Design, Tuning and Evaluation of Parallel Multilevel ILU Preconditioners

José I. Aliaga[1], Matthias Bollhöfer[2], Alberto F. Martín[1], and Enrique S. Quintana-Ortí[1]

[1] Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071–Castellón, Spain, Tel.: (+34) 964-728278, Fax: (+34) 964-728486; {aliaga,martina,quintana}@icc.uji.es.

[2] Institute of Computational Mathematics, TU-Braunschweig, D-38106 Braunschweig, Germany, Tel.: (+49) 531-391-7536, Fax: (+49) 531-391-8206; m.bollhoefer@tu-braunschweig.de.

Abstract. In this paper, we present a parallel multilevel ILU preconditioner implemented with OpenMP. We employ METIS partitioning algorithms to decompose the computation into concurrent tasks, which are then scheduled to threads. Concretely, we combine decompositions which obtain significantly more tasks than processors, and the use of dynamic scheduling strategies in order to reduce idle threads, which it is shown to be the main source of overhead in our parallel algorithm. Experimental results on a shared-memory platform consisting of 16 processors report remarkable performance for our approach.

Key words: Sparse linear system, incomplete LU factorization, parallel algorithm, OpenMP, shared-memory multiprocessor.

Related conference topics: Parallel and Distributing Computing, Numerical Algorithms for CS&E.

1 Introduction

The solution of large sparse linear systems is an ubiquitous problem in chemistry, physics, and engineering applications. Often, sparse direct solvers are used to deal with these problems, but the large amount of memory they sometimes require (due to, e.g., excessive fill-in in the factors), in practice limits the size of the problems these methods can solve. In recent years, iterative methods based on Krylov subspaces combined with preconditioners as, e.g., incomplete LU decompositions, have become popular and successful in many application problems. Among these methods, ILUPACK[1] (*Incomplete LU decomposition PACKage*) is a novel software package based on approximate factorizations which enhances

[1] http://www.math.tu-berlin.de/ilupack.

the performance of the process in terms of a more accurate solution and a lower execution time.

Our mid-term goal is to develop a parallel package to solve large sparse linear systems on shared-memory multiprocessors (including novel multicore processors) using the same techniques employed in ILUPACK. In an earlier paper [1], we presented a parallel preconditioner for the solution of sparse linear systems with symmetric positive definite coefficient matrix, and an OpenMP-based implementation of this preconditioner. In this paper, we extend previous work with the following new contributions:

- We discuss in detail the foundations of the design of the parallel preconditioner.
- We evaluate the preconditioner using a benchmark collection with problems arising from different domains such as, e.g., computational fluid dynamics (CFD) or circuit simulation.
- We split the task tree deeply in order to decompose the computation into a large number of fine-grain tasks.
- We provide a detailed explanation of the experimental performance. In particular, we use the KOJAK [17] and VAMPIR [16] performance analysis tools to gain a complete understanding of the performance of the parallel algorithm.

The paper is structured as follows. In Section 2 we briefly review the basis of the preconditioning procedure underlying ILUPACK. Next, in Section 3, we offer further details on the design of our parallel preconditioner. Section 4 gathers data from numerical experiments with our parallel algorithm implementation. Concluding remarks and future research goals follow in Section 5.

2 An Overview of ILUPACK

ILUPACK is a preconditioning package to solve large sparse linear systems via preconditioned Krylov subspace methods, where the preconditioner is based on multilevel ILUs. We will focus on the computation of the preconditioner since this is the most challenging task from the parallelization viewpoint.

The rationale behind the computation of the preconditioner is to obtain an incomplete LU decomposition of the coefficient matrix A in such a way that element growth in the inverse triangular factors remains bounded, thereby improving the quality of the preconditioner [2–5, 15]. ILUPACK adopts the following combination of steps in a multilevel manner in pursue of this goal:

1. A static pre-ordering and scaling of the system to transform A as

$$A \rightarrow P^T D_1 A D_2 Q = \hat{A}. \tag{1}$$

   The reordering strategy is intended to reduce the fill-in, while scaling is applied in order to balance the size of the entries.

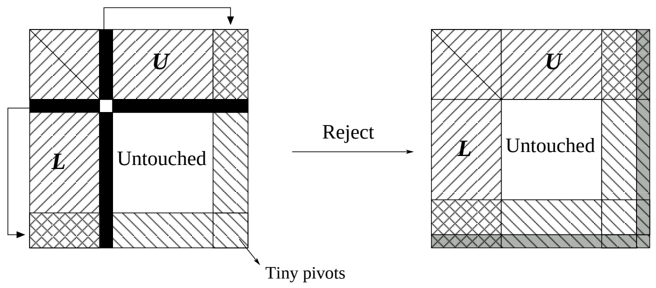


Fig. 1. ILUPACK pivoting strategy.

2. A partial incomplete LU factorization of $\hat{A}$ is applied, where rows/columns associated with "tiny" diagonal pivots are permuted to the bottom right corner of the matrix; see Figure 1. For the multilevel ILU, a bound κ for the (norm of the) inverse triangular factors is prescribed. Tiny pivots are thus defined as those which cause these inverses to exceed κ. These rows/columns are not considered further in this step: their decomposition is delayed until the next level. Thus, we obtain a partial approximation of the following reordered system

$$\hat{P}^T \hat{A} \hat{P} = \begin{pmatrix} B & F \\ E & C \end{pmatrix} \approx \begin{pmatrix} L_B & 0 \\ L_E & I \end{pmatrix} \begin{pmatrix} D_B & 0 \\ 0 & S \end{pmatrix} \begin{pmatrix} U_B & U_F \\ 0 & I \end{pmatrix}, \tag{2}$$

where L_B, U_B^T are unit lower triangular factors, D_B is a diagonal matrix, and S is the approximate Schur complement, where tiny pivots reside. The Schur complement corresponds to:

$$S \approx C - L_E D_B U_F. \tag{3}$$

Elements of magnitude ϵ/κ are dropped from the factors as well as from S during the computation, where ϵ is a user-defined tolerance. For details, see [5].
3. A multilevel configuration, which recursively applies the previous two steps to S, completes the partial decomposition of step 2.

3 Parallel Multilevel Preconditioners

3.1 Task Decomposition

The first step in the development of a parallel preconditioner consists of splitting the procedure that computes the preconditioner into tasks, and identifying the dependencies among these. The starting point of our approach is the *elimination tree* [8] of a sparse symmetric matrix ordering. In the context of the sparse Cholesky factorization, the elimination tree can be used as a source of coarse-grain parallelism [9]. Thus, e.g., if T_i and T_j are independent subtrees of the elimination tree (i.e., neither root node of the subtrees is a descendant of the other), then the operations on the rows corresponding to the nodes in T_i can be proceed independently from those corresponding to the nodes in T_j. Hence these computations can be done simultaneously by separate processors with no communication between them. Therefore, we can define a new tree, which we call the *task (dependency) tree*, where the leaf tasks correspond to the independent subtrees, and the intermediate dependent tasks correspond to the ancestor nodes of these subtrees in the elimination tree; see Figure 2.

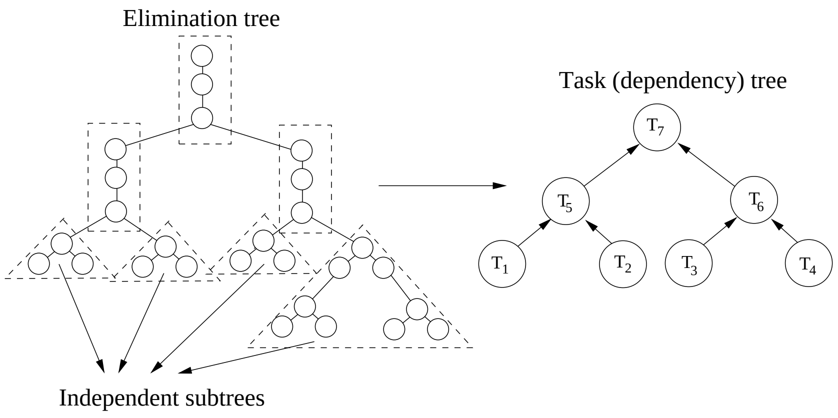


Fig. 2. Elimination tree and an associated task tree of height 2.

These abstractions reveal the parallelism in the sparse Cholesky factorization, and can also be used to obtain a multilevel ILU preconditioner in parallel. We now discuss how this preconditioner is computed using the task tree in Figure 2 (right). We first note that this task tree yields a partition of the coefficient matrix A. For the discussion, we consider the matrix reordering which corresponds to a breadth traversal of the task tree from bottom to top:

$$A \rightarrow \left(\begin{array}{cccc|cc|c} A_{11} & 0 & 0 & 0 & A_{15} & 0 & A_{17} \\ 0 & A_{22} & 0 & 0 & A_{25} & 0 & A_{27} \\ 0 & 0 & A_{33} & 0 & 0 & A_{36} & A_{37} \\ 0 & 0 & 0 & A_{44} & 0 & A_{46} & A_{47} \\ \hline A_{15}^T & A_{25}^T & 0 & 0 & A_{55} & 0 & A_{57} \\ 0 & 0 & A_{36}^T & A_{46}^T & 0 & A_{66} & A_{67} \\ \hline A_{17}^T & A_{27}^T & A_{37}^T & A_{47}^T & A_{57}^T & A_{67}^T & A_{77} \end{array}\right). \qquad (4)$$

Here the separator lines represent the partition with respect to the levels of the task tree. In order to compute the factorization in parallel, we split A into the sum of four submatrices:

$$A = P_1^T \underbrace{\left(\begin{array}{c|c|c} A_{11} & A_{15} & A_{17} \\ \hline A_{15}^T & A_{55}^1 & A_{57}^1 \\ \hline A_{17}^T & (A_{57}^1)^T & A_{77}^1 \end{array}\right)}_{A_1} P_1 + P_2^T \underbrace{\left(\begin{array}{c|c|c} A_{22} & A_{25} & A_{27} \\ \hline A_{25}^T & A_{55}^2 & A_{57}^2 \\ \hline A_{27}^T & (A_{57}^2)^T & A_{77}^2 \end{array}\right)}_{A_2} P_2 +$$

$$P_3^T \underbrace{\left(\begin{array}{c|c|c} A_{33} & A_{36} & A_{37} \\ \hline A_{36}^T & A_{66}^3 & A_{67}^3 \\ \hline A_{37}^T & (A_{67}^3)^T & A_{77}^3 \end{array}\right)}_{A_3} P_3 + P_4^T \underbrace{\left(\begin{array}{c|c|c} A_{44} & A_{46} & A_{47} \\ \hline A_{46}^T & A_{66}^4 & A_{67}^4 \\ \hline A_{47}^T & (A_{67}^4)^T & A_{77}^4 \end{array}\right)}_{A_4} P_4, \tag{5}$$

where $P_1,\ P_2, \ldots, P_4$ denote block permutations, and $A_{55} = A_{55}^1 + A_{55}^2$, $A_{57} = A_{57}^1 + A_{57}^2$, $A_{66} = A_{66}^3 + A_{66}^4$, $A_{77} = A_{77}^1 + A_{77}^2 + A_{77}^3 + A_{77}^4$. In this paper, we have selected $A_{55}^1 = A_{55}^2$, $A_{57}^1 = A_{57}^2$, $A_{66}^3 = A_{66}^4$, $A_{67}^3 = A_{67}^4$, and $A_{77}^1 = A_{77}^2 = A_{77}^3 = A_{77}^4$, to comply with (5). In (5), the separator lines for a given A_i, represent the partitioning with respect to the path from leaf task T_i to the root.

Then, tasks $T_1,\ T_2, \ldots, T_4$ compute, respectively, multilevel ILU decompositions of $A_1,\ A_2, \ldots, A_4$ following the approach presented in Section 2. We focus next on how the decomposition of A_1 is carried out in a parallel environment. The other submatrices are treated analogously. First, step 1 from Section 2 is restricted to the A_{11} block, i.e.:

$$P = Q = \left(\begin{array}{c|c} P_{11} & 0 \\ \hline 0 & I \end{array}\right),\ D_1 = D_2 = \left(\begin{array}{c|c} D_{11} & 0 \\ \hline 0 & I \end{array}\right), \tag{6}$$

with P_{11} and D_{11} of the same row dimension as A_{11}. Next, in step 2 tiny pivots are moved to the end of the A_{11} block, instead of the end of A_1. Thus,

$$\hat{P}_1^T A_1 \hat{P}_1 = \left(\begin{array}{cc|c|c} B_{11} & F_{11} & \hat{A}_{15} & \hat{A}_{17} \\ F_{11}^T & C_{11} & \tilde{A}_{15} & \tilde{A}_{17} \\ \hline \hat{A}_{15}^T & \tilde{A}_{15}^T & A_{55}^1 & A_{57}^1 \\ \hline \hat{A}_{17}^T & \tilde{A}_{17}^T & (A_{57}^1)^T & A_{77}^1 \end{array}\right), \tag{7}$$

and we obtain the following approximate partial factorization:

$$\hat{P}_1^T A_1 \hat{P}_1 \approx \left(\begin{array}{cc|c|c} \hat{U}_{11}^T & 0 & 0 & 0 \\ \tilde{U}_{11}^T & I & 0 & 0 \\ \hline U_{15}^T & 0 & I & 0 \\ \hline U_{17}^T & 0 & 0 & I \end{array}\right) \left(\begin{array}{cc|c|c} D_{11} & 0 & 0 & 0 \\ 0 & S_{11} & S_{15} & S_{17} \\ \hline 0 & S_{15}^T & S_{55}^1 & S_{57}^1 \\ \hline 0 & S_{17}^T & (S_{57}^1)^T & S_{77}^1 \end{array}\right) \left(\begin{array}{cc|c|c} \hat{U}_{11} & \tilde{U}_{11} & U_{15} & U_{17} \\ 0 & I & 0 & 0 \\ \hline 0 & 0 & I & 0 \\ \hline 0 & 0 & 0 & I \end{array}\right). \tag{8}$$

The multilevel approach from Section 2 is repeated on the remaining Schur complement $(S_{ij})_{i,j=1,5,7}$, until either S_{11} is empty or when trying to decompose S_{11}, the number of rows factorized is small with respect to the size of S_{11}.

After computing the multilevel ILU decompositions of $A_1,\ A_2, \ldots, A_4$, we define how the parallel algorithm proceeds with the next level inside the task tree. To do this, we form matrices S_5 and S_6, which are the input matrices for T_5

and T_6, respectively. Now, we show how S_5 is built from the Schur complements computed by T_1 and T_2:

$$S_5 = \underbrace{\left(\begin{array}{cccc|c} S_{11} & 0 & S_{15} & & S_{17} \\ 0 & 0 & 0 & & 0 \\ S_{15}^T & 0 & S_{55}^1 & & S_{57}^1 \\ \hline S_{17}^T & 0 & (S_{57}^1)^T & & S_{77}^1 \end{array}\right)}_{S_1} + \underbrace{\left(\begin{array}{cccc|c} 0 & 0 & 0 & & 0 \\ 0 & S_{22} & S_{25} & & S_{27} \\ 0 & S_{25}^T & S_{55}^2 & & S_{57}^2 \\ \hline 0 & S_{27}^T & (S_{57}^2)^T & & S_{77}^2 \end{array}\right)}_{S_2} . \tag{9}$$

Equation (9) is the key to understand how task interaction works: once T_1 and T_2 are completed, so that their corresponding Schur complements S_1 and S_2 are available, the corresponding processes in charge of T_1 and T_2 send S_1 and S_2 to the process in charge of T_5, which then computes $S_5 = S_1 + S_2$. Tasks T_3, T_4 and T_6 operate in the same way with S_3, S_4 and S_6. Then, T_5 and T_6 compute a partial multilevel ILU decomposition of S_5 and S_6, as $T_1, T_2, \ldots, T_4$ did with submatrices $A_1, A_2, \ldots, A_4$. This recursive parallel process is completed when the root task T_7 fully factorizes $S_{77}^{1,2,3,4}$, which is computed by the interaction of T_5, T_6 and T_7 once the second level of the tree is completed.

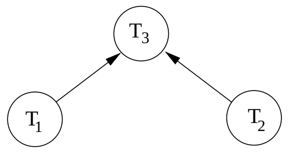
This process yields a template for the computation of multilevel preconditioners based on partial approximations such as, e.g., [14]. The procedure is easily generalized for task trees of height larger than 2, and also for non-complete[2] task trees. Finally, from the mathematical point of view, the parallel approach introduces artificial structure levels to compute the preconditioner in parallel, which may not be necessary in the sequential algorithm; see Figure 3. Therefore, the computations as well as the preconditioners computed by the sequential and the parallel algorithms are, in general, different. The same happens when computing parallel preconditioners with task trees of different heights, as we will see in Section 4.

3.2 Quality of the Task Trees

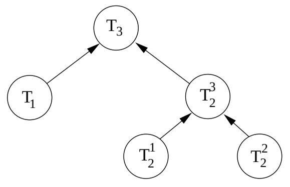
The performance of the parallel algorithm directly depends on properties of the task tree such as its shape or the computational costs associated with its tasks. Assuming the number of processors is a power of two, a desirable task tree should present the following properties:

1. It should be a complete binary task tree of height $log_2(p)$, where p is the number of processors, as this enables enough concurrency for p processors (a task tree of this shape has p leaf tasks).
2. Leaf tasks should concentrate the computational cost of the whole process: in a complete binary task tree the number of processors which can operate in parallel is divided by two when we move one level higher in the tree.
3. Leaf tasks should have similar costs as, otherwise, a static mapping of tasks to processors can lead to an unbalanced distribution of the computational load.

[2] In this context, a complete task tree is one in which all leaves present the same height.

(a) 

$$\longrightarrow \left(\begin{array}{cc|c} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ \hline A_{13}^T & A_{23}^T & A_{33} \end{array}\right)$$

(b) 

$$\longrightarrow \left(\begin{array}{ccc|c|c} A_{11} & 0 & 0 & 0 & A_{13} \\ 0 & A_{22}^{11} & 0 & A_{22}^{13} & A_{23}^{1} \\ 0 & 0 & A_{22}^{22} & A_{22}^{23} & A_{23}^{2} \\ \hline 0 & (A_{22}^{13})^T & (A_{22}^{23})^T & A_{22}^{33} & A_{23}^{3} \\ \hline A_{13}^T & A_{23}^{1}{}^T & A_{23}^{2}{}^T & A_{23}^{3}{}^T & A_{33} \end{array}\right)$$

Fig. 3. Task tree height vs. number of artificial structural levels. (a) A task tree of height one introduces two levels. (b) If task T_2 of task tree (a) is split further into two leaves, the resulting task tree introduces one additional level.

In practice, those three properties are difficult to satisfy simultaneously. For the first two properties, remind that the task tree is constructed from the elimination tree, whose shape strongly depends on the selected coefficient matrix ordering. Thus, given an elimination tree, there is no guarantee that the desired number of independent subtrees (p) will be obtained at a certain level ($log_2(p)$). Besides, there is no guarantee either that the independent subtrees which are identified will not concentrate the bulk of the computational cost; such case happens, e.g., when the number of nodes of the independent subtrees is relatively small compared with the number of nodes of the whole elimination tree.

In order to address these two difficulties, we employ the *MultiLevel Nested Dissection* (MLND) sparse matrix ordering algorithm provided by the METIS library[3] [10]. MLND produces fill-reducing orderings with balanced elimination trees and a high degree of concurrency. Furthermore, intermediate tasks obtained as a by-product of MLND orderings correspond to node graph separators. MLND has been found to produce very small node separators, so that independent subtrees are likely to concentrate most of the nodes of the elimination tree.

MLND helps with the first two items, but we still have to address the difficulties associated with the third property: the computational costs of tasks are unknown before execution and they can be heterogenous. The cost of a task depends on many factors such as, e.g., the size of the associated block, its density and sparsity pattern, the growth of the inverse factors, etc. Unfortunately, many of these are unknown until the approximation has been computed. Therefore, it can be interesting to obtain more leaf tasks than processors and use dynamic scheduling to deal with tasks irregularity as, assuming that leaves concentrate the major part of the computation, load unbalance in the computation of leaf tasks is likely to become the main source of overhead in the parallel algorithm.

[3] `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.`

The leaf tasks which present a "high" relative cost are a potential source of load unbalance. Our approach focuses on these tasks, which are further split into finer-grain tasks. As we mentioned before, these costs are unknown a priori, so that we must use an heuristic which yields an estimation of the cost of a leaf task. Currently, our heuristic is based on the number of nonzeros of the submatrix being approximated. For example, the heuristic of the relative cost of task T_1 (see Figure 2) is $h_1 = \frac{nnz(A_1)}{nnz(A)}$, where $nnz(M)$ denotes the number of nonzero elements of the matrix M. Moreover, we must define the criterion to decide whether to split a given leaf task. Currently, we split those leaves which satisfy $h_i > \frac{1}{f}$, where f is a user-defined threshold for the splitting mechanism. Thus, larger values of f yield a higher task tree. In this paper, we have selected f to be a modest multiple of the number of processors p. Roughly, if we select $f = kp$, we get at least k tasks per processor. For example, assuming a complete task tree of height $log_2(p)$ and $k = 1$, where $h_1 \approx h_2 \ldots \approx h_p$, we obtain p leaves. However, if some leaves at level $log_2(p)$ have a high value of h_i, then some branches are split more than others, yielding a non-complete task tree with more leaves than processors.

3.3 Implementation Details

We now discuss some aspects related with the OpenMP implementation. The task tree is constructed sequentially, prior to the parallel computation of the preconditioner. Then, the computation of the preconditioner begins, and tasks are executed by threads following a dynamic scheduling strategy, which maps tasks to threads on execution time. In general, an optimal scheduling executes *ready* tasks (i.e., those with their dependencies fulfilled) with highest computational cost as soon as possible. Our scheduling strategy applies this requirement only to the leaves of the task tree, as those concentrate the bulk of the computation. Concretely, it always priorizes leaves over inner tasks; among leaves, it schedules those with higher computational time first. The priority between inner tasks depends on our implementation, which we discuss next.

The key of our implementation is a *ready queue* which contains those tasks with their dependencies resolved. Tasks are dequeued for execution from the head, and new ready tasks are enqueued at the tail of this structure. Leaf tasks are initially enqueued in order of priority following our heuristic approach; see Figure 4 (a). The execution of tasks is scheduled dynamically: whenever threads become idle, they monitor the queue for pending tasks. When a thread completes execution of a task, the task dependent on it is examined, and if its dependencies have been resolved, then it is enqueued at the ready queue by this thread. Figure 4 (b) illustrates an example scenario of how this mechanism works with $p = 2$ and the task tree of Figure 4 (a). As can be observed, when thread 2 (i.e., P_2) completes the execution of T_2, dependencies of T_3 are resolved and T_3 becomes ready. Idle threads continue to dequeue tasks until all tasks have been executed. Similar mechanisms have been proposed for irregular codes as part of the Cilk project [11] and for dense linear algebra codes in the FLAME project [6].

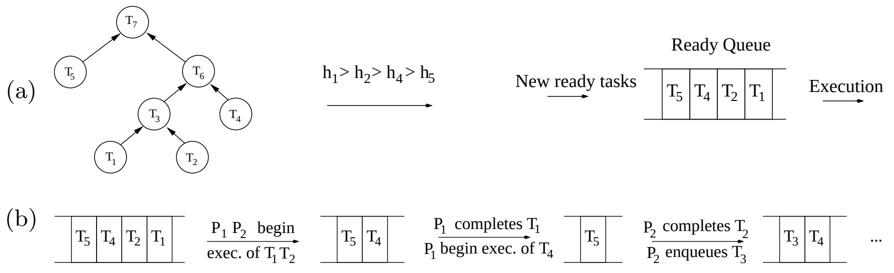


Fig. 4. Dynamic scheduling implementation. (a) The initial state of the ready queue. (b) An example scenario of how dynamic scheduling works with $p = 2$. Note that in this situation there has been a heuristic missestimation: T_1 is completed before T_2.

4 Experimental Results

All experiments in this section were obtained on a SGI Altix 350 CC-NUMA multiprocessor consisting of 16 Intel Itanium2@1.5 GHz processors sharing 32 GBytes of RAM via a SGI NUMAlink interconnect. IEEE double-precision arithmetic was employed in all experiments, and one thread was binded per processor. In both the serial and parallel algorithms we used ILUPACK default values for the condition estimator (κ=100), as well as default the drop tolerances for the L factor (ϵ_1=10^{-2}) and the Schur complement (ϵ_2=10^{-2}).

Table 1 characterizes the benchmark matrices from the UF sparse matrix collection [7] employed in the evaluation. As shown there, they arise from different application areas. Table 2 reports the results obtained from the execution of the serial algorithm in ILUPACK: serial execution time and fill-in factor (ratio between the number of nonzero elements in the triangular factors produced by the algorithm and the coefficient matrix). In the multilevel configuration, the input matrix for the first level has been pre-ordered with MLND, while the input matrices for the rest of the levels are not pre-ordered.

Code	Group/Name	Problem Domain	Rows/Cols.	Nonzeros
M1	*GHS_psdef/bmwcra_1*	Structural Problem	148770	10641602
M2	*Wissgott/parabolic_fem*	Computational Fluid Dynamics	525825	3674625
M3	*Schmid/thermal2*	Thermal Problem	1228045	8580313
M4	*AMD/G3_circuit*	Circuit Simulation Problem	1585478	7660826

Table 1. Benchmark matrices selected to test the parallel multilevel ILU.

For the parallel algorithm, the multilevel configuration of the sequential algorithm is kept. For simplicity, we only present results for $f = p, 2p, 4p$, with $p = 2, 4, 8, 16$ processors, but note that our implementation also allows values of

Code	M1	M2	M3	M4
Time (secs.)	98.6	9.63	22.4	32.1
Fill-in factor	7.2	4.4	4.3	5.0

Table 2. Results from the execution of the serial algorithm in ILUPACK.

f and p which are not power of two, e.g., $p = 12$ and $f = 3p$, and values of f which are not a multiple of p. Therefore, we use six different values of f, resulting in as many different preconditioners. Some of these are computed varying the number of processors; thus, e.g., $f = 8$ is used for $p = 2, 4, 8$ processors. Table 3 reports the number of leaves in the task trees and the fill-in factor for the corresponding preconditioner obtained by various choices of f. The fill-in factors are similar to those attained by the serial algorithm in ILUPACK, except for matrix M_1, for which the fill-in tends to be more reduced. The height of the tree depends on the value of f. As stated in Section 3.1, the computed preconditioners may differ when employing task trees of different heights. Therefore, the fill-in may change.

To give a rough idea of how our approach will perform, we simulate how dynamic scheduling would work in case of $h_i = c_i$, where c_i is the relative computational cost of a given task T_i. For example, assuming the task tree of Figure 4 (a), with $h_1 = 0.39$, $h_2 = 0.29$, $h_4 = 0.2$ and $h_5 = 0.1$, and $p = 2$, then the leaf tasks scheduling events are simulated to happen in the following order: T_1 scheduled to P_1, T_2 scheduled to P_2, T_4 scheduled to P_2, and T_5 scheduled to P_1. If we define $P_1 = h_1 + h_5$ and $P_2 = h_2 + h_4$, we can compute the variation coefficient $c_h = \sigma_h / \bar{h}$ with σ_h the standard deviation and $\bar{h}$ the average of these values (i.e., P_1 and P_2). Table 3 reports the value of c_h (expressed as a percentage) of each combination for all four matrices M1-M4, fraction, and number of processors. A ratio close to 0% implies an accurate heuristic balancing (e.g., M2/$f = 16$/$p = 16$), while a ratio significantly large (e.g., M1/$f = 16$/$p = 16$) indicates heuristic unbalance. As can be observed in Table 3, it is a good strategy to employ the same task tree while reducing the number of processors. For example, for matrix M1/$f = 16$, c_h is reduced by factor of 73% when we use $p = 4$ instead of $p = 16$ processors. Therefore, in general, a larger number of leaves per processor increases heuristic balancing. For matrix M2 this difference is not observed. This is due to the structure of the graph underlying M2, which allows MLND to produce high balanced elimination trees. This can be observed from the fact that, for $f = 2, 4, 8, 16$, we obtain complete binary task trees with f leaves.

Table 4 reports the execution time and the speed-up of the parallel algorithm. The speed-up is computed with respect to the parallel algorithm executed using the same task tree on a single processor. A comparison with the serial algorithm in ILUPACK offers superlinear speed-ups in many cases (see execution times in Table 1) and has little meaning here. Table 4 also reports the variation coefficient c_t (expressed as a percentage), which takes into consideration measured data

M1					M2					M3					M4				
f	Leaves	Fill-in	p	c_h	f	Leaves	Fill-in	p	c_h	f	Leaves	Fill-in	p	c_h	f	Leaves	Fill-in	p	c_h
2	2	6.3	2	1	2	2	4.4	2	0	2	4	4.3	2	1	2	5	5.0	2	0
4	6	4.6	2	1	4	4	4.4	2	0	4	7	4.3	2	0	4	11	5.0	2	0
			4	4				4	0				4	1				4	0
8	10	4.2	2	0	8	8	4.4	2	0	8	12	4.3	2	1	8	19	5.0	2	1
			4	8				4	0				4	1				4	1
			8	9				8	0				8	2				8	2
16	20	3.4	4	3	16	16	4.5	4	0	16	22	4.3	4	0	16	38	5.0	4	0
			8	9				8	0				8	2				8	1
			16	11				16	0				16	2				16	2
32	36	2.9	8	5	32	34	4.5	8	2	32	43	4.4	8	1	32	74	5.0	8	1
			16	9				16	2				16	2				16	1
64	66	2.7	16	5	64	70	4.4	16	3	64	83	4.4	16	1	64	158	5.0	16	0

Table 3. Number of leaf tasks of the tasks trees constructed, fill-in generated for each selected value of f and variation coefficient for each combination of f and p.

from the execution instead of estimations, as c_h do. A ratio close to 0% indicates a balanced distribution of the workload (e.g., M2/$f = 16/p = 16$), while a ratio significantly large (e.g., M1/$f = 16/p = 16$) indicates an unbalanced distribution of the computational load.

For matrices M2, M3, M4, and $p = 2, 4, 8, 16$, it was enough to select $f = p$ in order to get remarkable speed-ups (close to linear). This situation was well predicted by our current heuristic, as can be observed by comparing the values of c_h and c_t in Tables 3- 4. However, for matrix M1 it is not enough to select $f = p$; see, e.g., what happens for $f = 8/p = 8$ and $f = 16/p = 16$, where the speed-up are 4.74 and 9.26 respectively. This situation was not well predicted by our heuristic, which makes too optimistic predictions in these situations. A closer inspection of the task tree for $f = 16$, reveals 20 highly irregular leaf tasks to be mapped to $p = 16$ processors. In this case, dynamic mapping is useless, and hence the computational load is unbalanced. However, using this task tree with $p = 8$ and $p = 4$ leads to better results: when obtaining more leaves per processor, dynamic mapping is able to balance the computational load. Further splitting the task tree led to better performance in many situations, as happened, e.g., for M1/$p = 8$, where the speed-up was increased from 4.74 ($f = 8$) to 7.54 ($f = 32$). Finally, the relationship between c_t and the speed-up, confirms that the computational cost is concentrated on the leaves, and hence load balancing in the computation of these tasks led to high parallel performance.

We also traced the execution of our parallel algorithm with the KOJAK and VAMPIR performance analysis tools [17, 16]. Concretely, we focus on the influence of the potential sources of overhead in our parallel algorithm as idle threads, synchronization, access to shared resources on the SMM, CC-NUMA data local-

M1					M2					M3					M4				
f	p	Time(sec.)	Speed-Up	c_t	f	p	Time(sec.)	Speed-Up	c_t	f	p	Time(sec.)	Speed-Up	c_t	f	p	Time(sec.)	Speed-Up	c_t
2	2	43.7	1.92	1	2	2	4.88	1.97	0	2	2	11.1	1.97	0	2	2	16.1	1.98	0
4	2	28.6	1.96	2	4	2	4.77	1.99	1	4	2	11.0	1.98	1	4	2	16.1	1.98	1
	4	15.9	3.52	11		4	2.41	3.93	1		4	5.58	3.91	1		4	8.20	3.89	1
8	2	28.1	1.92	4	8	2	4.70	1.99	0	8	2	11.1	1.96	2	8	2	16.2	1.96	2
	4	15.2	3.57	10		4	2.37	3.95	0		4	5.59	3.89	2		4	8.22	3.87	2
	8	11.4	4.74	35		8	1.23	7.63	2		8	2.89	7.54	2		8	4.20	7.57	2
16	4	13.2	3.81	4	16	4	2.33	3.98	0	16	4	5.53	3.92	2	16	4	8.09	3.90	2
	8	8.55	5.91	18		8	1.20	7.77	1		8	2.82	7.67	2		8	4.23	7.46	3
	16	5.45	9.26	37		16	0.65	14.3	3		16	1.46	14.8	2		16	2.26	14.0	3
32	8	6.00	7.54	2	32	8	1.20	7.61	2	32	8	2.77	7.76	1	32	8	4.14	7.60	2
	16	3.44	13.1	7		16	0.65	14.0	3		16	1.45	14.8	2		16	2.21	14.2	4
64	16	3.61	13.0	7	64	16	0.62	14.6	3	64	16	1.42	14.9	2	64	16	2.16	14.6	2

Table 4. Performance of the parallel multilevel ILU.

ity, etc. We observed that idle threads are the main source of overhead, while the influence of other factors is negligible. For example, for $p = 16$ and matrix M1, the mean time to access the shared queue used for the dynamic scheduling implementation is approximately 40 μsecs. Therefore, as we employ large-grain parallelism (i.e., f a modest multiple of p), this overall synchronization overhead does not hurt performance.

In some situations dynamic mapping could have further reduced idle threads with a better heuristic estimation. To reduce idle threads it is mandatory to schedule those tasks which are in the critical path (that is, those leaf tasks with higher cost) as soon as possible. As mentioned in Section 3.3, we priorize the execution of leaves with higher heuristic by inserting them first in the queue. If some leaves costs are missestimated dynamic mapping can not comply with this requirement. Figure 5 shows a capture of the global timeline view of VAMPIR when displaying a trace of our parallel algorithm with M1$/f = 16/p = 8$. For each thread (i.e., each horizontal stripe), the display shows the different states and their change over execution time along a horizontal time axis. Concretely, black bars represent time intervals where threads are executing tasks, while white bars represent other thread states, such as idle or synchronization states. As can be observed, the two boxed leaf tasks present a high relative cost, and have been scheduled after other leaves with less computational cost, due to heuristic missestimation. Here, delays due to idle threads are further reduced by splitting the task tree: thus, the efficiency attained for M1$/f = 16/p = 4$ is 95%.

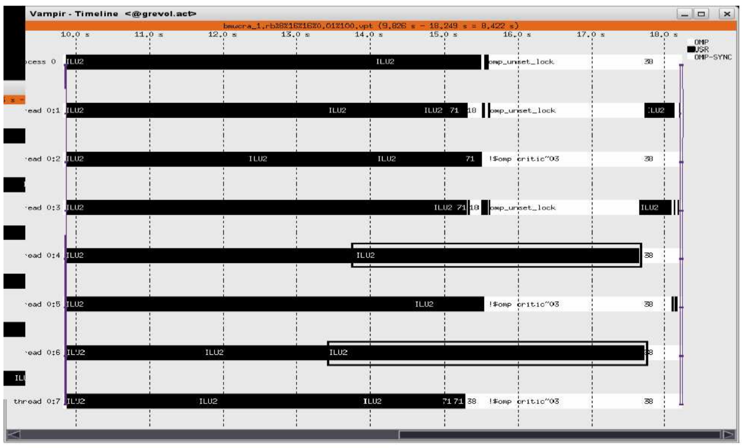


Fig. 5. VAMPIR global timeline view for M1/$f = 16/p = 8$. The two boxed leaf tasks present a high relative cost, and have been scheduled after other leaves with less computational cost.

5 Conclusions and Future Work

We have presented in detail the design foundations of a parallel multilevel preconditioner for the iterative solution of sparse linear systems using Krylov subspace methods. Our OpenMP implementation internally employs the serial routines in ILUPACK. MLND ordering, task splitting, and dynamic scheduling of tasks are used to enhance the degree of parallelism of the computational procedure. The use of large-grain decompositions ($f = p$) led to poor performance in some situations. This can be solved by obtaining finer-grain decompositions ($f = 4p$) combined with dynamic scheduling to deal with tasks irregularity. Remarkable performance has been reported on a CC-NUMA platform with 16 Itanium2 processors and we have gained more insights on the performance attained with the performance analysis tools employed.

Future work includes:

- To compare and contrast the numerical properties of the preconditioner in ILUPACK and our parallel preconditioner.
- To develop parallel algorithms to apply the preconditioner to the system, and to solve the system iteratively.
- To develop parallel preconditioners for non-SPD linear systems.
- To analyze alternative heuristics.

Here we would like to point out that preliminary experiments regarding to the first two aspects, confirm that the quality of the computed parallel preconditioners mildly depends on the task tree. This is currently under investigation.

Acknowledgments

This research has been supported by the CICYT project TIN2005-09037-C02-02 and FEDER, and the DAAD D-07-13360/*Acciones Integradas Hispano-Alemanas* programme HA2007-0071. Alberto F. Martín was also supported by a research fellowship from the *Generalitat Valenciana* (BFPI/06). This work was partially carried out during a visit of Alberto F. Martín to the TU-Braunschweig supported by the programme *Plan 2007 de Promoción de la Investigación* of the Universidad Jaime I.

References

1. J. I. Aliaga, M. Bollhoefer, A. F. Martín, and E. S. Quintana-Ortí, *Parallelization of Multilevel Preconditioners Constructed from Inverse-Based ILUs on Shared-Memory Multiprocessors*, in: Parallel Computing: Architectures, Algorithms and Applications, eds. C. Bischof et. al., 287–294, 2007.
2. M. Bollhoefer, *A Robust ILU Based on Monitoring the Growth of the Inverse Factors*, Linear Algebra Appl., **338**, no. 1-3, 201–218, 2001.
3. M. Bollhoefer, *A robust and efficient ILU that incorporates the growth of the inverse triangular factors*, SIAM J. Sci. Comput., **25**, no. 1, 86–103, 2003.
4. M. Bollhoefer and Y. Saad, *On the relations between ILUs and factored approximate inverses*, SIAM J. Matrix Anal. Appl., **24**, no. 1, 219–237, 2002.
5. M. Bollhoefer and Y. Saad. Multilevel preconditioners constructed from inverse–based ILUs. *SIAM J. Sci. Comput.*, 25(5):1627–1650, 2006.
6. E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, and Robert van de Geijn, *SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures*, in: Proceed. 19th ACM SPAA'07, pp. 116–125, 2007.
7. T. Davis, "University of Florida sparse matrix collection", `http://www.cise.ufl.edu/research/sparse/matrices`.
8. T. Davis, *Direct methods for sparse linear systems*, SIAM Publications, 2006.
9. W. Demmel, J. R. Gilbert, and X.S. Li. An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination. *SIAM J. Matrix. Anal. Appl.*, 20(4): 915–952, 1999.
10. G. Karypis and V. Kumar, *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs*, SIAM J. Sci. Comput., **20**, no. 1, 359–392, 1998.
11. C. Leiserson and A. Plaat, *Programming parallel applications in Cilk*, SINEWS: SIAM News, 1998.
12. "OpenMP Arch. Review Board: OpenMP specifications", `http://www.openmp.org`.
13. Y. Saad, *Iterative Methods for Sparse Linear Systems*, SIAM Publications, 2003.
14. Y. Saad. Multilevel ILU with reorderings for diagonal dominance SIAM J. Sci. Comput., **27**, no. 3, 1032–1057, 2005.
15. O. Schenk, M. Bollhöfer, and R. A. Römer, *On Large Scale Diagonalization Techniques for the Anderson Model of Localization*, SIAM Review, **50**, no. 1, 91–112, 2008.
16. VAMPIR: Visualization and Analisys of MPI Resources. `http://www.vampir-ng.de`.
17. F. Wolf. Automatic performance analysis on parallel computers with SMP Nodes. Ph.D. thesis, RWTH Aachen, Forschungszentrum Jülich, February 2003.