RAL-TR-1998-051

Multifrontal Parallel Distributed Symmetric and Unsymmetric Solvers[1]

P. R. Amestoy[2], I. S. Duff[3] and J.-Y. L'Excellent[4]

ABSTRACT

We consider the solution of both symmetric and unsymmetric systems of sparse linear equations. A new parallel distributed memory multifrontal approach is described. To handle numerical pivoting efficiently, a parallel asynchronous algorithm with dynamic scheduling of the computing tasks has been developed. We discuss some of the main algorithmic choices and compare both implementation issues and the performance of the LDL^T and LU factorizations. Performance analysis on an IBM SP2 shows the efficiency and the potential of the method. The test problems used are from the Rutherford-Boeing collection and from the PARASOL end users.

Keywords: MPI, distributed memory architecture, sparse matrices, multifrontal direct methods.

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

---

[1] This work has been partially supported by the PARASOL project (EU ESPRIT IV LTR project 20160).

Current reports available by anonymous ftp from matisa.cc.rl.ac.uk in the directory "pub/reports". This report is in file amdeRAL98051.ps.gz. Also published as Technical Report TR/PA/98/22 from CERFACS.

[2] amestoy@enseeiht.fr, ENSEEIHT-IRIT, 2 rue Camichel, Toulouse, France.

[3] I.Duff@rl.ac.uk, Rutherford Appleton Laboratory, Chilton, Didcot, Oxon, 0X11 0QX England, and CERFACS, Toulouse, France.

[4] excelle@cerfacs.fr, CERFACS, Toulouse, France.

Department for Computation and Information
Atlas Centre
Rutherford Appleton Laboratory
Oxon OX11 0QX

July 27, 1998.

Contents

1 Introduction

This work has been performed as a Work Package within the PARASOL Project. PARASOL is an ESPRIT IV Long Term Research Project (No 20160) for "An Integrated Environment for Parallel Sparse Matrix Solvers". The main goal of this Project, which started on January 1 1996, is to build and test a portable library for solving large sparse systems of equations on distributed memory systems. There are twelve partners in five countries, five of whom are code developers and five end users. The software is written in Fortran 90 and uses MPI for message passing. There are routines for both direct and iterative solution of symmetric and unsymmetric systems. The final library will be in the public domain.

The PARASOL Consortium is managed by PALLAS in Germany and consists of

- leading European research organizations with internationally recognized experience and an established track record in the development of parallel solvers (CERFACS, GMD-SCAI, ONERA, Rutherford Appleton Laboratory (RAL), University of Bergen);
- industrial code developers who define the requirements for PARASOL, are providing test cases generated by their finite-element packages, and will use the developed software in production mode (Apex Technologies, Det Norske Veritas (DNV), INPRO, MacNeal-Schwendler (MSC), Polyflow);
- two leading European HPC software companies who will exploit the project results and are providing state-of-the-art programming development tools (GENIAS, PALLAS).

For more information, see the project web site at `http://www.genias.de/parasol`.

CERFACS and RAL with the collaboration of ENSEEIHT-IRIT are developing the direct solver based on a multifrontal approach originally developed by Duff and Reid (1983,1984) and extended to shared memory computers by Duff (1986) and Amestoy and Duff (1989,1993) and subsequently to a prototype version using PVM by Espirat (1996). The integration of this direct code into the PARASOL Library and comments on the performance of earlier versions of the code can be found in Amestoy, Duff, L'Excellent and Plecháč (1998).

We discuss some important aspects of multifrontal methods in Section 2 and describe the main implementation issues for distributed memory machines in Section 3. We consider a performance analysis of the algorithm and show the results of some numerical experiments with the code in Section 4 before presenting some concluding remarks and pointers to future work in Section 5.

Throughout this paper we will show the performance of our algorithms on a set of test problems. These test problems consist of symmetric and unsymmetric problems from the Harwell-Boeing collection (Duff, Grimes and Lewis 1992), the

forthcoming Rutherford-Boeing Sparse Matrix Collection (Duff, Grimes and Lewis 1997). and problems from the PARASOL end users and are shown in Table 1.1.

Problem name	Type	Order	Nonzeros	Origin
GOODWIN	UNS	7320	324784	Rutherford-Boeing
BCSSTK15	SYM	3948	60882	Harwell-Boeing
WANG3	UNS	26064	177168	Rutherford-Boeing
INV-EXTRUSION-1	UNS	30412	1793881	PARASOL
MIXING-TANK	UNS	29957	1995041	PARASOL
B5TUER	SYM	162610	4036144	PARASOL
BMW7ST_1	SYM	141347	7339667	PARASOL
CRANKSEG_1	SYM	52804	10614210	PARASOL
CRANKSEG_2	SYM	63838	7106348	PARASOL
OILPAN	SYM	73752	1835470	PARASOL
QUER	SYM	59122	1462811	PARASOL

Table 1.1: Description of test problems.

2 Multifrontal methods

The multifrontal method for the solution of sparse linear equations is a direct method based on the LU factorization of the matrix. We refer the reader to our earlier papers (Amestoy and Duff 1989, Duff and Reid 1983, Duff and Reid 1984) for full details of this technique. In the following, we will consider multifrontal methods that solve the assembled system

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

both when $\mathbf{A}$ is symmetric and when it is unsymmetric.

In both cases, the structure of the matrix is first *analysed* to determine an ordering that, in the absence of any numerical pivoting, will preserve sparsity in the factors. An approximate minimum degree ordering strategy is used on the symmetrized pattern $\mathbf{A} + \mathbf{A}^T$, and this analysis phase produces both an ordering and an assembly tree. The assembly tree is then used to drive the subsequent numerical factorization and solution phases. At each node of the tree, a dense submatrix (called a *frontal matrix*) is assembled using data from the original matrix and from the children of the node. Pivots can be chosen from within a submatrix of the frontal matrix (called the *pivot block*) and eliminations performed. The rows and columns of the pivot block are fully summed, meaning that no further contributions to them will come from rows or columns later in the pivotal sequence. The resulting factors are stored for use in the solution phase, and the Schur complement (the *contribution block*) is passed to the parent node for assembly at that node. In the numerical factorization phase, the tree is processed from the leaf nodes to the root

(if the matrix is reducible, we have a forest, and each component tree of the forest will be treated similarly and independently). The subsequent forward and backward substitutions during the solution phase process the tree from the leaves to the root and from the root to the leaves, respectively. A crucial aspect of the assembly tree is that it defines only a partial order for the factorization since the only requirement is that a child must complete its elimination operations before the parent can be fully processed. It is this freedom that enables us to exploit parallelism in the tree (*tree parallelism*).

In the unsymmetric case, threshold pivoting is used to maintain numerical stability so that it is possible that the pivots selected at the analysis phase are unsuitable. In the numerical factorization phase, we are at liberty to choose pivots from anywhere within the pivot block (including off-diagonal pivots) but it still may be impossible to eliminate all variables from this block. The result is that the Schur complement that is passed to the parent node may be larger than anticipated by the analysis phase and so our data structures may be different from those forecast by the analysis. This implies that we need to allow dynamic scheduling during numerical factorization. In the symmetric positive-definite case only static scheduling is required. However, in this present work, we will use dynamic scheduling for symmetric systems because we want to use our code to solve problems that are not positive definite and it provides more flexibility for load balancing.

In both the unsymmetric and symmetric cases, data is first assembled at a node combining the Schur complements from the children with data from the original matrix. The original matrix data comprises rows and columns corresponding to variables that the analysis forecasts should be eliminated at this node. This data is usually supplied in so-called arrowhead format, with the matrix ordered according to the permutation from the analysis phase and row 1 preceding column 1 followed by row 2 (from the diagonal) and column 2 and so on, where the columns are not supplied in the symmetric case because they are identical to the rows. This data and the contribution blocks from the children are assembled (or summed) into a frontal matrix using indirect addressing (sometimes called an extended add operation).

Eliminations are then performed on the assembled frontal matrix. A right-looking factorization can be used and is blocked so that cache effects can be reduced and use can be made of the Level 3 BLAS (Dongarra, Du Croz, Duff and Hammarling 1990*b*, Dongarra, Du Croz, Duff and Hammarling 1990*a*). This can be done by eliminating a fixed number of pivots (`nb`, say). When numerical pivoting is required, the fully summed rows must be updated during these eliminations but the major part of the frontal matrix is not updated until the computations on the fully summed rows are completed whence the remaining rows can be updated using Level 3 BLAS kernels. It is possible either to use parallel versions of the Level 3 BLAS or to update the rows in independent strips. This gives rise to so-called *node parallelism*.

A version of the multifrontal code for shared memory computers was developed by Amestoy and Duff (1989) and was included in Release 12 of the Harwell

Subroutine Library (HSL 1996) as code `MA41`. This was the basis for Version 1.0 of MUMPS that was released in May 1997.

3 Description of the main implementation issues

The current version of MUMPS ("MUltifrontal Massively Parallel Solver") solves the linear system of equations $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is either unsymmetric, or symmetric positive definite. Main features include the solution of the transposed system, error analysis, iterative refinement, scaling of the original matrix, and the possibility for the user to input a given ordering.

In the current version of MUMPS (Version 2.3), both tree and node parallelism are exploited, and we distribute the pool of work among the processors, but our model still requires an identified host node to perform the analysis phase, distribute the incoming matrix, collect the solution, and generally oversee the computation. All routines called by the user for the different steps are SPMD, and the distinction between the host and the other processors is made by the MUMPS code. The code is organized with a designated host node and other processors as follows (notice that the following steps are easily implemented within the controlling strategy of the PARASOL Library):

1. Analysis. The host performs an approximate minimum degree algorithm based on the symmetrized pattern $\mathbf{A}+\mathbf{A}^T$, and carries out symbolic factorization. A mapping of the multifrontal tree is then computed, and symbolic information is transferred from the host to the other processors. Using this information, the processors estimate the memory necessary for factorization and solution.

2. Factorization. The host sends appropriate entries of the original matrix to the other processors that are responsible for the numerical factorization. The numerical factorization on each frontal matrix is conducted by a *master* processor (determined by the analysis phase) and one or more *slave* processors (determined dynamically) as discussed later in this section. Each processor allocates an array for contribution blocks and factors; the factors must be kept for the solution phase.

3. Solution. The right-hand side is broadcast from the host to the other processors. These processors compute the solution using the (distributed) factors computed during Step 2, and the solution is assembled on the host.

We discuss, in the following subsections, implementation issues in a distributed environment and will focus on the description of the factorization phase since it is the most complicated and time consuming phase.

We first introduce common features of the unsymmetric and symmetric codes. We describe the static mapping strategy. We present the three types of parallelism

exploited during the factorization and solve phases and focus on the description of the factorization phase. Parallel implementation issues are then presented. Finally, we describe the main differences between the LU and the LDL^T factorizations.

For both the symmetric and the unsymmetric algorithms used in the code, we have chosen a fully asynchronous approach with dynamic scheduling of the computational tasks. Asynchronous communication was chosen to enable overlapping between communication and computation. Dynamic scheduling was initially used to accommodate numerical pivoting in the factorization. The other important reason for this choice is that, with dynamic scheduling, the algorithm has the potential to adapt itself at execution time, and can remap work and data to a more appropriate processor. In fact, we combine the main features of static and dynamic approaches. We use the estimation done during analysis to map some of the main computational tasks; the other tasks are dynamically scheduled at execution time. The main data structures (original matrix and matrix of the factors) are similarly partially mapped according to the analysis phase. Part of the initial matrix is replicated to enable rapid task migration without data redistribution.

3.1 Mapping

A mapping of the assembly tree to the processors is performed statically as part of the analysis phase. The main objectives of this phase are to control the communication costs, and to balance the memory used and the computation done by each processor. The computational cost will be approximated by the number of floating-point operations, and only the matrix of the factors will be taken into account when balancing the memory used by the processors.

In this section, we describe the algorithms used to map the assembly tree onto the processors and show how we have combined memory and work balancing criteria.

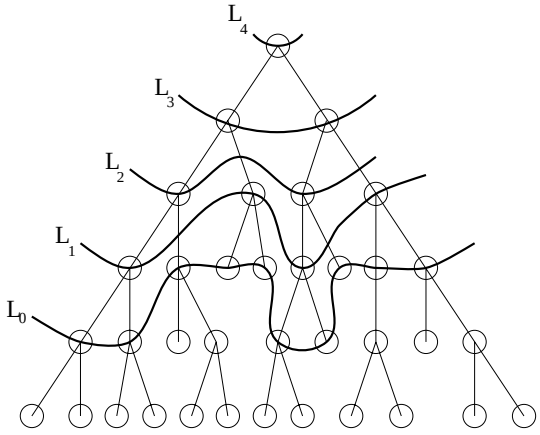

Figure 3.1: Decomposition of the assembly tree into levels.

The tree is processed from the bottom to the top, level by level (see Figure 3.1).

Level L_0 is determined using the Algorithm 1 (Geist and Ng 1989) and is illustrated in Figure 3.2. Then for $i > 0$, a node belongs to L_i if all its children belong to L_j, $j < i$. First, nodes of level L_0 (and the subtrees for which they are the root) are mapped. This first step is designed to balance the work in the subtrees and to reduce communication since all nodes in a subtree are mapped onto the same processor. Normally to get a good load balance it is necessary to have many more nodes in level L_0 than there are processors. Thus L_0 depends on the number of processors and a higher number of processors will lead to smaller subtrees.

Algorithm 1 – Construction and mapping of the initial level L_0

Let $L_0 \leftarrow$ Roots of the assembly tree
Repeat
Find the node q in L_0 whose subtree has largest computational cost
Set $L_0 \leftarrow (L_0 \backslash \{q\}) \cup \{children\ of\ q\}$ (See Figure 3.2)
Cyclic mapping of the nodes of L_0 onto the processors.
Estimate the load imbalance
Until *load imbalance* $<$ threshold

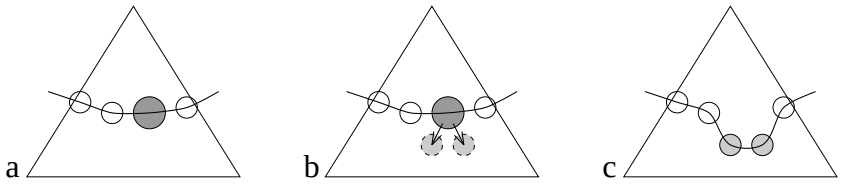

Figure 3.2: One step in the construction of the first level L_0.

The mapping of higher levels in the tree takes into account only memory balancing issues. For each processor, the memory load (total size of its factors) is first computed for the nodes at level L_0. For each level L_i, $i > 0$, each unmapped node of L_i is mapped to the processor with the smallest memory load and its memory load is revised.

The mapping is then used to explicitly distribute the permuted initial matrix onto the processors and to estimate the amount of work and memory required on each processor.

3.2 Sources of parallelism

We consider the condensed assembly tree of Figure 3.3, where the leaves are L_0 subtrees of the assembly tree.

There will be in general more leaf subtrees than processors, and therefore we can expect a good overall load balance of the computation at the bottom of the tree. However, if we only exploit the tree parallelism, the speed-up is very disappointing. The actual speed-up from this parallelism depends on the problem but is typically

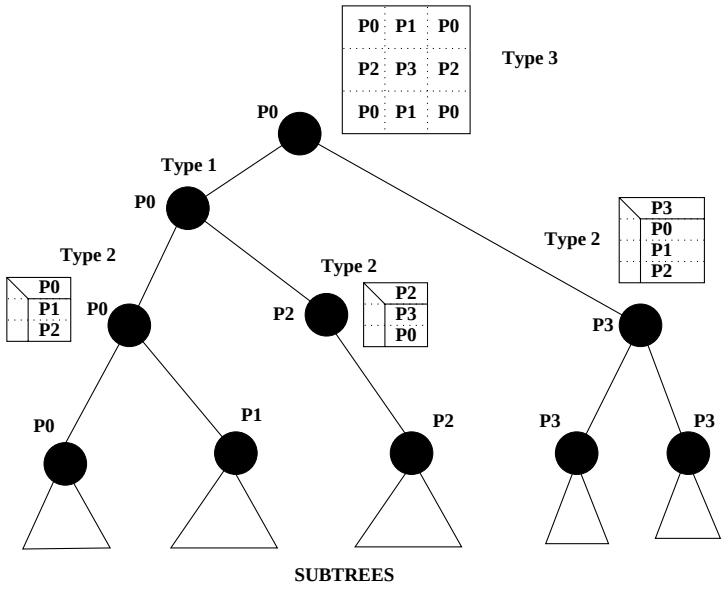

Figure 3.3: Distribution of the computations of a multifrontal assembly tree.

only 2 to 4 irrespective of the number of processors. This poor performance is caused by the fact that the tree parallelism decreases while going towards the root of the tree. Moreover, it has been observed (see for example Amestoy and Duff, 1993) that often more than 75% of the computations are performed in the top three levels of the assembly tree. It is thus necessary to obtain further parallelism within the large nodes near the root of the tree. The additional parallelism will be based on parallel versions of the blocked algorithms used during the factorization of the frontal matrices.

Nodes of the tree processed by only one processor will be referred to as nodes of *type 1* and the parallelism of the assembly tree will be referred to as *type 1 parallelism*. Further parallelism is obtained by doing a 1D block partitioning of the rows of the frontal matrix for nodes with a large contribution block. Such nodes will be referred to as nodes of *type 2* and the corresponding parallelism as *type 2 parallelism*. Finally, if the root node is large enough, then 2D block cyclic partitioning of the root frontal matrix is performed. The parallel root node will be referred to as a node of *type 3* and the corresponding parallelism as *type 3 parallelism*.

3.2.1 Description of type 2 parallelism

If a node is of type 2, one processor (called the master of the node) holds all the fully summed rows and performs the pivoting and the factorization on this block while other processors (so called slaves) perform the updates on the contribution rows (see Figure 3.4).

Macro-pipelining based on a blocked factorization of the fully-summed rows is

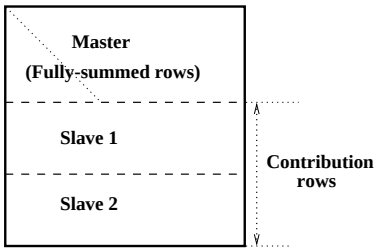

Figure 3.4: Type 2 nodes: partitioning of frontal matrix.

used to overlap communication with computation. The efficiency of the algorithm thus depends on both the block size used to factor the fully-summed rows and on the number of rows allocated to a slave process. During the analysis phase, based on the structure of the assembly tree, a node is determined to be of type 2 if its frontal matrix is sufficiently large. In terms of memory, the mapping algorithm assumes that the master processor holds the fully-summed rows and that any other processors might be selected as slave processes. As a consequence, part of the initial matrix is duplicated onto all the processors to enable efficient dynamic scheduling of computational tasks. At execution time, the master then first receives symbolic information describing the structure of the contribution blocks sent by its children. Based on this information, the master determines the exact structure of its frontal matrix and decides which slave processors will participate in the factorization of the node.

Further details on the implementation of type 2 nodes depends on whether the initial matrix is symmetric or not and will be given in Section 3.4.3.

3.2.2 Description of type 3 parallelism

In order to have good scalability, we perform a 2D block cyclic distribution of the root node. We use ScaLAPACK (Blackford, Choi, Cleary, D'Azevedo, Demmel, Dhillon, Dongarra, Hammarling, Henry, Petitet, Stanley, Walker and Whaley 1997) or the vendor equivalent implementation (PDGETRF for unsymmetric matrices and PDPOTRF for symmetric matrices).

Currently, a maximum of one root node, chosen during the analysis, is processed in parallel. This node is of type 3. The node chosen will be the largest root provided its size is larger than a computer dependent parameter. One processor, the so-called master of the root, holds all indices describing the root frontal matrix.

We define the root node as determined by the analysis phase, the *estimated* root node. Before factorization, the estimated root node frontal matrix is statically mapped onto a 2D grid of processors. We use a static distribution and mapping for those variables known by the analysis to be in the root node so that, for an entry in the estimated root node, we know where to send it and assemble it using functions involving integer divisions, moduli, ...

In the factorization phase, the original matrix entries and the part of the contribution blocks from the children corresponding to the estimated root can be assembled as soon as they are available. The master of the root node then collects the index information for all the uneliminated variables of its children and builds the structure of the frontal matrix. This symbolic information is broadcast to all participating processors. The contributions corresponding to uneliminated variables can then be sent by the children to the appropriate processors in the 2D grid for assembly, or directly assembled locally if the destination is the same processor. Note that, because of the requirements of ScaLAPACK, local copying of the root node is required since the leading dimension will change if there are any uneliminated variables.

3.2.3 Impact of parallelism on memory and work balance

Processor number	1	2	3	4	5
Original matrix ($\times 10^3$)	1920	2904	2475	2571	2059
LU factors ($\times 10^3$)	15927	15982	15993	16149	16117
Flop count ($\times 10^9$)	18.2	21.5	18.6	22.6	19.5

Table 3.1: Study of memory and work balancing on matrix CRANKSEG_1 using 5 working processors (that is, we exclude the host processor) and all levels of parallelism of the method. All sizes are in number of 64-bit reals per processor.

We show, in Table 3.1, the distribution of both the input matrix and the LU factors during the factorization of matrix CRANKSEG_1 on five working processors. The matrix is considered unsymmetric and has 10.6×10^6 nonzeros with 80.6×10^6 nonzeros in the LU factors. We see, in Table 3.1, how well the mapping algorithm balances the storage of the LU factors between the processors. Concerning the original matrix, we observe that the extra space due to duplication for type 2 node parallelization only represents around 10% of the size of the original matrix. Finally we see that, even if the algorithm does not aim to balance the work near the top of the tree, balancing the memory used for the factors also leads to a good balance for the floating-point operations.

3.3 Parallel implementation issues

To enable automatic overlapping between computation and communication, we have chosen to use fully asynchronous communications. For flexibility and efficiency, explicit buffering in the user space has been implemented. We have developed a Fortran 90 module to send asynchronous messages, based on immediate sends. We define a send buffer for each processor based on information from the analysis phase.

When we try to send contribution blocks, factorized blocks, ... we first check to see if there is room in the send buffer. Our module provides an equivalent of MPI_BSEND (Dongarra, Hempel, Hey and Walker 1995) with the advantage that messages are directly packed in the buffer and problems occurring when the buffer is full are overcome. Note that messages are never sent when the destination is identical to the source; in that case the associated action is performed directly locally, instead of the send.

Estimates of the minimum sizes needed for the send and receive buffers are computed by each processor prior to factorization. This estimation is based on the static mapping of the assembly tree and takes into account the three types of parallelism used during factorization. Note that, for example, using type 2 parallelism will significantly reduce the size of the contribution blocks sent between processors, and thus of the required buffers, as shown in Figure 3.5. Buffers are allocated on each processor at the beginning of the factorization.

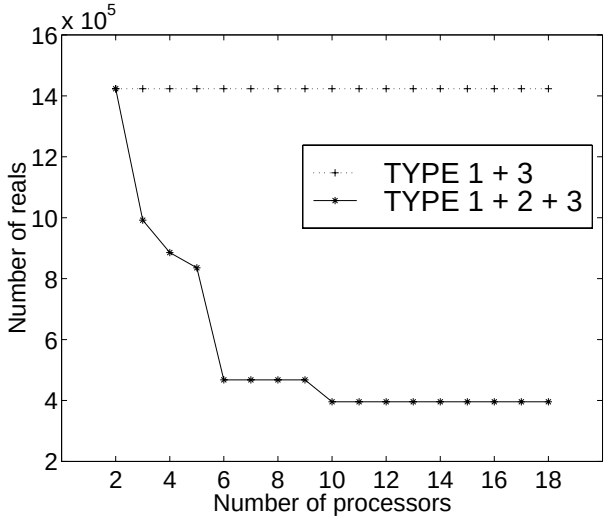

Figure 3.5: Impact of type 2 parallelism on the size (in number of 64-bit reals) of the send buffer. Test matrix is WANG3.

If there is not enough space to put the message in the buffer, the procedure requesting the send returns with an error code. In such cases, to avoid deadlock, the processor will try to receive messages until space becomes available in its local send buffer. Let us take a simple illustrative example. Processor A has filled-up its buffer doing an asynchronous send of a large message to processor B. Processor B has done the same to processor A. The next messages sent by both processors A and B will then be blocked until the other processor has received the first message. More complicated situations involving more processors can occur, but in all cases the key issue for avoiding deadlock is that each processor tries not to be the blocking processor.

MPI only guarantees that messages are non-overtaking, that is if a processor sends two messages to the same destination, then the receiver will receive them in the same order. For synchronous algorithms the non-overtaking property is often enough to ensure that messages are received in the correct order. With a fully asynchronous algorithm, based on dynamic scheduling of the computational tasks, it can happen that messages arrive "too early". In this case, it is crucial to be sure that the "missing" messages have already been sent so that blocking receives can be performed to process all messages that should have already been processed at this stage of the computation. As a consequence, the order used for sending messages is important. The impact on the algorithm design will be illustrated in Sections 3.4.1 and 3.4.3 during the detailed description of type 2 parallelism for LDL^T factorization.

A pool of tasks is used to implement dynamic scheduling. All tasks ready to be activated on a given processor are stored in the pool of tasks local to the processor. Each processor then executes the following algorithm.

Algorithm 2

while *(all nodes not processed)*
  if *local pool empty* **then**
    blocking wait for a message; process the message
  elseif *message available then*
    receive and process message
  else
    extract work from the pool, and process it.
  endif
end while

Note that priority is given to message reception. The main reasons for this choice are first that the message received might be a source of additional work and parallelism and second that the sending processor might be blocked because its send buffer is full.

3.4 LU versus LDL^T approaches

In this section, we describe the main differences between the symmetric and the unsymmetric algorithms. The symmetric code currently solves symmetric positive-definite systems, but it has been designed so that future developments like fully distributed LDL^T factorization with numerical pivoting and the detection of the null spaces, remain possible.

Taking into account the symmetry of the input matrix leads to a reduction in both the memory requirements (smaller input matrix, matrix of factors and frontal matrices) and the computational cost. Only the lower part of the original matrix is accessed and the LDL^T factorization is computed. Even if a significant part of

the implementation issues are shared by the LU and LDL^T factorizations, taking into account the symmetry implies major modifications in the assembly process, in the blocked factorization of nodes of type 1 and 2, and in the type 2 and 3 parallel algorithms.

Taking into account the symmetry for a node of type 3 was rather straightforward because our implementation is based on the use of ScaLAPACK (Blackford et al. 1997) routines (PDGETRF for the LU factorization and PDPOTRF for the LL^T factorization). Note that a parallel version of the LDL^T factorization for dense matrices does not exist in ScaLAPACK and that this issue will have to be addressed in a future release of the code that includes numerical pivoting for symmetric matrices.

3.4.1 Assembly process

An estimation of the frontal matrix structure (size, number of fully-summed variables) is computed during the analysis phase. The final structure and the list of indices in the front is however only computed during the assembly process of the factorization phase. The list of indices of a front is the result of a merge of the index lists of the contribution blocks of the children with the list of indices in the arrowheads associated with all the fully-summed variables of the front. Once the index list of the front is computed, the assembly of numerical values can be performed efficiently.

Let *inode* be a node of type 2. The master of *inode* defines the partition of rows of the frontal matrix into blocks, and chooses a set of slave processors that will participate in the parallel assembly and factorization of *inode*. It sends a message (identified by the tag DESC_STRIP) describing the work to be done on each slave processor. It also sends a message (with tag MAPROW) to all type 1 nodes and slave processors of type 2 nodes for the children of *inode*, giving them information on where to send their contribution blocks for the assembly process.

As already mentioned in Section 3.3, the order in which messages are sent is important. For example, a slave of *inode* may receive a contribution block before receiving the message of tag DESC_STRIP from its master. To allow this slave processor to safely perform a blocking receive on the missing DESC_STRIP message, we must ensure that the master of the node has sent DESC_STRIP before sending MAPROW. Otherwise we cannot guarantee that DESC_STRIP will actually be sent (for example, the send buffer might be full).

The main difference between the symmetric and the unsymmetric case is that a global ordering of the indices in the frontal matrices is necessary in the symmetric case to guarantee that all lower triangular entries in a contribution row of a child are in the lower triangular part of the corresponding row in the parent. We use the global ordering obtained during analysis, that is, the order in which variables would be eliminated if no numerical pivoting occurs.

Moreover, it is quite easy to perform a merge of sorted lists efficiently. If we assume that the list of indices of the contribution block of each child is sorted then the sorted merge algorithm will be efficient if the indices associated with the arrowheads are also sorted. Unfortunately, sorting all the arrowheads can be costly. Furthermore, the number of fully-summed variables (or number of arrowheads) in a front might be quite large and the efficiency of the merging algorithm might be affected by the large number of sorted lists to merge. Based on experimental results, we have observed that it is enough to sort only the arrowhead associated with the first fully-summed variable of each frontal matrix. The assembly process for the list of indices of the node is thus described in Algorithm 3.

Algorithm 3 Assembly of indices in a parent node

1. *Sorted merge of the sorted lists of the indices of the children and of the first arrowhead.*
2. *Build and sort variables belonging only to the other arrowheads (and not found at step 1)*
3. *Merge the sorted list built at step 2 with the sorted list obtained at step 1.*

The key issue for efficiency of Algorithm 3 is the fact that only a small number of variables are found at step 2. This has been experimentally validated. For example, on matrix WANG3, the average number of indices found at step 2 was 0.3. The numerical assembly can then be performed, row by row.

3.4.2 Factorization of type 1 nodes

Blocked algorithms are used during the factorization of type 1 nodes and, for both the LU and the LDL^T factorization algorithms, we want to keep the possibility of postponing the elimination of fully-summed variables. Note that classical blocked algorithms for the LU and LL^T factorizations of full matrices (Anderson, Bai, Bischof, Demmel, Dongarra, DuCroz, Greenbaum, Hammarling, McKenney, Ostrouchov and Sorensen 1995) are quite efficient, but it is not the case for the LDL^T factorization.

We will briefly compare kernels involved in the blocked algorithms. We then show how we have exploited the frontal matrix structure to design an efficient blocked algorithm for the LDL^T factorization.

Let us suppose that the frontal matrix has the structure of Figure 3.6, where A is the block of fully summed variables available for elimination. Note that, in the code, the frontal matrix is stored by rows.

During LU factorization, a KJI-SAXPY blocked algorithm (Amestoy and Duff 1989, Daydé and Duff 1991) is used to compute the LU factor associated with the block of fully summed rows (matrices A and C). The Level 3 BLAS kernel DTRSM

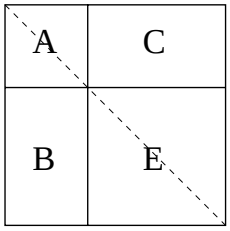

Figure 3.6: Structure of a type 1 node.

is used to compute the off-diagonal block of L (overwriting matrix B). Updating the matrix E is then a simple call to the Level 3 BLAS kernel, DGEMM.

During LDL^T factorization, a right-looking blocked algorithm is first used to factor the block column of the fully summed variables. Let L_{off} be the off diagonal block of L stored in place of the matrix B and D_A be the diagonal matrix associated with the LDL^T factorization of the matrix A. The updating operation of the matrix E is then of the form $E \leftarrow E - L_{off} D_A L_{off}^T$ where only the lower triangular part of E needs to be computed. No Level 3 BLAS kernel is available to perform this type of operation which corresponds to a generalized DSYRK kernel.

Note that, when we know that no pivoting will occur (symmetric positive definite matrices), L_{off} is computed in one step using the Level 3 BLAS kernel DTRSM. Otherwise, the trailing part of L_{off} has to be updated after each step of the blocked factorization, to allow for a stability test for choosing the pivot.

To update the matrix E, we have applied the ideas used by Daydé and Duff (1996) to design efficient and portable Level 3 BLAS kernels. Blocking of the updating is done in the following way. At each step, a block of columns of E (E_k in Figure 3.7) is updated. In our first implementation of the algorithm, we stored the scaled matrix

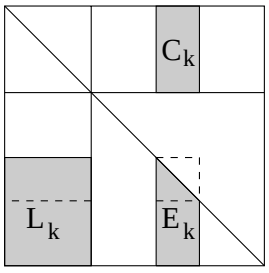

Figure 3.7: Blocks used for updates of the contribution part of a type 1 node.

$D_A L_{off}^T$ in matrix C, used here as workspace. Because of cache locality issues, the Megaflop rate was still much lower than that of the LU or Cholesky factorizations. In the current version of the algorithm, we compute the block of columns of $D_A L_{off}^T$ (C_k in Figure 3.7) only when it will be used to update E_k. Furthermore, to increase cache locality, the same working area is used to store all C_k matrices. This was possible because C_k matrices are never reused in the algorithm. Finally, the Level

3 BLAS kernel DGEMM is used to update the rectangular matrix E_k. This implies more operations but is more efficient on the IBM SP2 than the updates of the shaded trapezoidal submatrix of E_k using a combination of DGEMV and DGEMM kernels. Our final blocked algorithm is summarized in Algorithm 4.

Algorithm 4 LDL^T **factorization of type 1 nodes**
Blocked factorization of the fully summed columns
do *k = 1, nb_blocks*
Compute C_k *(block of columns of* $D_A L_{off}^T$*)*
$E_k \leftarrow E_k - L_k C_k$
end do

3.4.3 Parallel factorization of type 2 nodes

The differences between the symmetric and the unsymmetric case come from a modification of both the frontal matrix structure and the parallel algorithm. The modification of the matrix structure is illustrated in Figure 3.8. In both algorithms, the master processor is in charge of all the fully summed rows and the blocked algorithms used to factor the block of fully-summed rows are the ones described in the previous subsection.

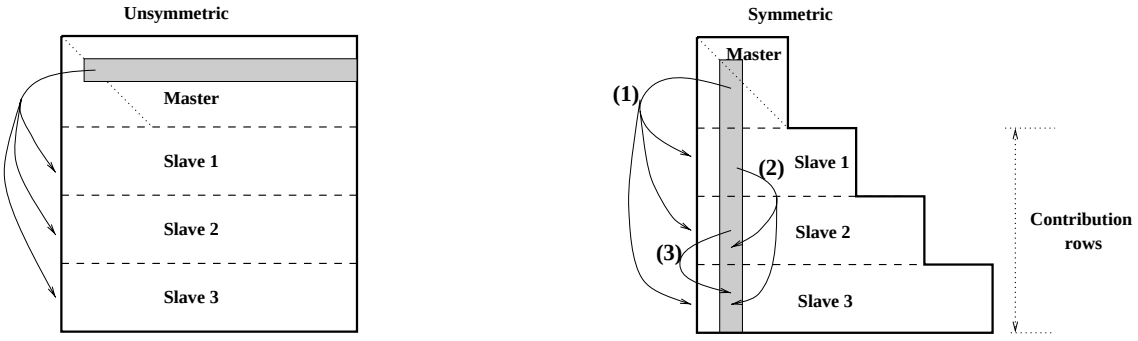

Figure 3.8: Structure of a type 2 node.

In the unsymmetric case, at each block step, the master processor sends the factorized block of rows to its slave processors and then updates its trailing submatrix. The behaviour of the algorithm is illustrated in Figure 3.9, where program activity is represented in black, inactivity in grey, and messages by lines between processes. The figure is a trace record generated by the VAMPIR package (Nagel, Arnold, Weber, Hoppe and Solchenbach 1996) from PALLAS. We see that, on this example, the master processor is relatively more loaded than the slaves.

In the symmetric case, a different parallel algorithm has been implemented. The master of the node performs a blocked factorization of only the diagonal block of fully-summed rows. At each block step, its part of the factored block of columns is

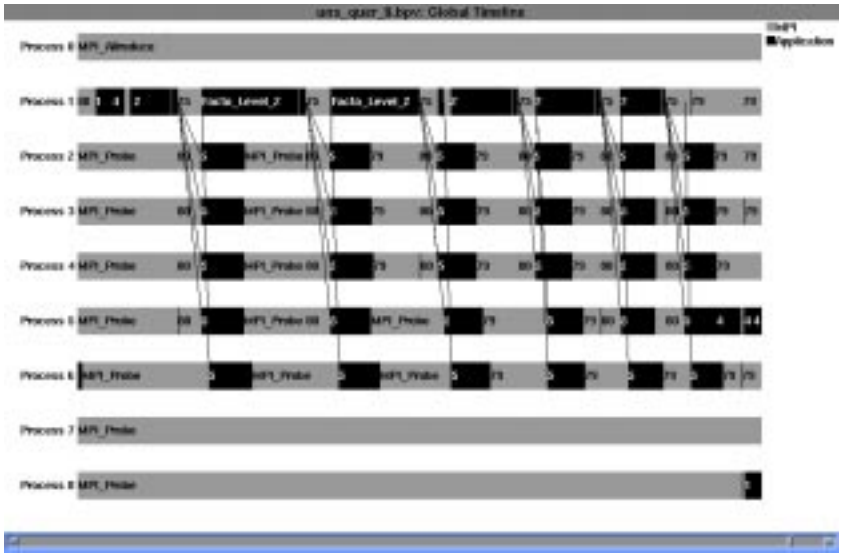

Figure 3.9: VAMPIR trace of an isolated type 2 unsymmetric factorization (Master is Process 1).

broadcast to all slaves ((1) in Figure 3.8). Each slave can then use this information to compute its part of the block column of L and to update part of the trailing matrix. Each slave, apart from the last one, then broadcasts its just computed part of the block of column of L to the following slaves (illustrated by messages (2) and (3) in Figure 3.8). Note that, to process messages (2) or (3) at step k of the blocked factorization, the corresponding message (1) at step k must have been received and processed.

We have chosen a fully asynchronous approach to implement the algorithm. Messages (1) and (2) might thus arrive in any order. The only property that MPI guarantees is that messages of type (1) will be received in the correct order because they come from the same source processor. When a message (2) at step k arrives too early, we have then to force the reception of all the pending messages of type (1) for steps smaller than or equal to k. This induces a necessary property in the broadcast process of messages (1): if at step k, message (1) is sent to slave 1, we must be sure that it will also be sent to other slaves. In our implementation of the broadcast, we first check availability of memory in the send buffer (with no duplication of data to be sent) before starting effective send operations. Thus, if the asynchronous broadcast starts, it will complete.

Similarly to the unsymmetric case, our first implementation of the algorithm is based on constant row block size. We can clearly observe from the corresponding execution trace in Figure 3.10 that the later slaves have much more work to perform than the others. To balance work between slaves, later slaves should hold less rows. This has been implemented using a heuristic that aims at balancing the total number of floating-point operations involved in the type 2 node factorization on each

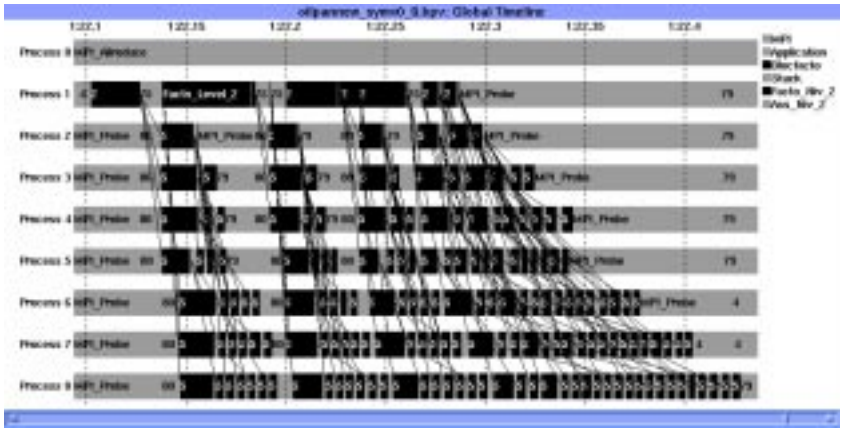

Figure 3.10: VAMPIR trace of an isolated type 2 symmetric factorization; constant row block sizes. (Master is Process 1).

slave. As a consequence, the number of rows treated varies from slave to slave. The corresponding execution trace is shown in Figure 3.11. We can observe that work on the slaves is much better balanced and both the difference between the termination times of the slaves and the elapsed time for factorization are reduced.

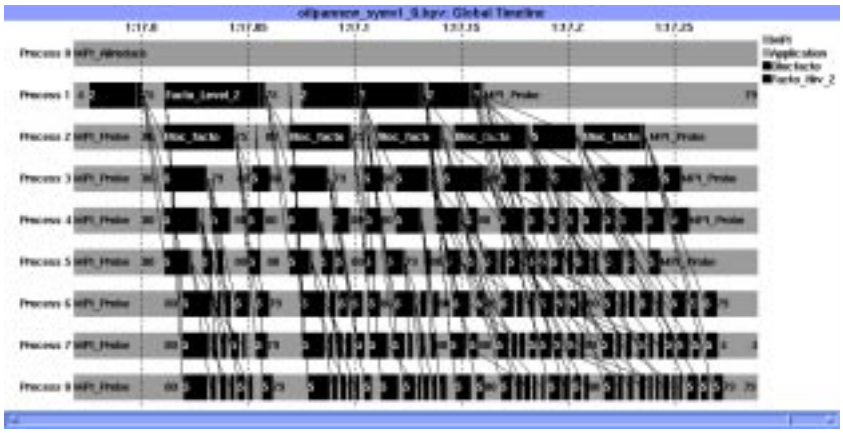

Figure 3.11: VAMPIR trace of an isolated type 2 symmetric factorization; variable row block sizes. (Master is Process 1).

However, the comparison of Figures 3.9 and 3.11 shows that firstly the number of messages involved in the symmetric algorithm is much larger than in the unsymmetric case; secondly, that the master processor performs relatively less work than in the parallel algorithm for unsymmetric matrices.

4 Performance

The results presented in this section have been obtained on a 34 processor IBM SP2 located at GMD (Bonn, Germany). Each node of this computer is a 66 MHertz processor with 128 MBytes of physical memory and 512 MBytes of virtual memory. An approximate Minimum Degree (AMD) ordering (Amestoy, Davis and Duff 1996) has been used to permute the initial matrix and all timings are given in seconds.

4.1 The theoretical speed-up of the methods

The maximum theoretical speed-up obtained for each type of parallelism is indicated in Tables 4.1 and 4.2. In these tables, we use a few typical matrices from our set of test problems. We do not take into account communication time and the number of processors available is assumed infinite. No account is taken of changes to the tree because of numerical pivoting. We compute the maximum theoretical speed-up by dividing the total number of flops during factorization by the number of flops on the longest path of the tree; for type 2 nodes, we suppose that the update on the slave nodes can be done for free. A node is considered to be of type 2 only if its contribution block is of size at least 200.

We also show speed-ups for the tree without the root in order to show the true gains from type 2 parallelism. This is masked in the results for the complete tree because of the amount of work performed at the root.

	Complete tree			Tree without root	
Matrix	Type 1	Type 1+2	Type 1+2+3	Type 1	Type 1+2
BCSSTK15	1.98	4.58	10.0	2.35	11.1
WANG3	1.38	3.08	13.8	1.60	11.9
QUER	3.16	7.70	24.5	4.08	25.9

Table 4.1: Estimated speed-up for the unsymmetric solver.

	Complete tree			Tree without root	
Matrix	Type 1	Type 1+2	Type 1+2+3	Type 1	Type 1+2
BCSSTK15	1.98	5.61	16.7	2.35	22.6
WANG3	1.38	3.67	46.9	1.60	68.1
QUER	3.16	9.68	69.5	4.08	105.6

Table 4.2: Estimated speed-ups for the symmetric solver.

As mentioned in Section 3.2, we notice that the parallelism arising from the tree is very limited. This can be improved by using some other reordering techniques, for

example combining nested dissection and minimum degree. We are experimenting with such reorderings and plan to incorporate some code for this, developed from the RALPAR partitioning package (Fowler and Greenough 1998), within our analysis phase. This is also the topic of a collaboration with Roman and Pellegrini (LaBRI, Bordeaux) and will not be addressed further in this paper. However, we see that a significant speed-up increase is provided by parallelism of types 2 and 3. Note that our model is very simple and therefore optimistic. It is also interesting to notice that type 2 parallelism is better in the symmetric case than in the unsymmetric case. This is due to the fact that, in the symmetric case, the master process is only in charge of the diagonal block of fully summed variables whereas in the unsymmetric case the master also computes the off-diagonal block (block of U factors) of the frontal matrix.

4.2 Actual performance

We report, in Tables 4.3 and 4.4, some statistics on various test problems: the size of the factors (both estimated and actual for unsymmetric test problems because of numerical pivoting), the number of floating-point operations for elimination, the size of the root node, and the time for analysis.

For some of the symmetric problems, we also give statistics on the corresponding *unsymmetrized* problem, as this will allow us to compare the behaviour of the symmetric and unsymmetric codes.

	Nonzeros in factors ($\times 10^6$)		Flops ($\times 10^9$)		Size of	Time for
Matrix	estim.	actual	estim.	actual	root	analysis
WANG3	11.5	11.5	10.5	10.5	1601	7.3
INV-EXTRUSION-1	30.3	31.2	34.3	35.8	1913	5.2
MIXING-TANK	38.5	39.1	64.1	64.4	2985	5.6

Table 4.3: Statistics for unsymmetric test problems (ordering based on AMD).

We now report on numerical experiments on type 1, 2, and 3 parallelism on the test problem WANG3, and on two instances of the problem QUER; one symmetric and the other unsymmetric.

For the test problem WANG3 (Table 4.5), timings obtained on 1 and 2 working processors are high because of memory paging when using the virtual memory. For larger numbers of processors, distribution of memory suppresses this effect. We estimate the uniprocessor time without memory paging from the CPU time as 71.0 seconds, so speed-up should be compared to that. Comparing these results with the theoretical study of Table 4.1, we observe that the theoretical speed-up gives a good estimation of the actual speed-up for type 1 and type 2 parallelism. It shows the

Matrix	Nonzeros in factors ($\times 10^6$)	Flops ($\times 10^9$)	Size of root	Time for analysis
SYMMETRIC CODE				
B5TUER	30	13.2	1435	16.2
BMW3_2	59	44.9	2495	18.7
BMW7ST_1	31	15.4	1560	9.8
CRANKSEG_1	48	50.2	2161	10.8
CRANKSEG_2	73	101.9	3127	14.5
OILPAN	12	3.8	819	4.5
QUER	12	4.0	1043	3.0
UNSYMMETRIC CODE				
B5TUER	52	26.4	1435	25.4
BMW7ST_1	54	30.7	1560	19.8
CRANKSEG_1	80	100.4	2161	34.9
OILPAN	20	7.6	819	7.1
QUER	20	8.0	1043	5.5

Table 4.4: Statistics for symmetric test problems (ordering based on AMD).

Working processors	Time for factorization (in seconds)		
	Type 1	Type 1 + Type 2	Type 1 + 2 + 3
1	206.6	142.4	216.4
2	105.6	81.0	96.5
4	51.1	40.0	24.6
8	46.0	28.6	19.5
12	49.2	28.1	18.8
16	47.4	29.2	17.3
24	46.7	29.7	16.7
32	45.6	27.2	16.6

Table 4.5: Influence of the types of parallelism for WANG3. Estimated sequential CPU time without paging is 71.0 seconds.

good overlapping of communication with computation since the estimated speed-up did not take into account the communication time. For type 3 parallelism, the estimated speed-up is quite optimistic.

Working processors	Time for factorization		
	Type 1	Type 1 + Type 2	Type 1 + 2 + 3
1	284.1	275.3	299.2
2	113.1	108.0	109.1
4	27.0	22.7	19.1
8	20.7	19.4	15.2
16	18.5	14.8	11.5
24	18.2	12.8	10.1
32	17.6	13.0	10.5

Table 4.6: Influence of the type of parallelism for QUER, treated as unsymmetric. Estimated sequential CPU time without paging is 64.9 seconds.

Working processors	Time for factorization (in seconds		
	Type 1	Type 1 + Type 2	Type 1 + 2 + 3
1	133.5	141.7	150.4
2	31.3	22.6	21.0
4	18.3	15.7	12.9
8	14.4	12.7	9.3
16	12.7	10.5	6.4
24	12.5	8.7	6.6
32	12.1	8.7	5.8

Table 4.7: Influence of the type of parallelism for QUER, treated as symmetric. Estimated sequential CPU time without paging is 41.1 seconds.

The same data are given in Tables 4.6 and 4.7, for the test problem QUER treated as symmetric, and unsymmetric (respectively). The estimated uniprocessor CPU times without paging (64.9 and 41.1 for the unsymmetric and the symmetric codes respectively) show that the uniprocessor Megaflop rate of the symmetric code (97 Mflops) is not too far from the uniprocessor Megaflop rate of the unsymmetric code (122 Mflops). Again, we observe that type 2 and 3 parallelism provide a significant increase in performance, and that paging has occurred for small numbers of processors. The higher speed-ups obtained on matrix QUER, treated as unsymmetric, with type 2 parallelism (5.0) compared with matrix WANG3 (2.6) reflect the difference in the estimated speed-up shown in Table 4.1. However, we do not benefit from the larger theoretical speed-up of type 2 parallelism on symmetric matrices compared with unsymmetric matrices (theoretical speed-up

of 9.68 compared to 7.70). The effective maximum speed-up obtained with the symmetric and the unsymmetric codes are in fact comparable being 4.7 and 5.0 respectively. This can be explained by the fact that, as already illustrated in Figures 3.9 and 3.11, although the master is in charge of relatively less work in the symmetric case, the parallelism of type 2 involves an increase in the communication flow and more irregularity of the distribution is required to correctly balance work on the slaves. Our present modification, illustrated in Figure 3.11, redistributes the block rows so that the number of flops performed by each slave is balanced but does not take into account the greater communication with later slaves.

We see, however, that parallelism of type 3 for symmetric matrices provides relatively larger speed-up increases than for unsymmetric matrices. As a result, on 32 processors, the factorization time of the symmetric code is almost half that of the unsymmetric code. The Megaflop rate for the symmetric and the unsymmetric factorizations is thus comparable (around 750 Mflops).

More results on all the large symmetric problems of our set are reported in Table 4.8. Results with the unsymmetric code are shown in Table 4.9. Some of the symmetric matrices could not be processed with the unsymmetric code because of the increase in memory requirements.

In these two tables, the time for distribution is the time for distributing the permuted initial matrix from the host processor onto the other processors; then the times for factorization and for solve are reported for various numbers of working processors. The analysis is sequential and so the time, reported in Tables 4.3 and 4.4, is independent of the number of processors. The distribution of the initial matrix is relatively time consuming because the same processor is in charge of sending pieces of the input matrix to all other processors and the communication network quickly gets saturated. Furthermore, if the initial matrix is large, paging can occur at this step, for example for CRANKSEG_1 or BMW7ST_1 when treated as unsymmetric. Using an initially distributed matrix on entry could lead to a better redistribution time and could avoid paging problems during this phase, especially in the case where the initial matrix does not fit in the physical memory of the host processor, but this would also require a more complicated interface to enable the user to provide a distributed matrix.

For the symmetric matrices appearing in both Tables 4.8 and 4.9, we can compare the performance of the symmetric and unsymmetric codes. We see that, even if the most important phases (assembly, factorization of types 1 and 2 nodes) of the LDL^T factorization are intrinsically more complicated than during LU factorization, the LDL^T gets full benefit from the symmetry and is usually almost twice as fast as the LU factorization. This was not the case in an earlier version of the symmetric code and much effort has been spent on optimizing low-level kernels for symmetric matrix calculations.

On large matrices and on a small number of processors, the problem of page swapping can have a somewhat extreme influence on the time for solution. This is

Matrix	Working processors	Time for distrib	Time for facto	Time for solve
B5TUER	4	12.0	151.05	122.67
B5TUER	8	12.6	31.18	2.16
B5TUER	16	13.9	17.81	1.92
B5TUER	24	17.4	14.71	1.59
B5TUER	32	18.4	12.46	2.16
BMW7ST_1	5	17.7	258.05	193.88
BMW7ST_1	8	13.6	47.40	27.32
BMW7ST_1	16	12.4	19.51	2.02
BMW7ST_1	24	15.4	17.85	1.99
BMW7ST_1	32	30.3	15.21	1.42
OILPAN	2	6.3	40.75	19.3
OILPAN	4	6.4	13.61	0.96
OILPAN	8	7.3	8.96	0.92
OILPAN	16	7.3	7.18	0.77
OILPAN	24	6.8	6.46	0.73
OILPAN	32	7.4	6.06	0.72
QUER	1	4.5	150.41	142.58
QUER	2	4.5	21.00	0.93
QUER	4	4.4	12.97	0.76
QUER	8	4.5	9.27	0.73
QUER	16	4.8	6.41	0.59
QUER	24	5.1	6.63	0.64
QUER	32	5.4	5.85	0.54
BMW3_2	8	129.8	369.52	237.74
BMW3_2	16	124.6	125.50	47.98
BMW3_2	24	145.2	45.17	5.60
BMW3_2	32	134.5	32.87	6.82
CRANKSEG_1	8	77.0	480.22	170.51
CRANKSEG_1	16	90.1	252.11	24.37
CRANKSEG_1	24	96.8	65.53	2.32
CRANKSEG_1	32	125.4	59.68	2.92
CRANKSEG_2	16	159.4	1045.34	90.33
CRANKSEG_2	24	249.7	457.26	39.39
CRANKSEG_2	32	222.6	139.66	11.13

Table 4.8: Results for the symmetric version of the code.

Matrix	Working processors	Time for distrib	Time for facto	Time for solve
B5TUER	8	164.2	111.45	58.9
B5TUER	16	181.2	29.35	3.77
B5TUER	24	177.7	26.72	3.15
B5TUER	32	168.0	26.40	2.67
WANG3	1	0.6	216.44	133.03
WANG3	2	0.7	96.54	36.09
WANG3	4	0.6	24.58	0.70
WANG3	8	0.6	19.47	0.77
WANG3	12	0.6	18.99	0.67
WANG3	16	0.6	17.35	0.70
WANG3	24	0.6	16.68	1.04
WANG3	32	0.7	16.62	0.78
INV-EXTRUSION-1	4	5.79	546.42	123.71
INV-EXTRUSION-1	8	6.27	131.26	17.25
INV-EXTRUSION-1	16	6.8	54.6	1.17
INV-EXTRUSION-1	24	8.2	54.9	1.40
INV-EXTRUSION-1	32	9.6	55.6	1.39
BMW7ST_1	8	209.9	163.07	106.99
BMW7ST_1	16	195.5	37.96	3.60
BMW7ST_1	24	177.0	35.13	3.50
BMW7ST_1	32	202.1	33.99	3.72
MIXING-TANK	8	7.0	321.07	87.32
MIXING-TANK	16	8.0	68.61	1.73
MIXING-TANK	24	9.6	61.36	1.25
MIXING-TANK	32	11.6	60.87	1.23
OILPAN	2	10.4	119.69	157.60
OILPAN	4	11.0	21.11	0.74
OILPAN	8	10.8	15.35	0.72
OILPAN	16	11.1	12.54	0.68
OILPAN	24	12.6	11.63	0.72
OILPAN	32	13.0	11.55	0.79
QUER	1	8.5	299.19	340.16
QUER	2	13.7	109.09	97.35
QUER	4	8.5	19.11	0.79
QUER	8	8.9	15.23	0.60
QUER	16	14.6	11.54	0.57
QUER	24	9.2	10.13	0.73
QUER	32	14.2	10.50	1.33
CRANKSEG_1	32	333.8	168.07	9.83

Table 4.9: Results for the unsymmetric version of the code (symmetric matrices are expanded).

due to the fact that computational time is dominated by memory access time due to page swapping. Therefore, accessing the relatively large matrix of the factors twice, as is done in the solve phase, is more critical than the number of actual floating-point operations.

Generally, we observe that our distributed memory approaches correctly exploit the memory available leading to superlinear speed-ups. They also exploit well the parallelism of the assembly tree and, even if additional tuning might still be done on type 2 parallelism, the overall speed-up of the codes is satisfactory. Finally one of the main properties of our parallel LDL^T factorization is that its Megaflop rate is comparable to that of the parallel LU factorization.

5 Conclusions and perspectives

From the results of Section 4, we can conclude that the current version of our MUMPS code does parallelize well and produces comparable speed-ups to shared memory variants, at least on a small number of processors. It is difficult to fully assess the scalability because of memory effects on small numbers of processors and insufficiently large problems for many processors. Certainly the Achilles heel for the code, as for all direct methods, is that of storage, somewhat exacerbated for the current code because of the need to estimate storage requirements in advance. This is one aspect on which we plan to work further. Certainly we plan to test the code on the ORIGIN 2000 computer in Bergen that has a far larger memory that should mitigate against paging effects.

We are currently studying other orderings including the use of dissection algorithms and their combination with minimum degree. Not only should this help the parallelism but often the overall number of floating-point operations is reduced.

We also plan to investigate further the dynamic scheduling of tasks from type 2 nodes based on estimates of the load on each processor.

Our current symmetric code has been developed from a code for unsymmetric matrices and has retained the capability of postponing eliminations for numerical reasons. This functionality can help in detecting rank and in developing an algorithm for null-space detection and determination of the null-space basis which will be needed by our code when used within PARASOL as a local solver within the domain decomposition codes, for example when using the Neumann-Neumann algorithm as described by Mandel (1993).

Acknowledgment

We would like to thank John Reid of the Rutherford Appleton Laboratory for helpful comments on an earlier draft.

References

Amestoy, P. R. and Duff, I. S. (1989), 'Vectorization of a multiprocessor multifrontal code', *Int. J. of Supercomputer Applics.* **3**, 41–59.

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

Amestoy, P. R., Davis, T. A. and Duff, I. S. (1996), 'An approximate minimum degree ordering algorithm', *SIAM J. Matrix Analysis and Applications* **17**(4), 886–905.

Amestoy, P. R., Duff, I. S., L'Excellent, J.-Y. and Plecháč, P. (1998), Parasol. an integrated programming environment for parallel sparse matrix solvers, Technical Report RAL-TR-98-039, Rutherford Appleton Laboratory. To appear in Proceedings of Conference HPCI 1998.

Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S. and Sorensen, D. (1995), *LAPACK Users' Guide, second edition*, SIAM Press.

Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D. and Whaley, R. C. (1997), *ScaLAPACK Users' Guide*, SIAM Press.

Daydé, M. J. and Duff, I. S. (1991), 'Use of Level 3 BLAS in **LU** factorization in a multiprocessing environment on three vector multiprocessors, the CRAY-2, the IBM 3090 VF and the Alliant FX/80', *Int. J. Supercomputer Applics* **5**(3), 92–110.

Daydé, M. J. and Duff, I. S. (1996), A blocked implementation of Level 3 BLAS for RISC processors, Technical Report RAL-TR-96-014, Rutherford Appleton Laboratory. Also ENSEEIHT-IRIT Technical Report RT/APO/96/1 and CERFACS Report TR/PA/96/06.

Dongarra, J., Hempel, R., Hey, A. J. G. and Walker, D. W. (1995), 'MPI : A Message Passing Interface Standard', *Int J. Supercomputer Applications* **8**, (3/4).

Dongarra, J. J., Du Croz, J., Duff, I. S. and Hammarling, S. (1990*a*), 'Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms: Model implementation and test programs', *ACM Trans. Math. Softw.* **16**(1), 18–28.

Dongarra, J. J., Du Croz, J., Duff, I. S. and Hammarling, S. (1990*b*), 'A set of Level 3 Basic Linear Algebra Subprograms.', *ACM Trans. Math. Softw.* **16**, 1–17. http://www.netlib.org/blas/blas3-paper.ps.

Duff, I. S. (1986), The use of vector and parallel computers in the solution of large sparse linear equations, *in* P. Deuflhard and B. Engquist, eds, 'Large scale scientific computing. Progress in Scientific Computing Volume 7', Birkhäuser, Boston, pp. 331–348.

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

Duff, I. S. and Reid, J. K. (1984), 'The multifrontal solution of unsymmetric sets of linear systems', *SIAM J. Scientific and Statistical Computing* **5**, 633–641.

Duff, I. S., Grimes, R. G. and Lewis, J. G. (1992), Users' guide for the Harwell-Boeing sparse matrix collection (Release I), Technical Report RAL 92-086, Rutherford Appleton Laboratory.

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

Espirat, V. (1996), Développement d'une approche multifrontale pour machines à mémoire distribuée et réseau hétérogène de stations de travail, Technical Report Rapport de stage de 3ieme année, ENSEEIHT-IRIT, Toulouse, France.

Fowler, R. F. and Greenough, C. (1998), RALPAR - RAL Mesh Partitioning Program. Version 2.0, Technical Report RAL-TR-98-025, Rutherford Appleton Laboratory.

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

HSL (1996), *Harwell Subroutine Library. A Catalogue of Subroutines (Release 12)*, AEA Technology, Harwell Laboratory, Oxfordshire, England. For information concerning HSL contact: Dr Scott Roberts, AEA Technology, 552 Harwell, Didcot, Oxon OX11 0RA, England (tel: +44-1235-434988, fax: +44-1235-434136, email: Scott.Roberts@aeat.co.uk).

Mandel, J. (1993), 'Balancing domain decomposition', *Communications in Numerical Methods in Engineering* **9**, 233–241.

Nagel, W. E., Arnold, A., Weber, M., Hoppe, H.-C. and Solchenbach, K. (1996), 'VAMPIR: Visualization and Analysis of MPI Resources', *Supercomputer* **12**(1), 69–80.