 2 for a sketch of the pivoting process). This finally results in a partial factorization of a permuted system. Rejected columns/rows can be viewed as *bad* pivots with respect to the inverse constraint $\|L^{-1}\| \leq \nu$.

The factorization procedure stops when only rejected pivots remain in the rows and columns yet to be factorized. In other words, the trailing principal submatrix starting at some row/column $\hat{k} + 1$ only contains elements from those rows and columns which have been rejected. The computation of the Schur complement with respect to the factorized $\hat{k} \times \hat{k}$ leading part completes the first *algebraic level* (see Figure 2). For the next algebraic level, the whole method is restarted on the Schur complement. This recursive process transforms this approach into a multilevel method we will refer to as the MIC (decomposition). The computation can be expressed algorithmically as follows:

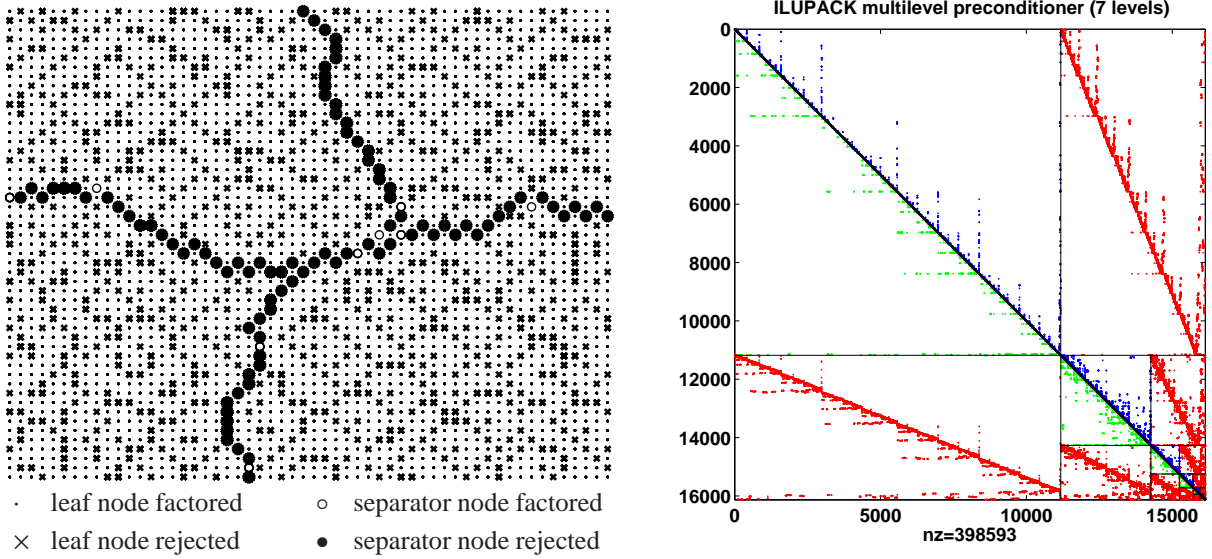


Figure 1: sequential MIC (first algebraic level) applied to 2D Laplacian (left), sequential MIC skeleton accumulated over all levels (right).

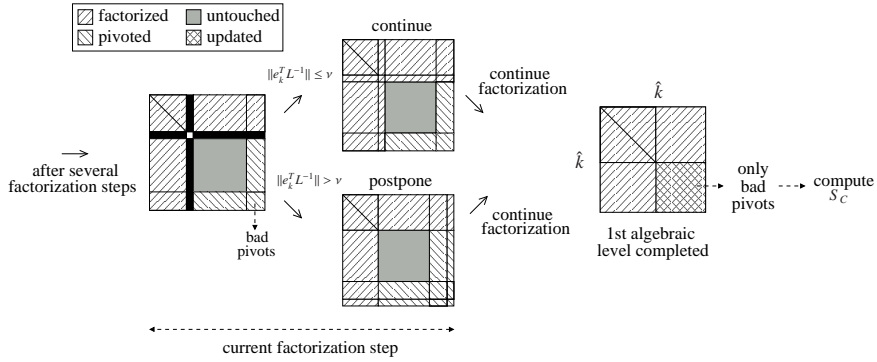


Figure 2: ILUPACK pivoting strategy.

1. Reorder $A \rightarrow P^T A P = \hat{A}$ using some fill-reducing ordering matrix P . In addition, a diagonal scaling $A \rightarrow S A S$ is also applied such that the diagonal entries become one. For simplicity, hereafter we will not mention this preprocessing step.

2. Compute a partial incomplete Cholesky decomposition of \hat{A} :

$$\hat{P}^T \hat{A} \hat{P} \equiv \begin{pmatrix} B & F^T \\ F & C \end{pmatrix} = LDL^T + E, \quad (6)$$

where $C \in \mathbb{R}^{n-\hat{k}, n-\hat{k}}$ is the trailing principal containing only elements from rejected rows and columns, E refers to some error matrix due to dropping (see [3] for details), and L, D are defined as in (5), with $\|L^{-1}\| \lesssim \nu$.

3. Proceed to the next algebraic level by repeating steps 1 and 2 with $A \equiv S_C$ until S_C is void or “dense enough” to be factorized using a dense Cholesky solver.

The arithmetic intensity (ratio of flops to memory accesses) of the kernels in the MIC decomposition is usually much lower than that of the operations arising in sparse direct methods, basically because of the presence of dropping.

In the MIC, entries are dropped on-the-fly based on available numerical information so that, in general, it is not possible to know the sparsity pattern of the factor(s) in advance via, e.g., a preliminary symbolic analysis. Hereby, exploiting (dense) level-3 BLAS for the MIC exhibits little appeal other than at step 3. Nevertheless, the solution of linear systems via the MIC is considered a competitive alternative to BLAS-3-based direct methods whenever the fill-in in the approximate factors is within a modest multiple of the number of nonzero elements in A .

2.3. Parallel multilevel factorization

Our parallelization approach is *algebraic*, i.e., it is exclusively based on the information derived from the sparsity pattern of the linear system. In particular, we exploit the connection between symmetric sparse matrices and undirected graphs. Consider the graph $G_A = \{V, E\}$ associated with matrix $A = (a_{ij})$, consisting of nodes $V = \{1, \dots, n\}$ and edges $E = \{\{i, j\} : a_{ij} \neq 0 \wedge i \neq j\}$. Thus, G_A reflects the nonzero pattern (excluding the diagonal) of A .

Nested dissection is a heuristic ordering strategy that starts by finding a *small* subset of G_A , called vertex or node separator, which splits G_A into two subgraphs of roughly *equal dimension*. These two subgraphs are disconnected as there is no edge in G_A between them. Thus, they identify blocks of rows/columns of A which can be factorized independently (hence in parallel). The subgraphs are next ordered recursively via nested dissection, if their dimension is still “large”, or, e.g., using minimum degree otherwise. The result of applying this ordering procedure is a permuted system matrix $A \rightarrow \Pi^T A \Pi = A_1$. In Figure 3, G_A is recursively split into four subgraphs (1,1), (1,2), (1,3) and (1,4), first using separator (3,1) and then repeatedly by separators (2,1) and (2,2). Gaussian elimination applied to the corresponding reordered system matrix allows to factorize the diagonal blocks associated with the subgraphs (1,1), (1,2), (1,3) and (1,4) independently. After that, the elimination proceeds with (2,1) and (2,2) in parallel until, finally, separator (3,1) is treated. This property is captured by the task dependency tree in Figure 3. Parallelism revealed by the task tree (*tree parallelism*) has been heavily used in parallel sparse direct methods over the past years (see, e.g., [15, 16]).

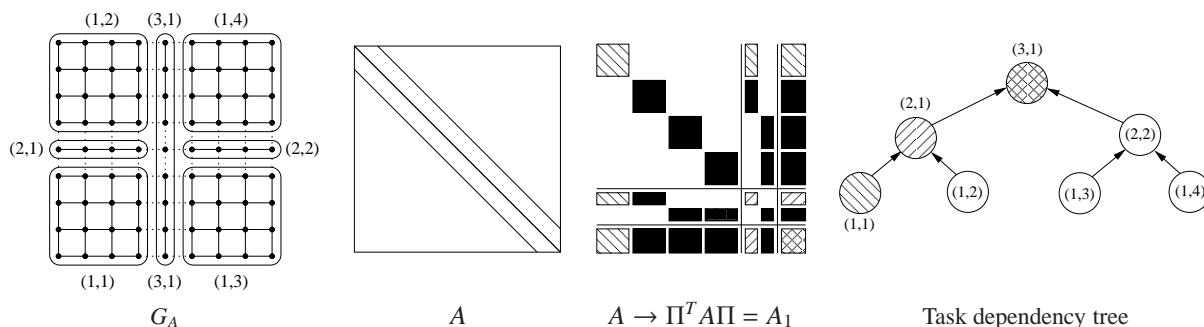


Figure 3: Nested dissection ordering. From left to right: nested dissection, natural ordering, nested dissection reordering, and task dependency tree.

The task tree imposes a certain order in which the ILU preconditioner can be computed. The MIC starts by processing the leaves of the task tree and proceeds bottom-up towards the root. Initially, the diagonal blocks associated with leaf nodes can be processed by the framework described in subsection 2.2 while other parts are updated only. We refer to this level as the bottom tree level. However, the MIC is forced to complete the processing of all nodes in the current tree level before moving up one level in the tree hierarchy. The top half of Figure 4 illustrates how the MIC proceeds at the bottom tree level. Bad pivots identified by the inverse-based strategy are postponed within the blocks corresponding to this tree level. Once the first local algebraic level is completed in the bottom tree level, the MIC enters the second local algebraic one, and the process is repeated until the set of bad pivots is “small enough”. The computation eventually enters the next tree level and incorporates the remaining bad pivots, rejected in the last local algebraic level of the previous tree level. The bottom half of Figure 4 shows how the input matrix for the next tree level is constructed from the Schur complement resulting from the bottom tree level. We will refer to matrices like those in Figure 4 as the *global matrices* of the parallel MIC. The MIC continues processing the levels of the tree until it reaches the root, where the sequential framework completes the computation.

Our parallel approach exploits that each node of the current tree level is associated with the factorization of certain blocks of the matrix. In Figure 3, for example, fill patterns are used to specify the correspondence between nodes

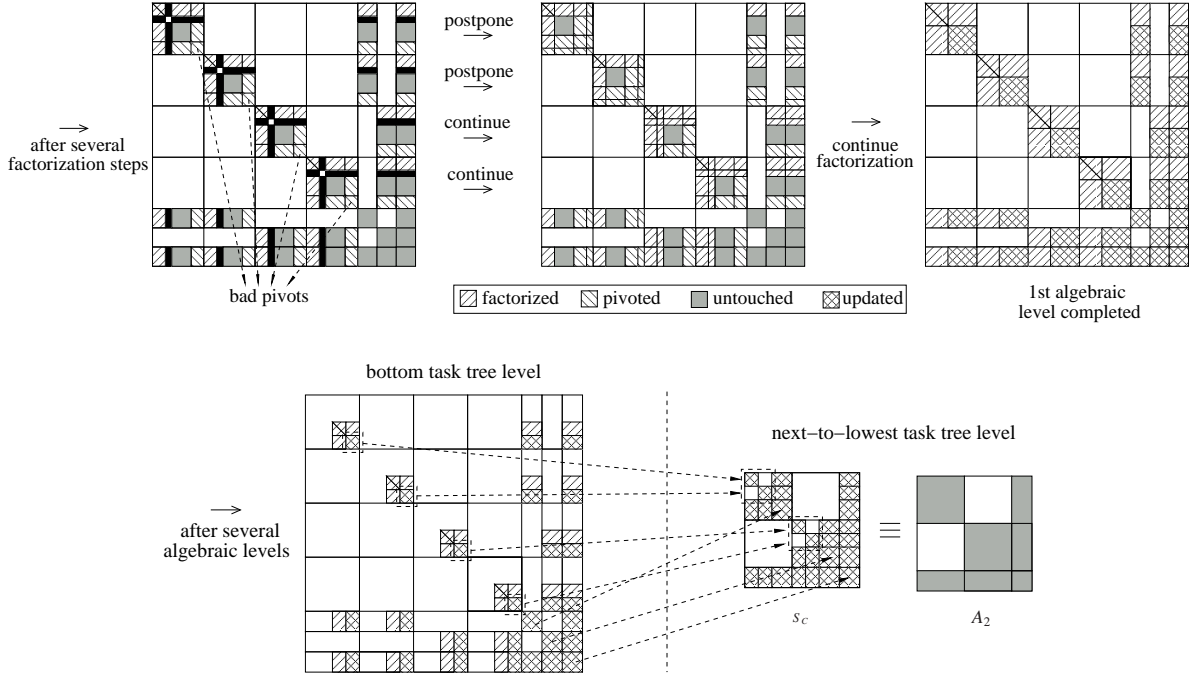


Figure 4: Sketch of the parallel MIC. Top: the MIC within the bottom tree level. Bottom: once all algebraic levels within the bottom tree level are processed, the MIC enters the tree level above it.

of the task dependency tree and blocks of the matrix. Also, the computations within a node only involve updates on blocks which will be factorized in ancestor nodes that are along the path to the root node. Following the example in Figure 3, task (1,1) only updates those blocks that will be later factorized in tasks (2,1) and (3,1). Hereafter, we will refer to blocks that are updated in a task as its *contribution blocks*. To increase the degree of parallelism, the updates from descendant nodes to an ancestor node are kept in separate contribution blocks with different data structures; thus, e.g., updates from tasks (1,1) and (1,2) to (2,1) are kept in separate contribution blocks so that they can be performed locally/independently. The entries of the global matrices can be recovered by adding the corresponding entries of the contribution blocks. Operating in this manner, the factorization of the blocks within the same tree level can proceed in parallel. When these computations are completed, the updates of the contribution blocks can also proceed concurrently.

Let us elaborate this idea further. Although our algorithm can be easily generalized for non-complete binary task trees, we assume for simplicity that we start from a complete binary task tree with s leaves of height $h = \log_2 s + 1$. The input global matrix for the bottom tree level, corresponding to the reordered matrix $A \rightarrow \Pi^T A \Pi = A_1$, is split into the *tree-path additive* sum

$$A_1 = (M^{(1,1)})^T A^{(1,1)} M^{(1,1)} + \dots + (M^{(1,s)})^T A^{(1,s)} M^{(1,s)}, \quad (7)$$

where $A^{(1,j)}$ contains certain blocks of the original matrix and $M^{(1,j)}$ is an appropriate block permutation matrix; see Figure 5. The contribution blocks of the leaves are initialized with the original entries of A_1 divided by 2^{l-1} , where l is the tree level the contribution block belongs to. For example, in Figure 5, the contribution blocks of task (1, 2) corresponding to nodes (2, 1) and (3, 1) are initialized with the original entries of A_1 divided by 2 and 4, respectively.

The parallel computation starts at the bottom tree level with one independent task per leaf. All tasks employ the algorithm in subsection 2.2, but differ in the input data: task (1, j) starts with $A \equiv A^{(1,j)}$. Within one task and a local algebraic level, the following partial factorization is first computed:

$$(\hat{P}^{(1,j)})^T A^{(1,j)} \hat{P}^{(1,j)} \approx L^{(1,j)} D^{(1,j)} (L^{(1,j)})^T,$$

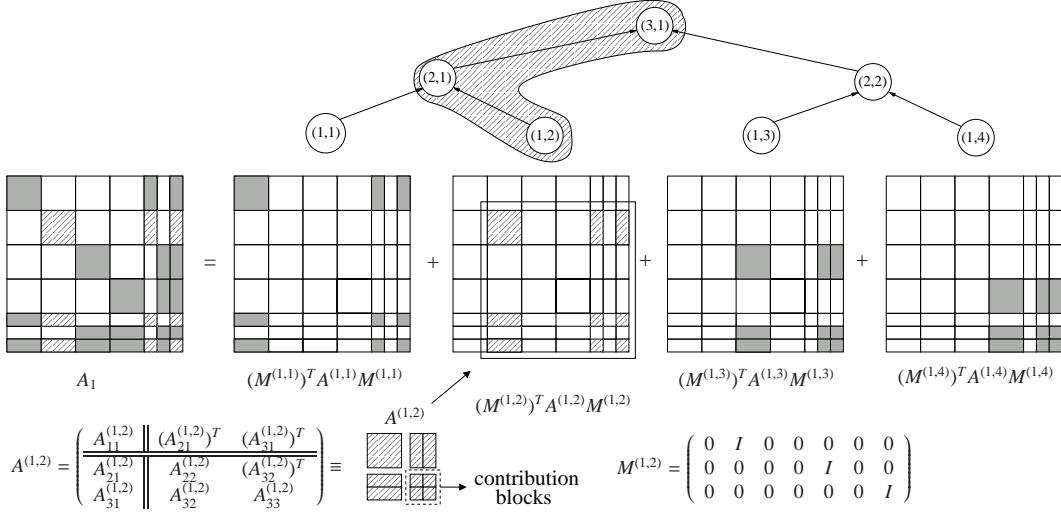


Figure 5: Tree-path additive matrix splitting of A_1 .

where

$$L^{(1,j)} = \begin{pmatrix} L_{B,11}^{(1,j)} & 0 & 0 & \dots & 0 \\ L_{F,11}^{(1,j)} & I & 0 & \dots & 0 \\ \hline L_{21}^{(1,j)} & 0 & I & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{h1}^{(1,j)} & 0 & 0 & \dots & I \end{pmatrix}, \quad D^{(1,j)} = \begin{pmatrix} D_{B,11}^{(1,j)} & 0 & 0 & \dots & 0 \\ 0 & S_{C,11}^{(1,j)} & (S_{C,21}^{(1,j)})^T & \dots & (S_{C,2h}^{(1,j)})^T \\ \hline 0 & S_{C,21}^{(1,j)} & S_{22}^{(1,j)} & \dots & (S_{2h}^{(1,j)})^T \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & S_{C,2h}^{(1,j)} & S_{2h}^{(1,j)} & \dots & S_{hh}^{(1,j)} \end{pmatrix} = \left(\begin{array}{c|c} D_{B,11}^{(1,j)} & 0 \\ \hline 0 & S_C^{(1,j)} \end{array} \right). \quad (8)$$

Here $S_{C,11}^{(1,j)}$ refers to those nodes which were rejected inside a local algebraic level, and the double partitioning lines represent the boundaries between blocks to be factorized by task $(1,j)$ and its contribution blocks. The next local algebraic level of the MIC then proceeds forward with the Schur complement $S_C^{(1,j)}$ as the input. The computation continues processing local algebraic levels until the set of bad pivots is “small enough”. Then, the parallel algorithm proceeds with the computations corresponding to the tasks in the next upper level of the tree. Each task $(2,k)$ of this tree level constructs a new submatrix $A^{(2,k)}$ composed by those parts of the system that were rejected by its children, along with a summation of their contribution blocks. That is,

$$A^{(2,k)} = \begin{pmatrix} A_{22}^{(2,k)} & (A_{32}^{(2,k)})^T & \dots & (A_{h2}^{(2,k)})^T \\ A_{32}^{(2,k)} & A_{33}^{(2,k)} & \dots & (A_{h3}^{(2,k)})^T \\ \vdots & \vdots & \ddots & \vdots \\ A_{h2}^{(2,k)} & A_{h3}^{(2,k)} & \dots & A_{hh}^{(2,k)} \end{pmatrix}, \quad (9)$$

with

$$A_{22}^{(2,k)} = \begin{pmatrix} S_{C,11}^{(1,2k-1)} & 0 & (S_{C,21}^{(1,2k-1)})^T \\ 0 & S_{C,11}^{(1,2k)} & (S_{C,21}^{(1,2k)})^T \\ S_{C,21}^{(1,2k-1)} & S_{C,21}^{(1,2k)} & S_{22}^{(1,2k-1)} + S_{22}^{(1,2k)} \end{pmatrix}, \quad A_{m2}^{(2,k)} = \left(S_{C,m2}^{(1,2k-1)}, \quad S_{C,m2}^{(1,2k)}, \quad S_{m2}^{(1,2k-1)} + S_{m2}^{(1,2k)} \right), \quad m = 3, \dots, h,$$

and $A_{mn}^{(2,k)} = (S_{mn}^{(1,2k-1)} + S_{mn}^{(1,2k)})_{m,n=3,\dots,h}$. Note that this leads to the tree-path additive splitting of the global matrix

$$A_2 = (M^{(2,1)})^T A^{(2,1)} M^{(2,1)} + \dots + (M^{(2,s/2)})^T A^{(2,s/2)} M^{(2,s/2)}$$

corresponding to the second tree level (see A_2 in Figure 4).

In addition, at every algebraic level of the local MIC performed by each task (j, k) , a diagonal scaling $A^{(j,k)} \rightarrow S^{(j,k)}A^{(j,k)}S^{(j,k)}$ and a fill-reducing ordering $A^{(j,k)} \rightarrow (P^{(j,k)})^T A^{(j,k)} P^{(j,k)}$ are applied. Both transformations must be restricted to the blocks of $A^{(j,k)}$ to be factorized by the task.

2.4. Computational aspects: bringing all together

Assume for simplicity that the task tree is binary and complete. The parallel MIC first computes s independent factorizations, corresponding to the leaf nodes; then, $s/2$ submatrices from the next-to-lowest level are constructed from parts of the system rejected by the children along with a summation of the contribution blocks. After this merging stage, matrices at this level are factorized and $s/4$ submatrices for the next level are built. This process is repeated at each level of the tree till the root node is reached.

We note however that the order in which tasks are executed (task schedule) is only constrained by the dependencies captured in the task tree. Therefore, in practice, there is no need for a global synchronization point (a barrier) between tree levels. We will exploit this in a mechanism to schedule tasks for execution at run time, which takes into account only dependencies among tasks. Our parallel algorithm to compute the MIC is sketched in pseudocode in Algorithm 1, and combines the building blocks presented so far with a dynamic task scheduling mechanism. Clearly the algorithm is also valid for incomplete task trees.

Algorithm 1: Computes the parallel MIC of the reordered system $A \rightarrow A_1 = \Pi^T A \Pi$

```

1 [  $\Pi, T$  ]  $\leftarrow$  nested_dissection( $G_A$ )                                 $\triangleright$  obtain task tree  $T$  and permutation  $\Pi$ 
2  $Q \leftarrow$  { leaves( $T$ ) }                                            $\triangleright$  initialize  $Q$  with all leaves of  $T$ 
3 mark all tasks of  $T$  as not executed
4 Begin parallel region
5   pid  $\leftarrow$  get_process_identifier()
6   repeat
7     while pending tasks in  $Q$  do
8       tid  $\leftarrow$  dequeue( $Q$ )                                        $\triangleright$  remove ready task from the head of  $Q$ 
9       map [tid]  $\leftarrow$  pid                                            $\triangleright$  process pid in charge of task tid
10      execute(tid)                                                     $\triangleright$  construct tid's submatrix and compute local MIC
11      mark tid as executed
12      if all dependencies of parent(tid) have been resolved then
13        enqueue(parent(tid),  $Q$ )                                        $\triangleright$  insert new ready task at the tail of  $Q$ 
14      end
15    end
16  until not all tasks executed
17 End parallel region

```

Algorithm 1 maintains a centralized (shared) queue Q containing only tasks with their dependencies fulfilled (ready tasks). The shared queue is initialized with all the leaves of the task tree, and then a group of “processes” (actually threads) is spawned (line 4). Processes enter the **while** loop when there are tasks ready for execution. The mapping of tasks to processes (line 8) is completely dynamic, which aims at improving load-balancing. The computational core of the algorithm is represented by the call to routine “execute” (line 10), which constructs the submatrix associated with a task and performs the corresponding computations. There are no synchronization points inside the routine; this is a consequence of splitting both the data structures and the computations of the MIC. When a process completes the execution of a task, it checks whether its sibling task has been already computed (this implies that all dependencies of its parent task have been resolved as the tree is binary) and, in such case, the process inserts the parent task in Q (line 13). For simplicity, details on the safe concurrent access to centralized data structures are omitted from the algorithm.

Algorithm 1 only exploits tree parallelism at levels below the root. It also checks the degree of sparsity of the Schur complement at the root task, and, “in case it is dense enough”, employs the LAPACK (dense) factorization

routine with parallelism extracted from a multi-threaded implementation of the level 3 BLAS. In parallel sparse direct methods, tree parallelism is combined with additional types of parallelism (e.g., pipelining parallelism [16] or node parallelism [15]; see also the references therein) because, frequently, a large bulk of the computation occurs at the higher levels of the tree, where the degree of tree parallelism is already more limited. For example, results in [15] report that often more than 75% of the computations are performed in the top three levels of the assembly tree (a structure similar to the task tree) in a multifrontal sparse direct method. In the parallel MIC this situation hardly occurs, mainly because of dropping. As our experiments will demonstrate, tree parallelism provides enough concurrency for a moderate number of processors in the case of MIC.

We next explain how the task tree T is obtained. Starting from a tree consisting of only one node (the root), T is constructed (line 1) from top to bottom, by splitting those leaf tasks which present a “high” estimated computational cost into two leaves. (A binary tree is thus ensured to be obtained.) The heuristic cost $h^{(1,k)}$ estimated for a given task $(1, k)$ is defined as the number of edges of the corresponding subgraph of G_A . A leaf task is split into two leaves if $h^{(1,k)} > \frac{|E|}{f}$, with f a parameter of our heuristic approach and $|E|$ the number of edges of G_A . We choose f so that, in general, there will be more leaf tasks than processors, improving the probability of attaining a good overall load balance of the computation; we found experimentally that $f \in [p, 2p]$, with p the number of processors, are appropriate choices for most examples [17, 18]. Algorithm 1 assigns higher priority to leaf tasks over tasks with descendants (since Q is initialized with all the leaves of the task tree and the tasks with descendants are inserted at the tail of Q). The order in which Q is initialized (line 2) also determines the execution schedule for the leaf tasks, and we initialize Q with the leaves in descending order of their estimated costs. The purpose is to prioritize execution of the leaf tasks with higher computational cost so as to reduce load unbalance due to their late schedule.

We consider two alternative approaches for the preprocessing step in line 1 of Algorithm 1. Both versions use node-based multilevel nested dissection orderings (MLND) provided by SCOTCH [19]. They differ in the reordering strategy applied to the independent subgraphs corresponding to the leaves of the task tree. The first strategy, ND-HAMD-A, executes the separator-finding mechanism repeatedly on these independent subgraphs until their size is “small enough” (e.g., $|E| = 100$). Then it switches to minimum degree to process small subgraphs. After the preprocessing is completed, the local MIC uses local reordering strategies which preserve the structure of the initial partitioning into tasks. Specifically, only scaling is applied in the initial local algebraic level, since this block has already been reordered by the preprocessing step. However, for any subsequent local algebraic level, Halo-AMD [20] is used. In the second strategy, ND-HAMD-B, independent subgraphs fulfilling $\frac{h^{(1,k)}}{|E|} \leq \frac{1}{f}$, are not further ordered via nested dissection and preprocessing is stopped. This reduces the cost of the preprocessing represented in line 1 significantly, and leaves the reordering algorithms to the local MIC. In our case the local MIC of each task is configured such that Halo-AMD reorderings are applied at each local algebraic level. The advantage of the ND-HAMD-B comes from switching earlier to less expensive fill-reducing heuristics. This can be done concurrently exploiting the implicit parallelism obtained from the preprocessing step.

2.5. Numerical comparison between the sequential and the parallel preconditioner

For the parallel MIC the algebraic levels have to reveal the structure of the task tree. This might lead to the impression that the parallelization approach severely changes the semantics of the original sequential preconditioner. Experimental results will show that in practice this is hardly the case. In this section we will explain why the parallel MIC is expected to be numerically close to its sequential counterpart.

MLND [21] has been considered to be well suited for parallel computation and helpful as a fill-reducing ordering as well. We next examine how the sequential MIC behaves when applied to a system reordered by MLND. At this point, it is important to review the role of the inverse-based approach in the computation. According to [3, 22], at each step i of the incomplete Cholesky decomposition $A \approx LDL^T$, the i -th row of L^{-1} is required to satisfy $\|e_i^T L^{-1}\| < \nu$. In general, we have that

$$\|e_i^T L^{-1}\|_\infty = \max_{\|z\|_\infty=1} |e_i^T L^{-1} z|,$$

and an inexpensive estimate for this norm consists in choosing a specific vector \hat{z} , with entries from $\{\pm 1\}$, such that $v_i = |e_i^T L^{-1} \hat{z}|$ is close to its upper bound $\max_{\|z\|_\infty=1} |e_i^T L^{-1} z|$; see [22] or [23] for details. The estimate $v_i = |y_i|$ is obtained while solving the system $Ly = \hat{z}$. Therefore, at step i of the incomplete Cholesky factorization y is updated by $y_j := y_j - l_{ji} y_i$ for $j > i$ such that $l_{ji} \neq 0$. Since in general \hat{z} is constructed to obtain a large $|y_j|$, it is likely that those components of $|y|$ which are involved in many forward substitution steps become large. By construction of the

MIC level	total size	task level		
		1	2	3
1	16129	15879	123	127
2	4951	4733	107	111
3	1865	1665	99	101
4	883	689	94	100
5	408	239	79	90
6	185	40	66	79
7	56	0	22	34

Table 1: Distribution of each algebraic level with respect to the task levels.

MLND, the separators are expected to lead to more fill-in than the leaf nodes. That is, when the sequential code is applied to the reordered system, it is likely that vertices attached to the separators are initially rejected more often. This is confirmed by our numerical experiments.

Example 2. *To illustrate how the sequential MIC postpones bad pivots we pursue Example 1 for the Laplacian problem. Numerical experiments show that, at the first algebraic level of the MIC, only 30% of the nodes associated with leaf tasks but 87% of the nodes associated with separators are rejected (see Figure 1, left).*

When the MIC enters the second algebraic level, 35% of leaf nodes are rejected while 92% of separator nodes are rejected again. This trend continues until the bulk of leaf nodes has been eliminated. Only after that, the MIC begins to accept most of the separator nodes. Table 1 illustrates the distribution of each algebraic level. One can see in particular that the bulk of the algebraic levels is concentrated in task level 1 (leaf tasks) until algebraic level 5.

The results from Example 2 are not uncommon. We observed for several sample matrices that most of the separator nodes are postponed over several levels of the MIC until the parts associated with leaf nodes are mostly factored. In this sense, the sequential MIC wastes computation time rejecting separator nodes which, by construction, are postponed in the parallel MIC. This observation reinforces our proposition that the parallel MIC exhibits fundamentally the semantics of the sequential algorithm.

3. Parallel PCG

3.1. Application of the MIC

The application of the preconditioner $M = LDL^T$ to the original system involves solving the linear system $Mz = r$ at each step of the PCG, where $r \in \mathbb{R}^n$ and $z \in \mathbb{R}^n$ are, respectively, the residual and preconditioned residual iterates of the PCG. In the MIC, this operation is more challenging, because in addition to partial factorizations, one has to deal with other matrix operations such as scalings or permutations. For simplicity, the following discussion only considers permutation matrices as those generated by ILUPACK pivoting strategy.

The application of the MIC can be described as solving the following linear system recursively:

$$Mz = r \equiv \hat{P}^T \begin{pmatrix} L_B & 0 \\ L_F & I \end{pmatrix} \begin{pmatrix} D_B & 0 \\ 0 & \tilde{S}_C \end{pmatrix} \begin{pmatrix} L_B^T & L_F^T \\ 0 & I \end{pmatrix} \hat{P}z = r, \quad (10)$$

where \tilde{S}_C is the approximation that is obtained when S_C is replaced by the recursive application of the multilevel factorization approach. Therefore, at each algebraic level, the computation is composed of the following three steps:

1. Set $\begin{pmatrix} \hat{r}_B \\ \hat{r}_C \end{pmatrix} := \hat{P}r$; solve $\begin{pmatrix} L_B & 0 \\ L_F & I \end{pmatrix} \begin{pmatrix} y_B \\ y_C \end{pmatrix} = \begin{pmatrix} \hat{r}_B \\ \hat{r}_C \end{pmatrix}$ for $\begin{pmatrix} y_B \\ y_C \end{pmatrix} = y$.
2. Solve $\tilde{S}_C \hat{z}_C = y_C$ by recursively applying (10) with $z \equiv \hat{z}_C$ and $r \equiv y_C$.
3. Solve $\begin{pmatrix} L_B^T & L_F^T \\ 0 & I \end{pmatrix} \begin{pmatrix} \hat{z}_B \\ \hat{z}_C \end{pmatrix} = \begin{pmatrix} \bar{y}_B \\ \bar{y}_C \end{pmatrix} = \begin{pmatrix} D_B^{-1} y_B \\ \hat{z}_C \end{pmatrix}$; and finally set $z := \hat{P}^T \begin{pmatrix} \hat{z}_B \\ \hat{z}_C \end{pmatrix}$.

The recursion in step 2 is finalized when invoked with input $\tilde{S}_C = L_C D_C L_C^T$, the approximation of S_C obtained as a result of the last algebraic level. In this case the system is solved directly. Hereafter, we will refer to steps 1 and 3 as *forward substitution* (FS) and *backward substitution* (BS), respectively. Actually, in the FS step only the leading block of y is computed from forward substitution, applied to $L_B y_B = \hat{r}_B$, while y_C is obtained as $y_C := \hat{r}_C - L_F y_B$. In the BS step, only the leading block of \hat{z} is computed by backward substitution in $L_B^T \hat{z}_B = D_B^{-1} y_B - L_F^T \hat{z}_C$.

3.2. Application of the parallel MIC

The application of the parallel MIC essentially operates in the same manner as its sequential counterpart, but now the FS and BS steps are spread over the tree levels, and therefore entering/leaving the recursive step 2 may imply moving up/down in the tree hierarchy. We next consider the algorithm in terms of the global matrices and vectors because it is easier to describe, although its implementation actually splits the overall vectors conformally with the computation of the parallel MIC (see [24] for details).

The parallel application of the MIC begins at the bottom tree level, proceeding bottom-up towards the root node. The recursion described in subsection 3.1 is applied at each local algebraic level within the current tree level, until all local contributions of this tree level for y_B and y_C have been computed. Then step 2 employs an approximation \tilde{S}_C that has been computed for the upper tree levels only (see Figure 6). Therefore, entering the recursive call at this point implies moving the parallel application of the MIC up by one tree level, where the input right-hand side vector for this level, r , incorporates the contributions for y_C computed by the previous tree levels. The algorithm continues processing FS steps bottom-up, until it reaches the last algebraic level within the root task. Then, the systems $L_C y = r$ and $L_C^T z = D_C^{-1} y$ are solved and computation proceeds next top-down towards the bottom level of the tree. The algorithm backtracks to each algebraic level within the current tree level until the lower tree level is reached; see Figure 7. The figure shows that, at the last BS step of the current tree level, the application of the inverse permutation \hat{P}^T to \hat{z} recovers the partitioning in z corresponding to the lower tree level. Computation then backtracks to the lower tree level, where \hat{z} incorporates the contributions for \hat{z}_c resulting from the local BS steps of the upper tree levels.

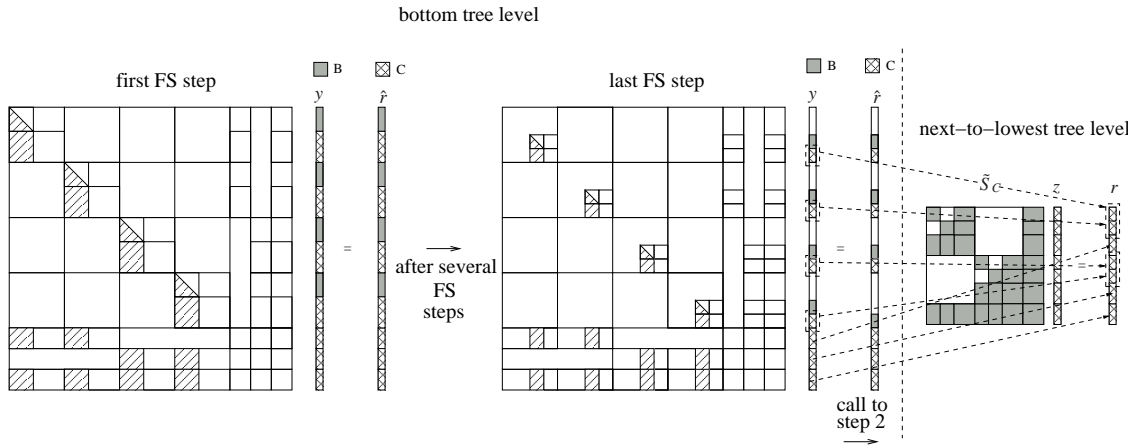


Figure 6: The application of the parallel MIC first proceeds bottom-up towards the root task. From left to right: FS steps within the bottom tree level and how the computation enters the recursion from the bottom to the next-to-lowest tree level.

3.3. Task scheduling

The operations involved in the application of the MIC (sparse matrix-vector products and sparse triangular solves) are memory-bounded computations exhibiting hardly any data reuse. Therefore, memory related issues should be considered carefully in the design of the mapping and scheduling mechanisms for the parallel algorithm. Our first choice is the use of a static mapping of leaf tasks to the processes during the execution of the PCG; i.e., a given process always executes the same leaf tasks for all the applications of the MIC. Provided there is no process migration during execution, this improves cache hit rate and reduces interprocessor communication, because each process repeatedly

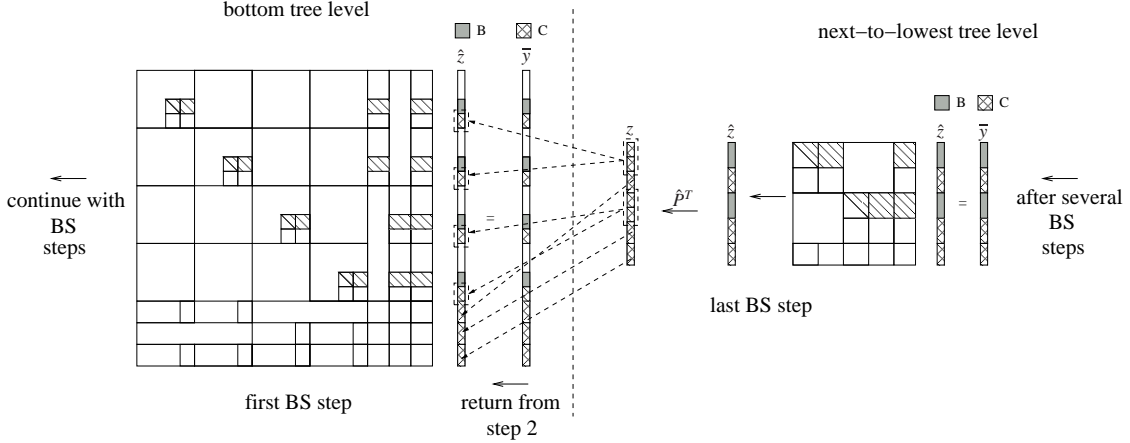


Figure 7: After reaching the root task, the application of the parallel MIC proceeds top-down towards the bottom tree level. From right to left: last BS step of the next-to-lowest tree level, backtracking from the next-to-lowest to the bottom one, and first BS step of the bottom tree level.

accesses the same data structures (executes the same tasks). Although, in principle, this work distribution can be explicitly computed via static mapping heuristics, in our approach we use instead the mapping resulting from the computation of the MIC (recorded during execution of line 9 of Algorithm 1). The same applies to tasks with descendants; however, during the backtracking (top-down) of the algorithm, under certain circumstances which may result in idle processes, at execution time we adopt decisions which may alter the pre-assignment of the tasks to processes.

The recursion phase (bottom-up) essentially follows the sketch of Algorithm 1, but now each process maintains its own queue of ready tasks, Q_{pid} . This implies replacing Q by Q_{pid} everywhere in Algorithm 1, except in line 13, where Q is replaced by $Q_{map[tid]}$. Besides, each process initializes its task queue in parallel with initializations by other processes/queues, with the leaves pre-assigned to it ($Q_{pid} = \{ tid \in leaves(T) : map[tid]=pid \}$) ordered by their estimated costs. The number of floating-point arithmetic operations performed by the task is now used to estimate the costs. The backtracking (top-down) of the algorithm is sketched in Algorithm 2. Each process maintains its own *priority* queue of ready tasks H_{pid} , which prioritizes the execution of tasks with descendants over leaf tasks. A given task will not become ready until all its ancestors have been executed, and therefore only tasks with descendants and leaf tasks which are independent of each other can be simultaneously in H_{pid} . If leaves were given priority over tasks with descendants, the execution of the tasks belonging to the subtrees rooted at the tasks with descendants would be delayed. This delay may result in idle processes, waiting for the tasks which belong to these subtrees to become ready. The execution of Algorithm 2 is asynchronous (there are not global barriers between tree levels), and it is possible that a task with descendants becomes ready while the process this task has been mapped to is already executing a leaf task. Whenever this occurs, the execution of this task is assigned to the process which resolves the dependencies of the task with descendants (line 10). Otherwise, this task is inserted in $H_{map[chil,tid]}$ (line 12) according to the prescribed mapping of tasks with descendants to processes.

We have observed experimentally that, when the number of processes is moderate (up to 16), the inexpensive heuristics described above provide fairly acceptable solutions for the mapping and scheduling of the parallel application of the MIC (see, e.g., [24]). We have no experimental results for a larger number of processes.

3.4. Other operations

The iteration in the PCG method consists of a repeated sequence of operations: application of the preconditioner, matrix-vector products, inner products, and $axpy$ updates. In our parallelization of the PCG method, we split the last three computations and the associated data structures conformally with the application of the preconditioner. In particular, we consider two types of splittings, structurally equivalent to (7), for the vectors involved in the operations of the PCG.

For the first type of splitting, the right-hand side vector b as well as the residual vector r are stored as additive

Algorithm 2: Computes the backtracking of the parallel application of the MIC

```

1 mark all tasks of  $T$  as not backtracked
2 pid  $\leftarrow$  get_process_identifier()
3 repeat
4   while pending tasks in  $H_{\text{pid}}$  do
5     tid  $\leftarrow$  extract( $H_{\text{pid}}$ ) ▷ extract highest priority task from  $H_{\text{pid}}$ 
6     backtrack(tid) ▷ construct tid's subvectors and compute local BS steps
7     mark tid as backtracked
8     for child_tid  $\in$  children(tid) do
9       if child_tid has descendants and process map [ child_tid ] currently executing a leaf task then
10        | insert(child_tid,  $H_{\text{pid}}$ ) ▷ proc. pid now in charge of child_tid
11        else
12        | insert(child_tid,  $H_{\text{map}}[\text{child\_tid}]$ ) ▷ proc. map [child_tid] in charge of child_tid
13        end
14      end
15    end
16 until not all tasks backtracked

```

sums,

$$r = (M^{(1,1)})^T r^{(1,1)} + \dots + (M^{(1,s)})^T r^{(1,s)}, \quad r^{(1,j)} = (r_1^{(1,j)} \parallel r_2^{(1,j)} \dots r_h^{(1,j)})^T,$$

where the entries of the contribution blocks $r_2^{(1,j)}, \dots, r_h^{(1,j)}$ store partial contributions to the original entries of r as in (7). The second type of splitting is based on *redundant* copies. Here, the approximate solution x , the preconditioned residual z , and the search direction p are split as

$$M^{(1,j)} p = p^{(1,j)}, \quad p^{(1,j)} = (p_1^{(1,j)} \parallel p_2^{(1,j)} \dots p_h^{(1,j)})^T, \quad j = 1, \dots, s,$$

where the entries of the contribution blocks $p_2^{(1,j)}, \dots, p_h^{(1,j)}$ store redundant copies of the original entries of p . It can be shown [25] that, using this combination for the data splittings of the iteration vectors r, x, z, p , all the parallel operations within the body of the PCG can proceed without explicit conversions between the two types of splittings.

4. Computational results

In this section we evaluate the performance of our parallel solver applied to a pair of challenging 3D application problems and a subset of irregularly structured matrices from the UF sparse matrix collection. The section also includes a comparison of the performances of our solution, PARDISO, BlockSolve95, and Hypre-Euclid. PARDISO [26, 27] is a state-of-the-art parallel supernodal direct solver. BlockSolve95 is a parallel library for the scalable iterative solution of symmetrically structured linear systems based on ILU(0)-preconditioned Krylov solvers. Hypre is a library for the parallel preconditioning and iterative solution of sparse linear systems. Euclid provides parallel incomplete level of fill factor preconditioning, i.e., the computation and application of parallel ILU(k) preconditioners. In contrast to our parallel solver and PARDISO, BlockSolve95 and Hypre-Euclid target distributed-memory multiprocessors, relying on MPI for message passing.

We describe first the environment setup *common to all the experiments* in this section. In the MLND [21] preprocessing step and the Halo-AMD fill-reducing orderings [20], we set the SCOTCH [19] parameters to default values. In general, the optimal parameter choice is highly problem dependent, but an exploration of the parameter space is out of the scope of this paper. The same applies to ν and τ , which control, respectively, the norm of the inverse and drop tolerances in the incomplete factorization process (see step 2 at subsection 2.2). These two parameters determine the fill-in, and hence the computational cost and storage required for the MIC preconditioner. Larger values of ν or τ lead to cheaper but (maybe) ineffective preconditioners, whereas smaller values can lead to prohibitively high fill-in; we refrain from optimizing ν and τ , and set their values to $\nu = 5$ and $\tau = 10^{-2}$. In the PCG method, we set the initial

solution vector guess $\tilde{x}_0 \equiv 0$, and the iteration is stopped when the criterion described in [28] is satisfied. All experiments employ IEEE double-precision arithmetic for the numerical calculations. The target platform is a SGI Altix 350 CC-NUMA shared-memory multiprocessor consisting of 8 nodes with 32 GBytes of RAM connected via a SGI NUMALink network. Each node is composed of two Intel Itanium2@1.5 GHz processors (256 KBytes level-2 and 6 MBytes level-3 cache) with 4 GBytes of local memory. All codes were compiled by the C and F77 Intel compilers, version 9.0. We used OpenMP rev. 2.5 as provided by these compilers and Intel MKL library, version 10.0, for the BLAS-3 and LAPACK routines required by PARDISO and our parallel codes. The BlockSolve95 and Hypre-Euclid codes were linked with an optimized MPI library included in the SGI Message Passing Toolkit, version 1.12. The parallel environment is configured so that one thread is binded per processor and no thread migration occurs during the computation. The same applies to the MPI processes during the parallel execution of the codes in BlockSolve95 and Hypre-Euclid.

Example 3. We consider a standard benchmark problem for the solution of PDEs: the Laplacian equation

$$-\Delta u = f$$

in a 3D unit cube $\Omega = [0, 1]^3$ with Dirichlet boundary conditions $u = g$ on $\partial\Omega$. Although this regular problem is known to be best-suited for multigrid methods, we have selected it because of its large scale and applicability. For the discretization we use a uniform mesh of size $h = \frac{1}{N+1}$. The computational domain Ω is replaced by a grid $\Omega_h = \{(x_i, y_j, z_k) = (ih, jh, kh) \mid i, j, k = 1, \dots, N\}$ and the differential operator is replaced by finite differences

$$-\Delta u(x_i, y_j, z_k) \approx \frac{1}{h^2} \left(-u_{i-1,j,k} - u_{i,j-1,k} - u_{i,j,k-1} + 6u_{ijk} - u_{i+1,j,k} - u_{i,j+1,k} - u_{i,j,k+1} \right),$$

where $u_{ijk} \approx u(x_i, y_j, z_k)$. Because of Dirichlet boundary conditions, any unknown u_{ijk} such that $i, j, k \in \{0, N+1\}$ is explicitly available and becomes part of the right-hand side vector. The resulting linear system $Au = b$ has a sparse SPD coefficient matrix with seven nonzero elements per row, and $n = N^3$ unknowns. We choose $N = 100; 125; 150;$ and $N = 200$ in our experiments, which results in four benchmark SPD linear systems of order $n = 1,000,000; 1,953,125; 3,375,000;$ and $8,000,000$ unknowns.

Table 2 compares the performance of the sequential algorithm in ILUPACK (results for $p = 1$) with the performance of our parallel solver using $p = 8$ and $p = 16$ processors applied to Example 3. Two preprocessing alternatives are considered (see subsection 2.4): ND-HAMD-A and ND-HAMD-B, and separate results are provided for the MLND, MIC, and PCG stages. The execution of the MLND is completely serial (no parallelization has yet been attempted for this stage in our approach). Therefore, for the MLND stage, p is not really the number of processors employed in the execution; it is solely used to determine the degree of parallelism which needs to be identified in order to enhance the parallel performance of the MIC and PCG stages. For ND-HAMD-A, the execution time is independent of p : the graph is split to the same depth for $p=1, 8,$ and 16 (hereby the equal execution time of this stage for all three values of p), and this level is much deeper than it would be actually necessary for the efficient exploitation of parallelism. (Fill-reducing is the guiding principle for such a deep split.) On the other hand, the execution time of this stage using ND-HAMD-B increases as p gets larger since, in this case, the splitting is stopped as soon as enough parallelism has been detected. A larger value of p translates to additional levels of recursion in the nested dissection of G_A , and therefore to larger execution times. For the sequential ILUPACK ($p = 1$), the global cost is balanced among all three stages (see ND-HAMD-A), but for the parallel MIC and PCG stages, the preprocessing step tends to concentrate a larger bulk of the computational load as p increases. We expect that a parallelization of this stage will lead to a significant reduction of the cost of MLND, and this is currently being considered as an extension of our work.

The table also reports the total number of nonzero elements in the triangular factor(s) (in millions) of the MIC preconditioner, as well as the execution time and the parallel speed-up. As the number of processors is increased, the number of nonzero elements becomes larger but this is clearly compensated by the reduction of the execution times. Speed-ups of up to 8.62 and 15.5 are obtained using 8 and 16 processors, respectively. The superlinear speed-up is due to the NUMA memory layout of the target architecture. When only one processor is involved, the amount of memory employed by the sequential algorithm in ILUPACK is larger than the local memory per node. Accesses to nonlocal storage slow down the performance of the sequential algorithm. When 8 or 16 processors are employed, the size of the local data structures is considerably smaller and they fit into the local memories.

Finally the results in the table for the PCG stage show the number of iterations required for convergence, the execution time, and the parallel speed-up. The iteration count only increases slightly as p becomes larger, in a practical demonstration of the close semantics between the sequential and parallel preconditioners. This increase is more than paid off by the reduction of the execution attained by the parallel PCG. Superlinear speed-ups are also observed in this stage (up to 10.9 and 19.2 using, respectively, 8 and 16 processors), the reason being the same as that exposed for the MIC stage.

Preprocessing with ND-HAMD-B is superior to ND-HAMD-A. Although the execution time of the MIC stage is lower for the second option, the former is both less computationally expensive and produces higher quality preconditioners (less number of iterations) so that the global cost is more reduced.

n	p	MLND		MIC			PCG		
		Option ND-HAMD-X	T (sec.)	mnz_L $\times 10^{-6}$	T_p (sec.)	S_p (T_1/T_p)	#Iter.	T_p (sec.)	S_p (T_1/T_p)
100^3	1	A	19.8	15.0	30.0	1.0	64	61.3	1.0
	8	A	19.8	15.2	4.0	7.5	67	7.4	8.3
	16	A	19.8	15.3	2.5	12.3	67	4.3	14.1
	1	B	0.0	13.6	36.0	1.0	57	49.9	1.0
	8	B	5.7	14.0	4.9	7.4	63	6.6	7.6
	16	B	7.1	14.2	2.9	12.5	66	4.0	12.2
125^3	1	A	42.8	29.5	60.9	1.0	78	146.0	1.0
	8	A	42.8	29.8	8.2	7.5	81	19.4	7.5
	16	A	42.8	30.0	4.8	12.7	82	10.7	13.7
	1	B	0.0	26.8	73.7	1.0	68	116.8	1.0
	8	B	12.0	27.5	10.0	7.4	72	15.8	7.4
	16	B	15.3	28.0	5.6	13.2	78	9.4	12.4
150^3	1	A	82.2	51.1	108.0	1.0	90	296.8	1.0
	8	A	82.2	51.8	14.3	7.5	91	38.5	7.7
	16	A	82.2	52.0	8.1	13.3	93	21.5	13.8
	1	B	0.0	46.6	138.9	1.0	79	240.1	1.0
	8	B	24.7	47.8	17.8	7.8	83	32.8	7.3
	16	B	30.6	48.2	10.0	14.0	87	18.9	12.7
200^3	1	A	247.0	121.9	303.3	1.0	117	1,332.0	1.0
	8	A	247.0	123.2	35.2	8.6	118	122.2	10.9
	16	A	247.0	123.9	19.6	15.5	117	69.4	19.2
	1	B	0.0	111.1	366.0	1.0	103	1,010.2	1.0
	8	B	89.7	113.4	45.9	8.0	107	104.2	9.7
	16	B	108.0	114.6	24.9	14.7	107	57.7	17.5

Table 2: Performance results for Example 3.

Table 3 reports the performance of PARDISO and the ILUPACK-based solvers (sequential ILUPACK for $p=1$ and our parallel solver for $p=16$). The results for PARDISO are decomposed into the different stages in this solver (initial *Reordering*, *Symbolic analysis*, numerical *Factorization*, and triangular *Solve*). The column labeled as “ mnz_L ” in the results for PARDISO reflects the number of nonzero entries (in millions) in the Cholesky factor. The number of nonzeros as well as the execution time for PARDISO are significantly higher than those of the ILUPACK-based solvers. This confirms that sparse direct solvers are hardly competitive for 3D application problems. For $n > 125^3$ we expect that PARDISO will dramatically slow down because of considerable fill-in. On the other hand, the MIC preconditioner exhibits a remarkable scalability as the number of nonzero entries and execution time increase almost linearly with n .

Example 4. *The second example addresses an irregular 3D problem*

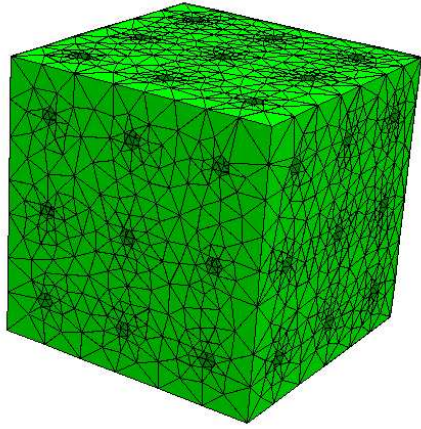
$$-\operatorname{div}(A \operatorname{grad} u) = f,$$

in a 3D domain (see the left part of Figure 8), where $A(x, y, z)$ is chosen with positive random coefficients. For the discretization, linear finite elements are used. The size and number of nonzero elements of the resulting sparse SPD linear systems depend on the initial mesh refinement level, and number of further mesh refinements. Based on the

n	p	PARDISO						ILUPACK-based	
		nnz_L $\times 10^{-6}$	T_{Re} (sec.)	T_{Sy} (sec.)	T_{Fa} (sec.)	T_{So} (sec.)	T_{All} (sec.)	nnz_L $\times 10^{-6}$	T_{All} (sec.)
100^3	1	786.1	16.9	5.3	1,714.7	20.5	1,757.4	13.6	85.9
	16	785.2	17.1	6.3	183.5	12.9	219.8	14.2	14.0
125^3	1	2,031.2	36.8	13.5	8,657.2	57.3	8,764.8	26.8	190.5
	16	2,029.5	37.4	15.5	802.8	31.1	886.8	28.0	30.3

Table 3: Performance comparison between PARDISO and our parallel solver for Example 3.

initial mesh as shown in the left part of Figure 8, the mesh refinement tool NETGEN⁶ refines the mesh up to two times based on the meshing levels (very coarse, coarse, moderate, fine, very fine) as provided by the software. The right part of Figure 8 presents the benchmark identifier, the benchmark code, the initial mesh refinement level, the number of further refinements, the number of unknowns, the number of nonzero elements in A , and the average number of nonzero elements in each row for the 12 benchmark linear systems we have selected for our experiments.



Id.	Code	Initial Mesh	# refs.	n	nnz_A	nnz_A/n
1	VC	very coarse	0	1,709	16,669	9.75
2	C	coarse	0	9,583	112,563	11.75
3	M	moderate	0	32,429	412,251	12.71
4	F	fine	0	101,296	1,368,594	13.51
5	VC2	very coarse	2	271,272	3,686,268	13.59
6	M1	moderate	1	297,927	4,134,255	13.88
7	VF	very fine	0	658,609	9,294,721	14.11
8	F1	fine	1	882,824	12,562,880	14.23
9	C2	coarse	2	906,882	12,854,824	14.17
10	VC3	very coarse	3	2,382,864	34,128,924	14.32
11	M2	moderate	2	2,539,954	36,768,808	14.48
12	VF1	very fine	1	5,413,520	78,935,174	14.58

Figure 8: Computational domain (of Example 4) in 3D with some holes inside (left), and benchmark matrices resulting from several discretizations of the computational domain (right).

Table 4 compares the performance of the sequential algorithm in ILUPACK with that of our parallel solver applied to a pair of relatively small cases and a pair of large ones in Example 4 (case F, M1, VC3, and VF1 in Figure 8). The results in Table 4 are similar to those presented in Table 2.

The small variation in the number of nonzero elements and the low increase in the number of iterations with larger p confirms that (for Example 4) our approach to parallelization preserves the mathematical basis of the sequential approach.

The speed-ups attained for the two small cases (F and M1) on $p=16$ processors correspond to parallel efficiencies of up to 70% and 64% for the MIC and PCG stages, respectively. However, for both large problems (VC3 and VF1), the parallel codes yield higher efficiencies (up to 100% and 108%, respectively). The cause for the superlinear speed-ups (for matrix VF1) are the same as mentioned earlier for Example 3 and $n = 200^3$.

Preprocessing with ND-HAMD-B is superior than ND-HAMD-A in terms of global execution time. However, the execution time of the MIC stage is higher for the former, and that of the PCG step is similar for both alternatives (as both the number of nonzero elements and number of iterations are very close; see, e.g., results for VC3).

Table 5 shows the interaction between the number of levels/leaves in the task tree and the execution times of the MIC and PCG stages determined by the dynamic scheduling policy (using ND-HAMD-B). The height of the tree

⁶<http://www.hpfem.jku.at/netgen>.

n	p	MLND		MIC			PCG		
		Option ND-HAMD-X	T (sec.)	nnz_L $\times 10^{-6}$	T_p (sec.)	S_p (T_1/T_p)	#Iter.	T_p (sec.)	S_p (T_1/T_p)
F	1	A	2.3	1.6	2.3	1.0	22	1.8	1.0
	8	A	2.3	1.6	0.4	6.0	24	0.3	6.4
	16	A	2.3	1.6	0.3	8.7	24	0.2	8.9
	1	B	0.0	1.5	2.6	1.0	23	1.8	1.0
	8	B	0.7	1.5	0.4	6.2	23	0.3	6.9
	16	B	0.9	1.5	0.3	9.4	23	0.2	9.3
M1	1	A	8.0	5.0	8.0	1.0	29	8.3	1.0
	8	A	8.0	5.1	1.3	6.2	29	1.2	7.2
	16	A	8.0	5.1	0.8	10.1	30	0.8	10.3
	1	B	0.0	4.7	9.8	1.0	28	7.7	1.0
	8	B	2.3	4.8	1.4	7.2	28	1.2	6.5
	16	B	2.7	4.8	0.9	11.1	29	0.8	9.9
VC3	1	A	82.1	44.9	83.9	1.0	52	136.8	1.0
	8	A	82.1	44.9	11.2	7.5	52	17.1	8.0
	16	A	82.1	44.9	6.5	13.0	53	10.0	13.7
	1	B	0.0	45.2	108.8	1.0	50	132.6	1.0
	8	B	21.5	45.0	13.3	8.2	52	17.1	7.7
	16	B	25.1	44.9	7.5	14.6	52	9.8	13.5
VF1	1	A	198.9	105.3	234.8	1.0	64	502.8	1.0
	8	A	198.9	104.2	27.0	8.7	63	51.7	9.7
	16	A	198.9	104.0	15.2	15.4	64	29.2	17.2
	1	B	0.0	100.7	279.7	1.0	60	362.3	1.0
	8	B	50.6	98.7	32.0	8.7	62	49.3	7.4
	16	B	60.9	99.1	17.4	16.1	62	27.7	13.3

Table 4: Performance results for Example 4.

and the number of leaves is fixed during the MLND stage as a function of f . We explore the results for a complete binary tree ($f = c$) and different values of $f = p$ times the value shown in the column labeled as f . As expected, a larger value of f leads to more leaves and/or higher task trees and therefore to a higher degree of parallelism. In general, this results in a lower execution time of the parallel solver. The column labeled as “ vc ” refers to the variation coefficient defined as the ratio between the standard deviation and the arithmetic mean of the sum of the computational costs of the tasks assigned to each thread. Thus, a lower value for vc indicates a more homogeneous distribution of the computational load. The results clearly connect lower execution times with a more balanced distribution of the workload. The column labeled as “Gain” illustrates the relative acceleration in the execution times attained by using values of $f = p \times 1.00, 1.25, 1.50$ with respect to those obtained with $f = c$. Values of $f > p \times 1.50$ do not lead to a significant reduction of the execution times. Two major conclusions can be extracted from this experiment: by manipulating f at the MLND stage, one can adjust the degree of parallelism available to subsequent stages; and the bulk of the computational load is concentrated at the leaves (this second observation is confirmed by the higher gains attained by increasing f).

Figure 9 reports the speed-up attained by the implementation of the MIC and PCG stages in our parallel solver for Example 4. In all these executions, f is set to $p \times 1.25$. For both stages, all matrix benchmarks (except for matrices with identifier 1 and 2) of this example deliver a higher speed-up when the number of processors is increased.

Table 6 compares the performance of PARDISO and that of the ILUPACK-based solvers. Although the number of nonzero elements for the smallest problem (F) is roughly 15 times larger for PARDISO, its overall computational time is only twice higher. This confirms that direct solvers are extremely efficient in exploiting level-3 BLAS performance during the numerical factorization. The table also shows that for the two larger cases of this example (M1 and VC3), PARDISO is no longer competitive due to the significant increase of the cost in terms of memory consumption (number of nonzero elements) and computational time (number of operations).

Example 5. We consider four large-scale SPD test matrices from the University of Florida sparse matrix collection⁷.

⁷<http://www.cise.ufl.edu/research/sparse/matrices>.

Code	p	f	Task Tree		MIC			PCG	
			#levels	#leaves	vc (%)	T_p (s)	Gain (%)	T_p (s)	Gain (%)
F1	8	c	4	8	28.3	5.9	0.0	6.3	0.0
	8	$\times 1.00$	5	11	17.8	4.9	16.3	4.7	24.8
	8	$\times 1.25$	5	14	10.7	4.4	24.7	4.2	33.1
	8	$\times 1.50$	6	16	5.0	4.3	26.7	4.3	31.5
	16	c	5	16	27.4	3.3	0.0	3.6	0
	16	$\times 1.00$	6	22	16.3	2.8	16.2	2.8	22.5
	16	$\times 1.25$	6	27	9.7	2.6	21.3	2.8	22.0
VC3	16	$\times 1.50$	7	32	4.7	2.5	24.4	2.9	18.0
	8	c	4	8	14.8	16.1	0.0	21.2	0.0
	8	$\times 1.00$	5	12	10.6	14.7	8.7	18.2	14.2
	8	$\times 1.25$	5	15	4.9	13.3	17.4	17.3	18.4
	8	$\times 1.50$	5	16	3.8	13.3	17.4	17.1	19.3
	16	c	5	16	19.0	8.8	0.0	13.0	0.0
	16	$\times 1.00$	6	22	12.4	7.9	10.4	10.5	19.2
	16	$\times 1.25$	6	28	7.0	7.4	15.8	9.8	24.6
	16	$\times 1.50$	7	33	6.4	8.2	6.9	10.8	19.2

Table 5: Role of parameter f and the dynamic scheduling policy for Example 4.

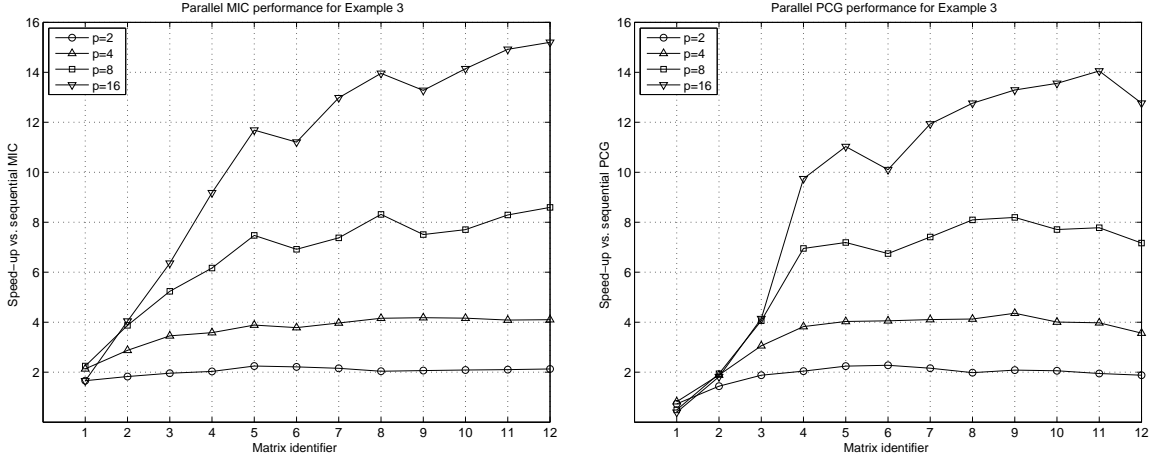


Figure 9: Performance of the parallel MIC (left) and PCG (right) stages for Example 4.

We have selected them in order to evaluate the performance of our parallelization approach with irregularly structured problems arising from very different application areas; see Table 7 for details.

Table 8 compares the performance of the sequential algorithm in ILUPACK with that of our parallel solver applied to Example 5. Although fine-tuning of the MIC stage on an individual basis yields better performance (i.e., less memory and/or time to solution) than that shown in Table 8, we refrain ourselves from optimizing ν and τ , and set their values to $\nu = 5$ and $\tau = 10^{-2}$. (We also made experiments with smaller drop tolerances, but these are skipped to keep the presentation simpler.) For our parallel solver, we use ND-HAMD-B preprocessing with f set to $p \times 2.0$ to let the dynamic scheduling policy improve load balancing. The results in Table 8 confirm that our approach yields remarkable parallel performance for the MIC and PCG stages despite of the irregular nature of these matrices. The mild increase in the number of PCG steps as well as the super-linear speed-ups which are attained in some cases (see e.g., the results for UF1) can be justified because of the underlying irregularity of these matrices which, to some extent, causes a different distribution of the nodes among the algebraic levels with increasing number of processors. We remark that the increase in the number of iterations is more than paid off by the reduction of the execution time yielded by the parallel PCG.

Code	p	PARDISO						ILUPACK-based	
		nmz_L $\times 10^{-6}$	T_{Re} (sec.)	T_{Sy} (sec.)	T_{Fa} (sec.)	T_{So} (sec.)	T_{All} (sec.)	nmz_L $\times 10^{-6}$	T_{All} (sec.)
F	1	24.4	1.6	0.3	5.7	0.7	8.3	1.5	4.4
	16	24.3	1.6	0.3	0.8	0.6	3.3	1.5	1.4
M1	1	97.7	6.0	0.9	36.9	2.7	46.5	4.7	17.5
	16	97.4	6.0	1.2	4.4	1.8	13.4	4.8	4.3
VC3	1	1,870.8	64.9	13.3	3,717.3	57.3	3,582.8	45.2	240.0
	16	1,867.5	66.9	15.8	377.0	24.4	484.1	44.9	42.4

Table 6: Performance comparison between PARDISO and our parallel solver for Example 4.

Code	Matrix name	Application area	n	nmz_A
UF1	af_shell3	Sheet metal forming	504855	17562051
UF2	bmwcra_1	Automotive crankshaft modeling	148770	10641602
UF3	G3_circuit	Circuit simulation problem	1585478	7660826
UF4	ldoor	Structural analysis	952203	42493817

Table 7: SPD test matrices with their code, name, application area of origin, order n and number of nonzeros nmz_A .

We next describe the configuration of the parallel algorithms in BlockSolve95 and in the Hypre-Euclid library for Example 5. These algorithms require the system to be distributed among the processors. We used the k -way partitioning heuristics included in METIS [21] to partition and distribute accordingly the coefficient matrix among the processors, as the amount of communication and load balancing of these algorithms heavily depend on the initial distribution of the coefficient matrix. In this direction, the k -way partitioning heuristics keep the amount of work associated with each subdomain roughly equal while minimizing the size of the edge separator. The degree of parallelism which can be exploited by BlockSolve95 depends on the multicoloring of the adjacency graph which the library internally computes. For Hypre-Euclid, the degree of parallelism which can be exploited during the $ILU(k)$ factorization depends on this initial distribution. In particular, the subdomain intersection graph should have a small chromatic number [9]. The k -way partitioning heuristics do not necessarily provide subdomain intersection graphs with a small chromatic number, though their use is justified because of the underlying irregularity of these matrices, where a natural partitioning by hand is not readily available. We used $k = 0$ and $k = 1$ for the $PILU(k)$ preconditioner in Euclid because these values lead to the best memory/time trade-offs. The rest of parameters of Euclid were set to default values. The parallel PCG solver in both libraries is initialized with the zero vector as the starting guess and the iteration is stopped when the number of iterations reaches 2000 or when the relative residual norm drops below 10^{-8} .

Table 9 compares the performance of BlockSolve95 and that of the ILUPACK-based solvers. The results for BlockSolve95 illustrate, from left to right, the execution time of the initial multicoloring of the adjacency graph, the number of nonzero entries (in millions) in the incomplete Cholesky factor, the execution time of the $IC(0)$ preconditioner, the number of iterations required for convergence (a dagger reflects that the PCG method did not converge within 2000 iterations), the execution time of the PCG solver, and the overall execution time without considering the initial k -way partitioning and distribution of the coefficient matrix. The results of the ILUPACK-based solvers refer to the number of nonzero elements in the MIC preconditioner and the overall execution time including the initial ND-HAMD-B preprocessing step. Although the multicoloring-based approach in BlockSolve95 attains almost linear speed-ups, it is not competitive to our approach in terms of overall solution time and robustness. This is in part due to the $IC(0)$ factorization, which drops any fill-in out of the original sparsity pattern of A , but also because of the multicoloring orderings, which lead to a high degradation in preconditioner quality (i.e., high iteration counts) for the matrices in our benchmark.

Table 10 reports the performance of Hypre-Euclid and the ILUPACK-based solvers. The results for Hypre-Euclid illustrate, for each value of the level-of-fill k , from left to right, the number of nonzero entries (in millions) in the incomplete L factor, the execution time of the $ILU(k)$ preconditioner, the number of iterations required for convergence (a dagger reflects that the PCG method did not converge within 2000 iterations), the execution time of the PCG solver, and the overall execution time without considering the initial k -way partitioning and distribution of the coefficient

n	p	MLND		MIC			PCG		
		Option ND-HAMD-X	T (sec.)	mnz_L $\times 10^{-6}$	T_p (sec.)	S_p (T_1/T_p)	#Iter.	T_p (sec.)	S_p (T_1/T_p)
UF1	1	B	0.0	19.5	33.6	1.0	331	307.9	1.0
	2	B	3.4	19.2	16.9	2.0	365	154.1	2.0
	4	B	4.8	19.2	8.5	4.0	371	69.8	4.4
	8	B	6.3	19.2	4.3	7.9	379	33.4	9.2
	16	B	7.5	19.3	2.3	14.6	403	20.1	15.3
UF2	1	B	0.0	10.2	24.6	1.0	756	321.9	1.0
	2	B	1.9	10.0	11.9	2.1	835	165.4	1.9
	4	B	3.1	10.3	5.7	4.3	866	75.8	4.2
	8	B	4.1	10.6	3.2	7.6	899	42.0	7.7
	16	B	5.0	10.7	1.7	14.1	914	28.8	11.2
UF3	1	B	0.0	12.5	29.5	1.0	121	120.7	1.0
	2	B	3.3	12.5	14.6	2.0	131	66.6	1.8
	4	B	5.1	12.5	6.2	4.8	124	25.3	4.8
	8	B	6.4	12.6	3.0	9.7	136	14.4	8.4
	16	B	7.9	12.6	1.6	18.4	130	7.6	15.8
UF4	1	B	0.0	36.2	61.8	1.0	409	740.0	1.0
	2	B	8.1	36.1	29.8	2.1	465	408.1	1.8
	4	B	11.3	36.1	14.9	4.1	464	186.3	4.0
	8	B	15.0	36.1	8.4	7.4	484	95.3	7.8
	16	B	18.0	36.7	4.3	14.4	476	46.9	15.8

Table 8: Performance results for Example 5.

matrix. The results of Table 10 show that, for serial computations, the inverse-based approach results in faster solution times and increased robustness. For example, the ILU(1) preconditioner could not successfully solve UF4, even consuming more memory than the MIC preconditioner. (We refer to [7] for an extensive empirical comparative study of the sequential version of ILUPACK and other preconditioners.) What is more relevant for the purpose of this paper, we can observe that our parallelization approach attains remarkable speed-ups compared to those of the Hypre-Euclid parallel solvers for these irregular matrices. The poor scaling achieved by Hypre-Euclid in the parallel computation of ILU(k) preconditioners is due to the combination of the following two factors. First, the k -way heuristic partitionings do not necessarily provide subdomain graphs with a small chromatic number, limiting the degree of parallelism which can be exploited during the elimination of the boundary nodes. Second, the computational cost involved in the elimination of the boundary nodes is high compared with that of the interior nodes. The higher speed-ups obtained by Hypre-Euclid in the parallel computation of ILU(0) preconditioners are due to the lower impact of this second factor, i.e., with $k = 1$ the elimination of the boundary nodes becomes much more computationally demanding, resulting in

Code	p	BlockSolve95					ILUPACK-based		
		T_{Col} (sec.)	mnz_L $\times 10^{-6}$	$T_{IC(0)}$ (sec.)	#iter.	T_{PCG} (sec.)	T_{All} (sec.)	mnz_L $\times 10^{-6}$	T_{All} (sec.)
UF1	1	24.1	9.0	10.6	1589	1650.3	1685.0	19.5	341.5
	8	2.9	9.0	1.3	1610	186.7	190.9	19.2	44.0
	16	1.4	9.0	0.6	1504	82.6	84.6	19.3	29.9
UF2	1	11.8	5.4	7.1	†	†	†	10.2	346.5
	8	1.6	5.4	0.9	†	†	†	10.6	49.3
	16	1.2	5.4	0.5	†	†	†	10.7	35.2
UF3	1	30.2	4.6	11.5	1828	2576.1	2617.8	12.5	150.2
	8	3.6	4.6	1.2	1825	347.3	352.1	12.6	23.8
	16	1.7	4.6	0.6	1829	166.2	168.5	12.6	17.1
UF4	1	89.3	21.7	154.2	1723	8220.0	8463.5	36.2	801.8
	8	8.7	21.7	17.4	1650	771.4	797.5	36.1	118.7
	16	3.8	21.7	6.5	1640	318.1	328.4	36.7	69.2

Table 9: Performance comparison between BlockSolve95 and our parallel solver for Example 5.

significantly lower speed-ups. In contrast to Hypre-Euclid, our parallelization is more robust because its degree of parallelism is not limited by an unproper distribution/coloring of the subdomain graph and more important, because the inverse-based approach efficiently controls the fill-in across the interfaces (i.e., the leaves concentrate the bulk of the computation).

Code	Hypre-Euclid											ILUPACK-based	
	p	$k = 0$					$k = 1$					$nmzL$ $\times 10^{-6}$	T_{All} (sec.)
		$nmzL$ $\times 10^{-6}$	$T_{ILU(k)}$ (sec.)	#iter.	T_{PCG} (sec.)	T_{All} (sec.)	$nmzL$ $\times 10^{-6}$	$T_{ILU(k)}$ (sec.)	#iter.	T_{PCG} (sec.)	T_{All} (sec.)		
UF1	1	9.0	9.0	1058	826.6	835.6	12.8	13.1	503	514.2	527.2	19.5	341.5
	8	9.0	4.9	1152	82.0	86.9	12.8	15.0	572	53.3	68.2	19.2	44.0
	16	9.0	4.5	1214	43.6	48.1	12.9	14.6	597	28.7	43.2	19.3	29.9
UF2	1	5.4	6.1	1401	398.5	404.6	15.9	41.0	815	589.0	630.0	10.2	346.5
	8	5.4	19.1	1426	60.4	79.5	15.6	221.2	845	62.6	283.8	10.6	49.3
	16	5.4	40.7	1444	40.0	80.7	14.8	614.9	855	53.9	668.8	10.7	35.2
UF3	1	4.6	14.5	909	394.9	409.4	6.1	14.9	497	235.1	250.0	12.5	150.2
	8	4.6	2.3	1244	65.1	67.4	6.1	2.7	673	41.3	44.1	12.6	23.8
	16	4.6	1.4	1271	32.0	33.4	6.1	1.8	685	20.7	22.5	12.6	17.1
UF4	1	21.7	25.8	†	†	†	47.1	69.4	†	†	†	36.2	801.8
	8	21.7	12.7	†	†	†	47.0	52.7	†	†	†	36.1	118.7
	16	21.7	12.5	†	†	†	46.9	54.9	†	†	†	36.7	69.2

Table 10: Performance comparison between Hypre-Euclid and our parallel solver for Example 5.

5. Conclusions

Our approach to the parallel iterative solution of sparse linear systems demonstrates that the degree of parallelism exposed by MLND is sufficient to efficiently exploit the hardware parallelism in shared-memory platforms with a *moderate* number of processors. Due to the low fill-in usual in iterative solvers based on ILUPACK, even a reduced number of processors can already provide reasonable execution times for large-scale sparse application problems. The mechanisms proposed in this paper include the use of MLND to identify concurrent tasks in the context of preconditioned iterative solvers, exploitation of task-parallelism extracted from the task dependency tree, dynamic scheduling to improve load-balancing, careful mapping of the tasks to processors to improve cache use and reduce communication time, and a parallelization for shared-memory multiprocessors based exclusively in standard tools as OpenMP.

Experimental results on a CC-NUMA platform with 16 processors, using several large scale examples, show that our approach yields a considerable reduction of the execution time for the MIC and PCG stages, while accommodating the mathematical semantics of the inverse-based preconditioning approach implemented in ILUPACK.

Acknowledgments

We thank Thomas Fischer and Mario Bebendorf from the University of Leipzig for providing us the data matrices for Example 4. We also thank François Pellegrini from INRIA Bordeaux for his help in setting up SCOTCH and his useful comments on its use.

References

- [1] A. George, Nested dissection of a regular finite element mesh, *SIAM Journal on Numerical Analysis* 10 (2) (1973) 345–363.
- [2] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd Edition, SIAM Publications, 2003.
- [3] M. Bollhöfer, Y. Saad, Multilevel preconditioners constructed from inverse-based ILUs, *SIAM J. Sci. Comput.* 27 (5) (2006) 1627–1650, special issue on the 8–th Copper Mountain Conference on Iterative Methods.
- [4] O. Schenk, M. Bollhöfer, R. A. Römer, On large scale diagonalization techniques for the Anderson model of localization, *SIAM Review* 50 (2008) 91–112.

- [5] O. Schenk, A. Wächter, M. Weiser, Inertia revealing preconditioning for large-scale nonconvex constrained optimization, *SIAM J. Sci. Comput.* (2008) to appear.
- [6] M. Bollhöfer, M. J. Grote, O. Schenk, Algebraic multilevel preconditioner for the helmholtz equation in heterogeneous media, *SIAM Journal on Scientific Computing* 31 (5) (2009) 3781–3805.
- [7] T. George, A. Gupta, V. Sarin, An Empirical Analysis of Iterative Solver Performance for SPD systems, Tech. Rep. RC 24737, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 (January 2009).
- [8] M. T. Jones, P. E. Plassmann, Scalable iterative solution of sparse linear systems, *Parallel Comput.* 20 (5) (1994) 753–773.
- [9] D. Hysom, A. Pothen, A scalable parallel algorithm for incomplete factor preconditioning, *SIAM J. Sci. Comput.* 22 (6) (2001) 2194–2215.
- [10] G. Karypis, V. Kumar, Parallel threshold-based ilu factorization, in: *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, ACM, New York, NY, USA, 1997, pp. 1–24.
- [11] M. M. monga Made, H. A. van der Vorst, Parallel incomplete factorizations with pseudo-overlapped subdomains, *Parallel Comput.* 27 (8) (2001) 989–1008.
- [12] Z. Li, Y. Saad, M. Sasonkina, pARMS: a parallel version of the algebraic recursive multilevel solver, *Numer. Alg. Appl.* 10 (2003) 485–509.
- [13] Y. Saad, B. Suchomel, ARMS: an algebraic recursive multilevel solver for general sparse linear systems, *Numer. Lin. Alg. w. Appl.* 9 (5) (2002) 359–378.
- [14] M. Hestenes, E. Stiefel, Methods of conjugate gradients for solving linear systems, *J. Research Nat. Bur. Standards* 49 (1952) 409–436.
- [15] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, J. Koster, A fully asynchronous multifrontal solver using distributed dynamic scheduling, *SIAM J. Matrix Anal. Appl.* 23 (1) (2001) 15–41.
- [16] O. Schenk, K. Gärtner, Two-level dynamic scheduling in pardiso: improved scalability on shared memory multiprocessing systems, *Parallel Comput.* 28 (2) (2002) 187–197.
- [17] J. I. Aliaga, M. Bollhöfer, A. F. Martin, E. S. Quintana-Ortí, Parallelization of multilevel preconditioners constructed from inverse-based ILUs on shared-memory multiprocessors, in: C. Bischof, M. Bücker, P. Gibbon, G. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.), *Parallel Computing: Architectures, Algorithms and Applications*, Vol. 38 of *Advances in Parallel Computing*, NIC, 2007, pp. 287–294.
- [18] J. I. Aliaga, M. Bollhöfer, A. F. Martin, E. S. Quintana-Ortí, Design, tuning and evaluation of parallel multilevel ILU preconditioners, No. 5336 in *Lecture Notes in Computer Science*, Springer, 2008, pp. 314–327.
- [19] SCOTCH: static mapping, graph partitioning, and sparse matrix block ordering package.
URL <http://www.labri.fr/perso/pelegrin/scotch/>
- [20] F. Pellegrini, J. Roman, P. Amestoy, Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering, Vol. 1586/1999 of *Lecture Notes in Computer Science*, Springer-Verlag, London, UK, pp. 986–995.
- [21] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs 20 (1) (1998) 359–392.
- [22] M. Bollhöfer, Y. Saad, On the relations between ILUs and factored approximate inverses, *SIAM J. Matrix Anal. Appl.* 24 (1) (2002) 219–237.
- [23] A. Cline, C. B. Moler, G. Stewart, J. Wilkinson, An estimate for the condition number of a matrix, *SIAM J. Numer. Anal.* 16 (1979) 368–375.
- [24] J. I. Aliaga, M. Bollhöfer, A. F. Martin, E. S. Quintana-Ortí, Scheduling strategies for parallel sparse backward/forward substitution, Tech. rep., Proceedings of the PARA 2008, 9-th International Workshop on State-of-the-Art in Scientific and Parallel Computing, submitted for publication (2008).
- [25] G. Haase, U. Langer, A. Meyer, The approximate dirichlet domain decomposition method. part I: An algebraic approach, *Computing* 47 (1991) 137–151.
- [26] O. Schenk, K. Gärtner, On fast factorization pivoting methods for sparse symmetric indefinite systems, *Electr. Trans. Num. Anal.* 23 (2006) 158–179.
- [27] O. Schenk, K. Gärtner, Solving unsymmetric sparse systems of linear equations with PARDISO, *J. of Future Generation Computer Systems* 20 (3) (2004) 475–487.
- [28] Z. Strakos, P. Tichy, Error estimation in preconditioned conjugate gradients, *BIT Numerical Mathematics* 45 (4) (2005) 789–817.