Direct Solvers for Sparse Systems

Ernst J. Haunschmid
Christoph W. Ueberhuber

Institute for Applied and Numerical Mathematics,
Technical University of Vienna,
Wiedner Hauptstrasse 8–10, A-1040 Vienna, Austria
E-Mail: {ernst,christof}@uranus.tuwien.ac.at

The work described in this report was supported by the Special Research Program SFB F011 "AURORA" of the Austrian Science Fund.

Contents

Introduction **5**

1 Ordering Sparse Matrices **6**
1.1 Orderings for Symmetric Problems 6
1.1.1 Minimum Degree Ordering 6
1.1.2 Nested Dissection 9
1.2 Orderings for Unsymmetric Problems 11
1.2.1 Markowitz Criterion 11
1.3 Software for Fill-in Reducing Ordering 11
1.3.1 HARWELL/MC47 11
1.3.2 METIS 12
1.3.3 ParMETIS 12
1.3.4 CHACO 14
1.3.5 WGPP 14

2 Direct Methods for Solving Linear Systems **16**
2.1 General Methods 16
2.2 Methods for Band Matrices 17
2.3 Variable Band Methods 18
2.4 Frontal Methods 18
2.4.1 Unifrontal Methods 19
2.4.2 Multifrontal Methods 21
2.4.3 Combined Unifrontal / Multifrontal Methods 22
2.5 Supernodal Methods 22
2.5.1 Symmetric Supernodes 22
2.5.2 Unsymmetric Supernodes 23
2.6 Static Pivoting 23

3 Software **25**
3.1 Software Packages 25
3.1.1 The PARASOL Project 26
3.1.2 PSPASES 29
3.1.3 WSSMP 30

3.1.4 Y12M . . . 32
3.1.5 UMFPACK . . . 32
3.1.6 PSSPD . . . 33
3.1.7 SuperLU . . . 33
3.1.8 SuperLU_MT . . . 33
3.1.9 SuperLU_DIST . . . 34
3.1.10 SPOOLES . . . 34
3.1.11 The Package S^+ . . . 39
3.2 Routines from Software Libraries . . . 39
3.2.1 Harwell Library . . . 39
3.2.2 IMSL Libraries . . . 44
3.2.3 NAG Libraries . . . 45
3.3 Routines from Scientific Packages . . . 46
3.3.1 PETSc . . . 46
3.3.2 MATLAB . . . 46
3.4 Matrix Collections, Utilities . . . 47
3.4.1 The Matrix Market . . . 47
3.4.2 The Harwell-Boeing Collection of Matrices . . . 47
3.4.3 The Rutherford-Boeing Collection of Matrices . . . 48
3.4.4 The University of Florida Sparse Matrix Collection . . . 48
3.4.5 SPARSE-BLAS . . . 49
3.4.6 SPARSKIT . . . 49
3.4.7 EMILY . . . 50
3.4.8 MatView . . . 50
3.4.9 SMMS . . . 51

4 Experimental Evaluation **53**
4.1 Experimental Setup . . . 53
4.1.1 Hardware Platforms . . . 53
4.1.2 Characteristics of the Test Matrices . . . 55
4.2 Performance of Serial Codes . . . 55
4.2.1 Effect of Tuning Parameters . . . 55
4.2.2 Performance of SuperLU . . . 57
4.2.3 Performance of Harwell Codes . . . 58
4.2.4 Performance of S^+ . . . 65
4.2.5 Comparison of Direct Solvers . . . 65
4.3 Performance of Parallel Codes . . . 67
4.3.1 Performance of SuperLU_MT . . . 67
4.3.2 Performance of S^+ . . . 70
4.3.3 Performance of SPOOLES . . . 71

5 Solving Large Sparse Systems Arising from IPMs **72**
5.1 The AURORA System 72
5.1.1 Problem Characteristics 73
5.1.2 Dynamic Stochastic Optimization 74
5.1.3 The Interior Point Method 74
5.1.4 Properties of the Augmented System Matrix 75
5.1.5 Properties of the Normal System 75
5.2 Numerical Experiments 78
5.2.1 Performance of Ng-Peyton Code 78
5.2.2 Performance of Harwell Codes 80
5.2.3 Performance of SuperLU 88
5.2.4 Performance of SPOOLES 89
5.2.5 Performance of the S^+ Package 89
5.2.6 Performance of PSPASES 90

Conclusion **93**

Bibliography **94**

Introduction

Solving large sparse systems of linear equations is at the core of many problems in engineering and scientific computing. Such systems are typically solved by two different types of methods—iterative methods and direct methods. The nature of the problem at hand determines which method is more suitable. A direct method for solving a sparse linear system of the form $Ax = b$ involves explicit factorization of the sparse coefficient matrix A into the product of lower and upper triangular matrices L and U. This is a highly time and memory consuming step; nevertheless, direct methods are important because of their generality and robustness. For linear systems arising in certain applications, they are the only feasible solution methods. In many other applications too, direct methods are often preferred because the effort involved in determining and computing a good preconditioner for an iterative solution may outweigh the cost of direct factorization. Furthermore, direct methods provide an effective means for solving multiple systems with the same coefficient matrix and different right-hand side vectors because the factorizations needs to be performed only once.

Chapter 1 describes several methods for ordering matrices in order to minimize fill-in during a subsequent factorization. Widely used software packages implementing these methods are presented. Chapter 2 gives an overview of direct methods for solving sparse linear systems. Software packages for solving sparse linear systems are presented in Chapter 3. Also tools and utilities for visualizing, manipulating, and converting sparse matrices are presented. Chapter 4 shows performance results for some of the software packages presented in the previous chapter. In Chapter 5 an application from the field of computational finance is introduced. Performance results for solving very large sparse linear systems arising from interior point methods (IPMs) used in that application are given.

Chapter 1

Ordering Sparse Matrices

Computing a fill-in reducing ordering of a sparse matrix seeks to generate a permutation of the columns (or rows) of a sparse matrix that minimizes the storage and computation required for the factorization of the matrix.

1.1 Orderings for Symmetric Problems

Partitioning and ordering techniques have seen major activity in recent years. Bisection and multisection techniques, extensions to orderings to block triangular form, and recent improvements and modifications to standard orderings such as minimum degree orderings are discussed in the next sections.

The most popular method for heuristically reordering sparse matrices prior to factorization is minimum degree ordering.

1.1.1 Minimum Degree Ordering

Although predated by some ten years by the paper of Markowitz [70] on unsymmetric matrices, scheme S2 in the paper by Tinney and Walker [87] established the main ordering for symmetric problems that has remained almost unchanged until present. Scheme S2 is commonly termed the *minimum degree ordering* because, at each stage, the pivot chosen corresponds to a node of minimum degree in the undirected graph associated with the reduced matrix. In matrix terms, this corresponds to choosing the entry from the diagonal that has the least number of entries in its rows within the reduced matrix.

This ordering algorithm has proved remarkably resistant to competitors and, although based only on a local criterion, does an excellent job of keeping subsequent work and fill-in low over a wide range of problems.

Minimum degree ordering is most easily described in terms of the elimination graph of A. The undirected graph $G \subset (V, E)$ of A contains a vertex $j \in V$ for every column in A, and an edge $e_{ij} \in E$ for every non-zero element a_{ij} of

A. Two vertices, i and j, are adjacent if there exists an edge e_{ij}, and the edge is incident to vertices i and j. The degree d_j of a vertex j is just the number of edges incident to j. The set $\text{adj}(j)$ is the set of vertices that are adjacent to vertex j.

The factorization of a symmetric matrix A can be depicted as a sequence of elimination graphs, $G^k = (V^k, E^k)$, where each G^k captures the non-zero structure of the matrix that remains after k columns of A have been eliminated. G^0 is the graph associated with the original matrix A.

In matrix terms, the elimination of a column causes an outer-product matrix, $\alpha z z^\top$, to be added into the remainder of A (where z is the subdiagonal part of the eliminated column). Since this matrix is symmetric, only its lower (or upper) triangular part must be computed.

In graph terms, G^k is obtained from the graph G^{k-1} by removing the node x_k corresponding to the kth eliminated column from V^{k-1}, removing all edges incident to x_k from E^{k-1}, and adding edges to E^{k-1} so that all neighbors of the eliminated node are adjacent. The added edges correspond to fill-in of the factor matrix. The sets of nodes that have been eliminated after k elimination steps will be referred to as $\bar{V}_k$.

As mentioned earlier, the minimum degree heuristic performs the ordering by eliminating at each stage k a node x_k that minimizes $\text{adj}_{G^{k-1}}(x_k)$ (where $\text{adj}_{G^k}(i)$ is the set of nodes adjacent to node i in graph G^k). Since the elimination of node x_k causes all neighbors of that node to become adjacent, the degree of a node can be used to compute a simpler upper bound on the amount of fill-in that could occur when the node is eliminated.

Of course, one could also use exact counts of fill-in rather than the bound provided by the node degree to select a node for elimination. However, this approach, known as minimum local fill-in or minimum of deficiency, is generally thought to provide limited quality advantages over minimum degree ordering while requiring significantly higher runtime.

Enhancements to Minimum Degree Ordering

As the minimum degree algorithm has evolved, several enhancements have been added to the basic algorithm. This section briefly describes widely used enhancements. A more complete discussion of these and other enhancements can be found in George and Liu [43].

Supernodes: Perhaps the most important enhancement to the minimum degree algorithm is the notion of a supervariable or supernode in G^k. A *supernode* is a set of nodes Q where for all pairs of nodes $i, j \in Q$

$$\text{adj}(i) \cup i = \text{adj}(j) \cup j.$$

Supernodes posses two crucial properties. The first is that all nodes in Q are eliminated consecutively in a minimum degree ordering. To understand the

reason for this property, note that all nodes in Q have the same degree, and that eliminating one node from Q decreases the degree of every other node in Q by one. Thus, if the degree of the original node were the minimum, then the degrees of the remaining nodes became the new minimum.

The second important property of a supernode in G^k is that it remains a supernode (or is subsumed by a larger supernode) in all subsequent graphs $G^l, l > k$.

A consequence of these two properties is that the minimum degree ordering algorithm can treat a supernode as one logical node. Supernodes can dramatically reduce the number of distinct nodes in the graph, resulting in significant reductions in ordering runtime.

Mass Node Elimination: Once a node is selected, all nodes that were in a complete subgraph (*clique*) containing that node, have degree one less and so can be eliminated without any extra fill-in, and subsequently all nodes can be eliminated. This is usually termed *mass node elimination.*

External Degree: Since there is no fill-in within a clique, a good measure of the "damage" done to the matrix by a potential pivot can be obtained from its *external degree*, which corresponds to the number of edges to nodes outside its clique. Many finite-element problems show a favorable clique structure, so it is quite natural to perform the minimum degree ordering on a graph where nodes in the same clique are treated as a single node.

Multiple Elimination: Another important enhancement to the minimum degree heuristic is multiple elimination (Liu [67]). In the *multiple minimum degree* (MMD) algorithm, an independent set of nodes of minimum degree is eliminated simultaneously. The degrees of the neighbors of these nodes are updated once the multiple elimination is complete. The benefit of multiple elimination is that it reduces the frequency of degree updates, since a node typically changes its degree only after multiple neighbors have been eliminated.

Approximate Node Degrees: A main issue to an efficient implementation of MMD is the update of the degree counts. Amestoy et al. [4] have designed an *approximate minimum degree* (AMD) ordering where the bound is equal to the degree in many cases. They have found that their AMD ordering is almost indistinguishable from the minimum degree ordering in quality but is very much faster to compute.

An unfavorable property of minimum degree ordering is that it may produce elimination trees that are not well balanced and so are not well suited as computational graphs for controlling parallel algorithms.

The use of dissection techniques (as described in Section 1.1.2) often gives much better balanced trees, although the inferior performance of the early dissection codes needs to be addressed for them to be viable.

Column Minimum Degree Ordering

Most column minimum degree ordering codes for asymmetric matrices are based on a symmetric minimum degree ordering. To reorder the columns of an asymmetric matrix A, a symmetric MMD ordering can be used on $A^\top A$. But each row of A induces a clique in $A^\top A$, so that A itself can be used to represent $A^\top A$ instead of forming the product explicitly. Such a clique representation of a symmetric graph is called the *generalized element* representation (Speelpenning [83]), or *quotient graph model* (George and Liu [43]).

1.1.2 Nested Dissection

As mentioned before, minimum degree algorithms are based on a local criterion. Dissection orderings, on the other hand, take a global view of the problem. The essence of a dissection technique is a bisection algorithm that divides the graph of the matrix into two partitions. If node separators are used, a third set will correspond to the node separators. Such a bisection is then used repeatedly in a nested fashion to obtain an ordering for the matrix.

Spectral Bisection

Various formulations of spectral bisection can be found in literature (Boppana [15], Pothen et al. [76], Simon [81]). One way to describe a partition is to assign a value of $+1$ to all the vertices in one set and a value of -1 to all the vertices in the other. If one denotes the value assigned to vertex i by $x(i)$, then the size of the cut set C corresponding to a partition vector x can be expressed as

$$|C| = \frac{1}{4} \sum_{(i,j)\in E} (x(i) - x(j))^2.$$

It is well-known that

$$\sum_{(i,j)\in E} (x(i) - x(j))^2 = x^\top L x,$$

where L is the Laplacian matrix of the graph G. $L = (l_{ij})$ is defined as

$$l_{ij} = \begin{cases} -1 & \text{if} \quad (i,j) \in E \\ d_i & \text{if} \quad i = j \\ 0 & \text{otherwise.} \end{cases}$$

This allows to write the graph partitioning problem as

$$\text{Minimize} \quad \frac{1}{4}x^\top Lx \quad \text{subject to} \quad x^\top 1_n = 0 \quad \text{and} \quad x_i = \pm 1, \tag{1.1}$$

where $1_n = (1, 1, \ldots, 1)^\top$. Since graph partitioning is an NP-hard problem one expects that there is no practical way to solve the minimization problem (1.1). By relaxing the discreteness constraint to $x^\top x = n$, one gets a continuous approximation that makes the algebraic problem much easier to solve. Fiedler [41] has shown that the eigenvector x_2, normalized as $||x_2||^2 = n$, corresponding to the largest non-zero eigenvalue solves the continuous problem. Hence this eigenvector is called *Fiedler vector* for short.

The solution of the continuous problem must then be mapped back to ± 1 by some appropriate scheme to define a partition. The median cut spectral bisection method chooses the median cut vector which can be obtained by finding the median value of the components of the Fiedler vector and mapping values above the median to $+1$ and values below to -1.

The actual computational challenge of the spectral bisection algorithm is the effective computation of the Fiedler vector. Normally some variant of the Lanczos algorithm is used since it does not require any manipulation of the Laplacian matrix.

There are clearly two conflicting goals: (i) balancing the bisection and (ii) minimizing the number of edges joining each set (or if a node separator is used, minimizing the number of nodes in the separator). For details see, e.g., Rothberg [78].

The spectral method requires much computing time, does not always yield optimal bisections, and naturally produces edge separators, requiring postprocessing to obtain a node separator set. For these reasons, spectral bisection is not strongly favored now, and there has been much research on alternatives.

Multilevel Schemes

One main approach to graph bisection is to perform graph reduction, compute a partition cheaply on the resulting coarse graph, and derive a partition of the original graph, using some kind of iterative improvement on the projection of this coarse partition onto the finer graph. This approach is nested and is termed a *multilevel scheme*. Multilevel schemes have been used by Hendrickson and Leland [55], Karypis and Kumar [62], and Hendrickson and Rothberg [57].

In most of these approaches, the dissection technique is only used for the top levels and the resulting subgraphs are ordered by a minimum degree scheme. This hybrid technique is used in many implementations (for example, Ashcraft and Liu [11] and Hendrickson and Rothberg [57]). In several studies on problems from structural analysis, Rothberg [78] and Ashcraft and Liu [11] have shown that dissection techniques can outperform minimum degree methods by an average of

15 % in terms of floating-point operations for Cholesky factorization using the resulting ordering, although the cost of these orderings is several times that of minimum degree.

Of course, dissection techniques are important for purposes other than generating an ordering for a Cholesky factorization. They can be used to partition an underlying grid for domain decomposition and are equally useful for the parallel implementation of many iterative methods.

Two of the major software products implementing graph partitioning based on the above mentioned techniques are CHACO (Hendrickson and Leland [56]) and METIS (Karypis and Kumar [62]).

1.2 Orderings for Unsymmetric Problems

The evolution that was just discussed for symmetric orderings has not happened in a similar way for unsymmetric systems, where usually a Markowitz ordering (Markowitz [70]) or a column ordering based on minimum degree techniques applied to the normal equations is used together with a threshold criterion for numerical pivoting.

1.2.1 Markowitz Criterion

The local fill-in is approximated most often by the so-called *Markowitz costs*, which are defined as follows. If $a_{ij}^{(k-1)}$ is chosen as the pivot element in the kth step of the Gaussian elimination process, then its Markowitz cost

$$\mathrm{MK}(a_{ij}^{(k-1)}) = \left(\text{non-zero}(z_{i+1-k}^{(k-1)}) - 1\right)\left(\text{non} - \text{zero}(s_{j+1-k}^{(k-1)}) - 1\right)$$

is an upper bound for the local fill-in. $z_i^{(k-1)}$ denotes the ith row and $s_j^{(k-1)}$ the jth column of the adjacency matrix B_k of the $(n-k+1) \times (n-k+1)$ principle minor of $A^{(k-1)}$.

1.3 Software for Fill-in Reducing Ordering

1.3.1 HARWELL/MC47

The MC47 code from the Harwell Subroutine Library (see Section 3.2.1) implements a minimum degree based ordering. Given a representation of the nonzero pattern of a symmetric matrix A, MC47 performs an approximate minimum degree ordering to compute a pivot order so that the number of the nonzeros in the Cholesky factors of A is kept low. At each step, the pivot selected is the one that minimizes an easily computed upper-bound on the external degree. Supervariable detection and mass elimination are also used. A permutation corresponding

to this ordering is returned, together with information to assist the subsequent numerical factorization of the matrix.

MC47 is typically faster than other minimum degree algorithms and produces comparable results to other minimum external degree algorithms in terms of fill-in and the number of floating-point operations needed to compute the Cholesky factorization. A full account of this method is given by Amestoy et al. [3].

MC47 is the symmetric analogue of the ordering used in the code of Davis and Duff [20].

1.3.2 METIS

METIS is a software package for partitioning large irregular graphs, partitioning large meshes, and computing fill-in reducing orderings of sparse matrices. The algorithms in METIS are based on multilevel graph partitioning techniques described in Karypis et al. [62]. METIS is written in ANSI C, and is easily ported to most UNIX systems.

METIS provides two programs, `oemetis` and `onmetis`, for computing fill-in reducing orderings of sparse matrices. Both of these programs use multilevel nested dissection to compute a fill-in reducing ordering. The nested dissection paradigm is based on computing a vertex-separator for the graph corresponding to the matrix. The nodes in the separator are moved to the end of the matrix, and a similar process is applied recursively for each one of the other parts.

Even though both METIS programs are based on multilevel nested dissection, they differ on how they compute the vertex separators. `oemetis` finds a vertex separator by first computing an edge separator using a multilevel algorithm, whereas `onmetis` uses the multilevel paradigm to directly find a vertex separator. The orderings produced by `onmetis` generally incur less fill-in than those produced by `oemetis`. In particular, for matrices arising in linear programming problems the orderings computed by `onmetis` are significantly better than those produced by `oemetis`. Furthermore, `onmetis` utilizes compression techniques to reduce the size of the graph prior to computing the ordering. Rows with the same sparsity structure can be represented by a single vertex whose weight is equal to the number of these rows. Such compression techniques can significantly reduce the size of the graph and substantially reduce the amount of time required by `onmetis`. However, when there is no reduction in graph size, `oemetis` is about 20 % to 30 % faster than `onmetis`.

Source code, documentation and technical reports are available over the internet from `www-users.cs.umn.edu/~karypis/metis`.

1.3.3 ParMETIS

PARMETIS is am MPI-based parallel library that implements a variety of al-

gorithms for partitioning unstructured graphs and for computing fill-in reducing orderings of sparse matrices. ParMetis is particularly suited for parallel numerical simulations involving large unstructured meshes. In this type of computation, ParMetis dramatically reduces the time spent in communication by decomposing the mesh and distributing it among the processors in a way that minimizes the number of interface elements.

The algorithms in ParMetis are based on multilevel partitioning and fill-in reducing ordering algorithms that are implemented in the serial package (see Section 1.3.2). ParMetis extends the functionality provided by Metis and includes routines that are especially suited for parallel computations and large scale numerical simulations. In particular, ParMetis provides the following four major functions:

- Partition an unstructured graph;
- Improve the quality of an existing partition;
- Repartition a graph that corresponds to an adaptively refined mesh;
- Compute a fill-in reducing ordering for sparse direct factorization.

Source code, documentation and technical reports are available over the internet from `www-users.cs.umn.edu/~karypis/metis/parmetis`.

Computing Fill-in Reducing Orderings

ParMetis provides `parometis` for computing a fill-in reducing ordering, suited for Cholesky factorization algorithms. `parometis` makes no assumptions on how the graph is initially distributed among the processors. It can effectively compute a fill-in reducing ordering for a graph that is randomly distributed as well as a graph that is nicely distributed.

The algorithm implemented by `parometis` is based on a multilevel nested dissection algorithm. This algorithm has been shown to produce low fill-in orderings for a wide variety of matrices. Furthermore, it leads to balanced elimination trees which is essential for parallel direct factorization. To achieve high performance, `parometis` first uses `parkmetis` to compute a high quality partition and accordingly redistributes the graph. Next it proceeds to compute the $\log p$ levels of the elimination tree concurrently, where p is the number of processors. When the graph has been separated into p parts, the graph is redistributed among the processors so that each processor receives a single subgraph, and a multiple minimum degree algorithm is used to order these smaller subgraphs.

1.3.4 CHACO

CHACO is a software package for graph partitioning. It allows for recursive application of several methods for finding small edge separators in weighted graphs. These methods include inertial (Simon [81]), spectral (Pothen et al. [76], Hendrickson and Leland [54]), Kernighan-Lin (Kernighan and Lin [63]) and multilevel (Hendrickson and Leland [55]) methods in addition to several simpler strategies. Each of these approaches can be used to partition the graph into two, four or eight pieces at each level of recursion. In addition, the Kernigham-Lin method can be used to improve partitions generated by any of the other algorithms.

Since CHACO (cf. User's Guide [56]) comprises a suite of different algorithms, the user can try different methods to find the one that works best for the specific problem. CHACO can be run on both symmetric multiprocessing (SMP) and massively parallel processing (MPP) platforms.

CHACO has been developed by B. Hendrickson (`bah@cs.sandia.gov`) and R. Leland (`leland@cs.sandia.gov`) at Sandia National Laboratories.

The source code is distributed along with technical documentation and sample input files. The code is available under research or commercial license, and enquiries should be directed to the authors. The CHACO Users's Guide and publications related to CHACO are available from `www.cs.sandia.gov/CRF/papers_chaco.html`.

1.3.5 WGPP

The *Watson Graph Partitioning Package* (WGPP) is a package of subroutines for partitioning graphs and computing fill-in reducing orderings of sparse matrices for their factorization. WGPP contains a collection of fast and high quality routines for unstructured graph partitioning and for generating robust fill-in reducing orderings for sparse matrices arising in different application domains.

The WGPP routines provide some useful options and added functionality that is absent from other packages. Some examples are:

- Options for preprocessing high-degree nodes and for compressing graphs with multiple degrees of freedom;
- Ability to generate the better of minimum degree and graph partitioning based ordering in a single call;
- Limited access to internal data structures through an advanced interface;
- User control over maximum tolerable imbalance among partitions;
- Some user control over sparse matrix ordering quality and ordering time tradeoff.

WGPP has been developed by A. Gupta at IBM Watson Research Center. It is available for IBM, SUN, and SGI systems; only the library file (no source code) is distributed.

The package can be obtained from the author by sending an e-mail to `anshul@watson.ibm.com`.

Chapter 2

Direct Methods for Solving Linear Systems

The use of a good pivotal ordering in Gaussian elimination significantly reduces both memory requirements and floating-point computation. This chapter discusses implementation strategies for achieving these goals without unreasonable overhead. Gaussian elimination can be divided into three phases:

Analysis phase, where the sparsity structure of the matrix is analyzed to find an ordering that preserves sparsity;

Factorization phase, where factor matrices are computed using information from the analysis phase;

Solution phase, where the factor matrices are used to solve the given system of linear equations.

Implementations can be divided broadly into two categories according to whether or not the analysis phase uses given numerical values.

Section 2.1 discusses the features of general solvers like MA48 (Duff and Reid [38]) from the Harwell Subroutine Library and Y12M (Zlatev et al. [92]).

The unifying theme of the remaining sections of this chapter is that an initial pivotal sequence is chosen from the sparsity pattern alone and is used unmodified (or only slightly modified) in the subsequent factorization phase.

2.1 General Methods

The typical features of the general approach are that pivoting for numerical stability is combined with pivoting for sparsity, and that sparse data structures are used throughout, even in the inner loops. As explained later, these features are drawbacks with respect to vectorization and parallelization.

To minimize fill-in during the elimination process the Markowitz criterion is used to select pivot elements. To control numerical stability the Markowitz selection is restricted to those pivot candidates satisfying

$$||a_{kk}|| \geq u||a_{ik}||, \quad i > k,$$

where $u \in (0, 1)$ is a preset threshold parameter, chosen to restrict the maximum possible growth in the numerical value of a matrix entry in a single step of Gaussian elimination to $(1 + 1/u)$. In practice, $u = 0.1$ has been found to provide a good compromise between maintaining stability and having the freedom to reorder to preserve sparsity. More details are given in Duff et al. [31].

The strength of the general approach is its satisfactory performance over a wide range of different sparsity structures. Therefore this approach is often the method of choice for many sparse unstructured problems. Some simplifications and performance gains can be obtained if the matrix is symmetric or banded.

MA48 from the Harwell Subroutine Library (Duff and Reid [38] and Y12M (Zlatev et al. [92] are typical implementations of the general method.

2.2 Methods for Band Matrices

Band matrices lead to very favorable performance behavior if Gaussian elimination without pivoting is applied. In this case no fill-in outside the original band is generated.

For symmetric positive definite matrices, no pivoting is needed to ensure numerical stability. In the unsymmetric case, interchanges are not needed if the matrix is diagonally dominant. In this case the band structure is preserved.

At the risk of numerical instability, one may continue to use these methods for matrices that are symmetric but not definite or are unsymmetric but not diagonally dominant. It is straightforward to check for instability by monitoring the size of each newly computed matrix entry (see Duff et al. [31]). Alternatively row interchanges can be applied to the matrix, thereby destroying symmetry in the symmetric case and increasing the bandwidth of U from $m + 1$ to $2m + 1$ in both instances. This information about band matrix fill-in can be exploited by providing enough storage locations for possible fill-in at the very beginning of program execution.

Banded systems occur very frequently in practice, sometimes as subproblems within larger computations. For example, methods for partial differential equations often solve many independent one-dimensional subproblems, each of which has a banded set of equations. The simplicity of such systems and the fact that they arise frequently has given rise to many special methods. Duff [30] discusses some of them, considering in particular techniques that are efficient on vector and parallel architectures.

2.3 Variable Band Methods

As for band matrices, a variable band form (also called skyline structure), is preserved during Gaussian elimination if no row or column interchanges are performed. No interchanges are needed, for example, when the matrix is symmetric and positive definite. In typical implementations (for example, in SPARSPAK) the lower triangular part of the matrix is stored by rows almost as conveniently as if the bandwidth were fixed. Additionally, a set of pointers to the positions of the diagonal elements is stored.

The difficulty of Gaussian elimination by rows is that there may be some zeros within the pivot row and the elements in the column being computed. In this case explicit zeros must be stored and used in later steps. There will be some multiplications by zero, operations that should be avoided in sparse matrix computations.

As shown in the next section, storing zeros explicitly and performing operations on them may improve overall performance, provided these operations can be done at higher performance rates (for instance, by using dense matrix BLAS-3 kernels) and thus amortizing the computational overhead. Full details on variable band methods and block band methods are given in Duff [31].

2.4 Frontal Methods

Frontal methods have their origin in the solution of finite-element problems in structural mechanics, but the are not restricted to this application nor need the matrix be symmetric and positive definite. It is, however, easiest to describe frontal methods in terms of this application (Irons [58]).

In a finite-element problem

$$A = \sum_l A^{[l]}, \tag{2.1}$$

where each $A^{[l]}$ has entries only in the principal submatrix corresponding to the variables in element l and represents the contributions from this element. The formation of the sum (2.1) is called *assembly* and involves the elementary operation

$$a_{ij} := a_{ij} + a_{ij}^{[l]}. \tag{2.2}$$

An entry is called *fully summed* when all contributions of the form (2.2) have been summed.

The basic operation of Gaussian elimination

$$a_{ij}^{(k+1)} := a_{ij}^{(k)} - a_{ik}^{(k)} {a_{kk}^{(k)}}^{-1} a_{kj}^{(k)}. \tag{2.3}$$

can be performed before all the assemblies (2.2) are complete, provided that the terms in the triple product in (2.3) are fully summed. Each variable can be

eliminated as soon as its row and column is fully summed, that is after its last occurrence in a matrix $A^{[l]}$. If this is done, the elimination operations will be confined to the submatrix of rows and columns corresponding to variables that have not yet been eliminated but are involved in one ore more of the elements that have been assembled. This permits all intermediate computations to be performed in a full matrix whose size increases when a variable appears for the first time and decreases when a variable is eliminated.

The pivotal order is determined from the order of assembly. If the elements are ordered systematically from one end of the region to the other, the active variables form a *front* that moves along it. For this reason the full matrix in which all arithmetics is performed is called the *frontal matrix* and the technique is called the *frontal method*.

As mentioned before, frontal methods are not restricted to finite-element problems. Indeed, the frontal method applied to a general (fully summed) problem can be viewed as a generalization of the variable band method (see Section 2.2). Therefore, such problems are called *assembled problems* (regardless whether they originate from a finite-element problem or not) and the corresponding system matrix is called an *assembled* matrix.

For assembled finite-element problems and general, non-finite-element problems, the rows may be taken as if they were unsymmetric finite-element matrices. The assembled rows are then always fully summed and a column becomes fully summed whenever the row containing its last nonzero is reached.

2.4.1 Unifrontal Methods

In a unifrontal scheme, the factorization proceeds as a sequence of partial factorizations and eliminations on dense submatrices, called frontal matrices. Although unifrontal methods were originally designed for the solution of finite-element problems (Irons [58]), they can be used on assembled matrices as well (Duff [29]) and only this version will be discussed in the following. For assembled systems, the frontal matrix can be written as

$$\begin{pmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{pmatrix}, \tag{2.4}$$

where all rows are fully summed (that is, there will be no further contributions to the rows in (2.4)) and the first block column is fully summed. This means that pivots can be chosen from anywhere in the first block column and, within these columns, numerical pivoting with arbitrary row interchanges can be accommodated since all rows in the frontal matrix are fully summed. It is assumed, without loss of generality, that the pivots that have been chosen are in the submatrix F_{11} of (2.4). F_{11} is factorized, the Gaussian elimination multipliers overwrite F_{21} and the Schur complement

$$F_{22} - F_{21}F_{11}^{-1}F_{12} \tag{2.5}$$

is formed using dense matrix kernels. The submatrix consisting of the rows and columns of the frontal matrix from which pivots have not yet been selected is called the *contribution block*. In the unifrontal case, this is the same as the Schur complement matrix (2.5).

At the next stage, further rows from the original matrix are assembled with the Schur complement to form another frontal matrix. The frontal matrix is extended in size, if necessary, to accommodate the incoming rows. The overhead is low (compared to a multifrontal method; see Section 2.4.2) since each row is assembled only once and there is never any assembly of two (or more) frontal matrices. The entire sequence of frontal matrices is held in the same working array. Data movement is limited to assembling rows of the original matrix into the frontal matrix, and storing rows and columns as they become pivotal. There is never any need to move or assemble the Schur complement into another working array. One important advantage of this method is that only this single working array resides in main memory. Rows of A can be read sequentially from disk into the frontal matrix. Entries in L and U can be written sequentially to disk in the order they are computed. A detailed description of frontal methods for assembled problems is given by Zitney [91].

The unifrontal method works well for matrices with small profile, where the profile of a matrix is a measure of how close the nonzero entries are to the diagonal and is defined as

$$\sum_{i=1}^{n} (\max_{a_{ij} \neq 0} (i - j) + \max_{a_{ji} \neq 0} (i - j)), \tag{2.6}$$

where it is assumed that the diagonal is nonzero so all terms in (2.6) are non-negative. For matrices that are symmetric or nearly so, the unifrontal method is typically preceded by an ordering method to reduce (2.6). For example, the reverse Cuthill-McKee (RCM) ordering is used (Cuthill and McKee [18]). This is typically faster than the sparsity preserving orderings commonly used by a multifrontal method (such as nested dissection and minimum degree orderings). However, for matrices with large profile, the frontal matrix can be large, and an unacceptable amount of fill-in can occur. In particular, effective profile reduction strategies for matrices whose pattern is very unsymmetric is lacking.

The unifrontal scheme can easily accommodate numerical pivoting. Because all rows in (2.4) are fully summed and regular partial pivoting can be performed, it is always possible to choose pivots from all the fully summed columns (unless the matrix is structurally singular).

As is the case for band techniques, frontal methods may be applied to any matrix. Of course, their success depends on the frontal size remaining small.

Unifrontal methods tend to produce more fill-in and require more arithmetic operations than local methods based on minimum degree orderings.

Two of the main reasons favoring band methods also support frontal techniques: (i) only the active part of the band or the frontal matrix need to be

held in main memory, and (ii) the arithmetic operations are performed on full matrices without any indirect addressing or complicated adjustments of data structures to accommodate fill-in. The avoidance of indirect addressing facilitates vectorization and use of Level 2 BLAS routines.

It is important to keep the size of the frontal matrix small since the work involved in an elimination step is quadratic in the size of the front.

2.4.2 Multifrontal Methods

This section discusses a generalization of the unifrontal method that is able to accommodate any ordering scheme based on the structure of the matrix, including minimum degree and nested dissection. Because more than one front is involved, it is called *multifrontal* method.

In a multifrontal method for a matrix with a symmetric sparsity structure, it is common to use an ordering such as minimum degree to reduce fill-in. Orderings like minimum degree tend to reduce fill-in much more than profile reduction orderings. The ordering is combined with a symbolic analysis to generate an assembly tree, where each node represents the elimination operations on a frontal matrix and each edge represents an assembly operation. When using the tree to drive the numerical factorization, the only requirement is that eliminations at any node cannot complete until those at the child nodes have been completed, giving added flexibility for issues such as exploitation of parallelism. As in the unifrontal scheme, normally the frontal matrix (2.4) cannot be factorized but only a few steps of Gaussian elimination are possible, after which the Schur complement (2.5) needs to be assembled with other data at the parent node.

In a multifrontal method for a matrix with an unsymmetric sparsity structure, the ordering, symbolic analysis, and numerical factorization are performed at the same time. The tree is replaced by a directed acyclic graph (DAG). A contribution block may be assembled into more than one subsequent frontal matrix.

The first pivot within a frontal matrix (called the *seed pivot* by Davis and Duff [22]) defines its size. This new frontal matrix is held in a larger working array, to allow room for the assembly of subsequent pivot rows and columns. The next pivots can either be taken from the fully summed part, or the non-fully summed part. If a potential pivot lies in the non-fully summed part of the frontal matrix then it is necessary to sum the row and the column containing it before this element can be used as a pivot. This is possible as long as its fully summed rows and columns can be accommodated within the larger working array, along with the contribution block and all previous pivot rows and columns of the frontal matrix. This is similar to the node amalgamation used in the symmetric-pattern multifrontal method, except that here the amalgamation is determined during numerical factorization, rather than during the symbolic analysis.

2.4.3 Combined Unifrontal / Multifrontal Methods

In the multifrontal method, work on a frontal matrix can be suspended, the matrix can be stored for later reuse, and a new frontal matrix can be generated. Accordingly, several frontal matrices are stored during the factorization and one or more of these are assembled (summed) when creating a new frontal matrix. Although this means that arbitrary sparsity patterns can be handled efficiently, extra work is required to sum the frontal matrices and thus the method can be costly because indirect addressing is required. The unifrontal method avoids this extra work by factorizing the matrix with a single frontal matrix. Rows and columns are added to the frontal matrix, and pivot rows and columns are removed. Data movement is simpler, but higher fill-in can result if the matrix cannot be permuted into a variable-band form with small profile.

Therefore, a technique has been developed that combines the favorable features of the two methods: (i) the lower data movement of unifrontal methods, and (ii) the lower fill-in of multifrontal methods.

This combined unifrontal/multifrontal method is implemented in UMFPACK Version 2 and Harwell/MA38.

2.5 Supernodal Methods

The idea of a supernode is to group together columns having the same sparsity structure, so that these columns can be treated as a dense matrix concerning storage and computation.

2.5.1 Symmetric Supernodes

Supernodes were originally used for sparse Cholesky factorization; the first published results are by Ashcraft et al. [9]. In the factorization $A = LL^\top$, a supernode is a range $(r : s)$ of columns of L with the same sparsity structure below the diagonal; that is, $L(r : s, r : s)$ is lower triangular, and every row of $L(s : n, r : s)$ is either dense or zero.

Ng and Peyton [71] analyzed the effect of supernodes in Cholesky factorization on modern uniprocessor machines with memory hierarchies and vector or superscalar hardware. All the updates from columns of a supernode are summed into a dense vector before the sparse update is performed. This reduces indirect addressing, and allows the inner loops to be unrolled. In effect, a sequence of column-column updates is replaced by a supernode-column update. The supernode-column update can be implemented using a call to a standard dense Level 2 BLAS kernel. This idea can be further extended to supernode-supernode updates, which can be implemented using dense Level 3 BLAS kernels. This technique reduces memory traffic by an order of magnitude, because a supernode in the cache can participate in multiple column updates. Ng and

Peyton reported that a sparse Cholesky algorithm based on supernode-supernode updates typically runs 2.5 to 4.5 times as fast as column-column algorithms. Indeed, supernodes have become a standard tool in sparse Cholesky factorization (see, for example, Rothberg and Gupta [79] or Simon et al. [82]).

To sum up, supernodes are advantageous for the following reasons:

1. The inner loop (over rows) has no indirect addressing. Thus sparse Level 1 BLAS can be replaced by dense Level 1 BLAS .
2. The outer loop (over columns in the supernode) can be unrolled to increase locality of memory references. Level 1 BLAS are replaced by Level 2 BLAS.
3. Elements of the source supernode can be reused in multiple columns of the destination supernode to reduce cache misses. Level 2 BLAS are replaced by Level 3 BLAS.

Supernodal methods allow the use of dense Level 3 BLAS for nearly all floating-point computations. The flop rates of modern sparse Cholesky factorization codes are nearly comparable to those of dense solvers (Ng and Peyton [71]).

2.5.2 Unsymmetric Supernodes

Supernodes in sparse Cholesky algorithms can be determined during symbolic factorization, before the numerical factorization starts. However, in sparse LU factorization, the nonzero structure cannot be predicted before numerical factorization, so supernodes must be identified on the fly. Furthermore, since the factors L and U are no longer transposes of each other, the definition of a supernode must be generalized.

Several ways to generalize the definition of supernodes to unsymmetric factorization have been suggested (for example, five different definitions are compared in Li [65]).

Supernodes are only useful if they actually occur in practice. The occurrence of symmetric supernodes is related to the clique structure of the chordal graph of the Cholesky factor, which arises because of fill-in during the factorization. Unsymmetric supernodes are harder to characterize, but they are also related to dense submatrices arising from fill-in.

2.6 Static Pivoting

Traditionally, partial pivoting is used to control the element growth during Gaussian elimination, making the algorithm numerically stable in practice. However, partial pivoting is not the only way to control element growth; there are a variety

of alternative techniques: pre-pivoting large elements to the diagonal, iterative refinement, using extra precision when needed, and allowing low rank modifications with corrections at the end of the computation. SuperLU_dist implements all these techniques to control the element growth during Gaussian elimination.

Currently, the code is under development; more information can be found at `www.nersc.gov/~xiaoye`.

Chapter 3

Software

As for linear systems with dense coefficient matrices, there are also commercial software products and public-domain software for large, sparse systems. Both offer facilities for storing and manipulating sparse matrices as well as methods for solving systems of linear equations.

A major difference between software for sparse linear algebra and software for dense linear algebra is that the standardization of program interfaces and functional properties of the software modules does not yet exist. This is due to the fact that the data structures for storing matrices and efficient methods for solving problems greatly depend on the problem itself.

3.1 Software Packages

Although much software is available that implements direct methods for solving sparse linear systems, little is within public domain. There are two reasons for this situation:

- often software for sparse systems is encapsulated within much larger packages (for example, structural engineering packages);
- often the work on developing sparse codes is funded commercially.

There are also several research codes that can be obtained from the authors but, since these usually lack full documentation and often require the support of the authors, these codes are not discussed in the following sections. For example, Liu's MMD ordering code is available from the author (`joseph@cs.yorku.ca`) upon request.

Among the public domain software for sparse systems there are some routines from the *Collected Algorithms of ACM* (available from NETLIB), mostly for matrix manipulation (for example, bandwidth reduction, ordering to block triangular form) rather than for solving linear systems.

Both Y12M and the Harwell Subroutine Library code MA28 are available from NETLIB, although the use of the newer and improved Harwell code MA48 is recommended. A research version of MA38, called UMFPACK, and SUPERLU, a C code implementing the supernodal factorization of Li [65] is available from NETLIB too. There are also some skeleton codes in the `misc` and the `linalg` collection in NETLIB.

Among the codes available under license are those from the Harwell Subroutine Library, a subset of which is also marketed by NAG under the title *Harwell Sparse Matrix Library*. The IMSL as well as the NAG library have some codes for the direct solution of sparse linear systems.

There are only a few packages for parallel computers: PSPASES [60] uses MPI for solving positive definite sparse linear systems. SUPERLU_MT [65] is a public domain version of SUPERLU for shared memory parallel computers. The Harwell code MA41 has also a version for shared memory computers.

PARASOL (a long term research project funded by the EU) develops a suite of direct and iterative sparse solvers for message passing systems that are primarily targeted to finite element applications.

The codes will be made publicly available at `www.genias.de/parasol`.

3.1.1 The PARASOL Project

PARASOL is an ESPRIT IV long term research project whose main goal is to build and test a portable library for solving large sparse systems of equations on distributed memory systems. The programs are written in Fortran 90 and use MPI for message passing. There are both direct and iterative routines for solving symmetric and non-symmetric sparse systems. The final library will be in the public domain.

The PARASOL Consortium is managed by PALLAS (`www.pallas.de`) in Germany and consists of

- leading European research organizations with internationally recognized experience and an established track in the development of parallel solvers (CERFACS, GMD-SCAI, ONERA, RAL, University of Bergen);
- industrial code developers who define the requirements for PARASOL, provide test cases generated by their finite-element packages, and will use the developed software in production mode (Apex Technologies, Det Norske Veritas, INPRO, MacNeal-Schwendler, 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 web site at `www.genias.de/parasol`.

The codes in the PARASOL library include direct methods, domain decomposition techniques, and multigrid algorithms. Within this project, RAL and CERFACS are involved in the development of direct solvers and are working in this context in close collaboration with ENSEEIHT-IRIT in Toulouse, France.

The PARASOL library contains four different solvers:

- MUMPS is a direct multifrontal sparse solver developed at RAL and CERFACS;
- HS is a hierarchical iterative solver developed at GMD;
- DDM is an iterative solver based on domain decomposition techniques developed at Parallab, University of Bergen;
- FETI is an iterative solver based on the dual Schwarz method developed at ONERA.

The PARASOL library provides its own communication and data exchange routines (the PARASOL interface) as a higher level message passing protocol based on MPI and designed for specific data structures that arise in finite-element approximations of partial differential equations and operations with sparse matrices.

The following sections discuss the PARASOL interface library and some aspects relevant to the implementation of MUMPS.

The PARASOL Interface

Different solvers and algorithms for the solution of sparse linear systems are included in the PARASOL library. The project group has designed and uses a higher level message passing protocol in order to overcome difficulties arising from the different data structures typically used by different algorithms. As the PARASOL library is designed for a distributed memory environment it uses the message passing interface MPI 1.1 as the basis for its own interface definitions. The PARASOL interface allows users and solver developers to exchange data between different modules of the package in a precisely defined way that is independent of the specific data structures needed internally in the user's codes or solvers.

PARASOL users have to define, configure and terminate the parallel environment in addition to controlling I/O resources. However, the PARASOL library offers tools for controlling synchronization of data exchanges. Every PARASOL routine is passed an instance descriptor as an argument and then performs operations only on data associated with this instance. The PARASOL library allows the user to run several PARASOL instances simultaneously. The instance operates on its own set of private data structures, although it can initialize and use other instances. This feature provides an easy mechanism for integrating different solvers

from the library into a new solver. For example, the domain decomposition solver can call the sparse direct solver. This new solver will use all parallel features of both solvers, without changing the data structures.

The PARASOL interface guide (Supalov [85]) gives a detailed description of the data exchange protocol and the data structures defined for PARASOL routines.

MUMPS

MUMPS (MUltifrontal Massively Parallel Solver) is a parallel sparse solver for distributed memory architectures using a multifrontal method (see Section 2.4.2). Details can be found in Amestoy and Duff [5], Duff and Reid [36], and in Duff and Reid [37].

The current version of MUMPS (Version 2.0) solves the system $Ax = b$, assuming that A is assembled and unsymmetric. The solution of a given problem is computed in three phases.

Analysis phase: The structure of the matrix is first analyzed to determine an ordering that, in the absence of any numerical pivoting, will preserve sparsity in the factors. In Version 2.0 of MUMPS, a minimum degree ordering strategy is used on the pattern of $A + A^\top$, and the analysis phase produces both an ordering and an assembly tree. The assembly tree is used to control the subsequent factorization and solution process.

Factorization phase: At each node of the tree, a dense submatrix (the frontal matrix) is assembled using data from the original matrix and from the sons of the tree node. Pivots can be chosen from within a submatrix of the frontal matrix (called the *fully summed block*; see Section 2.4) and eliminations are performed. The resulting factors are stored for use in the solution phase and the Schur complement (the *contribution block*) is passed to the father node of the tree 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, a forest is obtained, and each component tree of the forest will be treated similarly and independently. 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 son must complete its elimination operations before the father can be fully processed. That freedom gives the possibility to exploit parallelism in the tree.

Solution phase: 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.

Because the matrices are unsymmetric, threshold pivoting is used in the numerical factorization phase to maintain numerical stability so that it is possible that the

pivots selected by the analysis phase are unsuitable. Pivots can be chosen from anywhere within the fully summed block (including off-diagonal pivots) but it still may not be possible to eliminate all variables from this block. The result is that the Schur complement that is passed to the father node may be larger than anticipated by the analysis phase and so the resulting data structure may be different from those forecast by the analysis phase. These implies that a dynamic scheduling during numerical factorization is needed.

Sparse matrices are represented within the PARASOL library in the compressed column storage format.

The basic process within the PARASOL environment is called *an instance* and is described by its descriptor, which also encodes an MPI communicator used for data exchanges related to a given instance.

The underlying algorithm is a multifrontal one with a switch to SCALAPACK processing [16] towards the end of the factorization (and solution).

The codes will be made publicly available at `www.genias.de/parasol`.

3.1.2 PSPASES

PSPASES (Parallel SPArse Symmetric dirEct Solver) is an MPI based parallel stand-alone library intended to solve systems of linear equations with a sparse symmetric positive definite coefficient matrix.

PSPASES can be used on any parallel computer or network of workstations equipped with MPI, Fortran 90 and C language compilers. It has been tested on IBM SP computers, networks of IBM RS/6000 workstations, SGI Power Challenge, SGI Origin 2000, and Cray T3E computers.

The package is available from `www-users.cs.umn.edu/~mjoshi/pspases`.

Data Distribution

It is assumed that a system of order n is solved using $p = 2^k, k = 1, 2, 3, \ldots$ processors.

The input matrix A is assumed to be distributed among p processors using a block cyclic partitioning. The right hand side matrix B and the solution matrix X are also assumed to be distributed among processors using a block cyclic partitioning. The partitioning used for B and X must be identical, although the partitioning of A and B may be different. The local parts of the input matrix A are stored using a slightly modified and extended CRS format. The local parts of B and X are stored in two two-dimensional dense arrays.

Ordering

Fill-in reducing ordering is obtained using a parallel formulation of the multilevel nested bisection algorithm implemented in PARMETIS (Karypis and Kumar [61])

that has been found to be effective in producing orderings that are suited for parallel factorizations (see Section 1.3).

PSPASES allows advanced users to input a permutation order on rows of an unordered matrix. PSPASES would then reorder this matrix to realize the specified permutation. However, the user must be familiar with the algorithms of PSPASES to be able to provide such ordering information.

Numerical Factorization

For the numerical factorization a highly scalable method (Gupta et al. [50]) is used which is based on a multifrontal algorithm (see Section 2.4.2).

A collection of consecutive nodes in the elimination tree, each with only one child, is called a *supernode*. The nodes in a supernode are collapsed together to form the supernodal elimination tree. The serial multifrontal algorithm can be extended to operate on this supernodal tree by extending the single node operations performed while forming and factoring the frontal matrix. The frontal matrix corresponding to a supernode with l nodes is obtained by merging the frontal matrices of the individual nodes, and the first l columns of this frontal matrix are factored during the factorization of this supernode.

In the parallel formulation of the multifrontal algorithm, it is assumed that the supernodal tree is binary in the top $\log p$ levels. The portions of this binary supernodal tree are assigned to processors using a subtree-to-subcube strategy.

3.1.3 WSSMP

The *Watson Symmetric Sparse Matrix Package*, WSSMP, is a high performance, robust, and easy to use software package for solving large sparse symmetric systems of linear equations on IBM RS/6000 workstations and IBM SP systems. It uses a modified version of the multifrontal algorithm (Liu [69]; see Section 2.4.2) for sparse Cholesky factorization and a highly scalable parallel sparse Cholesky factorization algorithm (Gupta [45], Gupta et al. [51]). The package also uses scalable parallel sparse triangular solvers (Joshi et al. [59]) and a parallel version of the *Watson Graph Partitioning Package*, WGPP (Gupta [48], Gupta [47]), for computing fill-in reducing orderings. Sparse symmetric factorization in WSSMP has been measured at up to 440 Mflop/s on an RS/6000-397 and in excess of 20 Gflop/s on a 64 processor SP2 with RS/6000-397 nodes. These rates are 69 % and 49 %, respectively, of the theoretical peak performance (a single RS/6000-397 node running at 160 MHz has a theoretical peak performance of 640 Mflop/s).

An SMP version will be released in the near future. Neither the serial nor the parallel version has out-of-core capabilities and the problem must fit in the main memory for reasonable performance.

WSSMP is applicable to symmetric positive definite, quasi-definite, and indefinite matrices whose factorization is numerically stable irrespective of the pivot

sequence. For certain kinds of indefinite systems, special ordering techniques must be used. Also semi-definite matrices can be handled via appropriate use of the options provided by WSSMP. Pivoting is performed only for fill-in reduction; the factorization do not perform pivoting based on numerical data.

Since the input matrix is symmetric, only a triangular part is accepted as input. Both, the CSR format and the MSR format can be used for storing the matrix A. The performance of WSSMP is slightly better using the CSR format than using the MSR format.

WSSMP (Gupta et al. [49]), a faster version of PSPASES (see Section 3.1.2) with enhanced functionality for IBM RS/6000 workstations and IBM SP parallel computers, is available upon e-mail request from its author Anshul Gupta (`anshul@watson.ibm.com`).

More information about WSSMP and PWSSMP can be found at the web site `www.research.ibm.com/mathsci/ams/ams_WSSMP.htm`.

PWSSMP

The parallel version of the package, PWSSMP, performs ordering, factorization, solution, and iterative refinement on multiprocessor computers. Symbolic factorization, which is the least time consuming phase, is performed serially on processor P_0. Processor P_0 also bears a disproportionately high workload in the ordering phase compared to other processors. On heterogeneous machines, the performance of the parallel routines will be much better if processor P_0 has a fast CPU, high memory bandwidth, and a very large main memory.

The parallel routines can work in two modes:

- In the *0-master mode*, all data including the entire matrix A and right hand side b reside initially on processor P_0. This mode is easier to use and provides a quick way of migrating from a serial to a parallel implementation, because of identical calling sequences.

- In the *peer mode*, the initial data is distributed among the processors. There is no restriction on the relative amount of data on any processor, but using the peer mode with near uniform distribution of the input matrix will yield much better performance than the 0-master mode.

Since IBM SP systems may be heterogeneous with compute nodes differing from each other, the parallel version is designed to automatically detect heterogeneity and distribute the load accordingly, as far as possible. The performance of the PWSSMP solver on a heterogeneous configuration will vary depending on the relative location of fast and slow nodes in the system.

More information about WSSMP and PWSSMP can be found at the web site `www.research.ibm.com/mathsci/ams/ams_WSSMP.htm`.

3.1.4 Y12M

Y12M solves sparse systems of linear equations using a specially adapted Gaussian elimination. The subroutine `y12ma` is an easy to use *black box program* based on three subroutines.

`y12mb` converts the matrix supplied by the user in the COO format into an internal storage scheme.

`y12mc` performs an LU factorization of the matrix. To a great extent the pivoting strategy can be controlled to preserve both the numerical stability of the computations and the sparsity structure of the coefficient matrix. Moreover, a pivot sequence derived from a previous factorization of a matrix of the same sparsity structure can be used to reduce computational cost.

`y12md` solves the sparse system of linear equations using the LU factorization computed by `y12mc`.

The main task of `y12ma` is to call these three subroutines using suitable parameter settings. `y12mf` does the same except that it can perform an additional iterative refinement.

Y12M can be obtained from the `y12m` directory of NETLIB.

3.1.5 UMFPACK

UMFPACK is a program package for solving sparse, non-symmetric systems of linear equations using LU factorization. It contains subroutines for generating suitable pivot sequences, for constructing symbolic and numerical LU factorizations, and for computing the solution to a given LU factorization. The coefficient matrix must be specified in the COO format.

The implementation of the algorithms in UMFPACK—in contrast to many other software products—offers extensive opportunities for vectorization and parallelization. Previous implementations of direct methods have been disappointing in this respect. In Version 2 of UMFPACK a combined unifrontal/multifrontal approach (see Section 2.4.2) is used to combine the advantages of both methods. Version 1 was based on a multifrontal method. A detailed description of the algorithms used in UMFPACK is given in Davis, Duff [21].

UMFPACK can be obtained from the NETLIB; the directory `linalg` contains the file `umfpack.shar`. UMFPACK is also part of the Harwell Subroutine Library [1]. UMFPACK Version 2 implements the same algorithm as MA38 from the Harwell library. Although the routines have different names, the user interfaces are the same.

3.1.6 PSSPD

PSSPD (*Parallel Solution of Sparse Symmetric Positive Definite Systems*) is a program package for solving large, sparse, symmetric positive definite linear systems on distributed memory computers. PSSPD is implemented in ANSI C.

All routines of PSSPD—except the symbolic factorization—are performed in parallel. The numerical factorization and the triangular solution are implemented using the parallel multifrontal scheme described in Sun [84].

PSSPD is designed for solving sparse symmetric positive definite systems. However, it can be used to handle sparse symmetric positive semidefinite systems to some extent.

Information about the PSSPD package (and its author) is available from the web site `simon.cs.cornell.edu/Info/People/csun/psspd/index.html`.

3.1.7 SuperLU

The SUPERLU package contains a set of subroutines to solve sparse linear systems $AX = B$, where A is a square, non-singular, sparse matrix of order n. Matrix A does not have to be symmetric or definite; indeed, SUPERLU is particularly appropriate for matrices with very unsymmetric structure.

The package uses LU decomposition with partial pivoting. The columns of A may be preordered before factorization (either by the user or by SUPERLU); this preordering for sparsity is completely separate from the factorization. There are also routines for iterative refinement, equilibrating the system, estimating the condition number, calculating the relative backward error, and estimating the error bounds for the refined solutions available.

The factorization algorithm uses a graph reduction technique to reduce graph traversal time in the symbolic analysis. Computational loops are organized in a way that reduces data movements between levels of the memory hierarchy and that exploits dense submatrices in numerical kernels. The resulting algorithm is highly efficient on modern architectures.

SUPERLU is implemented in ANSI C. Also a Matlab interface is included, so that the factorization and solve routines can be called as alternatives to those built into Matlab.

SUPERLU as well as SUPERLU_MT are available from the `scalapack` directory of NETLIB .

3.1.8 SuperLU_MT

SUPERLU_MT is a parallel extension to the serial SUPERLU package. SUPERLU_MT is implemented in ANSI C, using multithreading extensions, for example, using POSIX threads. Currently, only the LU factorization routine, which is

the most time-consuming part of the solution process, is parallelized on machines with a shared address space. The other routines, such as column preordering and the forward and back substitutions are performed sequentially.

3.1.9 SuperLU_DIST

SuperLU and SuperLU_MT are highly efficient packages on workstations with deep memory hierarchies and shared memory parallel machines with a modest number of processors. Both packages are available from Netlib and are among the fastest available codes for solving sparse linear systems.

The shared memory algorithm for the Gaussian elimination with partial pivoting (as implemented in SuperLU_MT) relies on the fine-grained memory access and synchronization that shared memory computers provide. These features are necessary to manage the data structures needed as fill-in is created dynamically, to discover which columns depend on which other columns symbolically, and to use a centralized task queue for scheduling and load balancing. All these operations must be performed dynamically, because the computational graph does not unfold until run time. (This is in contrast to Cholesky algorithms, where pivoting is not needed to ensure numerical stability.) However, these techniques are too expensive on distributed memory machines.

Instead, for distributed memory machines, Li and Demmel [66] propose to not pivot dynamically, and so enable static data structure optimization, graph manipulation and load balancing, as with Cholesky algorithms (cf. Gupta and Kumar [52]) and yet remain numerically stable.

Numerical stability can be retained by a variety of techniques: pre-pivoting large elements to the diagonal, iterative refinement, using extra precision when needed, and allowing low rank modifications with corrections at the end. SuperLU_dist uses all these techniques to control the element growth during Gaussian elimination.

Currently, the code is not yet available to the public. More information can be found at `www.nersc.gov/~xiaoye`.

3.1.10 SPOOLES

Spooles is an attempt to provide scalability for solving sparse systems of linear equations for a diverse set of applications on a diverse set of parallel computers. An object oriented approach has been taken to providing sparse matrix technology which is different from the more traditional subroutine library approach.

Spooles is a package for solving sparse real and complex linear systems of equations, written in C using an object oriented design. To manage the complexity of over 120,000 lines of C code, encapsulation is the cardinal design rule. An object (there are 48 of them in Spooles) knows and operates on itself, but has

very limited knowledge of other objects. Encapsulation is a custom enforced by discipline rather than a language feature. As the object hierarchy of SPOOLES is very flat, only a few objects would benefit from inheritance (a feature supported in C++, but not in C). The development of SPOOLES has shown that good object-oriented design is possible without explicit language support.

At present, there is the following functionality:

- Compute multiple minimum degree, generalized nested dissection and multisection orderings of matrices with symmetric structure.
- Factorize and solve square linear systems of equations with symmetric structure, with or without pivoting for stability. The factorization can be symmetric $LDL^{\top}$, Hermitian LDL^{H}, or nonsymmetric LDU. A direct factorization or a drop tolerance factorization can be computed. The factorization and solution can be done in serial mode, multithreaded using Solaris or POSIX threads, or using MPI.
- Factorize and solve overdetermined full rank systems of equations using a multifrontal QR factorization, in serial or using POSIX threads.
- Solve square linear systems using a variety of Krylov iterative methods. The preconditioner is a drop tolerance factorization, constructed with or without pivoting for stability.

The SPOOLES package has been developed by members of Boeing Phantom Works. The development of SPOOLES was funded by DARPA and DoD with the purpose that others (academic, government, industrial and commercial organizations) could easily incorporate the data structures and algorithms into their application codes. It is an attempt to complete the algorithmic and software foundation for an industrial strength sparse matrix library for parallel computers. SPOOLES has been developed with the assistance of industrial users and has been designed for immediate use in packages like NASTRAN and other finite element software.

The algorithms for the numerical factorization and solution steps used in SPOOLES are based on established algorithms (Ashcraft et al. [7], Ashcraft et al. [10]). Innovative approaches for the ordering of sparse matrices have been included (Hendrickson and Leland [55], Hendrickson and Rothberg [57]).

SPOOLES is entirely within the public domain and there are no licensing restrictions. Version 2.2 of SPOOLES is available from NETLIB.

Matrix Ordering

SPOOLES has three options for ordering sparse matrices: multiple minimum degree (MMD), generalized nested dissection, and multisection. The MMD ordering

implemented in SPOOLES is very similar in quality and ordering time to the M-MD software of Liu [67]. The nested dissection and multisection orderings have been tested against two other state-of-the-art software packages, METIS [61] and SGI's EXTREME software (a descendent of the CHACO package [56] including some improvements). The quality of the three orderings is very similar over a large selection of structural analysis matrices.

Because of the object oriented nature of SPOOLES the user needs to have some knowledge about two important data structures that are used in the solution of sparse systems of equations: permutations and trees.

The vertex elimination tree is a representation of the order that the vertices in the adjacency graph will be eliminated. This order corresponds to the order that the rows and columns of a square matrix A will be factored. The dependencies of the ordering form a tree structure. The leaves of the tree represent vertices which can be eliminated first. The parents of those leaf nodes can be eliminated next, and so on, until finally the vertices represented by the root of the tree will be eliminated last.

The elimination tree illustrates the dependence of vertices. The basic rule is that a vertex depends only on his descendents and will affect only its ancestors. Therefore, the elimination tree gives the possibility to identify independent, parallel computation.

While the vertex elimination tree is useful to recognize data dependencies, it is not helpful in finding a useful granularity on which to base a factorization or solution process, either in serial or in parallel. It is important to group vertices together in some meaningful way to create data structures of coarser granularity that will be more efficient with respect to storage and computation. A first step towards this goal is the grouping of vertices that form a chain with no branches in the tree. Using this grouping the *fundamental supernode tree* can be generated.

A factorization based on the fundamental supernode tree requires no more operations than one based on an vertex elimination tree. There are many small supernodes at the lower levels of the tree. By *amalgamating* small but connected sets of supernodes together into larger supernodes, the algorithm reduces the overhead of processing all of the small supernodes at the expense of adding entries to the factors and operations to compute the factorization. This amalgamation of supernodes generally leads to an overall increase in computational efficiency. The result is called the *amalgamated supernode tree*.

There is one final step in constructing the tree that governs the factorization and solution. Large matrices will generate large supernodes at the topmost levels of the tree. The data structure for a top level supernode can be very large, too large to fit into main memory. In a parallel environment, SPOOLES follows the convention that each node in the tree is handled by one processor. Very large nodes at the top levels of the tree will severely decrease the potential parallelism.

The solution to both problems, too large data structures and limited parallelism, is to split large supernodes into pieces. The user can specify a maximum

size for the nodes in the tree, and split large supernodes into pieces no larger than this given limit. This will keep the data structures to a manageable size and increase the available parallelism. The resulting tree is called the *front tree* because it represents the final computational unit for the factorization, the frontal matrix.

Matrix Factorization

One of the most important design considerations for sparse direct solvers is the support of pivoting for numerical stability. In many cases pivoting for stability is not needed (for example, when matrices are diagonally dominant or well conditioned positive definite), but there are cases where pivoting is crucial.

For symmetric matrices the SPOOLES package uses the fast Bunch-Parlett algorithm (Ashcraft, Grimes, Lewis [10]). For unsymmetric matrices its analogous counterpart is used which is also known as *rook pivoting* (Foster [42]). Actually, the factorization $PLDUQ$ or $PU^{\top}DUP^{\top}$ is computed. When pivoting is enabled, the magnitude of entries in L and U are bound by some user supplied tolerance.

SPOOLES implements the fan-in algorithm for the numerical factorization (Ashcraft, Eisenstat, Liu [7]). When the computations for a given front are started, the processor owning the rows and columns for that front receives the modifications to that front from previously computed fronts which are owned by other processors. Every incoming modification is processed by the processor that owns the previously computed front.

SPOOLES and Thread Based Parallelism

The shared memory parallel version of SPOOLES is implemented using thread based parallelism. The multi-threaded code uses much of the serial code; the basic steps are the same and use the serial methods. The usage of SPOOLES for communicating the data for the problem and reordering the linear system is identical in the serial and the multi-threaded versions. Only the numeric factorization and solution steps are parallelized using threads. The main difference between the serial and the multi-threaded version is the scheduling of the computations.

While the storage for the factor matrices lies in one object, all processes access the data in a disjoint way. Front J is owned by one process that is responsible for factoring the front and computing its updates to all ancestor fronts. In other words, only the process that owns front J performs computations with that data.

One of the most important factors influencing the performance of a parallel factorization is how the computation is distributed. In SPOOLES the computation is distributed by how the fronts are assigned to the processes. The process that owns the front performs all of the computation with that data. SPOOLES provides four possible ways for distributing the ownership of fronts among the processes. and thus for influencing the amount of parallel communication required between

the processes and the overall efficiency of the parallel factorization and solution. The four types of owner maps are

Wrap map: This mapping assigns the front J to process $p = \mathrm{mod}(J, N)$, where N is the number of processors.

Balanced map: This mapping is an attempt to balance the workload between the processors. This map selects the fronts in a post-order traversal and assigns a front to the process with the fewest numbers of accumulated factor operations.

Subtree-subset map: Introduced in Heath [53], this mapping attempts to reduce the amount of communication by assigning subtrees of the front tree to subsets of the processes. The variant implemented in SPOOLES is based on a publication by Pothen and Sun [77]. While the mapping works fairly well for perfectly balanced trees, it suffers from load imbalance in most situations.

Domain decomposition map: This mapping, very similar to the subtree-subforest map introduced by Karypis and Kumar [60], attempts to remove the imbalance of operations. The basic idea is to designate a set of subtrees, or domains, to be owned by a single process and then assign these domains to the processes to balance the computation. The remaining fronts are assigned to processes using a balanced map.

The domain decomposition map requires an additional parameter `cutoff`, which specifies which fraction of the total number of operations is used to define the domain. It is recommended that the reciprocal value of `cutoff` is two times the number of processes.

It is extremely difficult to get an accurate estimate of the efficiency of a particular factorization for any of these mappings a priori. The user is encouraged to experiment with all of them for his particular application.

SPOOLES and MPI Based Parallelism

It is a very difficult to develop a general purpose sparse linear equation solver for a distributed memory computing environment. There are many tradeoffs between honoring user distribution of the matrix data and efficiency of the parallel factorization and solution. However, SPOOLES provides all the necessary routines and data structures to efficiently utilize distributed memory systems. Detailed information can be found in Ashcraft et al. [7].

3.1.11 The Package S$^+$

The S$^+$ package is a set of subroutines for solving general sparse linear systems of the form $AX = B$ on parallel architectures. A is a non-singular $n \times n$ sparse matrix, the right hand side B and the solution X are $n \times m$ dense matrices with real coefficients.

S$^+$ solves sparse linear systems using LU decomposition with partial pivoting and 2D data mapping and is based on previous work of Fu et al. [44]. S$^+$ uses the properties of elimination forests to guide supernode partitioning/amalgamation and execution scheduling. This design with 2D mapping effectively identifies dense structures without introducing too many zeros in the BLAS computation and exploits asynchronous parallelism with low buffer space cost. Two space optimization techniques are also incorporated into S$^+$ to improve the worst-case performance for static symbolic factorization.

The package is implemented in ANSI C. It uses MPI to handle communications and also uses the BLAS library for numerical operations. Version 1.0 of S$^+$ can run on both shared-memory and distributed-memory systems provided an MPI implementation is available. S$^+$ offers the following routines:

- `SP_Ordering` obtains permutation vectors P_c and P_r. The LU factorization of AP_c tends to have less fill-in than the LU factorization of A. P_r is applied after P_c to minimize the number of zero diagonal elements. `SP_Ordering` is a sequential routine and should only be called by one MPI node before calling `SP_Solve`.

  S$^+$ provides the column minimum degree ordering based on Liu's algorithm. The user can choose to order the matrix beforehand and bypass the ordering routine provided by S$^+$.

- `SP_Solve` is a routine for solving the linear system $AX = B$. Data distribution is done inside this routine.

Detailed information, related papers and the software is available from the web site `www.cs.ucsb.edu/research/S+`.

3.2 Routines from Software Libraries

3.2.1 Harwell Library

The *Harwell Subroutine Library* (HSL) is a collection of mostly Fortran 77 and some Fortran 90 programs that provide access to reliable numerical code covering a wide range of algorithms. It is arranged as a collection of packages, each of which consists of a set of program units. Each package performs a basic numerical task and has been designed to be incorporated into programs under the control of the user.

AEA Technology (a private sector company, formerly part of the United Kingdom Atomic Energy Authority, see also `www.aeat.co.uk`) is responsible for the distribution of HSL. HSL has started in 1963 and over this long period HSL has reached a high standard of reliability and has a world-wide reputation as a source of good numerical codes. A subset of the HSL is also marketed by NAG under the title of the *Harwell Sparse Subroutine Library*.

The MA package of the HSL is devoted to the direct solution of real sparse linear systems (both, single and double precision versions are provided); for complex matrices, the ME package with similar functionality is available.

The MA41 Routines

The MA41 set of routines solves sparse unsymmetric systems of linear equations. MA41 uses a parallel direct method based on a sparse multifrontal variant of Gaussian elimination. An initial ordering for the pivotal sequence is chosen using the sparsity pattern of $A + A^\top$. This ordering is later modified to enhance numerical stability. MA41 performs best on matrices whose sparsity pattern is symmetric or nearly so. Other HSL codes might be more appropriate for symmetric sparse matrices (for example MA27 or MA 47), or for very unsymmetric and very sparse matrices (for example MA38 or MA48). The sequential version of this routine is similar to MA37 but has a different interface, includes more options, and uses Level 2 and Level 3 BLAS for operations on dense matrices.

There is a Fortran 77 version of the code for uniprocessor machines. The parallel versions are machine dependent but only require simple operations and facilities like starting parallel tasks and locks. There are only two shared memory parallel systems on which MA41 has been tested extensively, namely the Cray Y-MP and the Alliant FX/80.

The MA41 package consists of only two routines:

`MA41ID` must be called to set the default values for the control parameters.

`MA41AD` has been designed as a user friendly interface to the multifrontal code. This routine, in addition to providing the normal functionality of a sparse solver, incorporates some pre- and post-processing. The routine enables the user to preprocess the matrix to obtain a maximum traversal so that the permuted matrix has a zero-free diagonal, to perform prescaling of the original matrix (a choice of scaling strategies is provided), to use iterative refinement to improve the solution, and finally to perform error analysis.

Pivot elements are chosen from the diagonal using the approximate minimum degree algorithm of Amestoy et al. [3] (see Section 1.1.1) and employing a generalized element model of the elimination thus avoiding the need to store the fill-in pattern explicitly. The elimination is represented as an assembly and elimination tree.

The parallelism of the elimination tree and the parallelism coming from the frontal matrix elimination steps are combined. The number of tasks created during factorization can be explicitly controlled by the user.

The MA42 Routines

The MA42 set of routines solves a sparse system of linear equations by the frontal method, optionally using direct access files for the matrix factors. Use is made of high level BLAS kernels. The code has low in-core memory requirements.

The matrix A may be input by the user in either of the following ways: (i) by elements in a finite element calculation, and (ii) by equations (matrix rows).

The following routines are provided in the MA42 package:

`MA42ID` must be called to set the default values for the control parameters.

`MA42AD` must be called for each element or equation to specify which variables are associated with it.

`MA42JD` can be used optionally. If the user wishes to perform a symbolic factorization, `MA42JD` must be called for each element or equation.

`MA42PD` opens and initializes direct access files. Only needed if the user decides to hold the data for the LU factors in direct access files.

`MA42BD` must be called for each element or equation to factorize A and optionally solve $Ax = b$.

`MA42CD` must be called for each element or equation to further solve systems $Ax = b$ or $A^\top x = b$.

A version of sparse Gaussian elimination is used in the MA42 package. The elimination algorithm is implemented using a unifrontal method.

A principle feature of MA42 is its ability to solve large problems in a predetermined and relatively small amount of in-core memory. To keep the amount of in-core memory low, it is important that the user orders the matrices $A^{(k)}$ so that the numbers of variables in the front (the front width) is small. For finite-element calculations, routine MC43 is recommended to obtain an efficient element ordering before MA42 is used. For equation entry, the equations should be ordered so that the matrix is banded.

The MA38 Routines

The MA38 set of routines solves a sparse unsymmetric system of linear equations using an unsymmetric multifrontal variant of Gaussian elimination.

The following routines are provided in the MA38 package:

`MA38ID` must be called to set the default values for the control parameters.

`MA38AD` computes the pivot ordering and performs both symbolic and numerical LU factorization of an $n \times n$ unsymmetric matrix.

`MA38BD` factorizes a matrix A that has the same nonzero pattern as a matrix previously factorized by `MA38AD`. It uses the same pivot order as found by `MA38AD`. `MA38BD` is generally significantly faster than `MA38AD`.

`MA38CD` uses the factors produced by `MA38AD` or `MA38BD` to solve $Ax = b$ or $A^\top x = b$, optionally performing iterative refinement.

Conventional sparse matrix factorization algorithms for general problems rely heavily on indirect addressing. This gives them an irregular memory access pattern that limits the floating-point performance of typical (parallel) vector machines and of cache based RISC computers. In contrast, the *classical* multifrontal method is designed with regular memory access in the innermost loops. This multifrontal method assumes structural symmetry and bases the factorization on an assembly tree generated from the original matrix and an ordering such as minimum degree. The computational kernel, executed at each node of the tree, is one or more steps of the LU factorization within a square, dense frontal matrix defined by the nonzero pattern of a pivot row and column. These steps of LU factorization compute a contribution block (a Schur complement) that is later assembled (added) into the frontal matrix of its parent in the assembly tree.

Although structural symmetry can be accommodated in the classical multifrontal method by holding the sparsity pattern of $A + A^\top$ and storing explicit zeros, this can have poor performance on matrices whose sparsity pattern is very unsymmetric. As one does not rely on the structural symmetry of the matrix A, the algorithms becomes more complicated. For example, the frontal matrices are now rectangular instead of square, and some contribution blocks must be assembled into more than one subsequent frontal matrix. As a consequence, it is no longer possible to represent the factorization by an assembly tree; the more general structure of a *directed acyclic graph* (DAG) is required. The MA38 routines are based on this unsymmetric pattern multifrontal approach. As in the symmetric multifrontal case, advantage is taken of the repetitive structure in the matrix by choosing more than one pivot element in each frontal matrix. Thus the algorithm can use higher level dense matrix kernels in its innermost loops (Level 3 Blas). The MA38 algorithm can thus achieve satisfactory floating-point performance.

The performance of the MA38 algorithms depends on two crucial factors: (i) the numerical factorization within dense, rectangular frontal matrices, (ii) the computation of the degrees of the rows and columns of the active submatrix. Both factors have been addressed in the implementation of the MA38 package (by using dense kernels and an AMD based ordering).

The MA47 Routines

The MA47 set of routines solves a sparse symmetric indefinite system of linear equations. The algorithm used is a direct method based on multifrontal sparse Gaussian elimination and is discussed in Duff and Reid [39].

The following routines are provided in the MA47 package:

`MA47ID` must be called to set the default values for the control parameters.

`MA47AD` prepares the data structures for the numerical factorization and chooses pivots for Gaussian elimination to preserve sparsity. The user may provide a pivot sequence, in which case the necessary information for `MA47BD` will be generated.

`MA47BD` factorizes a matrix A using the information from a previous call to `MA47AD`. The actual pivot sequence used may differ from that returned from `MA47AD`.

`MA47CD` uses the factors generated by `MA47BD` to solve a systems of equations $Ax = b$.

A version of sparse Gaussian elimination is used in the elimination process. It is implemented using a multifrontal method.

`MA47AD` chooses diagonal pivots of order 1 and 2 using the Markowitz criterion or accepts a sequence of such pivots supplied by the user. Because of the facility for handling matrices with zeros on the diagonal, the 2×2 pivots can be of the form

$$\begin{pmatrix} 0 & x \\ x & x \end{pmatrix} \quad \text{or} \quad \begin{pmatrix} 0 & x \\ x & 0 \end{pmatrix}$$

called *tile* and *oxo* pivots respectively. `MA47AD` employs a generalized element model (see Section 1.1.1) of the elimination, thus avoiding the need to store the fill-in pattern explicitly.

`MA47BD` factorizes the matrix by using the assembly and elimination ordering generated by `MA47AD`, but with additional numerical pivoting. The numerical pivoting can yield full 2×2 pivots in addition to the tile and oxo pivots above.

The pivotal strategy is explained by Duff et al. [32]. An explanation of the whole algorithm is given by Duff and Reid [39].

The MA48 Routines

The MA48 set of routines solves a sparse unsymmetric system of m linear equations in n unknowns using a general approach to Gaussian elimination (see Section 2.1).

The following routines are provided in the MA48 package:

`MA48ID` must be called to set the default values for the control parameters.

`MA48AD` prepares data structures for factorization, optionally permuting the matrix A to block upper triangular form, and optionally choosing a pivot sequence that leads to the factorization of the permuted matrix PAQ, using a pivotal strategy designed to compromise between maintaining sparsity and controlling loss of accuracy through roundoff. There is an option for the user to limit pivoting to the diagonal and an option to input the permutations P and Q.

`MA4BAD` factorizes a matrix A, given data provided by `MA48AD`. There are two main options within `MA48BD`, a "normal" call that performs further row permutations to control loss of accuracy by roundoff and a "fast" option that uses information from the normal call including additional row permutations.

`MA48CD` uses the factors produced by `MA48AD` to solve $Ax = b$ or $A^\top x = b$, optionally performing iterative refinement. An estimate of the error may be obtained.

Once a single matrix has been factorized by `MA48BD`, further matrices may be factorized more quickly if they have the same sparsity structure and the same pivot sequence is acceptable.

Pivoting is used to preserve sparsity in the factors and each pivot element a_{pj} must satisfy the stability test

$$|a_{pj}| \geq u \max_i |a_{ij}|$$

within the reduced matrix, where u is the threshold (with default value 0.1). The actual factorization is done by calling routine `MA50BD` to factorize each diagonal block. For the full matrix processing, options for the use of Level 1, 2 or 3 BLAS have been included. Which level of BLAS gives the best results depends on the computer system being used, the structure of the matrix and the availability of highly optimized BLAS. A discussion of the design of these routines is given by Duff and Reid [38].

3.2.2 IMSL Libraries

The IMSL Fortran library contains three subroutines for the solution of systems of linear equations with a general sparse coefficient matrix. The coefficient matrix must be supplied by the user in the COO format.

`IMSL/lftxg` performs an LU factorization of the matrix A. `IMSL/lfsxg` then solves the linear system by using the two factor matrices L and U. Given a matrix A and a right-hand side b, `IMSL/lslxg` solves the respective linear system by calling the two former subroutines.

In `IMSL/lftxg`, a direct method which estimates the local fill-in by the Markowitz costs has been implemented. By specifying certain parameters the user can control both the numerical stability of the method and the fill-in generated by the method. If the accuracy of the solution needs to be improved, this program offers iterative refinement as well.

For sparse, complex matrices an analogous set of subroutines with identical parameter lists is available. The names of those subprograms are identical to those for real matrices, except that the letter `x` (indicating real type) has been changed to `z` (indicating complex type).

For the solution of linear systems with symmetric positive definite matrices, the IMSL library offers a direct method based on Cholesky factorization.

Given a matrix A, whose lower triangular part is specified in the COO format, `IMSL/lscxd` performs a symbolic Cholesky factorization. This program computes a minimum degree ordering or uses a user-supplied ordering to set up the sparse data structure for the Cholesky factor L. The routine `IMSL/lnfxd` then computes the numerical entries in L very efficiently as all fill-in arising in the Cholesky factorization has already been considered in the sparse data structure for L. The numerical computations can be carried out in one of two ways. The first method performs a factorization using a multifrontal technique. This option requires more storage but in certain cases is faster. The second method is just the standard factorization method based on the sparse storage scheme.

Given the Cholesky factors of the coefficient matrix A and the right-hand side b, `IMSL/lfsxd` solves the system of linear equations. `IMSL/lslxd` performs all computations and subroutine calls necessary for the solution of a given linear system.

An analogous set of subroutines is provided for the direct solution (using Cholesky factorization) of linear systems with a sparse *Hermitian* positive definite coefficient matrix.

3.2.3 NAG Libraries

The NAG Fortran library, like the IMSL Fortran library, contains subprograms for the solution of sparse linear systems based on direct methods. `NAG/f01brf` performs an LU factorization of a matrix specified in the COO format. The numerical stability of the method can be controlled by several user definable parameters.

The pivot sequence obtained by the application of `NAG/f01brf` can be reused by the subprogram `NAG/f01bsf` if a matrix of the same sparsity structure has to be factorized. `NAG/f04axf` solves the linear system for a single right-hand side b provided that an LU factorization of the matrix A has already been computed.

3.3 Routines from Scientific Packages

Rapid prototyping in a comfortable programming environment is an important feature for many software developers. Both, MATLAB and PETSc have rich support for handling everything from problem formulation to post-analysis of the solution.

3.3.1 PETSc

PETSc is a large suite of data structures and routines for both uni- and parallel-processor scientific computing. Intended especially for the numerical solution of large-scale problems modeled by partial differential equations, PETSc provides a rich environment for modeling scientific applications as well as for rapid algorithm design and prototyping. PETSc includes scalable parallel preconditioners and Krylov subspace methods for solving sparse linear systems, as well as parallel nonlinear equation solvers and parallel ODE solvers. The high level PETScView tool provides profiling and visualization of PETSc programs.

PETSc is intended for use in large-scale application projects, and several ongoing computational science projects are built around the PETSc libraries. With strict attention to component interoperability, PETSc facilitates the integration of independently developed application modules, which often most naturally employ different coding styles and data structures.

PETSc is easy to use for beginners. Moreover, its careful design allows advanced users to have detailed control over the solution process. PETSc includes an expanding suite of parallel linear and nonlinear equation solvers that are easily used in application codes written in C, C++, and Fortran. PETSc provides many of the mechanisms needed within parallel application codes, such as simple parallel matrix and vector assembly routines that allow the overlap of communication and computation. In addition, PETSc includes growing support for distributed arrays.

PETSc can be found at `www.mcs.anl.gov/petsc/petsc.html`.

3.3.2 MATLAB

MATLAB is an integrated technical computing environment that combines numerical computation, advanced graphics and visualization, and a high-level programming language. MATLAB includes hundreds of functions for data analysis and visualization, numerical and symbolic computation, engineering and scientific graphics, modeling, simulation, and prototyping, programming, application development, and GUI design.

MATLAB offers many functions for sparse matrices, including sparse matrix generation, conversion between different storage formats, reordering algorithms,

direct sparse solvers, iterative linear equation solvers, incomplete factorization routines, visualization tools.

MATLAB offers the possibility to incorporate programs written in Fortran or C into MATLAB code. Some publicly available packages for sparse systems offer already such a MATLAB interface, for example, SUPERLU (see Section 3.1.7) and EMILY (see Section 3.4.7).

More details and information about this commercial product can be found at `www.mathworks.com`.

3.4 Matrix Collections, Utilities

The Harwell-Boeing collection of sparse matrices has been widely used for testing and comparison purposes of software for linear systems. This section gives an overview on publically available sparse matrix collections, and tools and utilities for manipulating, converting, and analyzing sparse matrices.

3.4.1 The Matrix Market

The MATRIX MARKET [14] provides convenient access to a repository of test data for use in comparative studies of algorithms for numerical linear algebra. Matrices as well as matrix generation software and services, from linear systems, least squares, and eigenvalue computations in a wide variety of scientific and engineering disciplines are provided. Tools for browsing through the collection or for searching for matrices with special properties are included.

Each matrix (and matrix set) has its own web page which provides details of matrix properties, visualization of matrix structure, and permits downloading of the matrix in one of several formats. Similarly, each matrix generator has a web page describing its properties. Generators are either static software which can be downloaded and included in the own applications, Java applets which generate matrices in the users web browser, or form-based requests to generate matrices at the MATRIX MARKET and return the matrices to the users browser.

Currently, 482 individual matrices and 25 matrix generators are available. The collection now includes the entire Harwell-Boeing Collection, Yousef Saad's SPARSKIT collection, and the nonsymmetric eigenvalue problem (NEP) collection of Bai, Day, Demmel and Dongarra. Users are urged to submit their own test matrices for possible inclusion in the MATRIX MARKET.

The MATRIX MARKET can be found at `math.nist.gov/MatrixMarket`.

3.4.2 The Harwell-Boeing Collection of Matrices

The *Harwell-Boeing Collection* is a compilation of large sparse matrices arising from different fields of application (mainly physics and engineering). A specific

scheme for storing these matrices, the *Harwell-Boeing* format, is available. Essentially, it corresponds to an extended CRS format with additional information (for example, the order of the matrix, the number of non-zero elements, etc.). In the *Harwell-Boeing* format, it is possible to store only the sparsity structure of the matrix (i. e., the positions of the non-zero elements are stored, not the values). It is also possible to store dense matrices together with one or more right-hand sides (including vectors representing the solution or starting vectors for an iterative method).

Information about the Harwell-Boeing matrix collection and software can be obtained from the web site `www.cerfacs.fr/harwell_boeing`. A comprehensive user's guide in POSTSCRIPT format includes a detailed description of the *Harwell-Boeing* format, and of all the matrices in the collection.

3.4.3 The Rutherford-Boeing Collection of Matrices

The *Rutherford-Boeing Collection* is a significantly updated version of the Harwell-Boeing matrix collection. The major differences are:

- More supplementary data are provided, such as orderings, estimates of solutions and eigensolutions, and geometric data.
- Format changes are provided to facilitate use of the collection from languages other than Fortran.
- More and larger matrices are contained in the collection.
- Internet access to the matrix collection is possible through the MATRIX-MARKET (`www.nist.gov/MatrixMarket`) and NETLIB (`www.netlib.org`).

The full documentation of the Rutherford-Boeing matrix collection can be found in Duff et al. [34].

3.4.4 The University of Florida Sparse Matrix Collection

The University of Florida sparse matrix collection is maintained by Tim Davis [19]. The collection includes about 700 sparse matrices from a wide field of applications. Each matrix has its own web page, with statistical information on the matrices and a small picture of the nonzero pattern. All matrices are available in the Harwell-Boeing format. More information and the matrices itself are available from the web site `www.cise.ufl.edu/~davis/sparse`.

3.4.5 SPARSE-BLAS

Analogous to the definition of BLAS (basic linear algebra subroutines for dense vectors and matrices), much effort has been spent to standardize the interfaces of basic operations in sparse linear algebra. One of the first proposals of a definition, regarding only vector operations (not regarding matrix-vector or matrix-matrix operations) was published in 1991 (Dodson, Grimes, Lewis [26]).

Almost all sparse direct solvers use kernels from dense linear algebra, typically (dense) BLAS are used. However, in case of iterative methods using preconditioning, a sparse version of the BLAS is appropriate (cf. Duff et al. [35].

3.4.6 SPARSKIT

SPARSKIT is a basic tool kit for sparse matrix computations written in Fortran 77. The CRS format is the basic format used in this package. SPARSKIT includes various procedures:

Input/Output routines allow files containing matrices stored in the Harwell-Boeing format to be read and written. Additionally there are some utilities for visualizing the sparsity structure of the matrix.

Format conversion routines: Most of the routines in SPARSKIT use the CRS format internally, but other commonly used storage schemes are also supported (COO, CCS, MRS, BCRS, BND, CDS, JDS, SKS). There are numerous format conversion routines which convert one storage scheme into another.

Unary matrix operations: There is a module consisting of a number of routines for transposing the matrix, extracting a specified part of the matrix, and retrieving information about the sparsity structure of the matrix.

Binary matrix operations: There is a module comprising routines for computing products and sums of sparse matrices in different ways.

Matrix-vector operations: This module offers (in its current form) certain triangular matrix solution routines in addition to the matrix-vector product which is especially important for iterative methods.

Matrix generation routines set up certain types of sparse matrices (e. g., matrices arising from 5-point or 7-point discretization of differential equations).

Statistics and information routines provide information on the sparsity structure of matrices stored in the *Harwell-Boeing* format.

The SPARSKIT package is public domain and has been made accessible through the Internet. It is available through anonymous FTP from `ftp.cs.umn.edu` as the file `SPARSKIT2.tar.Z` in the directory `/dept/sparse`.

3.4.7 EMILY

EMILY is a utility for visualizing large, sparse, unstructured matrices. Two fundamental problems arise in implementing such a matrix viewer: (i) which of the many possible sparse matrix data formats to use, and (ii) how to display a matrix with potentially more elements than displayable pixels? In choosing a matrix data structure, compatibility with existing software is the primary concern. Many numerical linear algebra packages for sparse systems are based on the compressed sparse column (CSC) format or its transpose, the compressed sparse row (CSR) format. CSC is the format used for the Harwell/Boeing collection of test matrices and is the internal representation used in the numerical package MATLAB [86] (see Section 3.3.2). This allows the matrix viewer to be used with a large base of standard test problems, and to be coupled with a matrix manipulation environment. The second problem—how to compress the matrix to fit the constraints of the display device—can be solved by using a mixture of weighted and unweighted area sampling, treating each input matrix value with equal area and using its numerical value as its contribution to the sampling function.

Other design features are also desirable in a matrix visualization tool. Having a variety of views of the matrix, ranging from a single element to the whole matrix, is needed to give different perspectives on the matrix. Multiple colormaps (palettes of displayable colors) are critical since linear systems come from a wide range of applications, and the tool must be able to provide the appropriate visual cues to match a user's acquired reactions. Providing multiple colormaps also helps focus the eye on different features and structures in the same view.

The disparate sources of linear systems also means the matrix entries can have arbitrarily large variations in magnitude. Providing logarithmic or linear scaling of input elements gives insight into the relative sizes of elements and allows for clearly viewing a wide range of elements.

The approach chosen in EMILY uses a Motif-based program. With the help of the mouse, the user can select a region of the matrix to zoom in on, expanding the region to see more detail. The windows opened up also display the matrix coordinates (in terms of row and column numbers) of the region being viewed, allowing the user to identify individual entries if desired. EMILY reads matrices stored in standard Harwell/Boeing format files. It can also be called from within MATLAB, so that users can perform manipulations on the matrices at a high level and quickly view the results.

EMILY is freely available from `www.cs.indiana.edu/scicomp/emily.html`.

3.4.8 MatView

MATVIEW is a tool for viewing and exploring large sparse matrices. Using MATVIEW, a sparse matrix with millions of elements can be reduced to a graphically viewable size for overviews of the high-level structure, or a detailed view can

be obtained by zooming in and navigating through small regions of the matrix. MatView provides:

- Efficient viewing of very large matrices (including huge matrix file support).
- Easy navigation, and zooming in and out of matrix details.
- A variety of data reduction functions (average, minimum/maximum, sum, variation).
- Custom color adjustments, or greyscale rendering.
- Viewing of sparsity structure only, if desired.
- Filtering of matrix values outside the color map range, if desired.
- Printing of matrix views, or saving views as PostScript files.
- Compatibility with the Matrix Market format, the Harwell-Boeing format, and the Matlab MAT files.
- Loading of compressed or gzipped matrix files.

MatView has been built using the Tcl/Tk system, and so should run on any computer where Tcl/Tk is available. MatView is an ongoing project, with continuing user interface additions and improvements, and there are plans to add pluggable modules for simple matrix transformations, matrix graphing support, and support for more matrix storage formats.

More information about MatView and the software itself can be found at the web site `www.epm.ornl.gov/~kohl/MatView`.

3.4.9 SMMS

The *Sparse Matrix Manipulation System* (Smms) is an open software architecture for the handling and interchanging of sparse matrix information for educational and research purposes. It is built around four main concepts: sparse matrices, permutation vectors, partition vectors and node set lists. Each of these concepts is implemented as a stream of ASCII data. These streams are operated upon by filters that process one or more items. Examples of filters include filters that produce permutation vectors, filters that add fill-in, filters that factor a matrix, and many more. Other routines in the system are used to generate sparse matrices and to display topology maps, factorization path trees, bipartite graphs and adjacency graphs for these matrices.

A virtue of this system is its ability for easily interchanging information among programs written by different programmers, in different languages, possibly in different computers at different sites.

SMMS is a collection of programs to process sparse matrices. In contrast to subroutine libraries, the programs in the SMMS package are commands, which can be executed by the user. Included are ordering routines, LDU factorization, visualization, orthogonal factorization, repeated solution, sparse vector methods, symbolic computation, sparse inversion, augmented matrix methods and much more. These routines are intended for educational and research applications. They complement and are able to work with other well known packages such as the Harwell routines, SPARSKIT, and MATLAB. In fact, new routines may be added to the basic SMMS routines in any language by anyone. The only requirement is that the SMMS rules have to be followed.

SMMS is freely available from `ftp://eceserv0.ece.wisc.edu:/pub/smms93`.

Chapter 4

Experimental Evaluation

This chapter gives an overview of the results obtained by numerical experimentation. First an overview of the computer systems and the properties of the matrices used in the experiments is given. Then detailed performance results for several sparse matrix packages are presented.

4.1 Experimental Setup

This section gives an short overview over the hardware platforms used in the tests and summarizes the characteristics of the matrices used.

4.1.1 Hardware Platforms

Numerical experiments were conducted in five different computing environments, but primarily on an SGI Cray Origin2000. The following five sections give a brief overview of the computer systems used.

SGI Cray Origin2000

The SGI Cray Origin2000 used in the experiments is a 64 processor system. All 64 MIPS R10000 processors are running at 250 MHz. Each processor has 4 MB second level cache. Important characteristics of the computer system are:

- **500 MFlop/s** peak performance per processor,
- 22 GB shared main memory,
- IRIX64 Version 6.5 operating system,
- MIPS Power Fortran 77 Compiler Version 7.2.1, and
- SGI/Cray Scientific Library (SCSL) used as BLAS library.

SGI Power Challenge XL

The computer system used consists of three machines, each having 18 MIPS R10000 processors running at 195 MHz. Each machine is equipped with 6 GB shared main memory; additionally, each processor has 2 MB second level cache. Important characteristics of the computer system are:

- **390 MFlop/s** peak performance per processor,
- 6 GB shared main memory, eight way interleaved,
- IRIX64 Version 6.2 operating system,
- MIPS Power Fortran 77 Compiler Version 7.0, and
- SGI Scientific Math. Library, `complib.sgimath`, used as BLAS library.

DEC AlphaServer 2100 4/275

The AlphaServer 2100 used in the experiments is a symmetric multiprocessor based on Digital's Alpha processor A21064A. Four processors share a common main memory of 1 GB which is accessed through a single memory bus; the bus has a peak data transfer bandwidth of 666 MB/s. Important characteristics are:

- **275 MFlop/s** peak performance per processor,
- 1 GB shared main memory,
- Digital UNIX Version 3.2 operating system,
- DEC Fortran 77 Compiler Version 3.7-063, and
- Digital Extended Math Library (DXML) used as BLAS library.

HP 9000-K460

The HP 9000-K460 used in these experiments is a symmetric multiprocessor equipped with four HP PA-RISC 8000 processors running at 180 MHz. Important characteristics are:

- **720 MFlop/s** peak performance per processor,
- 2 GB main memory,
- HP-UX 10.20 operating system,
- HP-UX Fortran 77 compiler Version B.10.20, and
- a vendor supplied BLAS library tuned for PA-RISC 2.0.

IBM RS/6000-390

This workstation is based on one IBM Power2 processor running at 67 MHz. Important characteristics are:

- **268 MFlop/s** peak performance,
- 256 MB main memory,
- AIX Version 4.1.4 operating system,
- xlf Version 3.2.4 Fortran 77 compiler, and
- IBM's ESSL Version 2.2.2 used as BLAS library.

4.1.2 Characteristics of the Test Matrices

The University of Florida sparse matrix collection [19] (see Section 3.4.4) contains 264 unsymmetric sparse matrices in assembled equation form. This set includes the Harwell-Boeing Sparse Matrix Collection [33] (see Section 3.4.2), Saad's collection [80], Bai's collection [12], and matrices from other sources. A subset of this collection, containing the larger matrices, has been used in the experiments.

Table 4.1 lists the unsymmetric matrices used in the benchmarks together with some characteristics of their nonzero structures. Structural symmetry s (denoted "Symmetry" in Table 4.1) is defined to be the fraction of the nonzeros matched by nonzeros in symmetric locations. The SPARSKIT routine `ANSYM` has been used to compute the values of s. None of the matrices are numerically symmetric.

4.2 Performance of Serial Codes

4.2.1 Effect of Tuning Parameters

Default values or parameter values suggested in relevant documents have been used in the experiments. Several parameters have been tuned to improve performance:

Threshold pivoting parameter u: Allowing larger multipliers results in (i) less fill-in, (ii) a decreasing factorization time, and (iii) a decreasing accuracy.

Number of columns to search: A larger number of rows/columns included in the pivot search process increases the factorization time, while the effect on storage requirements and accuracy is usually small.

Name	Symmetry	Size	Non-zeros	nnz/row
jpwh991	.947	991	6027	6.1
rdist1	.062	4134	9408	2.3
orsreg1	**1.00**	2205	14133	6.4
sherman3	**1.00**	5005	20033	4.0
sherman5	.780	3312	20793	6.3
saylr4	**1.00**	3564	22316	6.3
mcfe	.709	765	24382	31.8
lnsp3937	.869	3937	25407	6.5
gemat11	.002	4929	33185	6.7
orani678	.073	2529	90158	35.6
memplus	**1.00**	17758	99147	5.6
wang3	**1.00**	26064	177168	6.8
goodwin	.642	7320	324772	44.4
shyy161	.769	76480	329762	4.3
graham1	.724	9035	335504	37.1
onetone1	.010	36057	341088	9.4
lhr17c	.002	17576	381975	21.7
garon2	**1.00**	13535	390607	28.8
ex40	**1.00**	7740	458012	59.1
epb3	.723	84617	463625	5.4
psmigr1	.481	3140	543162	172.9
af23560	.997	23560	484256	20.5
lhr34c	.002	35152	764014	21.7
dense1000	**1.00**	1000	1000000	1000
rim	.646	22560	1014951	44.9
olafu	**1.00**	16146	1015156	62.8
ex11	**1.00**	16614	1096948	66.0
twotone	.276	120750	1224224	10.1
raefsky4	**1.00**	19779	1316789	66.6
raefsky3	**1.00**	21200	1488768	70.2
lhr71c	.002	70304	1528092	21.7
vavasis3	.001	41092	1683902	41.0
av41092	.001	41092	1683902	40.9
venkat01	**1.00**	62424	1717792	27.5
bbmat	.534	38744	1771722	45.7
appu	**1.00**	14000	1853104	132.3
rma10	**1.00**	46835	2374001	50.6
pre2	.358	659033	5959282	9.0

Table 4.1: Unsymmetric matrices used in the experiments.

MA48

Matrix	∞	20 %	10 %	0 %
lnsp3937	**1.0**	1.02	1.27	1.34
saylr4	**1.0**	1.04	1.06	4.47
sherman3	**1.0**	1.03	1.15	4.23

UMFPACK

Matrix	∞	20 %	10 %	0 %
lnsp3937	**1.0**	1.02	1.03	1.05
saylr4	**1.0**	1.04	1.06	1.12
sherman3	**1.0**	1.02	1.02	1.03

Table 4.2: Impact of size of "elbow room" on runtime of UMFPACK and MA48. Runtimes are given relative to infinite amount of "elbow room".

Density d to switch to dense matrix codes: The transition is made when the ratio of the number of entries in the reduced matrix to the number that it would have as a full matrix is greater than d. Smaller values of d result in decreased factorization times at the cost of increased storage requirements.

Block size in Level 3 BLAS operations: For most RISC-based architectures there is a wide range of optimal values for the block size. So the effect on factorization times is very small.

Size of "elbow room": Each routine requires a minimum size of working storage. If the amount of working storage provided is close to this minimum, the performance some routines (like MA48) may suffer dramatically, whereas in the majority of cases the impact on the performance is rather small (cf. Table 4.2).

4.2.2 Performance of SuperLU

Table 4.3 shows some performance statistics for the SUPERLU code running on the SGI Origin2000. All floating-point computations were done in double precision. A few matrices were symmetrically permuted by a symmetric minimum degree ordering on $A + A^\top$. For all other matrices, the columns were permuted by a minimum degree ordering of $A^\top A$.

In the SUPERLU code, `DGEMV` typically accounts for more that 80 % of the floating-point operations. The performance of the BLAS is clearly an upper bound on the overall performance achieved in the factorization. However, because symbolic manipulation takes a nontrivial amount of time, this bound could be much higher than the actual performance of the code.

The performance of the SuperLU code depends strongly on the sparsity structure of the system matrix. For example, matrices `memplus` and `gemat11` entail poor performance rates. Compared to other matrices, these two matrices are sparser, have less fill-in, and have smaller supernodes, so that supernodal techniques are less favorably applicable. `gemat11` is also highly unsymmetric, which makes the symmetric structural reduction technique less effective.

For some larger and denser matrices such as `raefsky4` a floating-point performance of about 175 Mflop/s can be achieved, which is more than a third of the theoretical peak performance of the machine.

Table 4.4 shows performance statistics for the SuperLU code running on the SGI PowerChallenge XL. In general, the SuperLU code on the Power Challenge shows the same performance behavior as on the Origin2000. Typically, the Origin2000 is about 30 % faster than the Power Challenge XL, which corresponds to the different clock rates of the processors (195 MHz versus 250 MHz).

Table 4.5 shows some performance statistics for the SuperLU code running on the HP K-460.

4.2.3 Performance of Harwell Codes

Table 4.6 shows performance results for MA38 running on the Cray Origin2000. The column "Refactorization" in Table 4.6 gives the time to factorize a matrix with the same sparsity structure as one which has already been factorized. For most matrices, this refactorization offers a significant saving. However, for the matrix `wang3`, the refactorization offered by MA38 was significantly slower than the original factorization. The memory requirements of MA38 are shown in Table 4.7. The amount of main memory used by MA38 is given in the column labeled "used" while the column labeled "required" gives the minimum amount of memory needed by MA38 to succeed (althoug many garbage collections might be required in this case).

Table 4.8 shows performance results for MA48 running on the Cray Origin2000. In general, MA48 needs much time for the symbolic factorization. For example, in the case of matrix `pre2` runtimes may be prohibitive. The memory requirements of MA48 are similar to those of MA38.

In general, no single code is clearly better than the others. The optimal choice of code depends on the type of problem being solved. As expected, MA41 achieves the shortest factorization times for problems with a nearly symmetric structure. For these problems, the ordering obtained by applying MC40 to the pattern of $A + A^\top$ enables MA42 to perform the matrix factorization more rapidly than both MA38 and MA48, but the frontal code has the disadvantage of generally producing many more entries in the factors than the other codes.

For problems which are far from symmetric in structure, the factorize time for MA38 is less than the sum of analyze time and factorize time for MA48, except for matrices which are very sparse.

Matrix	Factorization time [s]	Factorization performance [Mflop/s]	Solution time [s]	Solution performance [Mflop/s]	Fill-in ratio	Memory Requirements [MB]
jpwh991	0.33	37.65	0.02	23.34	26.54	7.74
rdist1	0.28	62.41	0.03	26.26	4.17	4.76
orsreg1	0.98	52.59	0.03	20.43	32.33	8.43
sherman3	0.42	110.11	0.03	26.16	19.58	4.98
sherman5	0.31	73.05	0.02	23.60	11.35	3.21
saylr4	0.61	102.52	0.04	26.74	23.96	6.12
mcfe	0.08	36.16	0.01	12.63	2.59	0.87
lnsp3937	1.20	30.03	0.01	12.33	22.03	2.23
gemat11	0.27	12.34	0.01	11.12	2.80	1.12
orani678	2.47	22.97	0.02	14.64	6.91	5.67
memplus	0.76	52.71	0.05	15.80	3.13	7.98
wang3	249.37	139.26	2.22	24.67	154.58	280.11
goodwin	4.70	107.66	0.21	26.29	8.50	31.34
graham1	14.68	97.03	0.69	13.01	13.38	51.13
shyy161	10.07	103.11	0.54	24.62	20.16	83.53
onetone1	21.69	117.08	0.39	24.01	13.72	56.79
lhr17c	2.15	50.98	0.31	10.96	4.45	22.23
garon2	9.82	129.86	0.32	29.82	12.22	52.20
ex40	22.12	103.29	0.61	18.91	12.59	63.86
epb3	14.86	120.65	0.86	23.20	21.51	119.74
psmigr1	156.48	106.28	1.07	16.28	16.04	88.78
af23560	32.29	153.38	0.79	32.90	26.83	136.75
lhr34c	4.13	57.03	0.29	24.11	4.58	45.44
dense1000	3.13	212.84	0.05	40.00	1.00	10.04
rim	19.13	110.67	0.90	22.15	9.82	112.75
olafu	17.88	165.57	0.54	29.35	7.81	83.89
ex11	73.39	194.53	1.31	30.28	18.08	210.15
twotone	190.87	65.08	2.50	19.78	20.20	302.15
raefsky4	61.12	175.54	1.03	31.61	12.25	166.68
raefsky3	55.97	138.50	1.26	22.45	9.50	147.25
lhr71c	9.02	54.42	0.73	19.51	4.66	92.26
vavasis3	468.19	138.77	2.76	28.95	23.73	420.45
av41092	497.39	130.63	4.00	19.98	23.73	420.45
venkat01	39.08	148.04	1.12	32.63	10.64	194.26
bbmat	446.00	99.95	3.61	27.80	28.32	540.49
appu	17406.13	92.61	19.12	19.60	101.09	1911.97
rma10	> 20000.00	–	–	–	–	–
pre2	13629.19	66.71	51.48	13.74	59.41	3245.67

Table 4.3: SuperLU on an Cray Origin2000 (500 Mflop/s peak performance).

Matrix	Factorization time [s]	Factorization performance [Mflop/s]	Solution time [s]	Solution performance [Mflop/s]	Fill-in ratio	Memory Requirements [MB]
jpwh991	0.43	28.99	0.03	17.97	26.54	7.74
rdist1	0.35	49.64	0.04	20.89	4.17	4.76
orsreg1	1.23	42.06	0.04	16.34	32.33	8.43
sherman3	0.56	82.59	0.04	19.62	19.58	4.98
sherman5	0.41	55.07	0.03	17.79	11.35	3.21
saylr4	0.80	78.26	0.05	20.41	23.96	6.12
mcfe	0.10	28.63	0.01	10.00	2.59	0.87
lns3937	1.59	22.44	0.01	9.21	22.03	2.23
lnsp3937	1.51	23.88	0.01	9.81	22.03	2.23
gemat11	0.35	9.61	0.01	8.66	2.80	1.12
orani678	3.30	17.18	0.03	10.95	6.91	5.67
memplus	0.98	40.79	0.06	12.23	3.13	7.98
wang3	317.28	109.45	2.82	19.39	154.58	280.11
goodwin	6.06	83.45	0.27	20.38	8.50	31.34
shyy161	13.25	78.36	0.71	18.71	20.16	83.53
onetone1	28.26	89.87	0.51	18.43	13.72	56.79
lhr17c	2.87	38.21	0.41	8.21	4.45	22.23
garon2	12.98	98.28	0.42	22.57	12.22	52.20
ex40	28.93	78.97	0.80	14.46	12.59	63.86
epb3	19.28	93.00	1.12	17.88	21.51	119.74
af23560	42.21	117.34	1.03	25.17	26.83	136.75
lhr34	5.99	39.08	0.40	18.25	4.82	47.71
lhr34c	5.54	42.55	0.39	17.99	4.58	45.44
dense1000	5.02	154.59	1.14	12.70	1.00	30.34
rim	25.71	82.34	1.21	16.48	9.82	112.75
olafu	24.10	122.83	0.73	21.77	7.81	83.89
ex11	95.86	148.93	1.71	23.18	18.08	210.15
twotone	252.78	49.14	3.31	14.94	20.20	302.15
raefsky4	80.23	133.73	1.35	24.08	12.25	166.68
raefsky3	74.54	104.00	1.68	16.86	9.50	147.25
lhr71c	11.55	42.49	0.94	15.23	4.66	92.26
vavasis3	609.12	106.66	3.59	22.25	23.73	420.45
av41092	650.63	99.86	5.23	15.27	23.73	420.45
venkat01	49.30	117.36	1.41	25.87	10.64	194.26
bbmat	592.77	75.20	4.80	20.92	28.32	540.49
appu	23248.61	69.34	25.54	14.67	101.09	1911.97

Table 4.4: SuperLU on a Power Challenge XL (390 Mflop/s peak performance).

Matrix	Factorization time [s]	Factorization performance [Mflop/s]	Solution time [s]	Solution performance [Mflop/s]	Fill-in ratio	Memory Requirements [MB]
jpwh991	0.26	47.66	0.02	29.54	26.54	7.41
rdist1	0.11	154.80	0.01	65.14	4.17	4.92
orsreg1	0.99	51.96	0.03	20.18	32.33	8.51
sherman3	0.21	217.86	0.02	51.76	19.58	5.17
sherman5	0.14	159.29	0.01	51.46	11.35	3.16
saylr4	0.67	93.63	0.04	24.42	23.96	6.07
mcfe	0.05	62.50	0.01	21.83	2.59	0.86
lns3937	1.10	32.39	0.01	13.30	22.03	2.24
lnsp3937	1.29	27.95	0.01	11.48	22.03	2.28
gemat11	0.23	14.28	0.01	12.87	2.80	1.12
orani678	2.12	26.79	0.02	17.08	6.91	5.69
memplus	0.85	46.96	0.06	14.08	3.13	8.31
wang3	287.31	120.87	2.56	21.41	154.58	293.53
goodwin	3.99	126.78	0.18	30.96	8.50	32.10
shyy161	9.08	114.37	0.49	27.31	20.16	86.18
onetone1	14.04	180.90	0.25	37.10	13.72	56.85
lhr17c	1.86	58.96	0.27	12.68	4.45	21.37
garon2	10.14	125.76	0.33	28.88	12.22	54.06
ex40	15.21	150.24	0.42	27.51	12.59	66.06
epb3	7.78	230.33	0.45	44.29	21.51	125.10
af23560	15.85	312.39	0.39	67.01	26.83	133.91
lhr34	3.36	69.78	0.23	32.60	4.82	50.05
lhr34c	2.73	86.26	0.19	36.47	4.58	45.86
dense1000	1.75	443.11	0.40	36.42	1.00	31.64
rim	21.83	96.99	1.03	19.41	9.82	114.25
olafu	13.32	222.26	0.40	39.40	7.81	82.43
ex11	31.20	457.62	0.56	71.23	18.08	214.99
twotone	84.59	146.84	1.11	44.63	20.20	316.84
raefsky4	71.08	150.95	1.20	27.18	12.25	165.74
raefsky3	47.56	163.00	1.07	26.42	9.50	152.99
lhr71c	3.65	134.49	0.30	48.21	4.66	94.02
vavasis3	457.11	142.13	2.69	29.65	23.73	408.17
av41092	310.15	209.49	2.49	32.04	23.73	411.34
venkat01	19.27	300.25	0.55	66.18	10.64	201.67
bbmat	350.16	127.31	2.83	35.41	28.32	546.61
appu	13913.87	115.85	15.28	24.52	101.09	1925.82

Table 4.5: SuperLU on an HP K-460 (720 Mflop/s peak performance).

Matrix	Factorization time [s]	Factorization performance [Mflop/s]	Refactorization time [s]	Refactorization performance [Mflop/s]	Solution time [s]
jpwh991	0.15	45.6	0.04	160.2	< 0.01
rdist1	10.82	112.4	7.26	166.3	0.08
orsreg1	0.55	69.4	0.18	201.1	0.01
sherman3	0.62	56.3	0.22	149.5	0.02
sherman5	0.47	44.3	0.14	142.0	0.01
saylr4	0.90	63.3	0.34	158.8	0.02
mcfe	0.17	18.9	0.04	75.1	< 0.01
lnsp3937	1.68	66.9	0.66	165.9	0.04
gemat11	0.24	3.8	0.06	12.2	0.01
orani678	1.15	10.5	0.14	82.6	0.00
memplus	1.34	1.8	0.22	9.1	0.05
wang3	288.18	163.8	370.43	126.9	2.14
goodwin	10.19	104.2	5.96	175.8	0.13
shyy161	36.68	103.1	20.50	183.0	0.36
graham1	37.88	112.6	22.09	191.9	0.31
onetone1	60.87	90.1	38.58	141.3	0.46
lhr17c	47.51	47.4	14.75	150.7	0.33
garon2	40.11	144.6	29.65	194.2	0.32
ex40	85.18	132.8	74.75	150.5	0.50
epb3	63.16	112.3	39.41	178.5	0.69
psmigr1	84.61	115.5	53.09	183.1	0.38
af23560	210.25	131.7	188.38	146.1	1.47
lhr34c	183.39	65.5	88.68	134.4	0.88
dense1000	5.54	122.0	3.47	192.3	0.03
rim	97.85	112.2	65.67	166.2	0.96
olafu	20.63	128.9	14.31	182.1	0.25
ex11	371.75	145.7	331.86	162.5	1.99
twotone	481.90	103.7	337.17	147.9	1.98
raefsky4	153.64	162.0	133.97	184.3	1.05
raefsky3	79.53	141.6	64.67	172.3	0.79
lhr71c	392.72	60.4	191.34	123.0	3.23
vavasis3	54.58	51.7	20.79	134.5	0.69
av41092	50.82	55.6	20.76	134.7	0.65
venkat01	59.41	120.6	41.63	169.0	0.66
bbmat	1980.43	123.2	1860.18	130.9	6.14
appu	8119.78	19.0	7440.59	20.5	20.70
rma10	288.50	123.0	257.35	137.0	2.49
pre2	18187.93	65.1	16826.97	70.3	33.00

Table 4.6: MA38 on an SGI Cray Origin2000 (500 Mflop/s peak performance).

Matrix	Memory [MB] used	required
jpwh991	2.2	1.1
rdist1	71.2	32.7
orsreg1	6.2	3.2
sherman3	7.1	3.8
sherman5	6.0	2.8
saylr4	10.2	5.3
mcfe	1.7	1.1
lnsp3937	18.5	8.3
gemat11	1.9	1.6
orani678	7.9	3.8
memplus	6.4	5.2
wang3	851.7	437.8
goodwin	87.3	45.8
shyy161	166.1	86.1
graham1	170.0	94.2
onetone1	220.3	103.9
lhr17c	118.9	64.7
garon2	212.8	100.5
ex40	381.7	159.8
epb3	349.8	171.9
psmigr1	378.3	172.2
af23560	609.9	291.3
lhr34c	419.1	176.6
dense1000	43.6	28.1
rim	362.3	215.7
olafu	162.0	98.0
ex11	949.2	416.8
twotone	813.1	424.6
raefsky4	522.1	272.4
raefsky3	292.7	198.8
lhr71c	519.8	273.9
vavasis3	237.7	122.2
av41092	237.7	122.2
venkat01	309.2	205.3
bbmat	2821.1	1201.4
appu	9047.5	4506.7
rma10	928.0	528.5
pre2	8871.0	5807.2

Table 4.7: Memory requirements of MA38 on an SGI Cray Origin2000.

Matrix	Symbolic Factorization time [s]	Factorization time [s]	Factorization performance [Mflop/s]	Refactorization time [s]	Refactorization performance [Mflop/s]	Solution time [s]
jpwh991	0.06	0.04	186.2	0.03	248.2	0.00
rdist1	81.25	19.48	56.7	18.23	60.6	0.15
orsreg1	0.56	0.23	156.3	0.19	189.2	0.01
sherman3	0.84	0.43	133.2	0.38	150.7	0.03
sherman5	0.52	0.16	135.2	0.14	154.6	0.01
saylr4	1.44	0.47	138.2	0.40	162.4	0.02
mcfe	0.19	0.06	96.6	0.04	145.0	0.00
lnsp3937	2.26	0.65	134.6	0.54	162.1	0.02
gemat11	0.08	0.03	18.8	0.01	56.4	0.01
orani678	0.27	0.09	118.8	0.07	152.7	0.01
memplus	0.33	0.11	45.7	0.06	83.8	0.03
wang3	560.06	263.35	14.1	265.10	14.0	2.12
goodwin	47.17	12.87	118.9	11.36	134.7	0.20
shyy161	637.40	142.76	45.2	156.47	41.2	1.15
graham1	315.80	94.25	32.1	86.13	35.1	0.62
onetone1	313.24	80.24	69.2	75.86	73.2	0.56
lhr17c	9.21	12.51	290.6	12.29	295.8	0.36
garon2	123.55	34.30	132.1	30.71	147.5	0.33
ex40	727.30	246.01	66.3	246.02	66.3	1.21
epb3	260.95	77.39	5.3	69.98	5.9	1.31
psmigr1	8.66	36.15	300.4	35.90	302.5	0.33
lhr34c	25.97	54.99	295.2	53.70	302.3	0.75
dense1000	1.13	2.58	25.8	2.45	27.2	0.12
rim	681.68	190.70	52.6	177.85	56.4	1.04
olafu	129.85	45.62	132.9	43.34	139.9	0.58
ex11	2439.56	856.17	116.8	852.86	117.2	3.56
twotone	777.23	294.22	92.8	299.74	91.1	1.76
raefsky4	513.33	212.61	119.7	196.36	129.6	1.53
raefsky3	360.36	143.03	97.7	137.02	102.0	1.27
lhr71c	100.50	121.16	255.9	116.86	265.3	2.26
vavasis3	257.06	48.76	9.0	44.73	9.8	0.95
av41092	250.22	50.37	8.7	44.84	9.8	1.15
venkat01	189.53	65.67	86.2	61.17	92.5	1.27
bbmat	23191.45	8511.94	49.8	7827.85	54.1	14.72
appu	329.68	5071.30	263.9	5018.95	266.6	19.02
rma10	3129.80	1090.91	33.4	1038.17	35.1	4.56
pre2	> 150000.00	–	–	–	–	–

Table 4.8: MA48 on an SGI Cray Origin2000 (500 Mflop/s peak performance).

4.2.4 Performance of S$^+$

Table 4.9 gives a few run time values obtained with S$^+$ and a comparison with run times of SuperLU and LazyS$^+$, an improved version of S$^+$ with advanced memory management. Table 4.10 shows the respective memory requirements.

Matrix	S^+	$SuperLU$	$LazyS^+$
raefsky4	658.0	857.6	606.7
memplus	344.6	322.3	44.2
TIb	1325.3	1360.6	73.9

Table 4.9: Run time of sequential code on an Alpha server 2100 4/275.

Matrix	S^+	$SuperLU$	$LazyS^+$
raefsky4	303.6	272.9	285.9
memplus	138.2	75.2	68.4
TIb	342.4	221.3	107.7

Table 4.10: Memory requirements [MB].

4.2.5 Comparison of Direct Solvers

Each of the codes MA38, MA41, MA42, MA48, and SuperLU is able to factorize general unsymmetric matrices. All these codes use dense matrix kernels (Blas [27]) to some extent. Each code has a set of input parameters that control its behavior; for most of them the recommended default values have been used, with a few exceptions noticed later. The default value of the threshold u varies with each code. For MA42, it is 0.01.

The results, shown in Table 4.11, include the following statistics for each code:

- Factorization time, which includes preordering and symbolic factorization, if any.
- Refactorization time, which is the numerical factorization of a matrix whose pivot ordering and symbolic factors are known. It excludes preordering and symbolic factorization.
- Solution time, excluding iterative refinement.
- Entries in the LU factors, in millions.
- Memory requirements, in MB.

	Symbolic factorization time [s]				
Matrix	MA38	MA41	MA42	MA48	SuperLU
av41092	–	25.7	56.3	250.2	–
twotone	–	13.7	14.5	777.2	–
onetone1	–	1.9	1.4	313.2	–
raefsky4	–	0.7	0.4	515.3	–

	Factorization time [s]				
Matrix	MA38	MA41	MA42	MA48	SuperLU
av41092	50.8	>10000	>10000	50.4	497.4
twotone	481.9	7951.8	>10000	294.2	190.9
onetone1	60.9	103.8	429.4	80.2	21.7
raefsky4	153.6	27.1	147.4	212.6	61.1

	Refactorization time [s]				
Matrix	MA38	MA41	MA42	MA48	SuperLU
av41092	20.8	>10000	>10000	44.8	497.4
twotone	337.2	7951.8	>10000	299.7	190.9
onetone1	38.6	103.8	429.4	75.9	21.7
raefsky4	134.0	27.1	147.4	196.4	61.1

	Solution time [s]				
Matrix	MA38	MA41	MA42	MA48	SuperLU
av41092	0.65	–	–	1.15	4.00
twotone	1.98	41.49	–	1.76	2.50
onetone1	0.46	2.57	3.73	0.56	0.39
raefsky4	1.05	1.46	9.28	1.53	1.03

	Entries in LU factors [$\times 10^6$]				
Matrix	MA38	MA41	MA42	MA48	SuperLU
av41092	39.38	–	–	60.84	39.95
twotone	9.75	43.45	–	26.34	24.73
onetone1	4.69	9.50	16.58	5.11	4.86
raefsky4	18.32	32.63	127.10	33.54	16.13

	Memory used [MB]				
Matrix	MA38	MA41	MA42	MA48	SuperLU
av41092	237.7	>10000.0	–	434.2	420.7
twotone	813.1	2403.1	–	1103.2	302.1
onetone1	220.3	433.5	28.5	291.3	56.8
raefsky4	522.1	120.4	50.7	645.2	166.7

Table 4.11: Serial codes on an SGI Cray Origin2000.

Since the codes being compared all offer quite different capabilities and are designed for different environments and different types of matrices, the results should not be interpreted as a direct comparison between them. For example, MA38 is designed for structurally unsymmetric matrices. A code like MA41 would normally be expected to perform much better when applied to (nearly) symmetric matrices. For example, matrix `av41092` is extremely unsymmetric and hence MA41 needs an enormous amount of run time and memory to factorize this matrix. On the other hand, MA41 can factorize matrix `raefsky4`, which has a symmetric sparsity pattern, faster than MA38. For MA38 and SuperLU no run times for the symbolic factorization are given since in these codes the symbolic and numerical factorization are combined into a single subroutine. MA41, MA42, and SuperLU do not offer a refactorization option, therefore the refactorization time is the same as the factorization time.

Only the run times obtained from the SGI Cray Origin2000 are compared in Table 4.11. Note, however, that the behavior of the codes can be strongly influenced by the computing platform being used (Duff and Scott [40]).

4.3 Performance of Parallel Codes

This section summarizes the performance of parallel codes running on the SGI Cray Origin2000 using a maximum number of 16 processors.

4.3.1 Performance of SuperLU_MT

Table 4.12 shows the speed-up of the multi-threaded SuperLU_MT code on the SGI Cray Origin2000. The speed-up is measured against the best sequential runtime achieved by SuperLU on a single processor of this machine.

The column labeled "$p = 1$" illustrates the overhead in the parallel code when compared with the sequential code, using the same blocking parameters. The structure of the parallel code SuperLU_MT, when run on a single processor, does not differ much from the sequential code SuperLU, except that a global task and various locks are involved. The extra work in the parallel code is purely integer operations.

The last two columns in Table 4.12 show the factorization time and the floating-point performance [Mflop/s], respectively, corresponding to $p = 16$, the largest number of processors used in the experiments (peak performance: 8 Gflops). Each experiment has been repeated at least three times; the run time variation was up to 30 %.

Table 4.13 shows the relative memory requirements of SuperLU_MT using p processors compared with the memory requirements of the serial version. The last column gives the absolute amount of memory needed when 16 processors are used.

Matrix	Speed-up $p=1$	$p=4$	$p=6$	$p=8$	$p=12$	$p=16$	Time	Mflop/s
jpwh991	0.97	1.74	1.57	1.27	0.80	0.32	1.04	17.8
rdist1	0.85	1.00	0.93	0.76	0.72	0.57	0.49	34.6
orsreg1	1.85	3.38	2.80	2.18	1.72	1.21	0.81	57.3
sherman3	0.78	1.14	1.14	0.98	0.65	0.33	1.26	34.4
sherman5	0.86	1.19	1.03	0.97	0.79	0.50	0.62	36.7
saylr4	0.80	1.36	1.33	1.36	1.20	0.75	0.81	72.4
mcfe	0.57	0.57	0.67	0.44	0.47	0.22	0.36	14.8
goodwin	0.98	3.01	3.48	3.79	4.39	2.64	1.78	282.1
shyy161	0.99	1.85	2.05	1.83	1.49	1.33	7.58	110.2
onetone1	0.77	2.72	3.49	3.44	3.73	3.23	6.71	388.6
lhr17c	0.92	1.67	1.94	1.72	1.44	0.90	2.40	45.8
garon2	1.00	2.89	3.71	4.02	4.50	3.11	3.16	403.0
epb3	0.74	1.83	2.28	2.18	1.62	1.33	11.19	148.7
af23560	0.97	2.66	3.69	4.36	4.76	4.67	6.92	701.8
lhr34c	0.91	1.82	1.77	1.89	1.40	1.32	3.14	75.1
dense1000	0.99	2.98	3.48	3.16	3.07	2.87	1.09	612.8
rim	0.93	3.10	4.60	4.94	5.27	2.73	7.02	286.4
olafu	0.88	2.76	3.47	3.78	3.96	3.84	4.66	653.8
twotone	1.00	3.69	4.83	5.39	6.22	7.50	25.44	477.0
raefsky4	1.00	2.94	4.26	4.93	6.57	6.72	9.09	1224.7
raefsky3	0.99	3.88	5.11	6.70	7.32	7.01	7.98	980.5
lhr71c	0.98	2.02	2.12	1.89	1.31	0.82	10.95	44.4
vavasis3	1.00	3.11	4.47	5.66	7.70	8.98	52.15	1231.0
av41092	0.90	3.26	5.05	5.88	7.45	9.64	51.61	1243.9
venkat01	0.78	2.55	3.48	4.02	4.29	2.47	15.83	367.7
bbmat	0.98	3.33	4.66	6.02	8.65	9.35	47.70	927.0

Table 4.12: Speed-up of SuperLU_MT compared to SuperLU on an SGI Cray Origin2000 (peak performance: $16 \times 500 = 8000$ Mflop/s). Absolute run times (in seconds) and floating-point performance are given for $p = 16$ processors.

	Relative Memory Requirements						
Matrix	$p = 1$	$p = 4$	$p = 6$	$p = 8$	$p = 12$	$p = 16$	[MB]
jpwh991	1.18	1.63	1.95	2.26	2.88	3.50	6.1
rdist1	1.09	1.72	2.13	2.55	3.39	4.22	20.1
orsreg1	1.09	1.46	1.70	1.95	2.43	2.92	13.0
sherman3	1.18	1.91	2.39	2.87	3.84	4.80	23.9
sherman5	1.16	1.90	2.40	2.90	3.89	4.88	15.7
saylr4	1.15	1.58	1.86	2.13	2.69	3.25	19.9
mcfe	1.36	2.19	2.66	3.16	4.19	5.18	4.5
goodwin	1.04	1.21	1.32	1.43	1.65	1.88	58.9
shyy161	1.12	1.78	2.22	2.65	3.54	4.41	368.7
onetone1	1.13	1.59	1.89	2.20	2.81	3.42	194.3
lhr17c	1.13	1.70	2.08	2.46	3.22	3.98	88.4
garon2	1.04	1.23	1.36	1.48	1.73	1.98	103.2
epb3	1.08	1.59	1.93	2.27	2.95	3.63	434.4
af23560	1.07	1.19	1.27	1.36	1.52	1.69	230.8
lhr34c	1.13	1.69	2.06	2.43	3.17	3.91	177.9
dense1000	1.02	1.10	1.15	1.21	1.31	1.42	14.3
rim	1.00	1.15	1.24	1.34	1.53	1.72	194.2
olafu	1.05	1.19	1.29	1.38	1.56	1.75	146.7
twotone	1.01	1.32	1.52	1.73	2.14	2.55	719.5
raefsky4	1.05	1.14	1.20	1.26	1.37	1.48	247.2
raefsky3	1.04	1.14	1.21	1.28	1.42	1.56	229.5
lhr71c	1.13	1.68	2.04	2.41	3.14	3.87	357.0
vavasis3	1.03	1.10	1.14	1.19	1.28	1.38	579.4
av41092	1.03	1.10	1.14	1.19	1.28	1.38	579.4
venkat01	1.08	1.31	1.46	1.62	1.93	2.23	434.0
bbmat	1.06	1.11	1.15	1.18	1.25	1.32	713.8

Table 4.13: Relative memory requirement of SUPERLU_MT compared to SUPERLU on an SGI Cray Origin2000. Absolute memory requirements (in [MB]) are given for $p = 16$ processors.

Table 4.14 shows the speed-up of the multi-threaded SuperLU_MT code on the SGI Power Challenge XL. The run time variation was less than 5 %. This might be due to the fact that there was no other load on the SGI Power Challenge XL during the experiments, whereas the SGI Cray Origin2000 was loaded with other jobs (but none of the jobs had to compete with other processes for CPU or memory requirements).

	Speed-up					
Matrix	$p = 1$	$p = 4$	$p = 8$	$p = 12$	Time	Mflop/s
memplus	0.69	1.64	1.79	1.73	0.38	4.45
gemat11	0.86	1.93	2.26	3.65	0.06	24.18
rdist1	0.88	1.61	1.54	2.27	0.41	34.16
orani678	0.70	1.76	2.42	2.55	0.41	36.55
mcfe	0.68	2.00	2.18	3.43	0.06	65.04
lnsp3937	0.98	3.07	3.70	3.98	0.33	128.70
lns3937	0.98	3.00	3.77	3.84	0.35	132.82
sherman5	0.85	2.37	2.98	3.23	0.22	118.63
jpwh991	0.81	2.38	3.60	5.24	0.08	237.99
sherman3	0.83	2.46	2.90	2.82	0.36	172.65
orsreg1	0.86	2.54	2.79	2.84	0.30	201.54
saylr4	0.94	2.79	3.50	4.76	0.36	299.38
shyy161	0.88	2.77	3.44	4.95	3.95	390.32
goodwin	0.86	3.58	5.16	5.97	1.37	492.98
venkat01	0.65	1.79	2.06	1.92	13.83	232.24
inaccura	0.88	2.77	3.94	5.18	8.51	490.61
bai	0.88	3.02	4.95	6.85	8.05	795.84
dense1000	0.85	2.63	3.31	3.97	0.82	810.69
raefsky3	0.91	2.98	5.63	6.58	10.52	1154.87
ex11	0.96	3.23	5.88	7.82	24.55	1148.22
wang3	0.87	2.24	3.49	3.86	19.39	750.41
raefsky4	0.94	3.15	5.12	6.51	30.20	1040.49
vavasis3	0.90	3.75	6.00	6.85	93.69	966.60

Table 4.14: Speed-up of SuperLU_MT compared to SuperLU on an SGI Power Challenge XL (peak performance: $12 \times 390 = 4680$ Mflop/s). Absolute run times (in seconds) and floating-point performance are given for $p = 12$ processors.

4.3.2 Performance of S$^+$

S$^+$ is an MPI based parallel direct solver routine which does everything from symbolic factorization to solving triangular systems. Table 4.15 gives speed-up

values of S^+ achieved for some matrices on an SGI Cray Origin2000. The last column gives the run times for S^+ using 12 processors. S^+ is not competitive with SuperLU_MT, even taking into account that the run times shown in Table 4.15 are total times required for the whole solution process, whereas Table 4.12 shows only the factorization times of SuperLU_MT. Moreover, several experiments did not succeed due to MPI related errors. This is rather a problem of the MPI implementation than a problem inherent in the S^+ package.

	Speed-up					
Matrix	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 12$	Time
af23560	**1.00**	1.29	1.94	3.00	4.63	29.1
goodwin	**1.00**	1.31	1.87	1.47	0.76	6.5
raefsky4	**1.00**	1.25	2.00	4.16	4.06	34.9
wang3	**1.00**	1.04	2.10	4.28	5.46	161.1

Table 4.15: Speed-up of S^+ on an SGI Cray Origin2000. Absolute run times (in seconds) are given for $p = 12$ processors.

Fu et al. [44] have compared SuperLU_MT with their S^+ code on a Sun Ultra-1 workstation and it has turned out that the two codes are competitive, both in terms of run times and memory requirements. Moreover, they have reported that S^+ can deliver up to 10 Gflop/s on 128 nodes of a 450 MHz Cray T3E when solving very large linear systems.

4.3.3 Performance of SPOOLES

Both, the MPI based version and the multi-threaded version of Spooles have been tested. Table 4.16 shows speed-up values of the multi-threaded version on an SGI Cray Origin2000. Only in a few cases it was possible to achieve some speed-up at all. The speed-up behavior of the MPI based version of Spooles is even worse.

	Speed-up							
Matrix	$p = 1$	$p = 2$	$p = 4$	$p = 6$	$p = 8$	$p = 12$	$p = 16$	Time
raefsky4	**1.00**	1.28	1.13	1.04	0.87	0.73	0.70	110.48
wang3	**1.00**	0.73	0.63	0.63	0.36	0.25	0.24	511.54
dense1000	**1.00**	0.72	0.98	0.90	0.69	0.56	0.68	15.85
venkat01	**1.00**	1.32	1.31	1.06	0.92	0.79	0.75	22.49
rma10	**1.00**	0.98	0.87	0.82	0.74	0.77	0.62	20.39

Table 4.16: Speed-up of Spooles on an SGI Cray Origin2000. Absolute run times (in seconds) are given for $p = 16$ processors.

Chapter 5

Solving Large Sparse Systems Arising from IPMs

The following sections discuss an application program from the field of dynamic portfolio management (Pflug and Swietanowski [74]) and the efficient solution of the mathematical problems arising in this application (Dockner et al. [25]. An *Interior Point Method* (IPM) [90] is used for solving the optimization problem. Extremely large and very sparse matrices arise from this approach and the respective linear systems have to be solved efficiently.

5.1 The AURORA System

AURORA is a software system for multiperiod asset liability management planning in a risky environment. Due to the complexity of the problem (with several millions of decision variables) the AURORA programs are written in Fortran 90 and HPF+ [13], which allows the execution on highly parallel computers. The AURORA software system exhibits the following features.

Modeling of the risk factors: The economic environment module is used to specify the risks under which the decisions are made. The system is open to include any possible kind of risk. Typically risks are connected with

- interest rates (like level and curvature of the term structure),
- exchange rates, and
- stock markets.

It is possible to include

- credit risk,
- liquidity risk, etc.

Modeling of risks is done in two steps. In Step 1 a stochastic model of the underlying random process is formulated and validated, using past data and/or the knowledge and the opinion of experts. In Step 2 a discrete scenario model is extracted automatically from this random process. This model is used in the subsequent dynamic stochastic optimization.

Modeling of the financial instruments: Assets (bonds, stocks, forward contracts, futures, options) and liabilities (credits and loans) are contracts which may be bought and sold at any decision moment. To each contract there are associated prices, transaction costs and cashflows. These quantities which depend on the scenario, i.e., on the presently unknown future development of the economic environment.

Modeling of the objective function: The objective of the planning model may be chosen according to the user's conceptions and needs. The main goals are to maximize (i) expected terminal wealth or (ii) expected discounted intermediate wealth. These two objectives are risk neutral. However, risk aversion may be included either in the objective or in the constraints. By changing the risk aversion parameter, different solutions with different investment into risky assets are generated.

More details about the AURORA planning system can be found in Pflug and Swietanowski [73], Dockner [25], Pflug and Swietanowski [74], and Pflug and Swietanowski [75].

5.1.1 Problem Characteristics

Currently the AURORA planning system is used for managing the asset and liability structure of a pension fund, and for making decisions about optimal asset allocation. Some of the characteristics of the model are:

- Long planning horizon (10 – 30 years);
- Length of decision period: Pension funds make investment decisions every six months;
- Investment in many different asset categories possible (national and/or international bonds, stocks, equities, etc.);
- Possible future development of assets is modeled by discrete stochastic processes (*discrete time discrete state Markov processes*);
- Structure of *objective function*: Maximize terminal expected wealth of the fund at the end of the planning horizon by taking into account a negative risk penalty.

5.1.2 Dynamic Stochastic Optimization

In the AURORA planning system the economic characteristics are modeled as a stochastic process with finite state space. The process is assumed to be a Markov chain, which is characterized by transition probabilities. However, the general history tree structure would also allow to consider non-Markovian processes. Some properties of the model are the following:

- Discrete time discrete state stochastic vector processes can be mapped onto a finite *scenario* (*history*) *tree* with nodes $n \in \mathcal{N} = \{1, \ldots, N\}$. The nodes at each level of the tree represent all possible states at a certain period of the planning horizon.
- The branching factor is determined by the discretization of the random variables. In the full scenario lattice (two categories of assets, liabilities) each node has $2^3 = 8$ successors.
- Decisions are made at each node of the tree: How much assets should be bought and sold subject to always being able to meet the liabilities (pay the pensions).
- Tree structures can be translated into linearly constrained stochastic dynamic optimization problems.
- The optimization problem can be formulated as a *linear optimization problem with linear constraints* (here: with *decomposable* objective function—this is not necessarily the case; see Dockner et al. [25])

$$\begin{aligned} \text{Maximize} \quad & f(x) = \frac{1}{N} \sum_{n \in \mathcal{N}} p_n {c_n}^\top x_n \\ \text{subject to} \quad & T_n x_{p(n)} + A_n x_n = b_n, \quad x_n \in X_n,\ n \in \mathcal{N}, \end{aligned} \tag{5.1}$$

  where $p(n)$ denotes the predecessor of node n.
- Constraint matrices $A_n \in \mathbb{R}^{m_n \times n_n}$ and $T_n \in \mathbb{R}^{m_n \times n_{p(n)}}$ result from simple cash balance and book-keeping equations.

5.1.3 The Interior Point Method

One possible approach is a direct solution of problem (5.2) using a primal-dual interior point method (IPM; see Wright [90]). The efficiency of a symmetric (possibly semi- or quasi-) definite factorization of a large linear system determines the efficiency of the whole method.

A search direction has to be determined at each step of the IPM. Applying first order optimality conditions to the Lagrange multiplier formulation of problem (5.2) yields an $(n+m) \times (n+m)$ sparse symmetric (possibly semi- or quasidefinite) linear system of equations (*augmented system*) with the system matrix

$$K = \begin{pmatrix} D_1 & A^\top \\ A & D_2 \end{pmatrix}, \tag{5.2}$$

where $A \in \mathbb{R}^{m \times n}$ is the constraint matrix and $D_1 \in \mathbb{R}^{n \times n}$, $D_2 \in \mathbb{R}^{m \times m}$ are diagonal matrices (see Fig. 5.1).

5.1.4 Properties of the Augmented System Matrix

In the particular application, the augmented system matrix K has the following properties (see Table 5.1).

- Its dimension grows very rapidly with the number of periods in the planning horizon; realistic problems (10 and more time periods) lead to matrices of *huge* size (see Table 5.1).
- $n : m \approx 1.7$ for large matrices.
- Larger matrices are even sparser than smaller ones, because the number of nonzeros $\mathrm{NZ}(K)$ grows only *linearly* with the dimension.
- $D_2 = 0$.

5.1.5 Properties of the Normal System

By a simple elimination step the augmented system matrix K can be reduced to the $m \times m$ *normal system* matrix

$$N = D_2 - AD_1^{-1}A^\top. \tag{5.3}$$

The normal system can be solved instead of the augmented system. This has several advantages:

- The most obvious advantage compared to solving the augmented system directly is the reduced size of the matrix N. As mentioned earlier, the dimension can be reduced by a factor of 2.8 for the matrices occurring in the financial optimization problem considered in this chapter.
- In general, the matrix N is potentially (much) denser than K. If the constraint matrix A contains one single dense column, then N is even a dense matrix!

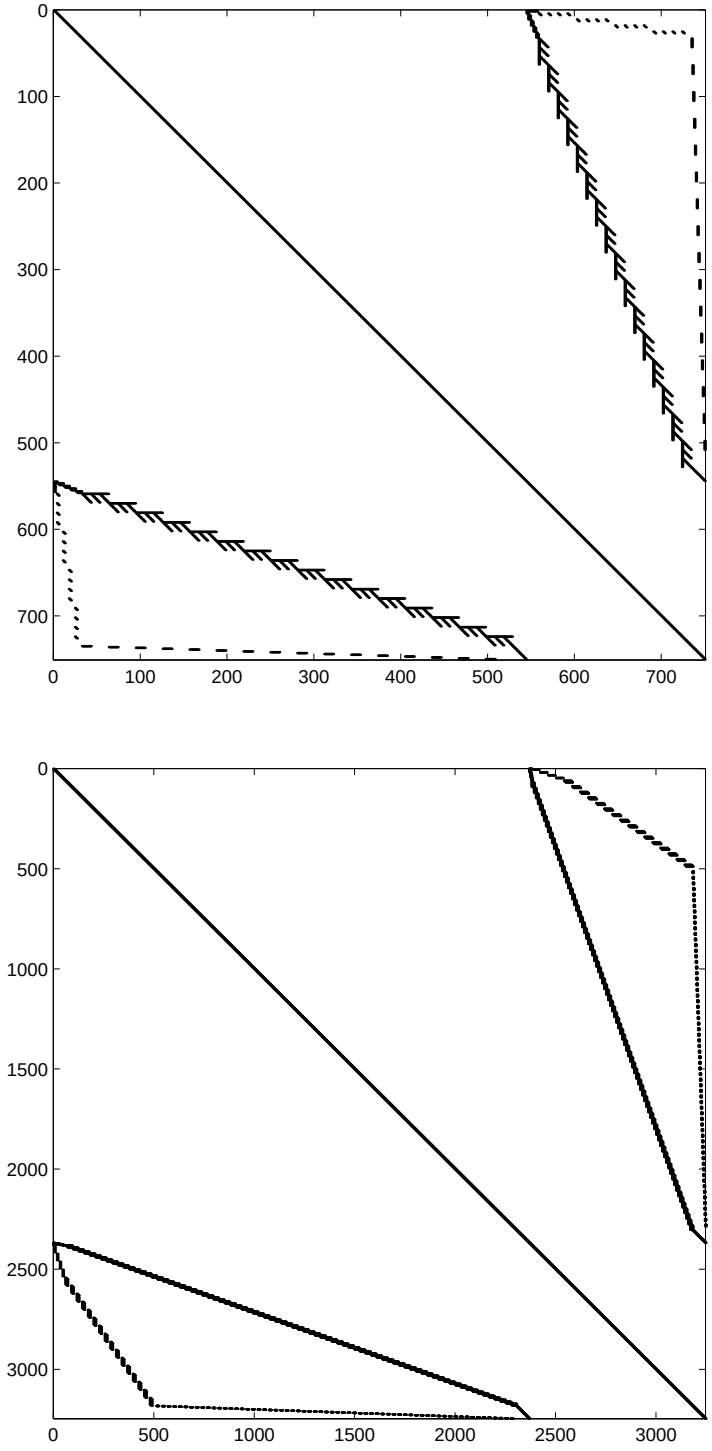

Figure 5.1: Sparsity structure of the augmented system matrix K in 2-period and 3-period models.

Periods	n	m	$n+m$	Sparsity
1	36	18	54	4.081 %
2	544	206	750	0.325 %
3	2,384	894	3,278	0.078 %
4	3,300	1,418	4,718	0.049 %
5	7,024	3,574	10,598	0.021 %
6	17,760	9,702	27,462	0.008 %
7	61,536	35,046	96,582	0.002 %
8	$3.11 \cdot 10^5$	$1.79 \cdot 10^5$	$4.91 \cdot 10^5$	$4.3 \cdot 10^{-4}$ %
9	$1.06 \cdot 10^6$	$6.11 \cdot 10^5$	$1.67 \cdot 10^6$	$1.3 \cdot 10^{-4}$ %
10	$5.15 \cdot 10^6$	$2.97 \cdot 10^6$	$8.13 \cdot 10^6$	$2.6 \cdot 10^{-5}$ %
11	$1.88 \cdot 10^7$	$1.09 \cdot 10^7$	$2.96 \cdot 10^7$	$8.6 \cdot 10^{-6}$ %
12	$6.89 \cdot 10^7$	$3.98 \cdot 10^7$	$1.06 \cdot 10^8$	$2.3 \cdot 10^{-6}$ %
13	$2.53 \cdot 10^8$	$1.46 \cdot 10^8$	$3.99 \cdot 10^8$	$6.4 \cdot 10^{-7}$ %
14	$9.28 \cdot 10^8$	$5.36 \cdot 10^8$	$1.46 \cdot 10^9$	$1.7 \cdot 10^{-7}$ %
15	$3.40 \cdot 10^9$	$1.97 \cdot 10^9$	$5.37 \cdot 10^9$	$4.7 \cdot 10^{-8}$ %
16	$1.25 \cdot 10^{10}$	$7.22 \cdot 10^9$	$1.97 \cdot 10^{10}$	$1.3 \cdot 10^{-8}$ %
17	$4.58 \cdot 10^{10}$	$2.65 \cdot 10^{10}$	$7.23 \cdot 10^{10}$	$3.5 \cdot 10^{-9}$ %
18	$1.68 \cdot 10^{11}$	$9.72 \cdot 10^{10}$	$2.65 \cdot 10^{11}$	$9.6 \cdot 10^{-10}$ %
19	$6.17 \cdot 10^{11}$	$3.57 \cdot 10^{11}$	$9.74 \cdot 10^{11}$	$2.6 \cdot 10^{-10}$ %
20	$2.26 \cdot 10^{12}$	$1.31 \cdot 10^{12}$	$3.57 \cdot 10^{12}$	$7.1 \cdot 10^{-11}$ %

Table 5.1: Augmented systems K from a financial decision problem. The column "Periods" refers to the number of time periods of the planning horizon, and the entries in the column "Sparsity" are given by the ratio $\mathrm{NZ}(K)/(n+m)^2$.

- In the finacial application discussed here the fill-in caused by the reduction of the augmented system is not very high, though (see Fig. 5.2). On average, the matrix N contains about *seven* nonzeros per row.
- Since $D_2 = 0$ in the particular application, (5.3) simplifies to $N = AD^{-1}A^\top$ (with $D := -D_1$).

5.2 Numerical Experiments

The following section discusses the performance of several codes when applied to the linear system occurring in the AURORA management system.

5.2.1 Performance of Ng-Peyton Code

The IPM solver currently included in the AURORA management system is a prototype which uses an individually adopted variant of the supernodal code of Ng and Peyton [72]. To reduce the order of the matrices arising from the IPM problems the solution process has been based on the normal system at the cost of forming the $ADA^\top$ product at each iteration step.

The Ng-Peyton code assumes that the matrix is positive definite, so that no pivoting is needed to ensure stability. However, the matrices arising from the IPM problems in the AURORA planning system are not guaranteed to be positive definite. As shown in Dockner et al. [25], every possible choice of pivot elements results in a stable algorithm. Therefore, pivoting for stability can be neglected, but some minor changes must be made in the algorithm.

Table 5.2 shows the total run times spent in Ng-Peyton routines during the solution of the IPM problem. For a given IPM problem the symbolic factorization is done only once. There is one $ADA^\top$ product, one numerical factorization and two triangular solutions at each iteration step of the IPM problem. The average run time of a single invocation of the Ng-Peyton routines is given in Table 5.3.

The forming of the $ADA^\top$ product is not part of the Ng-Peyton code, but it is needed to update the normal system at each iteration step of the IPM problem. Each IPM solver based on the normal systems will need this matrix update, whereas an IPM solver based on the augmented system will not need it.

All matrices arising from the solution of a particular IPM problem have the same sparsity structure; only the numerical values change at each iteration step. To reduce the total run time of the IPM solver, it is crucial that the linear solver being used offers a fast numerical factorization and a fast solution routine for triangular systems. The run time of the fill-in reducing ordering and of the symbolic factorization is less important for the total run time of the IPM solver since these operation are done only once.

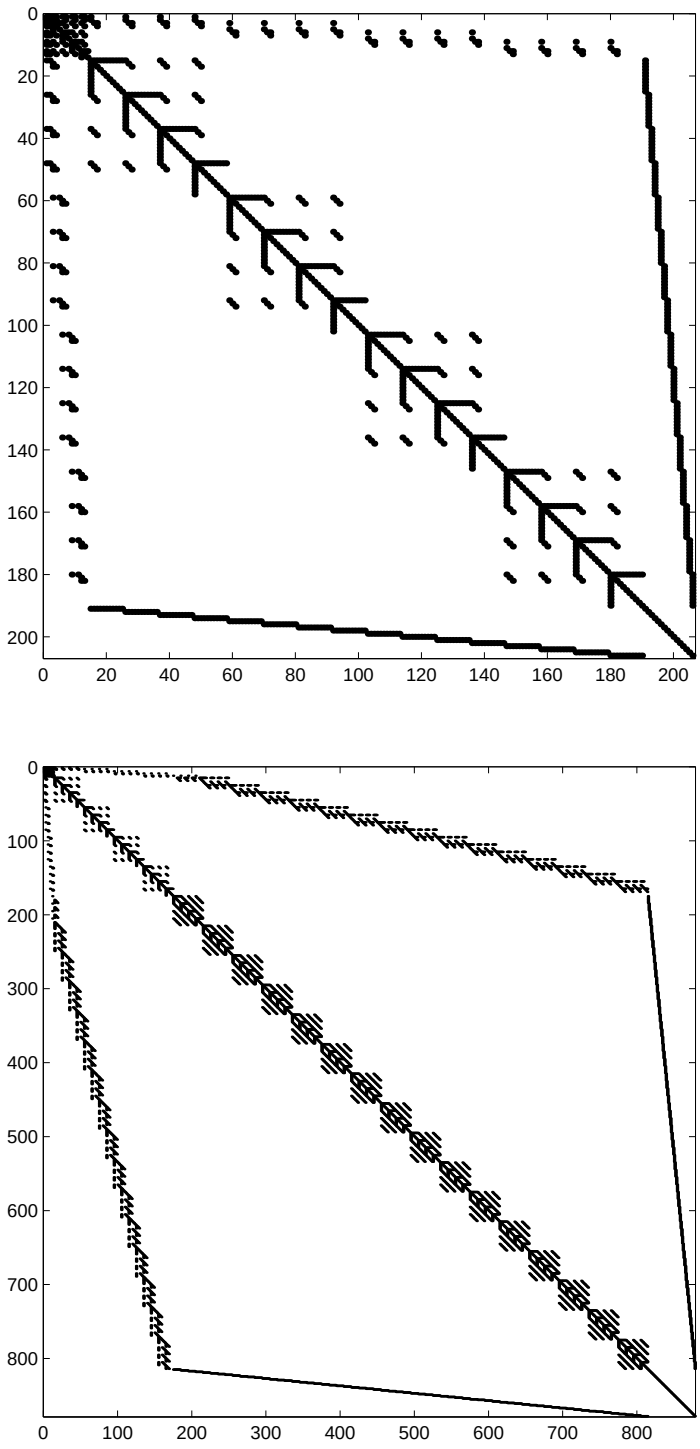

Figure 5.2: Normal system N in 2-period and 3-period models.

For the numerical experiments presented in the next sections the matrices A and D have been extracted from the IPM solver. The matrices Kxx are the system matrices of the augmented systems, whereas the matrices Nxx correspond to the normal systems. The field "xx" denotes the number of time periods.

Periods	Iteration Steps	Symbolic Factor.	Forming $ADA^\top$	Numerical Factor.	Triangular Solution
01	13	0.001	0.001	0.002	0.001
02	19	0.005	0.026	0.028	0.015
03	20	0.042	0.146	0.244	0.098
04	24	0.046	0.219	0.329	0.151
05	27	0.105	0.600	0.841	0.432
06	31	0.271	1.893	2.501	1.326
07	38	1.012	8.330	11.197	6.262
08	45	5.490	48.435	68.110	39.864
09	48	19.402	178.912	251.305	157.478
10	54	115.326	1254.913	1705.302	1296.048

Table 5.2: Total run times (in seconds) spent in Ng-Peyton routines during the solution of the IPM problem on an SGI Cray Origin 2000.

Periods	Symbolic Factor.	Forming $ADA^\top$	Numerical Factor.	Triangular Solution
01	0.001	< 0.001	< 0.001	< 0.001
02	0.005	0.001	0.002	< 0.001
03	0.042	0.007	0.012	0.003
04	0.046	0.009	0.014	0.003
05	0.105	0.022	0.031	0.008
06	0.271	0.061	0.081	0.022
07	1.012	0.219	0.295	0.085
08	5.490	1.076	1.514	0.453
09	19.402	3.727	5.236	1.675
10	115.326	23.239	31.580	12.227

Table 5.3: Average run time (in seconds) of a single invocation of the Ng-Peyton routines during the solution of the IPM problem on an SGI Cray Origin 2000.

5.2.2 Performance of Harwell Codes

As the current prototype of the AURORA system is based on the solution of the normal system exhaustive experiments have been made using this approach.

Alternatively the augmented system can be solved and it is shown that both approaches are comparable concerning run time and memory requirements.

Harwell MA48

The (set of routines) MA48 (see page 43) is a general solver designed for unsymmetric matrices. Therefore, the algorithm cannot fully exploit the symmetry of IPM matrices. Nevertheless, the performance of MA48 is quite astonishing.

Matrix	Symbolic	Factorization		Solution	
	Factor.	normal	fast	no refine	with refine
K05	0.11	0.04	0.01	0.01	0.06
K06	0.31	0.13	0.06	0.05	0.24
K07	1.16	0.46	0.20	0.17	0.88
K08	6.63	2.48	1.19	1.06	5.74
K09	29.57	11.43	6.91	6.88	29.75
K10	172.42	62.10	36.31	37.75	216.41
N05	0.06	0.02	0.01	< 0.01	0.02
N06	0.17	0.07	0.02	0.02	0.06
N07	0.71	0.31	0.17	0.13	0.66
N08	3.40	1.28	0.65	0.49	2.33
N09	17.48	6.58	4.20	3.35	18.07
N10	88.20	31.89	21.45	16.19	82.67

Table 5.4: Run times (in seconds) of MA48 (using *default* parameters) on an SGI Cray Origin 2000.

Table 5.4 shows the run times of MA48 routines with default parameter setting. The run times of MA48 routines using a parameter setting optimized for sparsity is given in Table 5.5. Several parameter settings have been tested, and it turned out that the optimal parameters greatly depend on the matrix being factorized (see, for example, matrices N08 and N10 in Tables 5.4 and 5.5).

Harwell MA38

The (set of routines) MA38 (see page 41) is a combined unifrontal/multifrontal solver for unsymmetric matrices. As expected, the performance of MA38 is not comparable to the performance of the Ng-Peyton code (see Table 5.6). The longer run times are partly due to the fact that MA38 has to perform more than twice the floating-point operations than the Ng-Peyton code.

MA38 can do subsequent factorizations of matrices with the same sparsity structure much faster. As the sparsity structure of the IPM matrices does not change, because at each iteration only the numerical values are updated, this

Matrix	Symbolic	Factorization		Solution	
	Factor.	normal	fast	no refine	with refine
K05	0.11	0.04	0.01	0.01	0.05
K06	0.31	0.13	0.06	0.05	0.24
K07	1.20	0.46	0.19	0.17	1.01
K08	9.00	3.39	1.75	1.79	9.20
K09	31.23	11.52	6.65	7.12	35.32
K10	151.88	52.12	28.13	28.66	158.70
N05	0.08	0.03	0.02	0.01	0.02
N06	0.15	0.06	0.02	0.01	0.06
N07	0.68	0.29	0.16	0.12	0.68
N08	4.10	1.65	0.99	0.67	3.70
N09	14.52	5.45	3.12	2.61	14.07
N10	80.18	30.32	18.16	14.99	81.37

Table 5.5: Run times (in seconds) of MA48 (using *optimized* parameters) on an SGI Cray Origin 2000.

feature of MA38 greatly improves the total run time of the IPM solver. The MA38 package includes also a very fast solution routine for triangular matrices. Optionally, the backward error of the computed solution can be improved by performing one (or more) steps of iterative refinement. For the solution of the linear system arising from the financial IPM problems there was no need to perform iterative refinement; the respective run times in Table 5.6 are given only for completeness.

MA38 uses two large arrays (one of type integer, one of type real) to store the sparse matrix A, the LU factors, and all internal data structures. For large matrices these arrays, which are included in the memory requirements shown in Table 5.7, account for more than 95 % of the total memory requirements.

The amount of main memory used by MA38 is given in Table 5.7 in the column labeled "used" while the column labeled "required" gives the minimum amount of memory needed by MA38 to succeed (although many garbage collections might be required in this case).

Table 5.8 shows the fill-in ratio and the extremely poor floating-point performance of MA38. The fill-in ratio r is defined as

$$r = \frac{(\text{nnz}(L) + \text{nnz}(U)) - \text{nnz}(A)}{\text{nnz}(A)} \tag{5.4}$$

There is almost no fill-in for large matrices arising from the normal systems. Due to the extreme sparsity of the matrices the performance rates are very poor.

Matrix	Factorization		Solution	
	normal	fast	no refine	with refine
K05	0.66	0.11	0.02	0.06
K06	1.60	0.39	0.10	0.27
K07	7.55	1.46	0.37	1.04
K08	35.26	5.47	1.26	3.68
K09	140.61	28.31	6.20	20.23
K10	818.74	128.90	29.72	100.24
N05	0.23	0.06	0.01	0.02
N06	0.53	0.15	0.02	0.07
N07	2.04	0.63	0.06	0.26
N08	11.30	3.50	0.44	1.64
N09	39.13	12.09	1.65	6.46
N10	223.39	72.00	10.63	40.79

Table 5.6: Run times (in seconds) of MA38 on an SGI Cray Origin 2000.

Matrix	Memory [MB]	
	used	required
K05	4.4	3.0
K06	9.6	6.7
K07	35.0	23.9
K08	158.3	109.5
K09	583.9	400.1
K10	2779.2	1868.7
N05	1.6	1.2
N06	3.5	2.9
N07	12.5	10.0
N08	64.0	50.6
N09	213.5	170.4
N10	1048.2	831.4

Table 5.7: Memory requirements of MA38.

Matrix	Fill-in ratio	Performance [Mflop/s]	
		normal fact.	fast fact.
K05	2.98	2.1	10.5
K06	1.81	1.2	4.1
K07	1.98	1.3	6.1
K08	1.18	0.7	3.2
K09	1.74	1.1	4.9
K10	1.42	0.6	2.7
N05	1.03	3.3	9.5
N06	0.54	2.2	7.2
N07	0.25	1.3	3.2
N08	0.17	1.1	3.0
N09	0.19	1.0	2.8
N10	0.17	0.9	2.5

Table 5.8: Fill-in ratio (as defined in (5.4)) and floating-point performance of MA38 on an SGI Cray Origin2000 (peak performance: 500 Mflop/s).

Harwell MA41

MA41 uses a parallel direct method based on a sparse multifrontal variant of Gaussian elimination (see page 40). An initial ordering for the pivotal sequence is chosen using the pattern of the matrix $A + A^\top$. Thus this code performs best on matrices whose pattern is symmetric, or nearly so. In general, MA47 is the better choice for symmetric matrices (compare Tables 5.9 and 5.10).

Harwell MA47

The (set of routines) MA47 (see page 43) is designed for the solution of sparse symmetric indefinite system of linear equations. MA47 is based on multifrontal sparse Gaussian elimination.

By default, MA47 uses $u = 0.001$ for threshold pivoting. Since this parameter is set on the assumption that the matrix is well-scaled, it is advisable to scale the matrix (for example, using MC30) before calling `MA47BD`.

The run times shown in Table 5.10 (obtained using the default parameter setting) are comparable with those of the Ng-Peyton code.

Like other Harwell routines, MA47 offers several parameters to optimize the performance of the code to a specific problem. Different settings for the following three parameters were used in the experiments.

`ICNTL(5)` has default value 5. It is used by `MA47AD` and `MA47BD` as the block size for the use of Level 3 Blas. Because symmetry is maintained during the

Matrix	Symbolic factorization	Numerical factorization	Triangular solution
K05	0.06	0.07	0.03
K06	0.18	0.19	0.13
K07	0.61	0.71	0.51
K08	5.76	4.40	4.78
K09	42.03	15.55	16.18
K10	4392.42	81.79	81.95
N05	0.03	0.03	0.01
N06	0.09	0.08	0.04
N07	0.59	0.33	0.28
N08	12.07	1.93	2.00
N09	137.94	8.27	10.39
N10	3652.70	32.19	29.82

Table 5.9: Run times (in seconds) of MA41 on an SGI Cray Origin2000.

Matrix	Symbolic Fact.	Numerical Fact.	Triangular Solution	Fill-in ratio	Memory [MB]
K05	0.1	0.1	0.02	1.4	0.4
K06	0.3	0.2	0.06	1.4	1.1
K07	1.2	1.0	0.27	1.5	4.0
K08	6.0	4.5	1.21	1.5	20.3
K09	29.0	17.9	4.92	0.4	97.4
K10	134.1	80.0	24.93	1.5	335.1
N05	0.1	0.0	0.01	0.7	0.2
N06	0.1	0.1	0.02	0.7	0.6
N07	0.4	0.4	0.08	0.7	2.3
N08	2.3	2.2	0.49	0.7	11.7
N09	9.1	7.3	1.59	0.7	39.1
N10	46.9	38.0	8.34	0.7	192.0

Table 5.10: Run times (in seconds), fill-in rations, and memory requirements of MA47 on an SGI Cray Origin 2000.

factorization, using Level 3 BLAS kernels may result in redundant floating-point operations.

`ICNTL(6)` has default value 5. `MA47AD` may amalgamate tree nodes in order to decrease the amount of indirect addressing. However, this may increase the number of floating-point operations. Two nodes are merged only if both involve less than `ICNTL(6)` eliminations.

`ICNTL(7)` has default value 4. Direct addressing and Level 2 BLAS are used by `MA47CD` on any block of the factorization having more than `ICNTL(7)` rows.

Table 5.11 shows the results obtained with various parameter settings. The results may be summarized as follows.

- The impact of the blocksize of Level 3 BLAS is small.
- Using Level 3 BLAS can result in more floating-point operations, but for this specific problem only a few (if any) redundant floating-point operations appeared (see column labeled "Mflop (red.)" in Table 5.11).
- The amalgamation factor `ICNTL(6)` has a significant influence on the fill-in, and thus on the number of floating-point operations needed in the factorization (see column labeled "Mflop (req.)" in Table 5.11).
- Smaller values for `ICNTL(6)` result (i) in larger elimination trees (the number of nodes in the elimination tree is given in Table 5.11), (ii) in smaller frontal matrices (the maximum frontsize is given in Table 5.11), (iii) in less fill-in, and (iv) in fewer floating-point operations in the factorization.
- A smaller number of floating-point operations does not automatically yield better run times.
- As described in the documentation, `ICNTL(7)` is only used in the solution process. Therefore, the run times for the symbolic and numerical factorization should not be affected by this parameter. However, Table 5.11 shows significant variations in the factorization run times. For each parameter setting at least five runs were made and the minimum run times are listed in the table.

The default parameter setting used for MA47 gives both, short run times and low memory requirements. Nevertheless, significant improvements are possible if the appropriate parameters are tuned to meet the characteristics of a given problem.

ICNTL(.)			Sym.	Num.	Triang.	Fill-in	Mflop	Mflop	Front-	Nodes	Mem.
5	6	7	Fact.	Fact.	Sol.	ratio	(req.)	(red.)	size	($\times 10^3$)	[MB]
1	5	4	9.5	8.6	1.56	0.7	30.6	0.0	12	98	39.1
2	5	4	9.9	7.7	1.66	0.7	30.6	0.8	12	98	39.1
3	5	4	9.9	7.7	1.65	0.7	30.6	1.3	12	98	39.1
4	5	4	9.8	7.7	1.71	0.7	30.6	2.5	12	98	39.1
5	5	4	8.3	7.2	1.50	0.7	30.6	0.0	12	98	39.1
6	5	4	9.8	7.5	1.64	0.7	30.6	0.0	12	98	39.1
7	5	4	9.4	7.4	1.57	0.7	30.6	0.0	12	98	39.1
8	5	4	8.2	7.1	1.41	0.7	30.6	0.0	12	98	39.1
9	5	4	9.6	7.5	1.66	0.7	30.6	0.0	12	98	39.1
10	5	4	8.3	7.2	1.56	0.7	30.6	0.0	12	98	39.1
12	5	4	8.8	7.3	1.50	0.7	30.6	0.0	12	98	39.1
16	5	4	9.8	7.4	1.63	0.7	30.6	0.0	12	98	39.1
5	1	4	10.0	10.1	0.81	0.1	13.5	0.0	6	450	39.1
5	2	4	9.4	8.9	0.70	0.4	19.6	0.0	7	257	39.1
5	3	4	9.7	8.2	0.84	0.6	25.4	0.0	9	170	39.1
5	4	4	9.3	8.1	0.86	0.6	27.1	0.0	11	153	39.1
5	5	4	8.3	7.2	1.50	0.7	30.6	0.0	12	98	39.1
5	6	4	9.0	7.5	1.62	0.7	31.1	0.0	15	98	39.1
5	7	4	9.0	9.6	1.58	1.4	61.7	0.0	17	72	47.5
5	8	4	9.0	10.1	1.51	1.6	71.3	0.0	19	65	50.6
5	9	4	10.0	9.3	1.14	1.3	62.7	0.0	20	52	45.9
5	10	4	8.8	9.7	1.41	1.5	69.5	0.0	23	49	47.9
5	12	4	8.8	9.9	1.38	1.5	74.5	0.0	27	47	49.4
5	16	4	9.1	17.3	1.56	3.1	201.9	0.0	35	27	80.7
5	5	1	8.9	7.4	1.62	0.7	30.6	0.0	12	98	39.1
5	5	2	8.5	7.2	1.53	0.7	30.6	0.0	12	98	39.1
5	5	3	9.2	7.6	1.64	0.7	30.6	0.0	12	98	39.1
5	5	4	8.3	7.2	1.50	0.7	30.6	0.0	12	98	39.1
5	5	5	8.9	7.5	1.56	0.7	30.6	0.0	12	98	39.1
5	5	6	8.8	7.4	0.98	0.7	30.6	0.0	12	98	39.1
5	5	7	8.6	7.3	0.91	0.7	30.6	0.0	12	98	39.1
5	5	8	8.2	7.2	0.74	0.7	30.6	0.0	12	98	39.1
5	5	9	7.7	6.9	0.62	0.7	30.6	0.0	12	98	39.1
5	5	10	8.3	7.2	0.75	0.7	30.6	0.0	12	98	39.1
5	5	12	8.7	7.5	0.66	0.7	30.6	0.0	12	98	39.1
5	5	16	8.7	7.5	0.87	0.7	30.6	0.0	12	98	39.1

Table 5.11: Run times (in seconds) and several characteristics of MA47 on an SGI Cray Origin 2000.

5.2.3 Performance of SuperLU

SuperLU is a direct solver based on the supernodal approach (see page 33). It has been designed especially for highly unsymmetric matrices.

Run times and performance rates of SuperLU are given in Table 5.12. The factorization routine is faster than the corresponding MA38 routine, but MA38 can do subsequent factorizations of matrices with the same sparsity structure much faster. Additionally, the fill-in reducing ordering is part of the factorization routine of MA38, whereas SuperLU strictly separates the ordering and the factorization process. Liu's MMD ordering has been incorporated into SuperLU. In the experiments, the matrices have been reordered prior to factorization using an MMD ordering based on the sparsity structure of $A^\top A$.

Matrix	Factorization time [s]	Factorization performance [Mflop/s]	Solution time [s]	Solution performance [Mflop/s]
K05	0.29	17.7	0.03	14.7
K06	0.55	10.3	0.07	10.4
K07	1.80	4.5	0.34	5.7
K08	7.77	2.9	1.45	6.1
K09	29.94	2.2	5.22	5.6
K10	215.40	1.3	34.93	4.0
N05	0.08	7.6	0.01	12.0
N06	0.17	6.3	0.02	13.2
N07	0.73	5.3	0.12	8.1
N08	3.78	5.0	0.69	7.1
N09	12.54	4.5	2.34	6.7
N10	78.86	3.8	14.16	5.5

Table 5.12: Run times (in seconds) and performance rates of SuperLU on an SGI Cray Origin2000 (peak performance: 500 Mflop/s).

Memory requirements and fill-in ratios of SuperLU are given in Table 5.13. For large matrices arising from the normal systems SuperLU produces much more fill-in than MA38 or MA48.

SuperLU_MT is a multi-threaded parallel package based on the serial algorithms in SuperLU (see page 33). Some experiments have been made with SuperLU_MT, but almost no speed-up could be observed. In some tests the parallel version was even slower than the serial code. Taking into account that already the serial code shows a very poor performance, it is not to be expected that a parallel direct solver designed for very unsymmetric matrices will show a satisfactory performance behavior for these particular IPM matrices.

Matrix	Fill-in ratio	Memory [MB]
K05	4.54	4.65
K06	2.60	9.82
K07	1.74	31.11
K08	1.46	152.66
K09	1.39	515.60
K10	1.38	2500.39
N05	1.28	1.41
N06	0.94	3.46
N07	0.94	12.54
N08	0.92	63.88
N09	0.84	212.06
N10	0.87	1041.91

Table 5.13: Fill-in ratio and memory requirements of SuperLU on an SGI Cray Origin2000.

5.2.4 Performance of SPOOLES

Spooles uses an object oriented approach to provide scalability on a diverse set of parallel computers (see page 34). As Spooles is written in C, routines from Spooles can be easily incorporated into applications written in Fortran. Several experiments were made with the serial version of Spooles, but most of them did not succeed due to internal errors. The few performance results which could be obtained were rather disappointing; neither run times nor memory requirements were comparable with the other packages discussed in this section.
The same is true for the multi-threaded version of Spooles and for the MPI based version of Spooles. Due to a restricted time frame no further investigations were made.

5.2.5 Performance of the S^+ Package

S^+ is an MPI based direct solver for unsymmetric systems (see page 39). Table 5.14 shows run times and memory requirements for the S^+ package.

S^+ is based on static symbolic factorization. The basic idea is to statically consider all possible pivoting choices at each elimination step and space is allocated for all possible nonzero entries. For the matrices arising from financial IPM problems, static factorization generates an excessive amount of fill-in which in turn costs a large amount of memory and run time for LU factorization. The factorization of the matrices K09 and K10 did not succeed due to excessive memory requirements.

Matrix	Ordering (MMD)	Solution (total)	Memory [MB]
K05	0.1	2.7	10.6
K06	0.3	9.2	25.6
K07	1.1	35.7	87.9
K08	6.2	2835.5	1561.4
K09	–	–	> 20000.0
K10	–	–	> 20000.0
N05	0.1	0.5	2.1
N06	0.3	1.0	5.0
N07	0.7	3.3	16.8
N08	4.4	23.9	84.8
N09	14.7	75.8	288.4
N10	84.8	436.5	1404.9

Table 5.14: Run times (in seconds) and memory requirements of S^+ on an SGI Cray Origin2000.

Some tests were made using more than one processor, but similarly to SuperLU_MT no speed-up could be observed. Instead, run times and memory requirements were growing with the number of processors being used.

5.2.6 Performance of PSPASES

Pspases is an MPI based direct solver for symmetric positive definite linear systems (see page 29).

Timing results have been obtained for the best numerical factorization performance. One of the parameters to be chosen is the blocksize b. Values of $b = 32$ or 64 turned out to be optimal for the numerical factorization. Results obtained with matrix N10 are given in Table 5.15 and Figure 5.3.

Another choice to be made concerns the ordering: serial ordering produces less fill-in than parallel ordering.

Most of today's parallel computers have communication subsystems whose performance is better for larger data sets being communicated. The optimal blocksize for achieving the highest factorization performance is usually smaller than that for the triangular solution (depending on the number of right hand sides). During factorization b^2 floating-point numbers have to be communicated, whereas during triangular solution only br numbers are communicated, where b is the blocksize and r is the number of right hand sides.

The Pspases routines require the number p of processing elements to be a power of 2 (and greater 1); so no run times for a serial version of Pspases are available. Accordingly, speed-up values given in Table 5.15 relate to the total

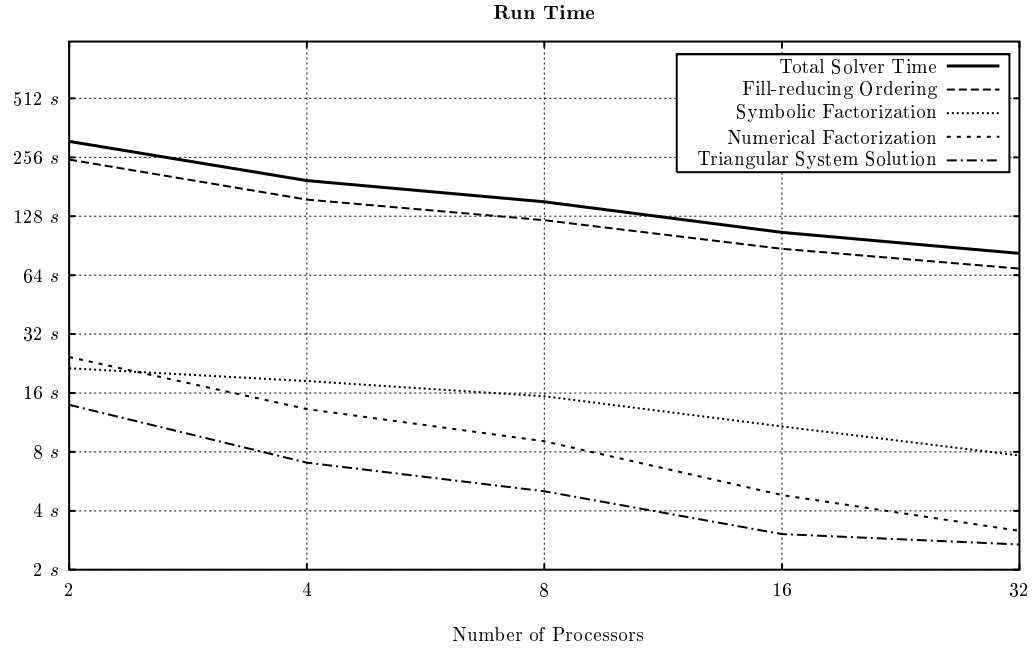

Figure 5.3: Run times of PSPASES routines on an SGI Cray Origin 2000.

	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
Fill-in reducing Ordering	249.4 s	155.8 s	122.3 s	87.3 s	69.1 s
Symbolic Factorization	21.4 s	18.3 s	15.3 s	10.7 s	7.6 s
Numerical Factorization	24.3 s	13.2 s	9.0 s	4.8 s	3.1 s
Triangular Systems Solution	13.9 s	7.0 s	5.0 s	3.0 s	2.6 s
Total Solver Time	309.2 s	194.5 s	151.8 s	105.9 s	82.6 s
Speed-up	**1.00**	1.58	2.04	2.91	3.74

Table 5.15: Run times of PSPASES on p processors of an SGI Cray Origin 2000.

solution time obtained on $p = 2$ processors.

PSPASES must run on a large number of processors to achieve run times comparable with those of the Ng-Peyton code. Moreover, the memory requirements of PSPASES are prohibitive, especially if very large systems are to be solved.

Conclusion

Often the solution of large sparse linear systems is the most time-consuming part in an application. However, the effort needed to employ sparse codes may be considerable. In particular organizing and managing the application data in preparation for and after the solution step may be costly. A major problem is that sparse matrices can be represented in many different ways. Most of them are problem specific and it may not be a trivial task to organize the data for a call to the linear equation solver. This task may be even more complicated if parallelism is to be exploited.

In this report many algorithms have been presented for the direct solution of large sparse linear systems and an overview of relevant software packages has been given. The performance results show that only some of them can be used as black-box routines. To achieve good floating-point performance the characteristics of the problems being solved have to be considered and an appropriate software package has to be selected.

As an example the AURORA planing system has been chosen. Difficulties occurring in the solution of the very large sparse linear systems arising from the optimization problem in this particular application have been pointed out. Several state-of-the-art software packages will be released in the near future and it will be interesting to see if any of them will improve the solution process of these particular sparse linear systems.

Bibliography

[1] AEA Technology: *Harwell Subroutine Library: Specifications (Release 12)*. AEA Technology, Harwell, 1995

[2] P. R. Amestoy: *Factorization of large sparse matrices based on a multifrontal approach in a multiprocessor environment*. INPT PhD Thesis TH/PA/91/2, CERFACS, Toulouse, France, 1991.

[3] P. R. Amestoy, T. A. Davis, I. S. Duff: *An approximate minimum degree ordering algorithm*. Technical Report TR/PA/95/09, CERFACS, Toulouse, 1995.

[4] P. R. Amestoy, T. A. Davis, I. S. Duff: *An approximate minimum degree ordering algorithm*. Report TR-94-039, University of Florida, 1994.

[5] P. R. Amestoy, I. S. Duff: *vectorization of a multiprocessor multifrontal code*. Int. J. of Supercomputer Applics. 3 (1989), pp. 41–59.

[6] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. J. Dongarra, J. Du Croz, S. Hammarling, A. McKenney, S. Ostrouchov, D. C. Sorensen: LAPACK *User's Guide*, 2nd ed. SIAM Press, Philadelphia 1995.

[7] C. Ashcraft, S. Eisenstat, J. W. H. Liu: *A fan-in algorithm for distributed sparse numerical factorization*. SIAM J. Scientific and Statistical Computing 11 (1990), pp. 593–599.

[8] C. Ashcraft, R. G. Grimes: SPOOLES*: An Object-Oriented Sparse Matrix Library*. to appear in Proceedings of the 1999 SIAM Conference on Parallel Processing for Scientific Computing

[9] C. Ashcraft, R. G. Grimes, J. G. Lewis, B. Peyton, H. Simon: *Progress in sparse matrix methods for large sparse linear systems on vector supercomputers*. Intern. J. of Supercomputer Applications 1 (1987), pp. 10–30.

[10] C. Ashcraft, R. G. Grimes, J. G. Lewis: *Accurate symmetric indefinite linear equation solvers*. Technical Report ISSTECH-95-029, Boeing Computer Services, 1995.

[11] C. Ashcraft, J. W. H. Liu: *Robust ordering of sparse matrices using multisection*. Technical Report ISSTECH-96-002, Boeing Information and Support Services, Seattle, 1996.

[12] Z. Bai, D. Day, J. Demmel, J. J. Dongarra: *Test matrix collection (non-Hermitian eigenvalue Problems), Release 1*. Technical Report, University of Kentucky, 1996.

[13] S. Benkner, E. Laure, H. Zima: HPF+: *An Extension of HPF for Advanced Industrial Applications*. Technical Report TR 99-1, University of Vienna, 1999.

[14] R. Boisvert, R. Pozo, K. Remington, R. Barrett, J. Dongarra: MATRIX MARKET : *a web resource for test matrix collections*, in R. Boisvert, Ed. *The Quality of Numerical Software: Assessment and Enhancement*. Chapman and Hall, London, 1997, pp. 125–137

[15] R. Boppana: *Eigenvalues and graph bisection: An average case analysis*. Proc. 28th Annual Symposium on Foundations of Computer Science, IEEE, 1987, pp. 280–285.

[16] L. S. Blackford, J. Choi, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley: SCALAPACK *User's Guide*. SIAM, Philadelphia, 1997.

[17] D. A. Calahan: *Parallel solution of sparse simultaneous linear equations*. Proceedings 11th Annual Allerton Conference on Circuits and System Theory, University of Illinois, pp. 445–452.

[18] E. Cuthill, J. McKee: *Reducing the bandwidth of sparse symmetric matrices*. Proceeding 24th National Conference of the Association for Computing Machinery, New Jersey, 1969, Brandon Press, pp. 157–172

[19] T. A. Davis: *University of Florida sparse matrix collection*. Available at the web site `www.cise.ufl.edu/~davis`, 1997.

[20] T. A. Davis, I. S. Duff: *An unsymmetric-pattern multifrontal method for sparse LU factorization*. Technical Report RAL 93-036, Rutherford Appleton Laboratory, 1993.

[21] T. A. Davis, I. S. Duff: *An Unsymmetric-Pattern Multifrontal Method for Sparse LU Factorization*. Technical Report TR-94-038, Computer and Information Science Dept., University of Florida, 1994.

[22] T. A. Davis, I. S. Duff: *An unsymmetric-pattern multifrontal method for sparse LU factorization*. SIAM J. Matrix Anal. Applic. 18 (1997), pp. 140–158.

[23] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, J. W. H. Liu: *A supernodal approach to sparse partial pivoting*. Technical Report UCB//CSD-95-883, Berkeley, 1995.

[24] J. W. Demmel, J. R. Gilbert, X. S. Li: *An asynchronous parallel supernodal algorithm for sparse Gaussian elimination*. Technical Report UCB//CSD-97-943, Berkeley, 1997.

[25] E. Dockner, G. C. Pflug, A. Swietanowski: *The AURORA Financial Management System: From Model Design to Parallel Implementation*. Technical Report AURORA TR1998-08, University of Vienna, 1998.

[26] D. S. Dodson, R. G. Grimes, J. G. Lewis: *Sparse Extensions to the Fortran Basic Linear Algebra Subprograms*. ACM Trans. Math. Softw. 17 (1991), pp. 253–263, 264–272.

[27] J. J. Dongarra, J. Du Croz, I. S. Duff, S. Hammarling: *A Set of Level 3 Basic Linear Algebra Subprograms*. ACM Trans. Math. Softw. 16 (1990), pp. 1–17, 18–28.

[28] J. J. Dongarra, J. Du Croz, S. Hammarling, R. J. Hanson: *An Extended Set of Fortran Basic Linear Algebra Subprograms*. ACM Trans. Math. Softw. 14 (1988), pp. 1–17, 18–32.

[29] I. S. Duff: *Design features of a frontal code for solving sparse unsymmetric linear systems out-of-core*. SIAM J. Sci. Comput. 5 (1984), pp. 270–280.

[30] I. S. Duff: *The influence of vector and parallel computers in the solution of large sparse linear equations*, in M. J. D. Powell and A. Iserles, eds., *The State of the Art in Numerical Analysis*. Oxford University Press, Oxford, 1987, pp. 359–407.

[31] I. S. Duff, A. M. Erisman, J. K. Reid: *Direct Methods for Sparse Matrices*. Oxford University Press, 1986.

[32] I. S. Duff, N. I. M. Gould, J. K. Reid, J. A. Scott, K. Turner: *Factorization of sparse symmetric indefinite matrices*. IMA J. Numerical Analysis 11 (1991), pp. 181-204.

[33] I. S. Duff, R. G. Grimes, J. G. Lewis: *Sparse matrix test problems*. ACM Trans. Math. Softw., 15 (1989), pp. 1–14.

[34] I. S. Duff, R. G. Grimes, J. G. Lewis: *The Rutherford-Boeing Sparse Matrix Collection*. Technical Report RAL TR-RAL-97-031, Rutherford Appleton Laboratory, 1997.

[35] I. S. Duff, M. Marrone, G. Radicati, C. Vittioli: *A set of Level 3 Basic Linear Algebra Subprograms for Sparse Matrices*. Technical Report RAL TR-RAL-95-049, Rutherford Appleton Laboratory, 1995.

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

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

[38] I. S. Duff, J. K. Reid: *MA48, A Fortran Code for direct solution of sparse unsymmetric linear systems of equations*. Technical Report RAL RAL-93-072, Rutherford Appleton Laboratory, 1993.

[39] I. S. Duff, J. K. Reid: *MA47, A Fortran code for direct solution of indefinite sparse symmetric linear systems*. Technical Report RAL RAL-95-001, Rutherford Appleton Laboratory, 1995.

[40] I. S. Duff, J. A. Scott: *A comparison of frontal software with other sparse direct solvers*. Technical Report RAL TR-RAL-96-102, Rutherford Appleton Laboratory, 1996.

[41] M. Fiedler: *A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory*. Czechoslovak Math. J., 25(100), pp. 619–633, 1975.

[42] L. V. Foster: *The growth factor and efficiency of Gaussian elimination with rook pivoting*. J. Comp. and Appl. Math., to appear.

[43] A. George, J. W. H. Liu: *The Evolution of the Minimum Degree Ordering Algorithm*. SIAM Review 31 (1989), pp. 1–19.

[44] C. Fu, X. Jiao, T. Yang: *Efficient Sparse LU Factorization with Partial Pivoting on Distributed Memory Architectures*. IEEE Transactions on Parallel and Distributed Systems 9 (1998), pp. 109–125.

[45] A. Gupta: *Analysis and Design of Scalable Parallel Algorithms for Scientific Computing*. PhD thesis, University of Minnesota, 1995.

[46] A. Gupta: *Graph partitioning based sparse matrix ordering algorithms for interior-point methods*. Technical Report RC 20467 (90480), IBM T. J. Watson Research Center, Yorktown Heights, NY, 1996.

[47] A. Gupta: WGPP: *Watson graph partitioning (and sparse matrix ordering) package: Users manual*. Technical Report RC 20453 (90427), IBM T. J. Watson Research Center, Yorktown Heigths, NY, 1996.

[48] A. Gupta: *Fast and effective algorithms for graph partitioning and sparse matrix ordering*. IBM Journal of Research and Development 41 (1997), pp. 171–183.

[49] A. Gupta, M. Joshi, V. Kumar: WSSMP*: Watson Symmetric Sparse Matrix Package*. IBM Research Report RC 20923 (92669), IBM T. J. Watson Research Center, Yorktown Heigths, NY, 1997.

[50] A. Gupta, G. Karypis, V. Kumar: *Highly Scalable parallel Algorithms for Sparse Matrix Factotization*. Technical Report TR-94-63, Department of Computer Science, University of Minnesota, 1994.

[51] A. Gupta, G. Karypis, V. Kumar: *Highly scalable parallel algorithms for sparse matrix factorization*. IEEE Transactions on Parallel and Distributed Systems 8 (1997), pp. 502–520.

[52] A. Gupta, V. Kumar: *Optimally scalable parallel sparse cholesky factorization*. in *The 7th SIAM Conference on Parallel Processing for Scientific Computinig*, SIAM Press, 1995, pp 442–447.

[53] M. Heath: *Communication reduction in parallel sparse Cholesky on a hypercube*. SIAM Press, 1987.

[54] B. Hendrickson, R. Leland: *An improved spectral graph partitioning algorithm for mapping parallel computations*. Technical Report SAND92-1460, Sandia National Laboratories, 1992.

[55] B. Hendrickson, R. Leland: *A multilevel algorithm for partitioning graphs*. Technical Report SAND93-1301, Sandia National Laboratories, 1993.

[56] B. Hendrickson, R. Leland: *The* CHACO *User's Guide. Version 2.0*. Technical Report SAND94-2692, Sandia National Laboratories, 1993.

[57] B. Hendrickson, E. Rothberg: *Improving the runtime and quality of nested dissection ordering*. Technical Report SAND96-0868J, Sandia National Laboratories, 1996.

[58] B. M. Irons: *A frontal solution program for finite elemet analysis*. Int. J. Numer. Methods Eng. 2 (1970), pp. 5–32.

[59] M. Joshi, A. Gupta, G. Karypis, V. Kumar: *Two dimensional scalable algorithms for solution of triangular systems.* Technical Report TR-97-024, Department of Computer Science, University of Minnesota, 1997.

[60] G. Karypis, V. Kumar: *A high performance sparse Cholesky factorization algorithm for scalable parallel computers*. Technical Report TR-94-41, Department of Computer Science, University of Minnesota, 1994.

[61] G.Karypis, V. Kumar: *Parallel multilevel graph partitioning*. Technical Report TR-95-036, Department of Computer Science, University of Minnesota, 1995.

[62] G. Karypis, V. Kumar: *A fast and high quality multilevel scheme for partitioning irregular graphs*. Technical Report TR-95-035, Department of Computer Science, University of Minnesota, 1995.

[63] B. Kernighan, S. Lin: *An efficient heuristic procedure for partitioning graphs*. Bell Systems Technical Journal 29 (1970), pp. 291–307.

[64] H. Kofler, E. J. Haunschmid, W. N. Gansterer, C. W. Ueberhuber *The Locality Property in Topological Irregular Graph Hierarchies*. Technical Report AURORA TR1998-13, University of Vienna, 1998.

[65] X. S. Li: *Sparse Gaussian Elimination on High Performance Computers*. PhD thesis, University of California, Berkeley, 1996.

[66] X. S. Li, J. W. Demmel: *Making Sparse Gaussian Elimination Scalable by Static Pivoting*. to appear in proceedings of Supercomputing 98.

[67] J. W. H. Liu: *Modification of the minimum degree algorithm by multiple elimination*. SIAM Trans. Math. Softw. 11 (1985), pp. 141–153.

[68] J. W. H. Liu: *On the storage requirements in the out-of-core multifrontal method for sparse factorization*. ACM Trans. Math. Softw. 12 (1986), pp. 249–264.

[69] J. W. H. Liu: *The Multifrontal Method for Sparse Matrix Solution: Theory and Practice*. SIAM Review 34 (1992), pp. 82–109.

[70] H. M. Markowitz: *The elimination form of the inverse and its application to linear programming*. Management Science 3 (1957), pp. 255–269.

[71] E. G. Ng, B. W. Peyton: *Block sparse Cholesky algorithms on advanced uniprocessor computers*. SIAM J. Scientific and Statistical Computing 14 (1993), pp. 1034–1056.

[72] E. G. Ng, B. W. Peyton: *A supernodal Cholesky algorithm for shared-memory multiprocessors*. SIAM J. Scientific Computing 14 (1993), pp. 761–769.

[73] G. C. Pflug, A. Swietanowski: *Parallel Decision Support for Financial Management Under Uncertainty*. Technical Report AURORA TR1998-07, University of Vienna, 1998.

[74] G. C. Pflug, A. Swietanowski: *The* AURORA *Financial Management System Documentation*. Technical Report AURORA TR1998-09, University of Vienna, 1998.

[75] G. C. Pflug, A. Swietanowski: *Dynamic Asset Allocation Under Uncertainty for Pension Fund Management*. Technical Report AURORA TR1998-15, University of Vienna, 1998.

[76] A. Pothen, H. Simon, K. Liou: *Partitioning sparse matrices with eigenvectors of graphs*. SIAM J. Matrix Anal., 11 (1990), pp. 430–452.

[77] A. Pothen, C. Sun: *A mapping algorithm for parallel sparse Cholesky factorization*. SIAM J. Sci. Comput. 14 (1993), pp. 1253–1257.

[78] E. Rothberg: *Exploring the tradeoff between imbalance and separator size in nested dissection ordering*. Technical Report (unnumbered), Silicon Graphics Inc., 1996.

[79] E. Rothberg, A. Gupta: *Efficient sparse matrix factorization on high-performance workstations — exploiting the memory hierarchy*. ACM Trans. Mathematical Software 17 (1991), pp. 313–334.

[80] Y. Saad: SPARSKIT*: a basic toolkit for sparse matrix computations, Version 2*. Computer Science Dep., University of Minnesota, 1994.

[81] H. D. Simon: *Partitioning of unstructured problems for parallel processing*. Proc. Conference on Parallel Methods on Large Scale Structural Analyis and Physics Applications, Pergammon Press, 1991.

[82] H. Simon, P. Vu, C. Yang: *Performance of a supernodal general sparse solver on the CRAY Y-MP: 1.69 GFLOPS with autotasking*. Technical Report TR SCA-TR-117, Boeing Computer Services, 1989.

[83] B. Speelpenning: *The generalized element method*. Tech. Report UIUCDCS-R-78-956, University of Illinois, 1978.

[84] C. Sun: *Efficient parallel solutions of large sparse SPD systems on distributed-memory multiprocessors*. Technical Report CTC92TR102, Cornell University, 1992.

[85] A. Supalov: PARASOL *Interface Specification*. German National Research Center for Information Technology, Institute for Algorithms and Scientific Computing, 1998.

[86] The MathWorks Inc.: *Using* MATLAB. Version 5.1, The MathWorks Inc., 1997

[87] W.F. Tinney, J.W. Walker: *Direct Solutions of Sparse Network Equations by Optimally Ordered Triangular Factorization*. Proc. of the IEEE 55 (1967), pp. 1801-1809.

[88] C.W. Ueberhuber: *Numerical Computation*. Springer-Verlag, Heidelberg 1997.

[89] R.C. Whaley: *Basic Linear Algebra Communication Subprograms: analysis and implementation across multiple parallel architectures*. Technical Report TR-CS-94-234, Computer Science Department, University of Tennessee, Knoxville, Tennessee, 1995.

[90] S.J. Wright: *Primal-Dual Interior Point Methods*. SIAM Press, Philadelphia, 1997

[91] S.E. Zitney: *Sparse matrix methods for chemical process separation calculations on Supercomputers*. Proc. Supercomputing'92, Minneapolis, 1992, IEEE Computer Society Press, pp. 414–423.

[92] Z. Zlatev, J. Wasniewski, K. Schaumburg: Y12M—*solution of large and sparse systems of linear algebraic equations*. Springer-Verlag, Berlin, 1981.