**RAL-TR-2002-028**

# Task scheduling in an asynchronous distributed memory multifrontal solver[1]

Patrick R. Amestoy[2] Iain S. Duff[3] and Christof Vömel[4]

**ABSTRACT**

We describe the improvements to the task scheduling for MUMPS, an asynchronous distributed memory direct solver for sparse linear systems. In the new approach, we determine, during the analysis of the matrix, candidate processes for the tasks that will be dynamically scheduled during the subsequent factorization. This approach significantly improves the scalability of the solver in terms of execution time and storage. By comparison with the previous version of MUMPS, we demonstrate the efficiency and the scalability of the new algorithm on up to 512 processors. Our test cases include matrices from regular 3D grids and irregular ones from real-life applications.

**Keywords:** Sparse linear systems, high performance computing, MUMPS, multifrontal Gaussian elimination, distributed memory code, task scheduling.

**AMS(MOS) subject classifications:** 65F05, 65F35, 65F50.

---

[1]Current reports available by anonymous ftp to ftp.numerical.rl.ac.uk in directory pub/reports. This report is available in compressed postscript as file amdvRAL2002028.ps.gz or as the PDF file amdvRAL2002028.pdf. The report is also available through URL http://www.numerical.rl.ac.uk/reports/reports.html. An extended version of this report was published as Technical Report TR/PA/02/105 from CERFACS.

[2]Patrick.Amestoy@enseeiht.fr, IRIT-ENSEEIHT, Rue Camichel, Toulouse, France.

[3]i.s.duff@rl.ac.uk, the work of this author was supported in part by the EPSRC Grant GR/R46441.

[4]Christof.Voemel@cerfacs.fr, CERFACS, 42 Ave G. Coriolis, 31057 Toulouse Cedex, France.

Computational Science and Engineering Department
Atlas Centre
Rutherford Appleton Laboratory
Oxon OX11 0QX

December 16, 2002

# Contents

# 1 Introduction

We consider the direct solution of sparse linear systems on distributed memory computers. Two state-of-the-art codes for this task, MUMPS and SuperLU, have been extensively studied and compared by Amestoy, Duff, L'Excellent and Li (2001*b*). Specifically, the authors show that on a large number of processors, the scalability of the multifrontal approach used by MUMPS (Amestoy, Duff, L'Excellent and Koster 2001*a*, Amestoy et al. 2001*b*) with respect to computation time and use of memory could be improved. This observation is the starting point for this current work.

The solution of a linear system of equations using MUMPS consists of three phases. In the *analysis* phase, the matrix structure is analysed and a suitable ordering and data structures for an efficient factorization are produced. In the subsequent *factorization* phase, the numerical factorization is performed. The final *solve* phase computes the solution of the system by forward and backward substitution using the factors that were just computed.

The numerical factorization is the most expensive of these three phases, and we now describe how parallelism is exploited in this phase. The task dependency graph of the multifrontal factorization is a tree, the so-called *assembly tree*. A node of this tree corresponds to the factorization of a dense submatrix (the *frontal* matrix), and an edge from one node to another describes the order in which the corresponding submatrices can be factorized. In particular, independent branches of the assembly tree can be factorized in parallel as the computations associated with one branch do not depend on those performed in the others. Furthermore, each node in the tree can itself be a source of parallelism. The ScaLAPACK library (Choi, Demmel, Dhillon, Dongarra, Ostrouchov, Petitet, Stanley, Walker and Whaley 1996) provides an efficient parallel factorization of dense matrices and is used for the matrix associated with the root of the assembly tree. But MUMPS offers another possibility for exploiting parallelism for those nodes that are large enough. Such nodes can be assigned a master process during analysis that chooses, during numerical factorization, a set of slave processes to work on subblocks of the dense matrix. This *dynamic* decision about the slaves is based on the load of the other processors, only the less loaded ones are selected to participate as slaves.

In order to address the scalability issues, we have modified this task scheduling and the treatment of the assembly tree during analysis and factorization. We now give a brief description of these new modifications to Version 4.1 of MUMPS (to which we sometimes refer as the old code or the previous version of MUMPS).

The objective of the dynamic task scheduling is to balance the workload of the processors at run time. However, two major problems arise from offering too much freedom to the dynamic scheduling. In the previous version of MUMPS, a master process is free to choose its slaves among all available processes. Since this choice is taken dynamically during the factorization phase, we have to anticipate it by providing enough memory on every process for the corresponding computational tasks. Since typically not all processes are actually used as slaves (and, on a large number of processors, often only relatively few are needed), the prediction of the required workspace will be overestimated. Thus, the size of the problems that can be solved is reduced unnecessarily because of this difference between the prediction and the allocation of memory by the analysis phase and the memory

actually used during the factorization. Secondly, decisions concerning a node should take account of global information in the assembly tree to localize communication. For example, by mapping independent subtrees to disjoint sets of processors so that all data movements related to a subtree are performed within the set, we can improve locality of communication and increase performance.

With the concept of *candidate processors*, it is possible to *guide* the dynamic task scheduling and to address these issues. The concept originates in an algorithm presented by Pothen and Sun (1993) and has also been used in the context of static task scheduling for sparse Cholesky factorization (Henon, Ramet and Roman 2002). In this paper, we show how it also extends efficiently to dynamic scheduling. For each node that requires slaves to be chosen dynamically during the factorization, we introduce a limited set of processors from which the slaves can be selected. While the master previously chose slaves from among all less loaded processors, the freedom of the dynamic scheduling is reduced so that the slaves are only chosen from among the candidates. This allows us to exclude all non-candidates from the estimation of workspace during the analysis phase and leads to a more realistic prediction of the workspace needed. Furthermore, the candidate concept allows us to structure the computation better since we can explicitly restrict the choice of the slaves to a certain group of processors and enforce for example a 'subtree-to-subcube' mapping principle (George, Liu and Ng 1989). (Throughout this paper, we assume that every processor has one single MPI process associated with it so that we can unambiguously identify a processor and a corresponding MPI process.)

We illustrate the benefits of the new approach by tests using a number of performance metrics including execution time, memory usage, communication volume, and scalability. Our results demonstrate significant improvements for all these metrics, in particular when performing the calculations on a large number of processors.

The rest of this paper is organized as follows. In Section 2, we review briefly the general concepts of the multifrontal direct solution of sparse linear systems. We introduce the assembly tree as a model for the tasks and task dependencies. We describe in Section 3 the possibilities for exploiting parallelism. We then introduce, in Section 4, the concept of candidate processors. In Section 5, we give an overview of how the candidate concept fits into the scheduling algorithm and present the algorithmic details in Section 6. Section 7 gives an overview of the test problems used in this paper. The presentation of our experimental results begins with parameter studies and detailed investigations of the improved algorithms in Section 8. Afterwards, we present a systematic comparison of the previous with the new version of the code on regular grid problems and general matrices in Section 9. Finally, we discuss possible extensions of our algorithm in Section 10 and present our conclusions and a brief summary in Section 11.

## 2 Tasks and task dependencies in the multifrontal factorization

We consider the direct solution of large sparse systems of linear equations

$$Ax = b$$

on distributed memory parallel computers using multifrontal Gaussian elimination. For an unsymmetric matrix, we compute its $LU$ factorization; if the matrix is symmetric, its $LDL^T$ factorization is computed.

The multifrontal method was initially developed for indefinite sparse symmetric linear systems (Duff and Reid 1983) and was then extended to unsymmetric matrices (Duff and Reid 1984). Because of numerical stability, pivoting is required in these cases in contrast to symmetric positive definite sparse systems where pivoting can be avoided. We are concerned with general unsymmetric and symmetric indefinite matrices in the following, for an overview of the multifrontal method for symmetric positive definite systems we refer to Duff and Reid (1983), Duff, Erisman and Reid (1986), and Liu (1992).

In this section, we describe the tasks arising in the factorization phase of a multifrontal algorithm. Specifically, we investigate the work associated with the factorization of individual frontal matrices and the order in which these factorizations can be performed.

The so-called *elimination tree* (Duff and Reid 1983, Liu 1990) represents the order in which the matrix can be factorized, that is, the order in which the unknowns from the underlying linear system of equations can be eliminated. For a dense matrix, the elimination tree is a chain and defines a complete ordering of the eliminations. However, for a general sparse matrix, the definition yields only a *partial* ordering which allows some freedom for the sequence in which pivots can be eliminated.

One central concept of all modern sparse direct solvers and, in particular, the multifrontal approach is to group (or *amalgamate*) columns with the same sparsity structure to create bigger *supervariables* or *supernodes* (Duff and Reid 1983, Liu, Ng and Peyton 1993) in order to make use of efficient dense matrix kernels. We will discuss later the advantages and dangers of amalgamation in the context of a distributed memory multifrontal code. We mention here that it is common to relax the criterion for amalgamation and permit the creation of coarser supernodes with extra fill-in that, however, improve the performance of the factorization (Duff and Reid 1983, Ashcraft and Grimes 1989). The amalgamated elimination tree is called the *assembly tree*.

We now investigate more closely the work associated with the factorization of the frontal matrix at an individual node of the assembly tree. Frontal matrices are considered as dense matrices and we can make use of the efficient BLAS kernels and avoid indirect addressing, see for example Dongarra, Duff, Sorensen and van der Vorst (1998). Frontal matrices can be partitioned as shown in Figure 2.1.

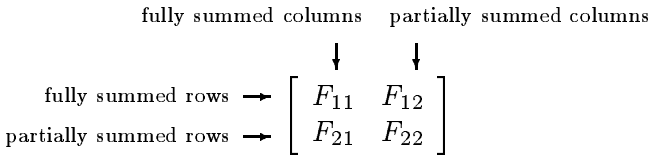


Figure 2.1: A frontal matrix.

Here, pivots can be chosen only from within the block of fully summed variables $F_{11}$. Once all eliminations have been performed, the Schur complement matrix $F_{22} - F_{21}F_{11}^{-1}F_{12}$ is computed and used to update later rows and columns of the overall matrix which are

associated with the parent nodes. We call this Schur complement matrix the *contribution block* of the node.

The notion of children nodes which send their contribution block to their parents leads to the following interpretation of the factorization process. When a node of the assembly tree is being processed, it assembles the contribution blocks from all its children nodes into its frontal matrix. Afterwards, the pivotal variables from the fully summed block are eliminated and the contribution block computed. The contribution block is then sent to the parent node to be assembled once all children of the parent (which are the siblings of the current node) have been processed.

We remark that possibly some variables cannot be eliminated safely from a frontal matrix because of possible numerical instability. In this case, their elimination will be delayed until stable pivots can be found. The corresponding fully summed rows and columns are added to the contribution block and are assembled at the parent node. The contribution block is then larger than was predicted during the analysis phase, and the data structure used in the factorization needs to be modified dynamically.

# 3 Parallelism in the multifrontal factorization

In the following, we identify different sources of parallelism in the multifrontal factorization and describe how these are exploited in MUMPS (Amestoy et al. 2001*a*).

## 3.1 The different types of parallelism

In Section 2, we mentioned that the tasks of multifrontal Gaussian elimination for sparse matrices are only *partially* ordered and that the task dependencies are represented by the assembly tree. A pair of nodes where neither is an ancestor of the other can be factorized independently from each other, in any order or in parallel. Consequently, independent branches of the assembly tree can be processed in parallel, and we refer to this as *tree parallelism* or *type 1 parallelism*.

A fundamental concept for the complete static mapping of assembly trees from model grid problems, the *subtree-to-subcube* mapping, was given by George, Liu and Ng (1989). This algorithm was then generalized for problems with irregular sparsity structure and unbalanced trees to the *bin-pack* mapping scheme by Geist and Ng (1989) and the *proportional* mapping approach by Pothen and Sun (1993). We will describe these algorithms in Section 3.2.

It is obvious that, in general, tree parallelism can be exploited more efficiently in the lower part of the assembly tree than near the root node. Experimental results presented by Amestoy, Duff and L'Excellent (2000) showed a typical speedup obtained from tree parallelism of less than five on 32 processors. These results are related to the observation (Amestoy and Duff 1993) that often more than 75% of the computations are performed in the top three levels of the assembly tree where tree parallelism is limited. For better scalability, additional parallelism is created from parallel blocked versions of the algorithms that handle the factorization of the frontal matrices.

The computation of the Schur complement of frontal matrices with a large enough contribution block can be performed in parallel using a Master-Slave computational model.

The contribution block is partitioned and each part of it assigned to a slave. The master processor is responsible for the factorization of the block of fully summed variables and sends the triangular factors to the slave processors which then update their own share of the contribution block independently from each other and in parallel. We refer to this approach as *type 2 parallelism* and call these nodes *type 2 nodes*.

Furthermore, the factorization of the dense root node can be treated in parallel with ScaLAPACK (Choi et al. 1996). The root node is partitioned and distributed to the processors using a 2D block cyclic distribution. We refer to this as *type 3 parallelism*. Note that a 2D distribution could also be used for frontal matrices other than the root node, but this is not exploited in MUMPS.

MUMPS performs the factorization of the pivot rows of a frontal matrix on a single processor. This can lead to performance problems, particularly if the frontal matrix has a large block of fully summed variables and only a relatively small contribution block. However, if the front size is big enough, it is possible to create artificial type 2 parallelism by splitting the pivot block, see Amestoy et al. (2001*a*) for a discussion of splitting.

## 3.2 Parallel task scheduling: main principles

From the point of view of scheduling, the different types of parallelism vary in their degree of difficulty. Apart from looking at each type of parallelism individually, it is also necessary to investigate their interaction. The main objectives of the scheduling approaches are to control the communication costs, and to balance the memory and computation between the processors. We describe in this section the techniques implemented in Version 4.1 of MUMPS which is described by Amestoy et al. (2000) and Amestoy et al. (2001*a*), and which has been extensively tested and compared with SuperLU (Amestoy et al. 2001*b*) and WSSMP (Gupta 2002). We also present the proportional mapping of Pothen and Sun (1993) from which we develop, in Section 4, our idea of the candidate-based scheduling that is used in the new version of MUMPS.

### 3.2.1 Geist-Ng mapping and layers in the assembly tree

We mentioned in Section 3.1 that, in general, only the lower part of an assembly tree can be exploited efficiently for tree parallelism. Our previous scheduling approach consists therefore of two phases. At first, we find the lower part of the assembly tree where enough tree parallelism can be obtained. Afterwards we process the remaining upper part of the tree additionally exploiting type 2 and type 3 parallelism.

The mapping algorithm by Geist and Ng (1989) allows us to find a layer in the assembly tree so that the subtrees rooted at the nodes of this layer can be mapped onto the processors for a good balance with respect to floating-point operations. Processor communication is avoided by mapping each subtree completely to a single designated processor. We call the constructed layer $L_0$. It marks the boundary between the lower part where scheduling exploits only tree parallelism (type 1), and the upper part where all three types of parallelism are used.

We consider the following top-down tree-processing approach (Geist and Ng 1989). We take as a potential layer $L_0$ the root nodes (or the root node, for an irreducible matrix) of the assembly tree. We check whether the nodes can be mapped onto the processors so

that the load on each processor is balanced up to a threshold. In general, this will not be the case, in particular if the number of processors used for the factorization is larger than the number of root nodes. We then modify the potential layer $L_0$ by replacing the node whose subtree has largest computational cost by its children. Again, we check if the mapping of the new layer is balanced up to the threshold, otherwise we repeat the previous substitution step for the node which has now the highest computational subtree costs, and so forth. The algorithm stops once the nodes in the potential layer $L_0$ allow a threshold-balanced mapping. Intuitively, we can think of the algorithm descending down the assembly tree, as illustrated in Figure 3.1.

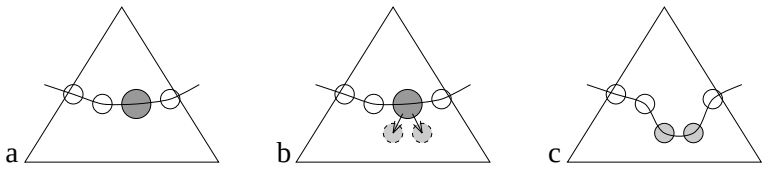


Figure 3.1: Geist-Ng algorithm for the construction of layer $L_0$.

The constructed initial layer $L_0$ induces a layer partition of the upper part of the assembly tree. Before the frontal matrix belonging to a node of the tree can be processed, all contributions from the descendants of the node have to be gathered. This leads to the following recursive definition. Given a node in layer $L_{i-1}$, the parent of this node belongs to $L_i$ if and only if all the children of this parent node belong to the layers $L_0, \ldots, L_{i-1}$. As the nodes in one layer can be only processed if all their children, belonging to the lower layers, have already been treated, the layer partition not only represents dependency but also concurrency of the multifrontal factorization. An example is shown in Figure 3.2.

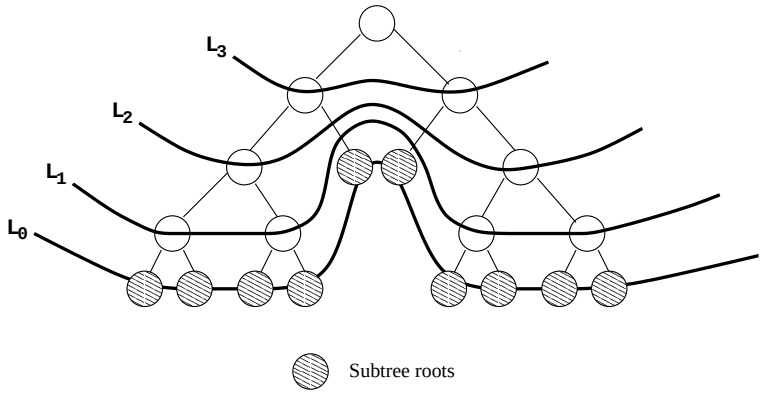


Figure 3.2: Layers in the assembly tree.

### 3.2.2 The proportional mapping of Pothen and Sun

The proportional mapping approach of Pothen and Sun (1993) represents an alternative approach to task scheduling in both regular and possibly irregular assembly trees. It consists of a recursive assignment of processors to subtrees according to their associated computational work.

The assembly tree is processed from top to bottom, starting with the root nodes. For each root node, we calculate the work associated with the factorization of all nodes in its subtree, and the available processors are distributed among the root nodes according to their weight. Each node thus gets its set of so-called *preferential* processors. The same partitioning is now repeated recursively. The processors that have been previously assigned to a node are now distributed among the children proportional to their weight given by the computational costs of their subtrees. The recursive partitioning stops once a subtree has only one processor assigned to it.

A main benefit of the proportional mapping is that communication is effectively localized within the processors assigned to a given subtree, with the partitioning guided from a *global* point of view taking account of the weight of subtrees. An illustration of the proportional mapping algorithm is given in Figure 3.3.

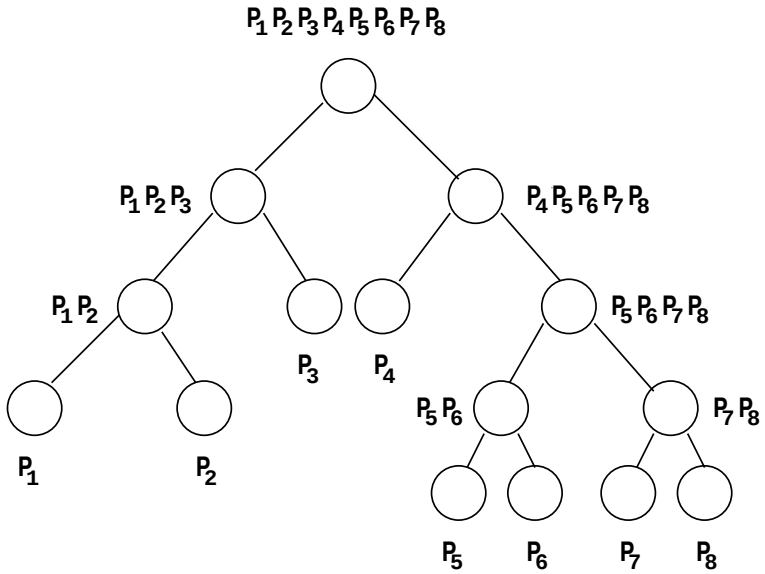


Figure 3.3: Proportional mapping of an assembly tree on eight processors.

While the proportional mapping approach has not been previously used in MUMPS, we mention it here due to its central importance for other sparse linear solvers like PaStiX (Henon et al. 2002) and because the concept of *candidate* processors for type 2 parallel nodes presented later exploits this idea.

### 3.2.3 Dynamic task scheduling for type 2 parallelism

It is possible to extend the static mapping to the tasks arising in the Master-Slave computational model for the factorization of type 2 parallel nodes. However, the static mapping is performed during the analysis phase on the basis of *estimated* costs of computational work and communication. These estimates can be inaccurate if pivots have to be delayed for numerical reasons. For a better equilibration of the actual computational work at run time, both the number and the choice of the slaves of type 2 nodes are determined *dynamically* during factorization as described by Amestoy et al. (2000) and

Amestoy et al. (2001*a*). When the master of a type 2 node receives the symbolic information on the structure of the contribution blocks of the children, the slaves for the factorization are selected based on their current workload, the least loaded processors being chosen. The master then informs the processes handling the children nodes which slaves are participating in the factorization of the node so that they can send the entries in their contribution blocks directly to the appropriate slaves.

The previous version of MUMPS exploits type 2 parallelism above layer $L_0$ as follows. If a node possesses a contribution block larger than a given threshold and the number of eliminated variables in its pivot block is large enough, then it will be declared a type 2 node and will be involved in the dynamic decision to schedule new activities. In the new version of MUMPS, we leave this concept principally unchanged; however, we restrict the freedom for the dynamic choice of the slaves. While, in the earlier algorithm, potentially every processor could be chosen as a slave during run time, in the new approach we restrict this selection to the candidates that have been chosen for a given node during the analysis phase. This is explained in detail in Section 4.2.

# 4 Combining the concept of candidates with dynamic task scheduling

The dynamic choice of the slaves of type 2 nodes during the factorization phase is an attempt to detect and adjust an imbalance of the workload between the processors at run time. It was shown to work very well on a small to medium (64) number of processors (Amestoy et al. 2001*a*, Amestoy et al. 2001*b*). However, the straightforward extension of this technique to a large number of processors often offers more freedom to the dynamic scheduling than can be exploited effectively. In this section, we first give a more detailed illustration of these shortcomings, and then propose as a solution an algorithm that exploits the concept of candidate processors.

## 4.1 Issues of dynamic scheduling

The first issue of dynamic scheduling concerns the memory management. In MUMPS, the amount of memory needed for each processor is estimated during the analysis phase and is reserved as workspace for the factorization. Consequently, if every processor can be possibly taken as a slave of a type 2 node, then enough workspace has to be reserved, on each processor, during the analysis phase for the potential corresponding computational task. However, during the factorization, typically not all processors are actually used as slaves. This leads to a severe overestimation by the analysis phase of the required work space with the possible consequence of exhausting the memory available on the processors.

Secondly, the choice of the slaves is completely local. When a type 2 node is to be processed, its master greedily takes the slaves that seem best to it; those processors that are less loaded (with respect to the number of floating-point operations) than itself at the time of the scheduling decision are selected as slaves. Thus, the decision about the slaves depends crucially on the instant when the master chooses the slaves (locality in time). Furthermore, no account is taken of other type 2 nodes in the tree that have to be processed (locality in space). Instead of sharing the available slaves so that other nodes

can be processed in parallel, a master might decide to take all of them, hindering the work on the other type 2 nodes and the treatment of other branches of the assembly tree. Furthermore, it is not possible in this approach to guarantee any locality of communication and data movement as, in principle, every processor can work on any type 2 node in the assembly tree. However, controlling locality is of great importance for modern computer architectures, for example SMPs like the IBM SP, where, for an MPI programming model, data movement within the shared memory of a node is cheap compared to communication across nodes.

### 4.2 Candidate processors for type 2 parallel nodes

In the following, we present a concept of *candidate processors* that naturally addresses the issues raised in Section 4.1. For each type 2 node that requires slaves to be chosen dynamically during the factorization because of the size of its contribution block, we introduce a limited set of processors from which the slaves can be selected. While the master previously chose slaves from among all less loaded processors, slaves are now only chosen from this list of candidates. This effectively allows us to exclude all non-candidates from the estimation of workspace during the analysis phase and leads to a tighter and more realistic estimation of the workspace needed. Secondly, we can expect a performance gain in cases as described in the previous section where greedy decisions of one type 2 master can no longer hinder processors from processing another node.

The candidate concept can be thought of as an intermediate step between full static and full dynamic scheduling. While we leave some freedom for dynamic decisions at run time, this is guided by static decisions about the candidate assignment during the analysis phase. We refer to Section 6.6 for a full description of the algorithmic details.

In Section 3.2.1, we described the layer structure of the assembly tree. As each layer of the assembly tree represents a view of concurrent execution, all type 2 nodes on the same layer are potential rivals for slave processors, see Section 4.1. By assigning the candidates to all type 2 nodes of a given layer simultaneously, we avoid isolated treatment of nodes and direct our candidate concept from a global view of complete layers.

The assignment and the choice of the candidate processors is guided using a proportional mapping as described in Sections 3.2.2 and 5.1. We partition the set of processors recursively, starting from the root, so that for each subtree there is a well defined subset of *preferential* processors which guides the selection of the candidates.

With this approach, we achieve

- Locality of communication as we limit the communication to those processors belonging to the subtree.
- Independence of computation as we limit the interaction of the processing of one subtree with the treatment of another independent one.

## 5 Task mapping and task scheduling in MUMPS

In this section, we give a generic description of the algorithm used by Version 4.1 of MUMPS (Amestoy et al. 2001*a*, Amestoy et al. 2001*b*) and discuss in general terms our

improvements to it as they have been integrated into the new version. A detailed discussion of the key modifications is given in Section 6. We speak, in the following, of task *mapping* when we refer to the assignment of master processors and candidates during the analysis phase, and of task *scheduling* when we refer to the dynamic choice of type 2 slaves during the factorization phase.

### 5.1 Task mapping algorithm during the analysis phase

We consider the task mapping during the analysis phase and compare the previous version with the new version of MUMPS. A first major point to emphasize is the greater flexibility and adaptivity of the new algorithm when mapping the upper part of the assembly tree (that is, above layer $L_0$). The former version, shown in Algorithm 1, performs a simple mapping of only the master nodes, while the new version, shown in Algorithm 2, treats the upper part layer-wise, mapping both master nodes and type 2 candidates. Using a layer-wise approach we take better account of the task dependency that will control the later factorization phase and, by analysing the quality of mapping decisions taken on previous layers, we can try to correct problems by influencing the mapping of the current layer. This adaptivity was conceptually impossible in the old mapping algorithm.

The second contribution of the new algorithm is of course the added features. A very important feature is the candidate concept guided by a proportional mapping partition of the processors. Furthermore, we have added to the treatment of each layer a preprocessing step that performs amalgamations and node splitting. Moreover, we have improved the construction of layer $L_0$ for better memory scalability. Lastly, we treat memory imbalances due to type 2 node mapping using a post-processing step.

We now present in more detail the previous version of the task mapping (Algorithm 1) and compare it afterwards with the new one, Algorithm 2.

**Algorithm 1** *Old task mapping algorithm.*

(1) Given the assembly tree of a sparse matrix $A$
(2) Build and map initial layer $L_0$
(3) Decide type of parallelism for nodes in upper part of tree
(4) Map master nodes of upper part of tree

The starting point (1) of the original algorithm is the assembly tree that was constructed from the elimination tree of a given sparse matrix using basic amalgamation and node splitting. From this assembly tree, the algorithm constructs, in step (2), an initial layer $L_0$ following the Geist-Ng approach presented in Section 3.2.1 with the objective of balancing the work between the processors. Afterwards, it is decided for which nodes type 2 or type 3 parallelism is exploited (3), and finally the masters of all nodes above layer $L_0$ are mapped (4) with the objective of balancing the memory. The choice of the slave processors for the type 2 nodes is left entirely to the dynamic scheduler during factorization, see Section 5.2.

The starting point ($1'$) of the new algorithm is the same assembly tree as for the old approach (1). In step ($2'$), we calculate a variant of the proportional mapping as introduced in Section 3.2.2 and whose algorithmic description is given later in Section 6.1. For each

**Algorithm 2** *New task mapping algorithm.*

($1'$) Given the assembly tree of a sparse matrix $A$
($2'$) Calculate relaxed proportional mapping, i.e. the *preferential processors*
($3'$) Build and map modified initial layer $L_0$
current_layer = 1
**while** there exist unmapped nodes on or above current_layer **do**
  ($4'$) Perform tree modifications if necessary
  ($5'$) Decide type of parallelism for the nodes on current_layer
  ($6'$) Map the tasks associated with the nodes on current_layer
  current_layer = current_layer + 1
**end while**
($7'$) Post-processing of the candidate selection to improve memory balance

node in the assembly tree, we obtain a set of *preferential processors* that will guide the selection and mapping of the candidate processors in step ($6'$). Step ($3'$) differs from the corresponding step (2) in the old algorithm insofar as the constructed initial layer has one additional property. Not only can it be mapped so that the computational work is balanced between the processors, but we also control better the memory demands of the subtree roots, see Section 6.2 for the details. Step ($4'$) performs amalgamations and node splitting to improve the nodes of the current layer. In step ($5'$), we decide which type of parallelism we exploit for the nodes of the current layer. Nodes are of type 2 when their contribution block is large enough, that is greater than a minimum block size. The list of tasks associated with the current layer includes the masters for the type 1 and type 2 nodes, and the type 2 candidates which are derived from the proportional mapping ($2'$), see Section 6.1. For the task mapping, we use a list scheduling algorithm that is described in Section 6.4. The main difference of the new mapping ($6'$) from the old one (4) is that we now pre-assign candidate processors for the type 2 nodes while in the former version, every processor was a potential type 2 slave. The post-processing step ($7'$) affects mostly the $LU$ factorization. Because the flop-based equilibration of the mapping step ($6'$) can lead to a particularly bad memory balance, we perform a remapping of the type 2 masters for improved memory balance, as described in Section 6.5.

## 5.2 Task scheduling during the factorization phase

In this section, we describe the task management of a processor during the factorization phase. Here, the old and the new version of the algorithm differ only in the way the type 2 slaves are chosen. In the previous algorithm, every processor was a potential slave for a type 2 node, whereas now only the candidates can be selected to work on the parallel update of the contribution block.

The task pool (1) of a processor can contain the following tasks: master of a type 1 node, master of a type 2 node, or slave of a type 2 node. MUMPS uses a stack as the data structure for the task pool; the processor adds new tasks (3) or extracts them from the pool (4), respectively. If, during the factorization, the task pool of the processor is empty, it will wait until it receives new tasks and then re-enter loop (2). If the processor works

**Algorithm 3** *Dynamic task scheduling performed on each processor during the factorization.*

```
(1) Given the task pool of one processor
while (2) Not all tasks processed do
  if Work is received from another processor then
    (3) Store work in pool of tasks
  else
    (4) Extract work from the task pool
    if Task is master of type 2 node then
      (5) Choose and notify the slaves for the type 2 node
    end if
    (6) Perform pivot elimination and/or contribution block update
  end if
end while
```

as a type 2 master, it chooses the slaves that will participate in the parallel contribution block update (5) before it starts the elimination of the pivotal block (6). Otherwise, if the processor is a type 1 master or a type 2 slave, it begins directly with the pivot elimination or the contribution block update, respectively (6).

In the new version of the algorithm, only step (5) is modified to ensure that the type 2 slaves are selected from among the candidates allocated for the type 2 node. We give the details of the algorithm for choosing the slaves in Section 6.6.

# 6 Details of the improved task mapping and scheduling algorithms

After the general discussion, in Section 5, contrasting the task mapping and scheduling for the old and new versions of MUMPS, we now describe the key points of the new algorithm in detail.

## 6.1 The relaxed proportional mapping

We give below, in Algorithm 4, an algorithmic description of one step of the proportional mapping presented in Section 3.2.2. The preferential processors given to a node are distributed among its children according to their weights. Note that we can *relax* the strict proportional mapping by multiplying the number of preferential processors $n_a$ by a relaxation factor $\rho \geq 1$ in step (2).

In step (1), we calculate the relative costs $c_r(s)$ of a child $s, s \in \{s_1, \ldots, s_i\}$ from the costs $c(s)$ for the factorization of all nodes in the subtree rooted at $s$ as

$$c_r(s) = \frac{c(s)}{\sum_{k=1}^{i} c(s_k)}. \tag{6.1}$$

From the relative weight of child $s$, we obtain its share of preferential processors in step (2) that can be relaxed by the factor $\rho$. A fundamental property of the proportional

**Algorithm 4** *One step of proportional mapping.*

Given a node $n$ with preferential processors $p_1, \ldots, p_{n_a(n)}$ and children $s_1, \ldots, s_i$
**for each** child $s$ of $n$ **do**
(1) Calculate relative costs $c_r(s)$ of child $s$, $0 \leq c_r(s) \leq 1$
(2) Calculate number of preferentials $n_a(s) = \max(1, \min\{\rho \times c_r(s) \times n_a(n), n_a(n)\})$ for child $s$
**end for**
(3) Cyclic assignment of the preferential processors for all children $s_1, \ldots, s_i$

mapping is that the preferential processors of a child $s$ are a subset of the preferential processors of its parent node $n$. This is ensured because $n_a(s) \leq n_a(n)$ in (2) and we always choose from the preferential processors of the parents at step (3). Here, we also make sure that each child has at least one processor, even if its cost is negligible. After we have calculated the number of preferential processors for all children, in step (3) we distribute the processors $p_1, \ldots, p_{n_a(n)}$ among the children. If the proportional mapping is strict ($\rho = 1$) and the number of preferential processors calculated from the relative weight $c_r(s)$ in equation (6.1) is an integer, each processor is assigned exactly once. Otherwise, and in particular if the proportional mapping is relaxed with $\rho > 1$, processors can become preferential for more than one child. Consequently, large values of $\rho$ will dilute the strict partition of the processors so that in the extreme case, $\rho \to \infty$, all processors become preferential for each node.

## 6.2 The Geist-Ng construction of layer $L_0$

We now give an algorithmic description of the construction of the initial layer $L_0$ that extends the Geist-Ng approach presented in Section 3.2.1.

**Algorithm 5** *The Geist-Ng algorithm.*

(1) Let $L_0$ contain all root nodes of the assembly tree
(2) Map layer $L_0$
**while** (3) Layer $L_0$ is not acceptable **do**
(4) Find node in $L_0$ with highest computational costs
(5) Replace this node by its children in $L_0$
(6) Map new layer $L_0$
**end while**

Starting with a potential layer $L_0$ consisting of the root nodes of the assembly tree (1), we first compute (2) a mapping of $L_0$ with the list scheduling heuristics described in Section 6.4. The former criterion for accepting the layer in step (3) demands that the load imbalance between the processors is smaller than a threshold. Here, the work associated with a node in $L_0$ is defined as the costs for computing the factors of the subtree rooted at the node and can be estimated during the analysis phase. If the mapping of layer $L_0$ is not acceptable, then the node with the highest costs is eliminated from the layer and replaced by its children (4, 5). A new mapping is computed (6) with the same algorithm as in (2).

The main problem of the algorithm is that balancing the computational work does not necessarily imply balancing the memory. Consider a node with a very small number of pivots but a big contribution block. The costs for the factorization depend mainly on the size of the pivotal block and are small, while the memory required to stack the contribution block is large. In order to take care of such situations, we propose the following approach. If a node with a large contribution block was in the upper part of the tree above $L_0$, it could either be amalgamated with its parent or become a type 2 node, and, in both cases, the memory problems would vanish. Thus, by eliminating such nodes from layer $L_0$ and replacing them by their children, we control the memory required for the subtrees in addition to balancing the work on layer $L_0$. The idea is to treat critical nodes in the latter part of the algorithm by moving them into the upper part of the tree.

Summarizing, we modify the criterion of acceptability (3) to demand that both the load imbalance for the mapping of $L_0$ is smaller than a threshold *and* that $L_0$ contains no nodes that would need to be amalgamated.

### 6.3 Choosing the number of candidates for a type 2 node

We now describe the role of the proportional mapping for the decision of which processors become candidates for a type 2 node. Our approach consists of two steps. For a given layer, we first determine for each type 2 node the *number* of candidate processors. In a second step, we choose the candidates from the available processors. (Thus, for a given node $n$ we determine first an integer number $n_c(n)$ that determines how many candidate processors $p_1, \ldots, p_{n_c(n)}$ have to be chosen in the second step.) The reason for this approach is the following. The selection of a candidate processor is conceptually similar to the selection of the master processors for the type 1 and type 2 nodes; all these are tasks that need to be mapped onto the set of processors. In Section 6.4, we describe the algorithm that we use to map the tasks associated with one layer in the assembly tree. By mapping the master and candidate processors together, we hope to obtain better load balancing.

**Algorithm 6** *Determining the number of candidates using the preferentials.*

Given a layer in the assembly tree
**for each** Type 2 node $n$ with $n_a(n)$ preferential processors in the layer **do**
(1) Determine the number of candidates by $n_c(n) = n_a(n)$.
**end for**
(2) OPTIONAL: Redistribute the total number of candidates of the layer among the layer's type 2 nodes according to their relative weights.

We have experimented with two different ways for determining the number of candidates for a given type 2 node and describe these in Algorithm 6. In the first approach, we select its preferential processors as candidates, thus setting the number of candidates equal to the number of preferentials. We emphasize that this approach is *not* equivalent to a relaxed proportional mapping as the candidates are only *potential* slaves for the factorization. In the second approach, we employ an additional post-processing step where we redistribute the candidates of the layer according to the relative weight of the nodes. As described in Section 3.2.2, the proportional mapping is calculated from the costs of

complete subtrees, not individual nodes. So it might happen that a small node has a large number of preferentials because it is the root of a large subtree, while a relatively large node on the same layer has only a small number of preferentials. In order to correct this, we can reassign candidates from small type 2 nodes as candidates of large type 2 nodes on the same layer by the optional step in Algorithm 6.

### 6.4 Layer-wise task mapping

The algorithm that we use for the mapping of the tasks of each layer is a variant of the well known list scheduling algorithm (Hochbaum 1996) where we first make a list of the tasks sorted by decreasing costs, and then maps the tasks in this order one after another to the processor that has the least work assigned so far. We remark that this heuristic can be proved to construct a schedule whose total makespan (that is, the time by which all jobs complete their processing) never exceeds twice the makespan of an optimal schedule (Hochbaum 1996).

As described in the previous sections, the tasks associated with a layer can include the following:

- The masters of all type 1 and type 2 nodes.
- For each type 2 node, the number of candidate processors determined using the node's preferential processors, see Section 6.3.
- The type 3 parallel node.

In the case of layer $L_0$, we employ the original list scheduling algorithm (Hochbaum 1996), however, for all upper layers $L_1, L_2, \ldots$ our algorithm is more complicated for two reasons. Firstly, we want to guide mapping decisions by the proportional mapping representing a global view of the tree. Secondly, we have to take care of constraints that arise either from explicit user-given limits on memory or work for each processor, or implicitly from the fact that any two candidate processors or any candidate and the master of a type 2 node have to be different from each other.

**Algorithm 7** *Generic mapping algorithm.*

```
(1) Create an ordered task list
while Task list not empty do
  (2) Extract the next task t_i from the list
  (3) Make a preference list for the processors
  while Task t_i not mapped to a processor do
    (4) Try to map t_i to next processor from the preference list
  end while
end while
```

The first two steps (1) and (2) of Algorithm 7 are identical to the original list scheduling approach: we create a list of all tasks that have to be mapped on the layer, that is, the work of the type 1 node masters, of type 2 node masters, and of type 2 node candidates (which have been obtained from the proportional mapping, as described in Section 6.1).

This list is then ordered by decreasing costs and the tasks are mapped in the order that they appear in the list.

Steps (3) and (4) are the generalization of the idea of mapping to the least loaded processor. In order to take account of the proportional mapping, we can simply propose mapping the task on the least loaded of the preferential processors coming from the proportional mapping. However, this is actually too simple as the mapping constraints for type 2 nodes that we mentioned above have to be respected. Our solution is that we create a *preference* list containing all the processors, at first the preferential ones ordered by decreasing workload and then the non-preferential ones ordered separately, also by decreasing workload. The first processor in the preference list that doesn't violate the mapping constraints will be the one to which the task is mapped.

## 6.5 Post-processing of the assembly tree for an improved memory balance in the $LU$ factorization

.

The mapping algorithm from Section 6.4 tries to balance the work between the processors. However, there is an important difference between symmetric and unsymmetric factorization with respect to memory. In the $LDL^T$ factorization, the master of a type 2 node only holds the pivotal block whereas, in the $LU$ factorization, the master stores the *complete* fully summed rows. The additional memory that a master requires for storing its part of the factors in the $LU$ factorization (with respect to the $LDL^T$ factorization) is illustrated in Figure 6.1. In both the $LU$ and the $LDL^T$ factorization, a type 2 slave reserves space for a part of the $L$ factor below the pivotal block as shown in Figure 6.2. Thus, in the case of the $LU$ factorization, the work equilibration can lead to memory imbalances if the same processor becomes master of several type 2 nodes.

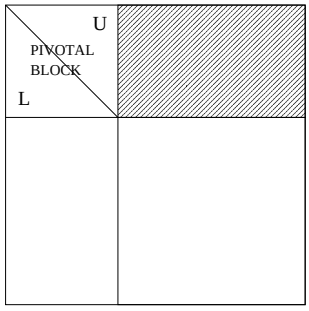


Figure 6.1: Additional memory needed by a type 2 master in the unsymmetric factorization.

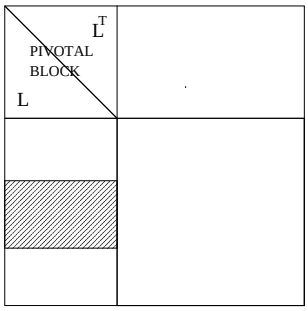


Figure 6.2: Memory reserved for the $L$ factor on a type 2 slave in both $LU$ and $LDL^T$ factorization.

We propose the following simple remedy. After the whole tree is mapped with the objective of balancing the work, we use a post-processing step to correct obvious memory problems.

In Algorithm 8, we process the upper part of the assembly tree from the top down (1), as the type 2 nodes creating the biggest problems are often near a root of the tree. By

**Algorithm 8** *Post-processing for better memory equilibration in the LU factorization.*

(1) Process the type 2 nodes in the tree from the root downwards
(2) For a node $n$ with master $p^M(n)$ select candidate $c^*(n)$ with smallest memory
**if** memory imbalance can be improved by swapping $p^M(n)$ and $c^*(n)$ **then**
  (3) Exchange the roles of master and candidate processor $p^M(n) \Leftrightarrow c^*(n)$
**end if**

swapping a master processor with one of the candidates, we still guarantee the benefits of the proportional mapping used in the candidate assignment, but locally improve the memory imbalance (steps 2 and 3).

### 6.6 The dynamic scheduling algorithm used at run time

We describe, in Algorithm 9, the dynamic scheduling algorithm used in MUMPS for the mapping of the slaves of a type 2 node at run time (Amestoy et al. 2001*a*) and show how the candidate concept influences the original approach. Furthermore, we identify and describe the role of the algorithmic parameter that controls the minimum granularity for type 2 parallelism at run time. We denote this parameter by $k_{\max}$.

**Algorithm 9** *Dynamic choice of the slaves of a type 2 node.*

Given a type 2 node $n$ with master processor $p^M(n)$ and children $s_1, \ldots, s_i$
(1) The masters of the children $p^M(s_1), \ldots, p^M(s_i)$ send symbolic data to $p^M(n)$
(2) $p^M(n)$ analyses its information concerning the load of all processors
(3) $p^M(n)$ decides the partitioning of the frontal matrix of node $n$ and chooses the slave processors $p_1^S(n), \ldots, p_j^S(n)$
(4) $p^M(n)$ informs all processors working on the children about the partition
(5) The numerical data is sent directly to the slaves $p_1^S(n), \ldots, p_j^S(n)$

Instead of first assembling all numerical data on the master of the type 2 node and distributing it afterwards to the slaves, a two-phase assembly process is used. At step (1), the master receives only the integer data describing the symbolic structure of the front. At step (2), the master analyses the information on the workload of the other processors. Each processor is responsible for monitoring its own workload status and for broadcasting significant changes to all other processors so that everyone has accurate information on the overall progress of the factorization. At step (3), the master processor $p^M(n)$ selects the least loaded among all processors as slaves. As a general rule, all processors that are less loaded than $p^M(n)$ are chosen as slaves. Then a partition of the frontal matrix onto the slaves is calculated.

The following constraint on the number of slaves, $n_s$, for a type 2 node selected during factorization is imposed. It must always satisfy

$$n_s \geq \max(\lfloor \frac{ncb}{k_{\max}} \rfloor, 1), \tag{6.2}$$

where $ncb$ denotes the number of rows in the contribution block. The parameter $k_{\max}$

controls the maximum work of a type 2 slave and thus the maximum buffer size permitted for the factorization of a type 2 node.

Once the slaves participating in the parallel update of the contribution block have been selected, they obtain the part of the symbolic information from the master $p^M(n)$ that is relevant for their work (4). Furthermore, they receive the corresponding numerical data from the processors working on the children (5).

In the candidate-based scheduling approach, we modify step (3) so that the slaves are *always* chosen among the candidates provided for the node. At first, we select all those candidates that are less loaded than the master processor. If the inequality (6.2) is not satisfied, additional candidates are chosen so that it holds. In order to be able to choose the slaves at factorization time among the candidates so that (6.2) is never violated, we must take care to provide enough candidates during analysis. If we provide only a minimum number of candidates so that (6.2) holds as equality, we enforce a static scheduling. In this case, all candidates must be selected as slaves during factorization. We have freedom for dynamic choices during the factorization only if we provide a number of candidates greater than $ncb/k_{\max}$. Consequently, the freedom offered to the dynamic scheduling can be measured by the number of extra candidates given for a type 2 node. On the other hand, the larger the number of candidates for a given node, the closer we come to the case of fully dynamic scheduling with all the possible drawbacks discussed in Section 4.1. In the following experiments, we will see that the dynamic scheduling works most effectively when $k_{\max}$ is large and (6.2) does not impose a significant restriction. Because of overall memory constraints, the scope for increasing the parameter is limited; however, we will see that the better memory estimates from the candidate approach greatly increase the range from which $k_{\max}$ can be chosen.

We remark that, in the case of the $LDL^T$ factorization, MUMPS precomputes a partition of the contribution block in order to guarantee that each of the slaves performs approximately the same amount of work (Amestoy et al. 2000). As the frontal matrix is symmetric (and only the lower triangular part is stored), rows at the bottom of the frontal matrix are longer (and thus associated with more work) than rows at the top.

# 7 The test environment

In this section, we present the test matrices that we use to illustrate the behaviour of our algorithm. Specifically, we consider in Section 7.1 matrices from regular grids and in Section 7.2 irregular ones from real-life applications. We mention that our set of regular grid problems includes those used by Amestoy et al. (2001*b*) which allows us to compare the performance of the new code with results already published.

For our tests, we use both a CRAY T3E-900 (512 processors, 256 MBytes RAM and 900 peak MFlops per processor) and an SGI Origin 2000 (32 processors, 16 GBytes shared memory, 500 peak MFlops per processor). We consider different orderings including nested dissection from SPARSPAK (George and Ng 1984) and METIS (Karypis and Kumar 1998), and Approximate Minimum Fill (Rothberg and Eisenstat 1998, Ng and Raghavan 1999).

### 7.1 Regular grid test problems

We consider a set of test matrices obtained from an 11-point discretization of the Laplacian on 3D grids of either cubic or rectangular shape, the grid sizes are reported in Table 7.1. The set of problems is chosen as in Amestoy et al. (2001*b*) and is designed so that when the number of processors increases, the number of operations per processor in the *LU* factorization stays approximately constant when employing a nested dissection ordering (George and Ng 1984).

| Processors | Rectangular grid sizes | | | Cubic grid size |
|---|---|---|---|---|
| 1 | 96 | 24 | 12 | 29 |
| 2 | 100 | 20 | 20 | 33 |
| 4 | 120 | 30 | 15 | 36 |
| 8 | 136 | 32 | 16 | 41 |
| 16 | 152 | 38 | 19 | 46 |
| 32 | 168 | 42 | 21 | 51 |
| 48 | 172 | 44 | 22 | 55 |
| 64 | 184 | 46 | 23 | 57 |
| 128 | 208 | 52 | 26 | 64 |
| 256 | 224 | 56 | 28 | 72 |
| 512 | 248 | 62 | 31 | 80 |

Table 7.1: 3D grid problems.

In Tables 7.2 and 7.3, we show the distribution of work for type 1 masters (T1), type 2 masters (T2M) and slaves (T2S), and the type 3 root node (T3). It can be seen that, when increasing the problem size and the number of processors used, the work of the type 2 slaves becomes a major part of the overall work. Thus, improving the mapping of the type 2 slaves through the candidate concept will have a great influence on the overall performance of the factorization, in particular on larger problems.

| | *LU* | | | | $LDL^T$ | | | |
|---|---|---|---|---|---|---|---|---|
| Processors | T1 | T2M | T2S | T3 | T1 | T2M | T2S | T3 |
| 1 | 100 | 0 | 0 | 0 | 100 | 0 | 0 | 0 |
| 2 | 85 | 0 | 0 | 15 | 85 | 0 | 0 | 15 |
| 4 | 45 | 7 | 34 | 14 | 45 | 2 | 39 | 14 |
| 8 | 28 | 7 | 49 | 14 | 28 | 2 | 56 | 14 |
| 16 | 18 | 5 | 63 | 14 | 18 | 2 | 65 | 15 |
| 32 | 7 | 4 | 75 | 14 | 8 | 1 | 77 | 14 |
| 48 | 7 | 4 | 75 | 15 | 8 | 1 | 77 | 14 |
| 64 | 5 | 3 | 78 | 14 | 5 | 1 | 81 | 13 |

Table 7.2: Percentage distribution of work for 3D cubic grid problems (nested dissection ordering).

| | $LU$ | | | | $LDL^T$ | | | |
|---|---|---|---|---|---|---|---|---|
| Processors | T1 | T2M | T2S | T3 | T1 | T2M | T2S | T3 |
| 1 | 100 | 0 | 0 | 0 | 100 | 0 | 0 | 0 |
| 2 | 88 | 0 | 0 | 12 | 88 | 0 | 0 | 12 |
| 4 | 84 | 1 | 3 | 12 | 84 | 1 | 3 | 12 |
| 8 | 49 | 5 | 34 | 12 | 49 | 3 | 36 | 12 |
| 16 | 25 | 5 | 58 | 12 | 25 | 2 | 61 | 12 |
| 32 | 16 | 4 | 68 | 12 | 16 | 2 | 70 | 12 |
| 48 | 14 | 4 | 70 | 12 | 12 | 2 | 74 | 12 |
| 64 | 10 | 4 | 74 | 12 | 10 | 1 | 77 | 12 |

Table 7.3: Percentage distribution of work for 3D rectangular grid problems (nested dissection ordering).

## 7.2 General symmetric and unsymmetric matrices

The matrices described in this section all arise from industrial applications and include test matrices from the PARASOL Project (PARASOL 2002), the Rutherford-Boeing Collection (Duff, Grimes and Lewis 1997), and the University of Florida sparse matrix collection (Davis 2002).

| Matrix name | Matrix type | Matrix order | Number entries | Origin |
|---|---|---|---|---|
| `bbmat` | symmetric | 38744 | 1771722 | Rutherford-Boeing |
| `ecl32` | symmetric | 51993 | 380415 | Rutherford-Boeing |
| `g7jac200` | symmetric | 59310 | 837936 | University of Florida |
| `twotone` | symmetric | 120750 | 1224224 | Rutherford-Boeing |
| `ship003` | unsymmetric | 121728 | 8086034 | PARASOL |
| `bmwcra_1` | unsymmetric | 148770 | 10644002 | PARASOL |

Table 7.4: Matrix order, type, and number of entries for the irregular test matrices.

In Table 7.4, we describe the characteristics of the test matrices arising from real life problems. In Table 7.5, we show for the irregular problems on 64 processors the distribution of work for type 1 masters (T1), type 2 masters (T2M) and slaves (T2S), and the type 3 root node (T3). We see that the work distribution depends heavily on the ordering used. The AMF ordering produces assembly trees that are rich in type 2 parallelism; on the other hand, the root nodes are so small that type 3 parallelism cannot be exploited effectively, in contrast to METIS. For all matrices apart from `bbmat`, the major part of the work is associated with the factorization of type 2 nodes, similar to the regular grid problems.

| | AMF | | | | METIS | | | |
|---|---|---|---|---|---|---|---|---|
| Matrix | T1 | T2M | T2S | T3 | T1 | T2M | T2S | T3 |
| bbmat | 43 | 3 | 54 | 0 | 57 | 7 | 30 | 6 |
| ecl32 | 14 | 8 | 78 | 0 | 29 | 8 | 52 | 11 |
| g7jac200 | 9 | 2 | 89 | 0 | 12 | 5 | 71 | 12 |
| twotone | 6 | 6 | 90 | 0 | 7 | 8 | 79 | 6 |
| ship003 | 7 | 7 | 85 | 1 | 14 | 10 | 65 | 11 |
| bmwcra_1 | 22 | 8 | 70 | 0 | 36 | 12 | 51 | 1 |

Table 7.5: Percentage distribution of work for irregular problems on 64 processors with different orderings.

# 8 Experimental investigation of algorithmic details

In this section, we study the influence and scope of parameters in the algorithms used by Version 4.1 and by the new version of MUMPS. Furthermore, we present a detailed investigation of isolated parts of the improved algorithm by typical examples of phenomena that we have observed in our experiments.

## 8.1 The impact of $k_{max}$ on volume of communication and memory

We first show the impact on the volume of communication and memory of the parameter $k_{\max}$ that controls the minimum granularity of the type 2 parallelism.

Our test matrix is from Section 7.1 and comes from an 11-point discretization of the Laplacian on a cubic grid of order 46, ordered by nested dissection. Here, we perform an *LU* factorization on an SGI Origin 2000 with 16 processors. This platform is well suited for testing the $k_{\max}$ parameter over a wide range of values because of its shared-memory architecture where a large amount of memory is available to all processors.

At first, we study the behaviour of Version 4.1 of MUMPS and then compare it with the new code.

The two graphs in the upper row of Figure 8.1 illustrate that with increasing $k_{\max}$, both the total volume of communication and the number of messages associated with dynamic scheduling decrease. If $k_{\max}$ is small, the required minimum number of slaves for a type 2 node and the corresponding communication volume will be large. With increasing $k_{\max}$, a single type 2 slave might be authorized to work on larger parts of a contribution block and the minimum number of slaves required during factorization becomes smaller. With $k_{\max}$ sufficiently large and thus the guaranteed minimum number of slaves from inequality (6.2) being no longer a constraint, the dynamic scheduling can freely choose slaves among the least loaded processors. Thus, further increases in $k_{\max}$ do not further reduce the volume of communication.

The graph in the left lower corner of Figure 8.1 shows the increase in estimated and actually used memory with increasing $k_{\max}$, and the graph in the right lower corner shows the decomposition of the estimated memory into the space reserved for the communication buffers, the *LU* factors, and the stack. As potentially every processor can be selected as a

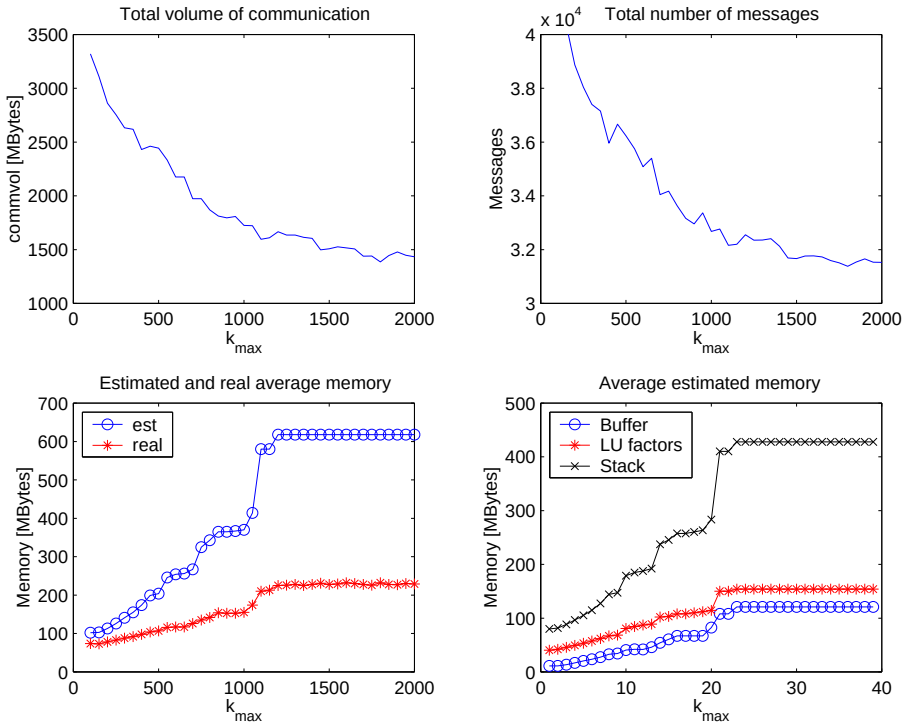


Figure 8.1: Impact of $k_{\max}$ on volume of communication and memory in Version 4.1 of MUMPS (Origin 2000, 16 processors).

slave during the factorization and the memory predicted depends monotonically on $k_{\max}$, the prediction during the analysis phase will lead to an increasing gap between real and estimated memory as can be seen in the graph on the lower left. On the lower right, we see that the main contribution to the overestimation of the memory is the stack. As slaves stack their part of the contribution block until it can be received by the processors working on the parent of the node, the stack has to grow when $k_{\max}$ increases. Furthermore, a single type 2 slave is authorized to work on larger parts of a contribution block.

When weighing memory estimation and communication volume against each other, the best value for $k_{\max}$ is so that it reduces the memory overestimation but at the same time limits the communication volume sufficiently.

We now investigate the behaviour of the new candidate-based code on the same test matrix. Candidates are assigned without relaxation and layerwise redistribution, following the proportional mapping of the assembly tree. From the two graphs in the bottom row of Figure 8.2 we observe the expected better estimation of memory. Compared to the corresponding graphs in Figure 8.1, the growth of the gap between estimated and real memory is significantly smaller. As the type 2 slaves can only be chosen from the candidates, the non-candidates can be excluded thus making the estimation tighter and more realistic. Furthermore, the two graphs in the top row of Figure 8.2 indicate that the communication volume in the new version of MUMPS drops faster with increasing $k_{\max}$ than it does for the previous version. This can be explained by the restricted freedom

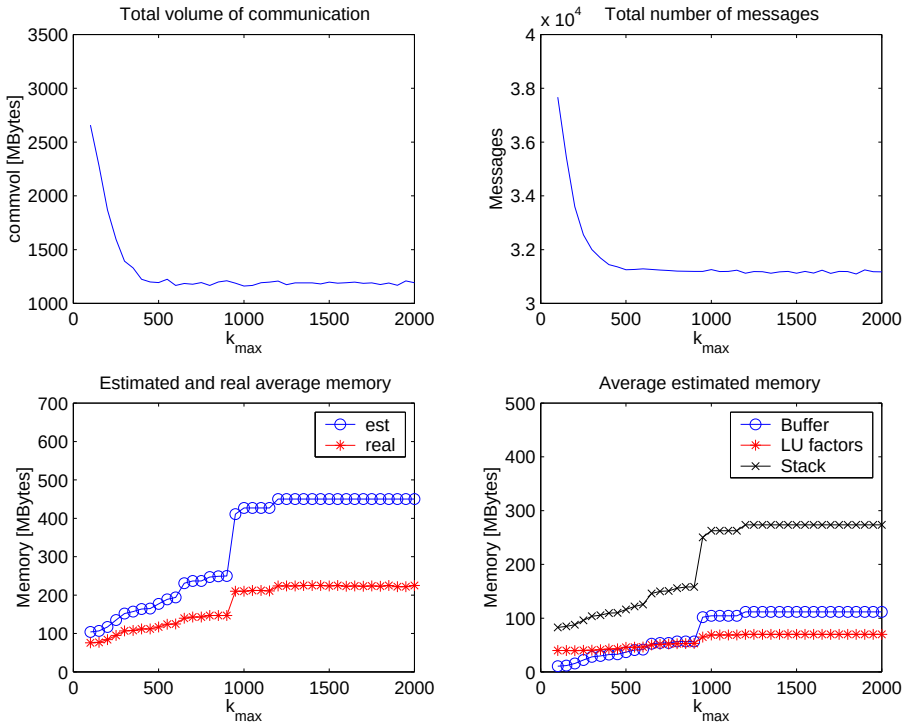


Figure 8.2: Impact of $k_{\max}$ on volume of communication and memory in the new version of MUMPS (Origin 2000, 16 processors).

for the dynamic scheduling, so that actually less parallelism is created and fewer slaves are chosen during factorization. Thus, if we want to reduce the communication volume in the new code, we are not obliged to increase $k_{\max}$ substantially with the consequent drawback of overestimating the memory. Instead, we can choose $k_{\max}$ relatively small and have the benefits of a relatively realistic memory estimation together with a reduced communication volume.

## 8.2 The impact of $k_{max}$ on performance

In the following, we show the impact of the parameter $k_{\max}$ on the factorization time, using as a test matrix an 11-point discretization of the Laplacian on a cubic grid of order 51, ordered by nested dissection. We perform an *LU* factorization on a CRAY T3E with 64 processors. (We have reduced the problem size from that in Table 7.1 so that we have enough flexibility with respect to memory for this parameter study.) Furthermore, because of limited memory and in order to separate the different algorithmic parameters, we use a candidate assignment without relaxation. For a study of the influence of relaxation we refer to Section 8.3.

The CRAY T3E is well suited for providing reliable timing for performance measures because the processors are guaranteed to run in dedicated mode for a single task. On the other hand, the T3E has a distributed-memory architecture with a fairly small amount of

memory per processor, so that we can vary the parameter $k_{\max}$ only in a relatively small range compared to the range possible on the Origin 2000 which has a shared memory.

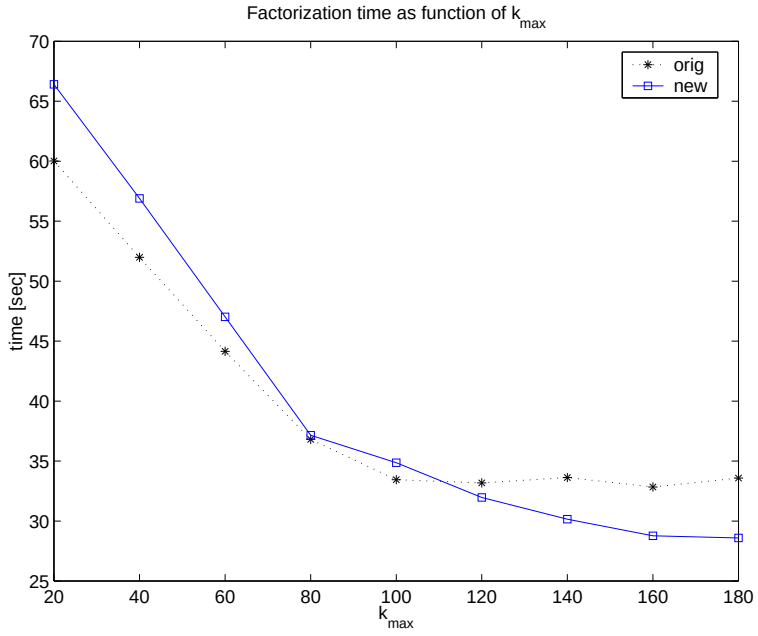


Figure 8.3: Impact of $k_{\max}$ on the performance of the original and the new version of the *LU* factorization time (CRAY T3E, 64 processors).

From Figure 8.3, we see that with increasing $k_{\max}$, the factorization time decreases in both versions of the code as the minimum number of slaves required during factorization gets smaller and the dynamic scheduler is free to decrease unnecessary parallelism. However, the previous version of MUMPS needs much more memory than the candidate-based version, and thus the flexibility for increasing $k_{\max}$ is more strictly limited.

Once $k_{\max}$ is sufficiently large, a further increase in $k_{\max}$ shows no further improvements in performance. This corresponds to the results on the limited reduction in the volume of communication obtained in Section 8.1. We note that for the larger values of $k_{\max}$, the new version of the code performs better. We will confirm this observation by systematic studies on our set of test matrices in Section 9.

## 8.3 Modifying the freedom offered to dynamic scheduling

We now investigate the behaviour of the new code when modifying the assignment of candidates. We study two different approaches. As described in Section 6.3, we can increase the number of candidates given to a node by increasing its number of preferentials through relaxing the proportional mapping. Furthermore, according to Algorithm 6, we can modify the candidate assignment for a given layer by an optional redistribution of the candidates that takes account of the weight of the nodes relative to each other.

We first compare the performance of the candidate assignment with and without

layerwise redistribution. Afterwards, we show the impact of relaxation on the two assignment strategies.

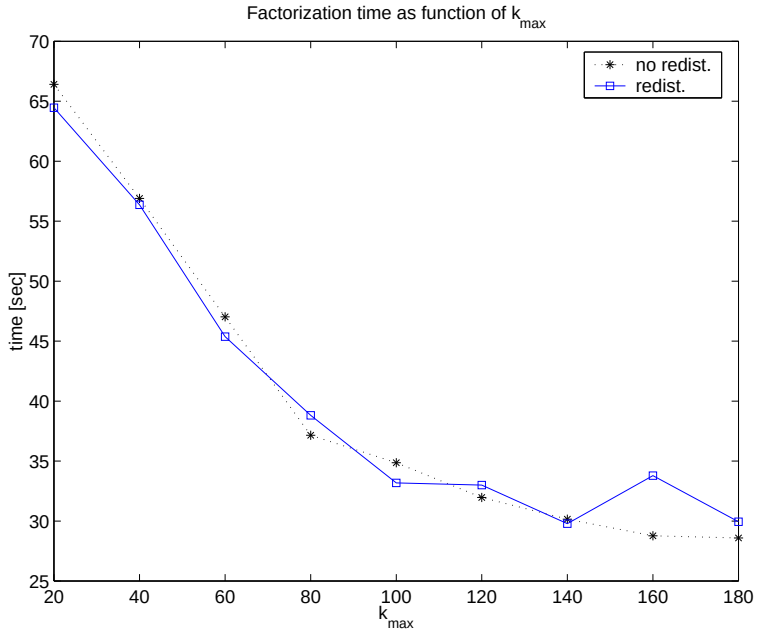


Figure 8.4: Comparison of the candidate assignment with (solid) and without (dotted) layer-wise candidate redistribution when increasing minimum granularity (LU factorization time on CRAY T3E, 64 processors, no candidate relaxation).

For our study, we use the same test case as in Section 8.2. Figure 8.4 shows the factorization time of the new version of MUMPS for the candidate assignment with and without layer-wise candidate redistribution as a function of the minimum granularity. We cannot find significant differences in the behaviour of the two approaches. This example is representative of the results we have obtained on the complete set of test problems.

We now investigate the impact of relaxation on the volume of communication, memory, and performance. In Figures 8.5 and 8.6, the horizontal axis denotes the percentage relaxation factor. We present the behaviour of the new version of MUMPS for the candidate assignment with and without layer-wise candidate redistribution as a function of the relaxation.

The two graphs in Figure 8.5 illustrate that, with increasing relaxation, both the total volume of communication and the number of messages related to dynamic scheduling increase because the flexibility for choosing the slaves during factorization becomes greater. Likewise, the memory estimation grows with increasing relaxation, see Figure 8.5. However, we do not observe a positive impact of relaxation on the performance of the algorithm; a possible interpretation is that, through the relaxation, we create additional parallelism that is not actually needed at run time. In general, we already have, without relaxation, enough freedom for a dynamic choice of the slaves. While this observation holds for all the experiments we have conducted, we are convinced that relaxation might show a positive impact on irregular problems from real-life applications. Unfortunately,

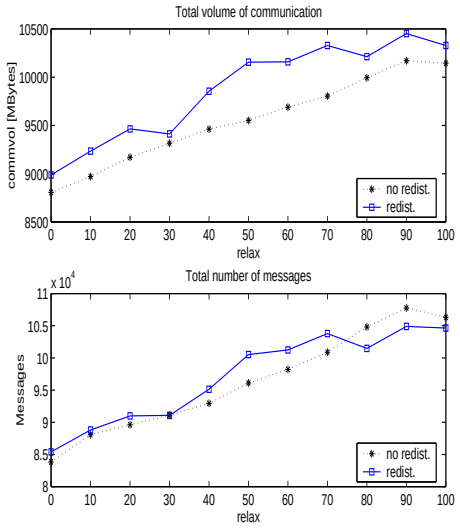


Figure 8.5: Amount of communication in original and modified candidate assignment when increasing the relaxation (CRAY T3E, 64 processors).

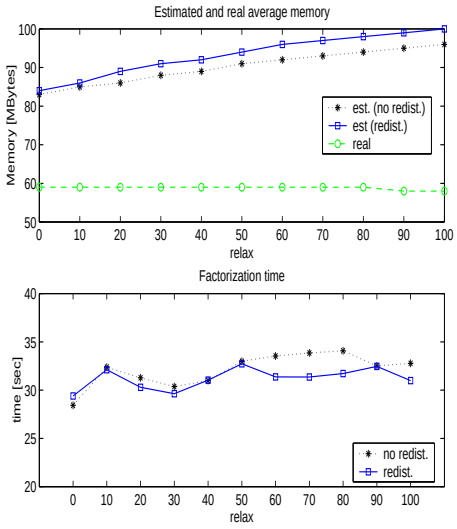


Figure 8.6: Memory estimation and performance of original and modified candidate assignment when increasing the relaxation (CRAY T3E, 64 processors).

the irregular problems available to us are not large enough to effectively exploit parallelism on a large number of processors, and we plan to investigate this further in the future, see the remarks in Section 10.

In conclusion, we note that the candidate approach without layer-wise redistribution and without additional relaxation already offers good results in our experiments. In the following, we focus therefore on the presentation of the results obtained with this algorithmic configuration.

## 8.4 Improved node splitting

Node splitting is useful when the pivot block is large relative to the size of the frontal matrix. It is discussed in detail by Amestoy et al. (2001*a*) and was introduced at the end of Section 3.1 and discussed briefly in Section 5.1. We now illustrate the additional capabilities of the new code for node splitting on the set of test matrices from Section 7.1. Those matrices are obtained from an 11-point discretization of the Laplacian on 3D cubic or rectangular grids with Approximate Minimum Fill (AMF) ordering and are described in Table 7.1. The AMF ordering produces long and thin trees from the regular grid problems which we can use to illustrate the problems of the splitting criterion used in the previous version of the code. Splitting was done in a preprocessing step and before the layer structure of the assembly tree was known. In order to prevent useless splitting below layer $L_0$ where no type 2 parallelism is exploited, the algorithm authorized node splitting only up to a fixed distance from the root node, where this distance depended only on the number of processors but not on the matrix. So it could happen that even

though there were nodes in the upper part of the tree that should have been split for better performance, the splitting was not performed.

| | Cubic grids (AMF) | | | Rectangular grids (AMF) | | |
|---|---|---|---|---|---|---|
| | Number of splittings | | | Number of splittings | | |
| | added by | total in | Nodes in | Added by | Total in | Nodes in |
| Processors | new code | new code | upper tree | new code | new code | upper tree |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 2 | 0 | 0 | 13 | 0 | 0 | 118 |
| 4 | 0 | 0 | 14 | 0 | 0 | 285 |
| 8 | 2 | 5 | 31 | 0 | 0 | 246 |
| 16 | 2 | 7 | 41 | 0 | 2 | 188 |
| 32 | 7 | 21 | 96 | 0 | 2 | 175 |
| 48 | 8 | 37 | 120 | 3 | 10 | 136 |
| 64 | 14 | 32 | 140 | 1 | 4 | 194 |
| 128 | 2 | 13 | 192 | 0 | 7 | 196 |
| 256 | 2 | 33 | 348 | 50 | 80 | 414 |
| 512 | Not enough memory in analysis | | | 61 | 107 | 830 |

Table 8.1: Improved splitting of the new code.

The new algorithm incorporates the splitting systematically in the upper part of the tree. Once layer $L_0$ is known, we can authorize splitting everywhere in the upper part of the tree to create more parallelism if this is useful. We illustrate the additional splitting in Table 8.1 for both cubic and rectangular grids. For each grid type, we show in the first of the three columns the additional number of splittings of the new code and compare them to the total number of splittings (including the splittings already performed by Version 4.1 of the code) and the total number of nodes in the upper part of the tree after splitting in the second and third columns, respectively. For example, for the rectangular grid on 512 processors, and with the same splitting criteria, the new code performs 61 splittings in addition to those already done by the old code, so that altogether 107 splittings are performed, resulting in an assembly tree with 830 nodes. In other words, in this example, the previous version of MUMPS missed 57% of the possible splittings.

| | Nmb | Operations | | Mem. est. | | Mem. real | | Facto. |
|---|---|---|---|---|---|---|---|---|
| Algorithm | type 2 | elim. | assem. | max | avg | max | avg | time |
| no splitting | 126 | 7.98e+11 | 1.10e+09 | 196 | 143 | 141 | 92 | 182 |
| with splitting | 140 | 7.98e+11 | 1.30e+09 | 167 | 150 | 119 | 96 | 145 |

Table 8.2: Comparison of the candidate-based $LU$ factorization with and without improved node splitting, cubic grid of order 57 on CRAY T3E with 64 processors (AMF).

We illustrate, in Table 8.2, the properties and benefits of the improved splitting in the case of the cubic grid of order 57 on 64 processors. The additional splitting slightly increases the number of assembly operations and also the average amount of memory

per processor. However, it creates additional parallelism by augmenting the number of type 2 nodes. This significantly improves the performance of the factorization. Moreover, memory can be balanced better between the processors because of the additional type 2 parallelism.

## 8.5 Improved node amalgamation

Amalgamation, particularly amalgamation that potentially increases fill-in, can be considered the opposite of splitting and can be useful when the pivot block is relatively small. In this section, we illustrate the improvements we have made concerning node amalgamation by using our test examples from Table 7.1 with a nested dissection ordering.

| | Cubic grids (ND) | | | Rectangular grids (ND) | | |
|---|---|---|---|---|---|---|
| | extra amalg | | total | extra amalg | | total |
| Processors | $L_0$ | total | nodes | $L_0$ | total | nodes |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 2 | 0 | 0 | 1 | 0 | 0 | 1 |
| 4 | 2 | 2 | 5 | 0 | 0 | 3 |
| 8 | 2 | 3 | 18 | 0 | 0 | 17 |
| 16 | 1 | 4 | 43 | 0 | 1 | 53 |
| 32 | 7 | 12 | 121 | 2 | 2 | 90 |
| 48 | 13 | 18 | 126 | 0 | 1 | 104 |
| 64 | 9 | 14 | 157 | 3 | 5 | 156 |
| 128 | 0 | 7 | 250 | 2 | 2 | 271 |
| 256 | 36 | 54 | 528 | 2 | 7 | 525 |
| 512 | 77 | 119 | 1371 | 66 | 93 | 1326 |

Table 8.3: Improved amalgamation of the new code.

In Table 8.3, we show, for both cubic and rectangular grids, the number of extra amalgamations the new code performed in layer $L_0$ of the Geist-Ng algorithm as described in Section 6.2 and the total number of extra amalgamations performed on all layers of the tree. We also give the total number of nodes in the upper part of the tree after all amalgamations have been performed.

We emphasize that, in the new version, we use the same amalgamation criteria as in the previous version and show, in the table, the amalgamations that are performed *in addition* to those performed before. Amalgamation in the previous version was only possible between a parent node and its oldest child; the greater freedom in the new code allows many more amalgamations as can be seen in particular for the large test matrices on 512 processors. For example, for the cubic grid on 512 processors, the new code performed 119 additional amalgamations, that is, 119 amalgamations more than the old code with the same amalgamation criteria. Among the additional amalgamations of the new code are 77 for layer $L_0$, so that the amalgamated assembly tree has 1371 nodes in the upper part.

We illustrate in Table 8.4 the properties and benefits of the improved amalgamation

in the case of the cubic grid of order 46 on 16 processors. The additional amalgamation decreases the number of assembly operations and allows a better memory balance because the stacking of several large type 1 nodes can be avoided.

| Type of | Operations | | Mem | | Mem real | | Fact. |
|---|---|---|---|---|---|---|---|
| amalgamation | assem. | elim. | max | avg | max | avg | time |
| Old | 2.44e+08 | 5.91e+10 | 187 | 121 | 175 | 97 | 19.4 |
| New | 2.35e+08 | 5.91e+10 | 108 | 95 | 82 | 71 | 18.7 |

Table 8.4: Comparison of the candidate-based $LDL^T$ factorization with and without improved node amalgamation, cubic grid of order 46 on CRAY T3E with 17 processors.

## 8.6 Post-processing for a better memory balance

On the CRAY T3E, we illustrate a shortcoming that we detected when testing an initial version of the candidate-based $LU$ factorization. We consider the cubic grid problem of order 72 from Table 7.1 ordered by nested dissection. We show the benefits obtained by remapping the masters of type 2 nodes for better memory balance as described in Section 6.5 and conclude that the post-processing is also crucial for obtaining good speedup.

Looking at the first rows (no postp.) of Table 8.5, we see that the flop-based equilibration of the scheduling algorithm leads to severe memory imbalance both in the estimated and the actual memory. In particular, the process needing the largest amount of (estimated) memory requests 179 megabytes, about 70% of the memory of the processor. For performance reasons, it is necessary to increase $k_{\max}$, see Section 8.2. However, this is impossible because of the strong memory imbalance, as augmenting $k_{\max}$ increases the memory estimations of the analysis phase considerably.

| Algorithm | $k_{\max}$ | Max est | Avg est | Max real | Avg real | Fact. time |
|---|---|---|---|---|---|---|
| no postp. | 80 | 179 | 117 | 172 | 102 | 165 |
| | 160 | Not enough memory | | | | |
| w. postp. | 80 | 136 | 117 | 123 | 102 | 152 |
| | 160 | 193 | 164 | 162 | 132 | 124 |

Table 8.5: Memory (in MBytes) and factorization time (in seconds) of the candidate-based $LU$ factorization with and without post-processing, cubic grid of order 72 with nested dissection.

In the last two rows of Table 8.5, we show the memory statistics when the post-processing is performed. We observe that the difference between average and maximum values for both the estimated and actual memory are much reduced. This allows us to double the $k_{\max}$ parameter for this test case and obtain better performance for the factorization. However, note that the estimate for the most loaded processor is

more accurate without post-processing. This is because the differences between memory estimation and actually used memory are mainly related to a processor being a type 2 candidate but not being chosen as a slave during factorization. Without post-processing, the major activity for the most loaded processor probably involves work associated with being master of several type 2 nodes, see Section 6.5. But after the post-processing, the processor exchanges its role as master with another and becomes a type 2 candidate so that its memory estimate can become less accurate.

# 9 Performance analysis

In the following, we compare the performance of the new MUMPS code with the previous version (Amestoy et al. 2001*b*) on the complete set of test problems presented in Section 7. All these tests were performed on the CRAY T3E of the NERSC computing center at Lawrence Berkeley National Lab in Berkeley, California.

## 9.1 Nested dissection ordering

In this section, we use the test matrices from Table 7.1 ordered by nested dissection.

We observe, from the results in Table 9.1, that for up to 64 processors, the new version has similar performance to the good results of the previous version. However, when more processors are used and the matrices become larger, the new code performs significantly better. Looking at the results on 128, 256 and 512 processors, we note the greatly improved scalability of the candidate-based code.

| | Cubic grids (ND) | | | Rectangular grids (ND) | | |
|---|---|---|---|---|---|---|
| Processors | flops | old | new | | old | new |
| 1 | 7.2e+09 | 23.2 | 23.2 | 4.5e+09 | 16.6 | 16.6 |
| 2 | 1.6e+10 | 29.1 | 29.0 | 9.5e+09 | 17.2 | 17.1 |
| 4 | 2.7e+10 | 27.4 | 23.9 | 1.8e+10 | 16.6 | 16.9 |
| 8 | 6.0e+10 | 30.1 | 29.5 | 3.7e+10 | 20.6 | 19.2 |
| 16 | 1.2e+11 | 30.8 | 31.8 | 7.3e+10 | 22.4 | 23.3 |
| 32 | 2.3e+11 | 43.3 | 42.2 | 1.4e+11 | 25.7 | 27.4 |
| 48 | 3.6e+11 | 53.0 | 57.5 | 1.8e+11 | 26.0 | 23.9 |
| 64 | 4.5e+11 | 59.0 | 52.9 | 2.4e+11 | 31.2 | 30.2 |
| 128 | 8.9e+11 | 93.4 | 72.7 | 4.9e+11 | 44.9 | 38.5 |
| 256 | 1.8e+12 | 163.5 | 119.4 | 7.7e+11 | 75.4 | 47.1 |
| 512 | 3.4e+12 | 599.6 | 189.1 | 1.4e+12 | 135.5 | 73.7 |

Table 9.1: Performance of the old and new *LU* factorization (time in seconds on the CRAY T3E).

Another major advantage of the new candidate-based code is that it better estimates the memory used for the factorization. In Table 9.2, we show the memory space for the *LU* factors of the old and the new version of MUMPS. We see that the candidate-based code significantly reduces the overestimation of the storage required, and that the gains increase with the matrix size and the number of processors.

| Processors | Cubic grids (ND) space used | Estimate old | Estimate new | Rectangular grids (ND) space used | Estimate old | Estimate new |
|---|---|---|---|---|---|---|
| 1 | 11.4 | 11.4 | 11.4 | 10.2 | 10.2 | 10.2 |
| 2 | 19.7 | 19.7 | 19.7 | 17.2 | 17.2 | 17.2 |
| 4 | 28.1 | 28.1 | 28.1 | 26.5 | 26.6 | 26.6 |
| 8 | 49.1 | 49.2 | 49.2 | 43.9 | 44.6 | 44.0 |
| 16 | 77.9 | 84.3 | 78.6 | 70.4 | 82.3 | 70.9 |
| 32 | 121.2 | 181.5 | 122.8 | 107.7 | 166.8 | 110.0 |
| 48 | 165.9 | 289.5 | 170.7 | 130.2 | 255.5 | 134.7 |
| 64 | 193.7 | 412.6 | 203.8 | 158.4 | 407.2 | 166.7 |
| 128 | 309.7 | 897.9 | 357.0 | 260.1 | 1108.0 | 296.4 |
| 256 | 504.4 | 2678.5 | 924.6 | 353.9 | 2420.5 | 478.0 |
| 512 | 780.4 | 4594.0 | 1369.7 | 541.6 | 5759.0 | 921.5 |

Table 9.2: Space for the $LU$ factors (number of reals $\times 10^6$).

The big gains of the new candidate-based code are a result of the individual improvements concerning splitting and amalgamation, reduced communication and the better locality of the computation as illustrated in Section 8. Furthermore, we need to decrease $k_{\max}$ in the large problems for the old version of MUMPS because of memory. This limits the performance as we saw in Section 8.2. On the other hand, we do not need to decrease $k_{\max}$ in the candidate-based code as the tighter estimates stay within the memory available.

As all regular test matrices are symmetric, we can also compare the old with the new candidate-based $LDL^T$ factorization. The results presented in Table 9.3 confirm those obtained for the $LU$ factorization. The candidate-based code shows a much better performance in particular for the large problems on a large number of processors due to improved locality of communication and computation, and because of the bigger scope for increasing the $k_{\max}$ parameter.

| Processors | Cubic grids (ND) flops | old | new | Rectangular grids (ND) flops | old | new |
|---|---|---|---|---|---|---|
| 1 | 3.6e+09 | 19.1 | 18.7 | 2.2e+09 | 13.5 | 13.1 |
| 2 | 8.0e+09 | 21.3 | 20.7 | 4.8e+09 | 13.1 | 12.9 |
| 4 | 1.3e+10 | 19.7 | 16.7 | 9.0e+09 | 11.5 | 12.4 |
| 8 | 3.0e+10 | 18.1 | 18.3 | 1.8e+10 | 15.2 | 12.9 |
| 16 | 5.9e+10 | 18.8 | 19.8 | 3.6e+10 | 13.8 | 13.2 |
| 32 | 1.1e+11 | 25.8 | 22.2 | 6.8e+10 | 15.5 | 15.3 |
| 48 | 1.8e+11 | 28.7 | 30.4 | 9.0e+10 | 14.2 | 14.8 |
| 64 | 2.2e+11 | 30.7 | 25.6 | 1.2e+11 | 17.6 | 16.8 |
| 128 | 4.4e+11 | 45.6 | 33.0 | 2.4e+11 | 33.5 | 20.3 |
| 256 | 9.1e+11 | 109.1 | 43.0 | 3.8e+11 | 45.2 | 18.4 |
| 512 | 1.7e+12 | 421.9 | 64.0 | 7.1e+11 | 195.5 | 24.3 |

Table 9.3: Performance of the $LDL^T$ factorization (time in seconds on the CRAY T3E).

Note that, because of the improvements in the scalability of the new code, MUMPS now compares favourably to SuperLU on a large number of processors. (The factorization time for SuperLU on 128 processors and the same nested dissection ordering is 71.1 seconds for the cubic and 56.1 seconds for the rectangular grid (Amestoy et al. 2001*b*).)

## 9.2 Approximate Minimum Fill (AMF) ordering

Recently a fairly large number of experiments have been conducted with several heuristics to reduce the fill-in (deficiency) during the elimination process (Ng and Raghavan 1999, Rothberg and Eisenstat 1998). The approximation of the deficiency used in our AMF code is based on the observation that, because of the approximate degree, we count variables twice that belong to the intersection of two elements adjacent to a variable in the current pivot list. This property of the approximate degree can be exploited to improve the estimation of the deficiency and the accuracy of the approximation proposed by Rothberg and Eisenstat (1998).

The AMF ordering produces trees that are difficult to exploit in MUMPS. The upper part of the tree where type 2 and type 3 parallelism can be exploited is usually a long and thin chain. In Table 9.4, we show the memory required to store the factors of the *LU* factorization for the different orderings. In the case of the cubic grid, the real space used by the factors is significantly larger than when using the nested dissection ordering. Furthermore, we note that for the rectangular grid, AMF actually needs the least space for the factors. However, the shape of the assembly tree still offers less potential for parallelism and we expect the factorization time for AMF-ordered matrices to be considerably longer than for the case of nested dissection. This is confirmed by the results in Tables 9.5 and 9.6.

| Grid | AMF Factors | ND Factors |
|---|---|---|
| Cubic | 247,804,999 | 193,785,687 |
| Rect | 148,102,032 | 158,402,018 |

Table 9.4: Number of entries in the factors by ordering for the *LU* factorization on 64 processors, grid sizes according to Table 7.1.

| Processors | Cubic grids (AMF) flops | old | new | Rectangular grids (AMF) flops | old | new |
|---|---|---|---|---|---|---|
| 1 | 8.6e+09 | 25.7 | 25.7 | 3.1e+09 | 13.4 | 13.7 |
| 2 | 2.1e+10 | 47.4 | 48.3 | 5.8e+09 | 21.7 | 22.1 |
| 4 | 3.8e+10 | 35.1 | 35.0 | 1.0e+10 | 22.5 | 23.7 |
| 8 | 1.0e+11 | 54.5 | 47.8 | 2.2e+10 | 27.8 | 27.6 |
| 16 | 1.9e+11 | 55.5 | 54.9 | 5.4e+10 | 34.8 | 32.6 |
| 32 | 3.8e+11 | 96.3 | 81.3 | 1.0e+11 | 50.7 | 49.8 |
| 48 | 4.8e+11 | 114.6 | 98.2 | 1.9e+11 | 71.0 | 67.4 |
| 64 | 8.0e+11 | 188.0 | 145.4 | 1.8e+11 | 46.4 | 43.3 |
| 128 | 1.7e+12 | 302.6 | 242.7 | 4.6e+11 | 118.9 | 114.6 |
| 256 | 4.1e+12 | 740.9 | 484.1 | 8.6e+11 | 262.5 | 208.6 |
| 512 | Not enough memory in analysis | | | 1.2e+12 | 325.7 | 264.7 |

Table 9.5: Performance of the $LU$ factorization (time in seconds on the CRAY T3E).

| Processors | Cubic grids (AMF) flops | old | new | Rectangular grids (AMF) flops | old | new |
|---|---|---|---|---|---|---|
| 1 | 4.3e+09 | 19.5 | 19.5 | 1.6e+09 | 11.3 | 11.3 |
| 2 | 1.1e+10 | 32.8 | 33.8 | 2.9e+09 | 18.6 | 18.7 |
| 4 | 1.9e+10 | 24.0 | 24.6 | 5.2e+09 | 19.5 | 19.9 |
| 8 | 5.1e+10 | 28.0 | 27.9 | 1.1e+10 | 15.0 | 14.9 |
| 16 | 9.5e+10 | 29.0 | 29.2 | 2.7e+10 | 15.4 | 16.7 |
| 32 | 1.9e+11 | 34.1 | 33.8 | 5.1e+10 | 20.1 | 20.9 |
| 48 | 2.4e+11 | 36.3 | 36.5 | 9.3e+10 | 24.6 | 25.0 |
| 64 | 4.0e+11 | 51.8 | 48.6 | 8.9e+10 | 23.6 | 23.7 |
| 128 | 8.4e+11 | 86.1 | 67.8 | 2.3e+11 | 38.6 | 34.5 |
| 256 | 2.1e+12 | 237.7 | 117.3 | 4.3e+11 | 74.6 | 67.7 |
| 512 | Not enough memory in analysis | | | 6.2e+11 | 196.1 | 73.0 |

Table 9.6: Performance of the $LDL^T$ factorization (time in seconds on the CRAY T3E).

## 9.3 Analysis of the speedup for regular grid problems

We now summarize the results of the previous sections by presenting a comparison of the speedup on the 3D grid problems.

Let $t_j$ denote the time to execute a given job involving $ops_j$ floating point operations on $j$ parallel processors. Then, we define the scaled *speedup*, $S_p$, for $p$ processors to be

$$S_p = \frac{t_1/ops_1}{t_p/ops_p}. \tag{9.1}$$

In Figures 9.1 and 9.2, we show the scaled speedup for the matrices ordered by nested dissection and in Figures 9.3 and 9.4 for the AMF ordering, respectively.

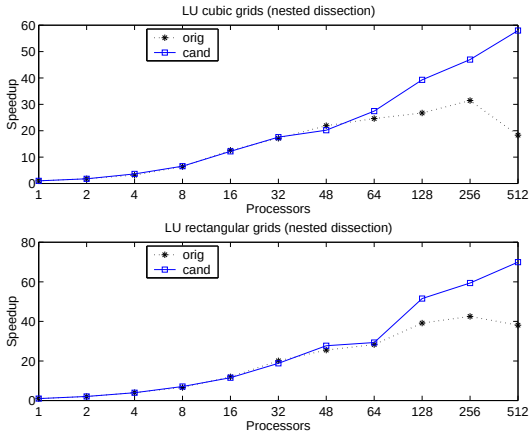


Figure 9.1: Comparison of the speedup of the $LU$ factorization for 3D grid problems ordered by nested dissection.

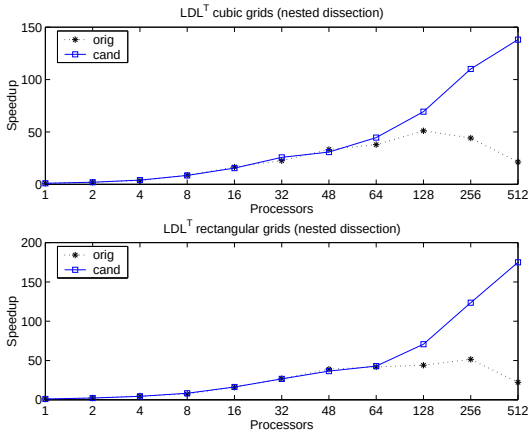


Figure 9.2: Comparison of the speedup of the $LDL^T$ factorization for 3D grid problems ordered by nested dissection.

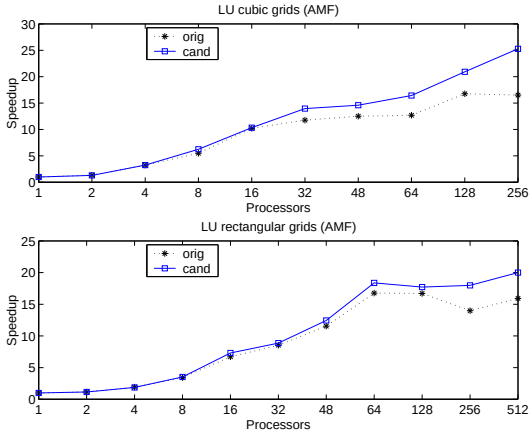


Figure 9.3: Comparison of the speedup of the $LU$ factorization for 3D grid problems ordered by AMF.

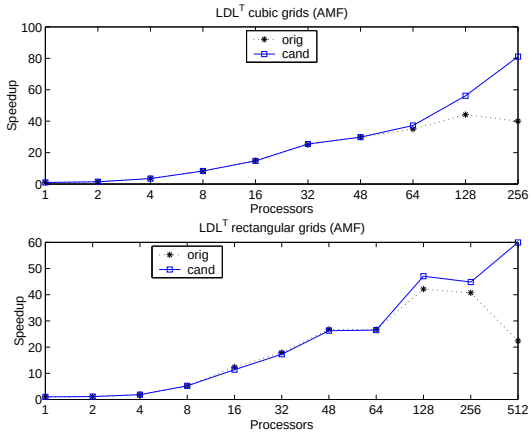


Figure 9.4: Comparison of the speedup of the $LDL^T$ factorization for 3D grid problems ordered by AMF.

## 9.4 Performance analysis on general symmetric and unsymmetric matrices

In this section, we compare the performance of the new mapping algorithm with the previous version on general symmetric and unsymmetric matrices. The main problem with this comparison is that our algorithm offers the biggest performance gains only on a large number of processors. However, the unsymmetric matrices available to us are either too small to offer enough potential for scalability on more than 64 processors, or they are too large to do the analysis (which is performed on only one processor). This was already observed in the analysis of the scalability of both MUMPS and SuperLU (Amestoy et al. 2001*b*) .

In order to compare the quality of the different orderings, we show the number of entries in the factors for the test matrices in Table 9.7.

While it is not always the best ordering, METIS consistently provides a good overall performance with respect to the number of entries in the factors.

| Matrix name | AMF | | METIS | |
|---|---|---|---|---|
| | Factors | Flops | Factors | Flops |
| bbmat | 37,734,384 | 2.8e+10 | 37,429,544 | 2.8e+10 |
| ecl32 | 31,862,069 | 3.5e+10 | 25,190,381 | 2.1e+10 |
| g7jac200 | 33,245,736 | 3.5e+10 | 43,496,678 | 5.5e+10 |
| twotone | 22,653,594 | 2.9e+10 | 25,537,506 | 2.9e+10 |
| ship003 | 68,199,143 | 9.6e+10 | 71,388,126 | 8.3e+10 |
| bmwcra_1 | 95,816,634 | 9.9e+10 | 78,012,686 | 6.1e+10 |

Table 9.7: Number of entries in the factors and number of operations during factorization by ordering ($LDL^T$ factorization for symmetric and $LU$ factorization for unsymmetric matrices).

| Matrix | Order | Alg | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|
| bbmat | AMF | old | 119.7 | 71.1 | 50.5 | 44.3 | 44.1 |
| | | new | 114.2 | 69.6 | 44.1 | 27.6 | 21.7 |
| | METIS | old | 39.5 | 24.2 | 14.5 | 11.8 | 9.6 |
| | | new | 37.7 | 22.2 | 14.1 | 10.8 | 8.8 |
| ecl32 | AMF | old | 45.2 | 25.7 | 19.9 | 16.6 | 16.0 |
| | | new | 44.4 | 24.2 | 19.0 | 16.0 | 14.5 |
| | METIS | old | 28.4 | 16.7 | 10.7 | 7.7 | 6.3 |
| | | new | 29.4 | 16.0 | 11.4 | 7.7 | 5.6 |
| g7jac200 | AMF | old | 166.0 | 77.3 | 63.4 | 40.2 | 41.8 |
| | | new | 171.6 | 78.3 | 61.3 | 38.6 | 33.7 |
| | METIS | old | - | 48.2 | 27.4 | 20.3 | 15.7 |
| | | new | - | 41.4 | 26.7 | 19.9 | 13.6 |
| twotone | AMF | old | 105.8 | 47.1 | 28.3 | 20.8 | 19.1 |
| | | new | 102.4 | 47.4 | 29.0 | 20.9 | 18.7 |
| | METIS | old | - | 26.9 | 19.1 | 13.3 | 11.4 |
| | | new | - | 27.9 | 17.7 | 11.9 | 11.2 |
| ship003 | AMF | old | - | 66.0 | 34.0 | 24.4 | 22.1 |
| | | new | - | 62.2 | 33.5 | 24.2 | 20.4 |
| | METIS | old | - | - | 29.2 | 18.2 | 12.3 |
| | | new | - | - | 28.4 | 18.0 | 12.0 |
| bmwcra_1 | AMF | old | - | - | 44.6 | 30.3 | 27.6 |
| | | new | - | - | 42.4 | 28.5 | 26.9 |
| | METIS | old | - | 36.6 | 20.1 | 13.5 | 8.5 |
| | | new | - | 35.7 | 20.9 | 13.2 | 8.4 |

Table 9.8: Performance of old and new code on the irregular test matrices (factorization time in seconds on the CRAY T3E).

In Table 9.8, we see that in general the new mapping algorithm performs similarly to the old one. As already noted, we would expect significant improvements on large matrices and on more than 64 processors. However, we notice some improvements for the AMF ordering on `bbmat` and `g7jac200`. However, since METIS generally provides better orderings, these improvements for AMF are not so relevant and only show the capacity of our algorithm to correctly handle irregular trees.

# 10 Perspectives and future work

In this section, we summarize the open questions that need further investigation.

In Section 8.3, we investigated the behaviour of the new code when modifying the assignment of candidates through relaxation and layer-wise redistribution. On the test cases that we have studied in the framework of this paper, these modifications have not shown a positive effect on the overall performance of the code. Still, there is an intuitive argument suggesting further experiments. The analysis phase tries to predict the actual factorization of the matrix and takes mapping decisions based on this symbolic factorization. However, there are cases where this approach might not be accurate enough; for example we do not take into account costs of communication between the processors as is done, for example, by the static scheduler of PaStiX (Henon et al. 2002). Since, during factorization, the assembly tree is treated from bottom up, we might expect mapping problems to have more severe influence towards the root of the tree. For this reason, we could decide to offer more freedom to dynamic scheduling near the root nodes so that unfortunate mapping decisions can be corrected dynamically there.

Furthermore, in Section 9.4 we have presented test results on a few large irregular test matrices from real life applications. We have already remarked that these matrices are still relatively small and do not offer enough sources of parallelism on a large number of processors. This study needs to be extended in order to be able to give reliable statements on the scalability of the new code also in real life applications.

Finally, our candidate based approach can be extended to take account of the system architecture, for example with respect to non-uniform communication costs on machines consisting of SMP nodes. We can modify the task scheduling so that processors which require expensive communications are penalized so that the master-slave communication costs are reduced.

# 11 Summary and conclusions

Previous studies of MUMPS, a distributed memory direct multifrontal solver for sparse linear systems, indicated that its scalability with respect to computation time and use of memory could be improved. In this paper, we have presented a new task scheduling algorithm designed to address these problems. It consists of an approach that treats the assembly tree layer by layer and integrates tree modifications, such as amalgamation and splitting, with the mapping decisions. As a major feature, we have introduced the concept of candidate processors that are determined during the analysis phase of the solver in order to guide the dynamic scheduling during the factorization.

We have illustrated key properties of the new algorithm by detailed case studies on selected problems. Afterwards, by comparison of the old with the new code on a large set of regular and irregular test problems, we have illustrated the main benefits of the new approach. These include improved scalability on a large number of processors, reduced memory demands and a smaller volume of communication, and the easier handling of parameters relevant for the performance of the algorithm. Finally, we have discussed possible extensions of our algorithm, in particular with respect to its use on SMP architectures.

### Acknowledgments

We are grateful to E. Ng for providing access to the CRAY T3E at NERSC. J. Koster and J. Y. L'Excellent gave helpful comments on an earlier version of this paper.

## References

P. R. Amestoy and I. S. Duff. Memory management issues in sparse multifrontal methods on multiprocessors. *Int. J. Supercomputer Appl.*, **7**, 64–82, 1993.

P. R. Amestoy, I. S. Duff, and J. Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computer Methods in Appl. Mech. Eng.*, pp. 501–520, 2000.

P. R. Amestoy, I. S. Duff, J. Y. L'Excellent, and J. Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal. Appl.*, **23(1)**, 15–41, 2001*a*.

P. R. Amestoy, I. S. Duff, J. Y. L'Excellent, and X. S. Li. Analysis, Tuning and Comparison of Two General Sparse Solvers for Distributed Memory Computers. *ACM Trans. Math. Software*, **27(4)**, 388–421, 2001*b*.

C. Ashcraft and R. G. Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Trans. Math. Software*, **15**, 291–309, 1989.

J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. *Computer Physics Communications*, **97**, 1–15, 1996. (also LAPACK Working Note #95).

T. Davis. University of Florida sparse matrix collection. URL: http://www.cise.ufl.edu/research/sparse/matrices/, 2002.

J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers.* SIAM Press, Philadelphia, 1998.

I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Software*, **9**, 302–325, 1983.

I. S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear systems. *SIAM J. Sci. Stat. Comput.*, **5**, 633–641, 1984.

I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, London, 1986.

I. S. Duff, R. G. Grimes, and J. G. Lewis. The Rutherford-Boeing Sparse Matrix Collection. Technical Report RAL-TR-97-031, Atlas Centre, Rutherford Appleton Laboratory, 1997. Also Technical Report ISSTECH-97-017 from Boeing Information & Support Services and Report TR/PA/97/36 from CERFACS, Toulouse.

A. Geist and E. Ng. Task scheduling for parallel sparse Cholesky factorization. *Int J. Parallel Programming*, **18**, 291–314, 1989.

A. George and E. Ng. SPARSPAK: Waterloo sparse matrix package user's guide for SPARSPAK-B. Research Report CS-84-37, Dept. of Computer Science, University of Waterloo, 1984.

J. A. George, J. W. H. Liu, and E. G.-Y. Ng. Communication results for parallel sparse Cholesky factorization on a hypercube. *Parallel Computing*, **10**, 287–298, 1989.

A. Gupta. Recent advances in direct methods for solving unsymmetric sparse systems of linear equations. *ACM Trans. Math. Software*, **28**(3), 301–324, 2002.

P. Henon, P. Ramet, and J. Roman. PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, **28(2)**, 301–321, 2002.

D. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*, chapter Approximation algorithms for scheduling, pp. 1–45. PWS Publishing, Boston, 1996.

G. Karypis and V. Kumar. *MeTis - A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices - Version 4.0.* University of Minnesota, 1998.

J. W. H. Liu. The Role of Elimination Trees in Sparse Factorization. *SIAM J. Matrix Anal. Appl.*, **11**, 134–172, 1990.

J. W. H. Liu. The multifrontal method for sparse matrix solution: Theory and Practice. *SIAM Review*, **34**, 82–109, 1992.

J. W. H. Liu, E. G. Ng, and W. Peyton. On finding supernodes for sparse matrix computations. *SIAM J. Matrix Anal. Appl.*, **14**, 242–252, 1993.

E. Ng and P. Raghavan. Performance of greedy heuristics for sparse Cholesky factorization. *SIAM J. Matrix Anal. Appl.*, **20**, 902–914, 1999.

PARASOL. PARASOL test data. URL: http://www.parallab.uib.no/parasol/data.html, 2002.

A. Pothen and C. Sun. A Mapping Algorithm for Parallel Sparse Cholesky Factorization. *SIAM J. Sci. Comput.*, **14(5)**, 1253–1257, 1993.

E. Rothberg and S. C. Eisenstat. Node selection strategies for bottom-up sparse matrix ordering. *SIAM J. Matrix Anal. Appl.*, **19**(3), 682–695, 1998.