High Performance Cholesky and Symmetric Indefinite Factorizations with Applications

Jonathan David Hogg

Doctor of Philosophy
University of Edinburgh
2010

Abstract

The process of factorizing a symmetric matrix using the Cholesky (LL^T) or indefinite (LDL^T) factorization of A allows the efficient solution of systems $A\boldsymbol{x} = \boldsymbol{b}$ when A is symmetric. This thesis describes the development of new serial and parallel techniques for this problem and demonstrates them in the setting of interior point methods.

In serial, the effects of various scalings are reported, and a fast and robust mixed precision sparse solver is developed. In parallel, DAG-driven dense and sparse factorizations are developed for the positive definite case. These achieve performance comparable with other world-leading implementations using a novel algorithm in the same family as those given by Buttari et al. for the dense problem. Performance of these techniques in the context of an interior point method is assessed.

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Jonathan David Hogg*)

Acknowledgements

I would like to thank my PhD supervisor Julian Hall for all his help and interest over the years. Among the other staff members at Edinburgh, I would like to single out my second supervisor Andreas Grothey for his support, and Jacek Gondzio who took on the role of supervising me for some projects.

My colleagues at RAL are also deserving of praise, with much of the work at the core of this thesis done during my year working with them. Specific thanks to Jennifer Scott and John Reid for correcting various technical reports and teaching me how to write better — both code and English.

My proofreader and girlfriend Gwen Fyfe, for reading and checking the English, even if most of the mathematics went over her head. Also for support and kisses.

Many improvements and corrections were also suggested by the Thesis examiners Jacek Gondzio and Patrick Amestoy.

Finally to my office mates and fellow PhD students down the years, for their support and conversation: Kristian Woodsend, Danny Hamilton, Hugh Griffiths, Ed Smith, and the rest.

I will breifly mention the research that didn't quite work out or make it into this thesis: deficient basis simplex methods, structured algebraic modelling languages, parallel generalised upper bound simplex codes and reduced interior point methods.

Contents

Notation

Notational conventions

Capital italic letters,	e.g. A, L, P	Matrices.
Bold lower case italic letters,	e.g. $\boldsymbol{b}, \boldsymbol{x}$	Vectors.
Lower case italic letters,	e.g. i, ϵ	Scalars.
Subscripted italic lower case letters,	e.g. x_i	i-th component of vector $\boldsymbol{x}$.
Double subscripted lower case italic letters,	e.g. a_{ij}	Entry in row i, column j of matrix A.
Double subscripted capital italic letters,	e.g. A_{ij}	Sub-block of matrix A in position (i, j).
Superscripted with brackets,	e.g. $\boldsymbol{x}^{(k)}, H^{(j)}$	Value of variable at superscripted iteration.
Calligraphic subscripted letter,	e.g. $\mathcal{A}_j$	Lower-triangular non-zero set of corresponding matrix e.g. A.
Subscripted with brackets,	e.g. $A_{(i:j)(k:l)}$	Submatrix of A containing rows i to j, and columns k to l.
Value with hat,	e.g. $\hat{\boldsymbol{x}}$	Exact analytic solution.

Specific variables and matrices

A	Matrix to be factorized ($n \times n$); alternatively the constraint matrix for a linear program ($m \times n$).
D	Diagonal matrix with either 1×1 or 2×2 blocks on the diagonal.
I	The identity matrix.
L	Lower triangular matrix; factor of $A = LL^T$ or $A = LDL^T$.
P	Permutation matrix.
S	Scaling matrix.
$\boldsymbol{b}$	Right hand side matrix in the equation $A\boldsymbol{x} = \boldsymbol{b}$.
$\boldsymbol{e}$	The vector of all ones.
$\boldsymbol{e}_i$	The ith column of the identity matrix.
$\boldsymbol{x}$	Unknown vector to be determined.
ϵ	Machine epsilon value under IEEE arithmetic.

Introduction

The solution of the system

$$A\boldsymbol{x} = \boldsymbol{b}$$

is one of the fundamental building blocks of numerical algorithms from optimization to structural modelling and statistics. The fast and accurate solution is critical for disciplines as wide ranging as engineering, chemistry, bioinformatics and social science. Such problems are normally categorised according to the properties of the matrix A. The principal split is into symmetric and unsymmetric problems; the symmetric case is simplest and algorithms that exploit this are generally faster than those that do not. Historically, many fast unsymmetric algorithms have evolved as generalisations of symmetric ones. Thus, in this thesis, we shall consider only the symmetric case.

The other common categorical split is into dense and sparse. A matrix A is considered to be sparse in a given context if a sparsity-exploiting algorithm gives better performance than a dense one. As processor speed has risen the most computationally challenging problems typically feature matrices that are both sparse and structured due to the limited way such large problems are generated. Exceptions of course exist, such as those encountered in various branches of data mining. This thesis will concentrate substantially on sparse matrices.

With the explosion of *multicore* processors in the desktop computers typically used by researchers, *all* algorithms must become parallel in order to fully exploit the available processing power. This has been referred to as "the end of the free lunch," where performance of numerical algorithms has stopped continuously improving as a result of ever increasing transistor densities in computer processors. The return of vector processing in the form of general purpose computing on graphics processing units (GPGPU) represents another fundamental change to underlying architectural assumptions that must be addressed. Future *manycore* systems are likely to exhibit features of both.

To support the new work in this thesis, the first three chapters (Part I) present existing work in the field and establish a solid theoretical foundation on which the new work set out in the remaining chapters (Part II) is based. Chapter 1 gives the context of modern computing, Chapter 2 the theoretical and practical basis to the practical solution of linear equations and Chapter 3 presents a basic theory of Interior Point Methods that represent a major use of linear solvers.

The contributions of this thesis presented in Part II are the development of several novel and practical methods to address the solution of $A\boldsymbol{x} = \boldsymbol{b}$ for large sparse symmetric matrices on modern architectures. The requirement to address numerical stability has the potential to seriously disrupt sparse matrix factorization, and Chapter 4 surveys scaling techniques that reduce the impact of badly scaled problems. On both multicore and GPU platforms significant differences between single and double precision computational speeds can be exploited to accelerate solution through the use of the practical mixed precision techniques developed in Chapter 5. To fully exploit current and future multicore and manycore architectures fine-grained and adaptive parallelism must be exploited. DAG-based dense and sparse Cholesky factorizations are described in Chapters 6 and 7 that expose and exploit such parallelism. Finally Chapter 8 takes the work of previous chapters and applies it in the context of interior point methods.

Part I

Background

Chapter 1

Coding for modern processors

Modern processors have many special hardware features to allow the fast, efficient and safe execution of standard operations. While most of these are primarily aimed at common desktop applications and games, many may also be exploited for high performance computing (HPC).

Modern compilers are written to generate code that uses these hardware features, but are limited by their heuristic understanding of an algorithm. For example a typical compiler will be able to generate vectorised versions of simple loops, however a more complicated (but equivalent) loop will not meet the compiler's requirements for vectorisation. Through the careful reformulation of our algorithms we can state useful information more clearly and explicitly to the compiler, avoiding problematic language features. The limiting case of such reformulation is hand-optimized code, though that is undesirable except in a small number of critical sections.

This chapter presents a quick tour of those features we need to pay explicit attention to in a linear algebra context, along with common techniques for dealing with them. The descriptions herein draw on many sources including my personal experience, but in particular the work of Biggar et al. [2005] and the Intel optimization guides [Intel Corporation 2006; 2007].

1.1 Programming language

Almost all results in this thesis are based on codes written using Fortran 95 [ISO 1997] with some Fortran 2003 [ISO 2004] features (particularly allocatable components of derived types). Common HPC languages are Fortran and C/C++, and as expertise was more readily available in the former it was chosen. Further, for work which became part of (or used) the HSL software library [HSL 2007] the use of Fortran was essential.

1.2 Overview

Current supercomputers and desktops are built around commodity multicore processors. For our purposes these can be viewed as a number of processing cores grouped into one or more processor packages. These processor packages also include a small amount of fast memory, known as cache, which is connected to one or more of the cores. Externally each processor package is connected to the main memory, in addition to peripherals such as disk or network resources. Clusters are a collection of such computing nodes connected via a network. Figure 1.1 shows a typical HPC node containing two quad core Intel processors.

There are three major constraints to how fast a program will run [Graham et al. 2004]:

Execution speed: How many floating point operations can the core complete per unit time?
Normally measured in floating point operations per second (FLOPS).

memory bandwidth: How fast can we supply data to the core to be operated on?
Normally measured in Megabytes per second (MB/s).

Memory latency: What is the delay between requesting data and receiving it?
Normally measured in either nanoseconds or cycles.

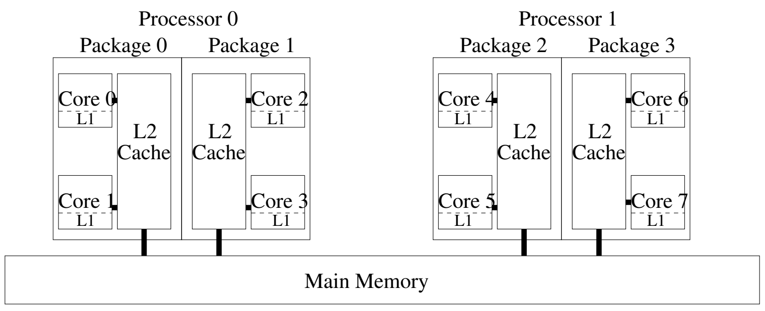


Figure 1.1: Typical HPC node

In an ideal world we would be limited only by execution speed. However most numerical codes are constrained by the memory properties.

1.3 Caches

In order to address both memory bandwidth and latency issues, a hierarchy of caches exists. A cache is a smaller and faster memory which holds a partial copy of main memory; if accessing a memory location which is already cached, then you will wait fewer cycles for the data than if it came from main memory.

Memory is divided into cache lines (typically 8 bytes), and the most recently accessed cache lines are kept in memory. When a memory request is made it is first requested from the cache; if it is not present then a *cache miss* occurs and the newly accessed cache line is retrieved from main memory, replacing the least recently referenced line in the cache. Specific cache lines from main memory are normally only stored in a limited number of possible locations in cache. This reduces the hardware and/or time required to locate the given line when it is required. The extreme of this optimization is a directly mapped cache, though this typically leads to high cache eviction rates and the consequent large number of cache misses this can cause. Typical caches in modern processors are 8 or 16 way associative (each cache line can be in one of either 8 or 16 places).

To reduce cache misses hardware prefetching is commonly used to bring lines in to cache before they are requested. Special hardware detects memory access patterns with a fixed stride, predicting which cache lines will be required next. Software prefetching, where the programmer or compiler inserts special prefetch instructions, may also be used.

Consider, for example, performing a naive matrix-vector multiplication. We loop over the matrix columnwise as it is stored in column-major order. On the first multiply we bring the data for a_{11} and x_1 into cache. As these entries sit on a cache line this also brings in (say) $a_{21}, a_{31}, a_{41}, x_2, x_3$ and x_4. As we progress the prefetcher notices that two stride one access sequences are occurring, and requests the cache line containing a_{51} before we require it. If we continue to the end of the first column, the prefetcher has brought entries a_{12} and x_{n+1} into cache. Clearly we will use the entries of A, but not the (non-existent) entries of $\boldsymbol{x}$: there is some wastage. Eventually we will have run out of space in our cache, so the least recently touched entries, $A_{(1:m)1}$ will be evicted to make room. The $\boldsymbol{x}$ entries are kept in cache as they are continuously reused.

There are up to three levels of cache between main memory and each processing core in common chips today. In this hierarchy each cache is smaller and faster than the one below it (with main memory at the bottom and the core at the top). The cache closest to the processor is known as the level 1 cache, the next is the level 2, and so forth. Typical cache characteristics are given in Table 1.1.

Techniques to exploit caches attempt to maximize computation with data once it has been loaded into cache. This can normally be achieved for dense linear algebra through the use of block data formats and blocked algorithms, with carefully tuned block sizes. Some advanced compilers can automatically reorder and block nested loops for optimal performance; this can be

		Pentium IV	Intel Core 2	AMD Istanbul
Level 1	Size	16KiB	32KiB	64KiB
	Latency	12	3	-
Level 2	Size	1-2MiB	2-6MiB	512KiB
	Latency	18-20	14	-
Level 3	Size	-	-	6MiB
	Latency	-	-	-
Main Memory	Size	1-64GiB		
	Latency	100+		

Table 1.1: Cache characteristics for common processors. Latency is in clocks for floating point operations.

encouraged by keeping such loops simple with an inner iteration count that is clearly constant across outer iterations.

If blocking is not possible then a constant stride access pattern, preferably of stride one, should be used. This allows the hardware prefetcher and compiler to prevent all but a few cache misses. The small stride means that fewer virtual memory lookups have to be performed and fewer cache lines are used.

The full exploitation of caches can result in an order of magnitude difference in performance for codes which have a high ratio of floating point operations to memory footprint, such as matrix-matrix multiplication.

1.4 Branch prediction

If we consider a single processing core, we find that it is not a simple black box. Each operation goes through a number of different stages, at a rate of one per clock cycle. These stages form a pipeline typically between 12 and 20 stages in length. Splitting of instructions into micro-ops and using out-of-order execution allows much of the top level cache latency to be effectively hidden. However, if a branch is encountered there is a problem: we must complete the branch comparison before we can determine the subsequent operations.

This problem is tackled by predicting which branch will be taken based on heuristics and past experience. This typically works well for loops (which repeat on most iterations) and for conditions which are either occasionally true or occasionally false. It does not work well where both branches have a similar likelihood of being taken. However, even with prediction, branches represent an overhead which is best avoided.

We can reduce the number of branch predictions required by reordering static conditions outside of loops. For example, if a debugging flag is set at the start of the program but is tested in an inner loop, we could instead have two loops: one for when the debugging flag is set, and one for when it is not. This avoids the overhead of the branching operation and reduces the number of entries in the branch prediction table, particularly if the compiler unrolls the loop. Algorithms can also be carefully written to avoid the need for some tests on boundary conditions by setting up a halo of values which produce the correct behaviour. For example, rather than testing if we are on the first or last iteration of a loop to avoid an out-of-bounds array access, we can make the array larger and set the 0th and $(n + 1)$th entries to a value which has no effect. An alternative is to duplicate the code inside the loop for these special cases and only run the loop over indices 1 through $n - 1$.

The number of expensive branch mispredictions (which flush the pipeline) can be reduced by reordering and/or combining nested conditional statements such that the majority of tests are predictable. We can transform conditional numerical statements into functions instead. Consider the equivalence of the following two lines of Fortran if `a` is either 0 or 1:

```
if(a.eq.0) b = b + 1
b = b + (1-a)
```

Level	Type of operation	Examples		Speed (MFLOPS)
1	Vector-Vector	daxpy	$\boldsymbol{y} \leftarrow \alpha \boldsymbol{x} + \boldsymbol{y}$	345
2	Matrix-Vector	dgemv	$\boldsymbol{y} \leftarrow \alpha A \boldsymbol{x} + \beta \boldsymbol{y}$	736
		dtrsv	$\boldsymbol{x} \leftarrow L^{-1} \boldsymbol{x}$	714
		dsyr	$A \leftarrow \alpha \boldsymbol{x} \boldsymbol{x}^T + A$	1531
3	Matrix-Matrix	dgemm	$C \leftarrow \alpha AB + \beta C$	9303
		dtrsm	$B \leftarrow \alpha L^{-1} B$	9171

Table 1.2: The Basic Linear Algebra Subprograms. Speeds are maximum achieved over a range of operation sizes measured on `fox` (see Table 1.4) using the Goto BLAS.

1.5 SIMD instructions

Commodity chips now include a variety of Single Instruction Multiple Data (SIMD) extensions (SSE, SSE2, 3DNOW! etc.) aimed at multimedia decoding. These allow the use of instructions which act on a vector of data and perform fused multiply-add and vector load operations more efficiently than standard operations. They use additional special floating point units, and are key to achieving full floating point performance.

While these operations are not available explicitly from Fortran, they can be produced by vectorising compilers if the parallelism is clearly exhibited through the use of array slice notation and/or simple explicit loops.

1.6 BLAS

Due to the dominance of dense matrix operations in common linear algebra codes, much optimization effort has been concentrated upon them. This has resulted in a number of platform specific libraries which share a common interface, known as the Basic Linear Algebra Subprograms, or BLAS [Lawson et al. 1979; Dongarra et al. 1986; Dongarra et al. 1990].

The BLAS are split into three levels, representing different algorithmic complexities. These are summarised in Table 1.2. Clearly the use of the higher level matrix-matrix operations is preferable to multiple calls to lower level matrix-vector or vector-vector routines. This considerably higher performance is due to the high computation to memory ratio, which allows full exploitation of the caches.

1.7 Parallel programming models

As almost all computers are now equipped with multicore processors, and most supercomputers consist of multiway-multicore nodes, it is essential to exploit parallelism to achieve good performance. There is a plethora of available parallel programming models, however most are not easily portable, or not available to Fortran programmers. This section summarises the main models we shall consider.

1.7.1 OpenMP

OpenMP is designed to allow the programming of shared memory systems through the addition of extra directives to an otherwise serial code. These directives take the form of special comments around sections of code which are to be parallellised. When an OpenMP compliant compiler is used these sections of code are translated into the correct parallel (threaded) code. If a non-OpenMP compilation is required, the same source code can be used to generate a serial program with the same functionality.

OpenMP has a clearly defined memory model which does not directly map onto hardware, but allows compilers to do clever and (hopefully) performance-enhancing tricks. All variables are either *global* or *thread private* . Each thread has its own version of thread private variables, and these may only be accessed by that thread. Global variables are shared between threads. However the OpenMP memory model allows each thread to have its own transient copy. These copies of the global variables are only guaranteed to be synchronised to the real variable when an OpenMP `flush` directive is encountered (most common OpenMP directives imply one or more flushes). Additional unrequested synchronisation of these transient copies can occur at any time. Further, care must be taken with explicit flushes to ensure that code may not be reordered by an optimizing compiler in a fashion that causes subtle bugs.

Rather than using OpenMP in its originally designed mode (for loops) we instead often use it as a portable basis to build more complex schemes. These make heavy use of locks and the `flush` directive. *Locks* are shared variables which may be set or unset using functions from the OpenMP run-time library, and in their simplest form correspond to binary semaphores [Dijkstra 1965]. Only one thread may have a lock set at any given time; other threads must wait for the lock to be released before they can acquire it. For example if threads A and B both attempt to acquire the lock by calling the subroutine `omp_set_lock(lock)`, then only one of them will return. The other will not return from that subroutine until the winning thread releases the lock by calling `omp_unset_lock(lock)`. The use of locks allows much finer control over synchronisation than traditional `atomic`, `barrier`, or `critical` synchronisation as they can be associated with particular elements of data structures and arrays.

OpenMP is not designed for use in a distributed memory configuration, though Intel support Cluster OpenMP [Intel Corporation 1996]. This translates the OpenMP programs into a distributed memory application, though good performance is only really possible when the computation work between synchronisations is large.

The OpenMP standard [OpenMP Architecture Review Board 2008] and the book Using OpenMP [Chapman et al. 2008] are good sources of more detailed information. OpenMP is directly supported by most major Fortran and C compilers.

1.7.2 MPI

MPI stands for Message Passing Interface, and is the most common method for programming distributed memory machines. It can also be used on shared memory nodes, where it typically performs well, but not as well as codes which take specific advantage of shared memory.

In this programming model, processes on different hosts send messages to each other containing data. Processes optionally block until they receive data needed to continue. More advanced features of MPI-2 include one-sided communication (direct access to another process's memory space) and parallel I/O.

MPI supports both Fortran and C, and more detailed information can be found in the books of Snir et al. [1996], Gropp et al. [1999] and the MPI standards [Message Passing Interface Forum 2008a; 2008b]. It is possible to combine MPI and OpenMP in the same program to reflect a cluster of SMP machines, though the reward for the extra effort may not be worthwhile.

1.7.3 POSIX threads

The basic mechanism for multi-tasking under unix-like systems is process forking, where a single process splits into two. The overheads of this mechanism are large, and the POSIX Threads (PThreads) API was designed as a light-weight alternative. PThreads are now supported by most operating systems. They can be easily accessed via C and thus wrapped for use from Fortran if necessary. These are often used to implement OpenMP and MPI on shared memory systems. We have chosen not to use this interface as there is no direct language support for Fortran and full portability is not guaranteed. While PThreads offer a wider range of synchonisation primitives than OpenMP (such as general semaphores and monitors), we do not require anything more than simple locks, and benefit from the higher level thread management interfaces offered by OpenMP.

1.7.4 Task-based

Recently there has been much work done on task-based parallelism. In this model the work is split into tasks, which may optionally have dependencies. These tasks are then scheduled on different processing cores until no work remains. During the execution of a task, additional tasks may be added as dependencies are satisfied.

	Single	Double
Sign + Exponent + Significand	$32 = 1 + 8 + 23$	$64 = 1 + 11 + 52$
Epsilon ($\min \epsilon : 1 + \epsilon \neq 1$)	$2^{-24} \approx 6.0 \times 10^{-8}$	$2^{-53} \approx 1.1 \times 10^{-16}$
Underflow Threshold	$2^{-126} \approx 1.2 \times 10^{-38}$	$2^{-1022} \approx 2.3 \times 10^{-308}$
Overflow Threshold	$(2 - 2^{-23}) \times 2^{127}$ $\approx 3.4 \times 10^{38}$	$(2 - 2^{-52}) \times 2^{1023}$ $\approx 1.8 \times 10^{308}$

Table 1.3: Summary of the IEEE 754-2008 standard, comparing single and double precision.

The most recent OpenMP 3.0 standard [OpenMP Architecture Review Board 2008] adds some limited tasking directives, while Intel has recently released its Thread Building Blocks (TBB). The effectiveness of such methods has been demonstrated in dense linear algebra through the PLASMA project [Buttari et al. 2006; Buttari et al. 2009; Song et al. 2009], and various compilers and libraries have been developed to assist in their use. These include SMP Superscalar (SMPSS) [Perez et al. 2008], Cilk [Randall 1998] and the work of Kurzak and Dongarra [2009].

A single standard for task-based programming has not yet emerged, though it is hoped the examples produced in this thesis will be considered during the evolution of what finally emerges.

1.8 IEEE floating point

Almost all common HPC platforms (with the notable exception of GPGPU) implement the full IEEE floating point standard [IEEE 2008]. This offers guarantees on behaviour of numerical calculations that are useful in error analysis, in addition to giving more portable numerical behaviour.

Floating point numbers are represented by a sign bit and two integers. These integers are the significand (mantissa), c, and the exponent, q, from which the value can be obtained using the formula

$$(-1)^s \times c \times 2^q$$

where s is the sign bit, having value 0 or 1. Special values of the exponent q are used to signal special values:

denormal number A value smaller than can be represented by the normal size of the exponent. Bits are borrowed from the significand to represent this number (with fewer significant digits). This allows gradual underflow to be used for calculations involving small numbers. Also known as subnormal numbers.

Infinity Can be positive or negative. Generated, for example, by division of a non-zero value by zero.

NaN Not a Number, generated by otherwise undefined operations.

We have found that performance of denormal calculations on recent Intel chips seems to have a large performance penalty, presumably due to a more limited execution pathway for this special case. Many processors allow denormal numbers to be flushed to zero rather than stored; doing so prevents the propagation of these values through calculations. For some problems it is highly advantageous to enable this feature (one problem ran over 100 times faster once this was enabled).

Various ranges of single (32-bit) and double (64-bit) precision numbers are given in Table 1.3. The epsilon value is the smallest value such that $1 + \epsilon$ and 1 have a distinct value. Overflow is the largest representable number, and underflow is the smallest normal number.

1.9 Machines

In this thesis we report results on a wide range of machines which have changed over time. The main machines used are detailed in Table 1.4, but software and compiler options vary between

	`curtis`	`fox`	`HPCx`	`seras`	`jhogg`
Processor Model	Pentium IV	Xeon E5420	Power5	Athlon64	Xeon E5520
Clock	2.80 Ghz	2.50Ghz	1.50Ghz	1.83GHz	2.27[1] Ghz
Cores	1	2×4	16×1	1×2	1×4
Theoretical Peak (1/all cores)		10 / 80	6 / 96		10.1 / 36.3
`dgemm` Peak (1/all cores)		9.3 / 72.8	5.0 / 70.4		9.6 / 33.9
Memory	1GiB	32GiB	32GiB	2GiB	2GiB

[1]The E5520 can boost a single core to a frequency of 2.53Ghz if machine is lightly loaded

Table 1.4: Machines used in this thesis, peak speeds measured in Gflop/s.

experiments, and are detailed at relevant points in the text. We note that `HPCx` refers to a single SMP node of the UK supercomputer.

Chapter 2

Solving linear systems

In this chapter we develop the theory to solve general equations of the form

$$AX = B \tag{2.1}$$

for X. A is a $n \times n$ symmetric matrix, X is a $n \times k$ unknown matrix and B is a $n \times k$ matrix.

If A were not square then either the equations are inconsistent, or there is a space of non-unique solutions. If A had full rank, but was not symmetric, then a solution can be found through a generalisation of the symmetric methods discussed here (for example through Gaussian elimination). Both of these cases are outwith the scope of this thesis.

Without loss of generality we shall consider only the case $k = 1$; if $k > 1$, then we can instead solve k systems with a single right hand side. However, most methods discussed can be efficiently extended to handle multiple right-hand sides. We hence consider

$$A\boldsymbol{x} = \boldsymbol{b}. \tag{2.2}$$

There are two broad approaches to the solution of such systems:

Direct Methods factorize A into the product of two or more matrices with which solves may be easily made. For example, the Cholesky factorization for positive definite A finds a lower triangular matrix L such that $A = LL^T$. Major advantages of this approach include the reuse of factors to solve repeatedly for different right-hand sides, and the black-box nature of the solver that allows it to be readily applied to almost any problem. The substantial disadvantage to these methods is that the computational effort can grow much faster than the dimension n; for dense systems the computational effort typically grows with the cube of n.

Iterative Methods employ an iterative scheme to reduce the error in an approximate solution to (2.2). These methods normally only require the ability to form the product $A\boldsymbol{x}$, and as such can have a lower complexity than direct methods, but only if a low iteration count is achieved. In practice, preconditioning is often necessary for good performance, so applicability is limited by the availability of a good preconditioner.

We cover direct methods in the majority of this chapter. First we address serial and parallel dense Cholesky factorization in Section 2.1. Section 2.2 describes the related sparse indefinite LDL^T factorization, while Sections 2.4 and 2.5 tackles their sparse equivalents. Section 2.6 addresses out-of-core working while Section 2.7 describes iterative techniques for refining solutions generated by these direct methods. Finally Section 2.8 describes some of the solver software we use in this thesis.

2.1 Dense Cholesky factorization

If A is positive definite (all its eigenvalues are positive) then the unique matrix factorization $A = LL^T$ exists. The proof of this statement (as given for Theorem 23.1 of

Algorithm 1 Cholesky, for in place factorization $A = LL^T$

```
Require: Symmetric Positive Definite $A$
  for $j = 1, n$ do
    $l_{jj} \leftarrow \sqrt{a_{jj}}$
    $L_{(j+1:n)j} \leftarrow A_{(j+1:n)j}/l_{jj}$
    $A_{(j+1:n)(j+1:n)} \leftarrow A_{(j+1:n)(j+1:n)} - L_{(j+1:n)j}L^T_{(j+1:n)j}$
  end for
```

Numerical Linear Algebra [Trefethen and Bau III 1997]) is constructive through Algorithm 1, and proceeds by induction. Making the inductive assumption that a unique $L_kL_k^T$ factorization exists for a $k \times k$ positive definite matrix A_k, consider a single step of Algorithm 1 to a $(k+1) \times (k+1)$ matrix A_{k+1}. This yields the following partial factorization:

$$A_{k+1} = \begin{pmatrix} a_{11} & \boldsymbol{w}^T \\ \boldsymbol{w} & K \end{pmatrix} = \begin{pmatrix} l_{11} & 0 \\ \boldsymbol{w}/l_{11} & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & K - \boldsymbol{w}\boldsymbol{w}^T/a_{11} \end{pmatrix} \begin{pmatrix} l_{11} & \boldsymbol{w}^T/l_{11} \\ 0 & I \end{pmatrix}.$$

Clearly the first diagonal entry is positive as A_{k+1} is positive definite, so $l_{11} = \sqrt{a_{11}}$ exists and is real. Set $A_k = K - \boldsymbol{w}\boldsymbol{w}^T/a_{11}$ and observe that as the next lower right principal submatrix of A, it is positive definite. By our inductive assumption, a unique $L_kL_k^T$ factorization of A_k exists and may be substituted in. Hence if the factorization exists for $k = 1$, it exists for all $k > 1$. A 1×1 positive definite matrix contains a single positive entry a_{11} and admits the unique factorization $\sqrt{a11}\sqrt{a_{11}}^T$.

Once we have obtained the factors L, we can solve the system $A\boldsymbol{x} = \boldsymbol{b}$ through simple forward and backward substitution:

$$\begin{aligned} L\boldsymbol{y} &= \boldsymbol{b} \\ L^T\boldsymbol{x} &= \boldsymbol{y}. \end{aligned}$$

This only requires the solution of two triangular systems. Counting the total number of floating point operations we see that Cholesky factorization is an $O(n^3)$ algorithm.

Algorithm 1 is backwards stable [Wilkinson 1968]. However, it may suffer in the presence of ill conditioning, though in practice this can be detected by monitoring the magnitude of the diagonal entries as the factorization progresses. It can then be addressed internally but should be reported to the user when a problem is encountered.

2.1.1 Serial implementation

Algorithm 1 can be reorganised to a block form as Algorithm 2, allowing the exploitation of level 3 BLAS for the computationally intensive components. These are matrix-matrix multiplies and triangular solves. The remaining operation, factorizing the diagonal block, represents a small number of operations on an amount of data that should fit in cache, and is performed using the element-wise Cholesky factorization (Algorithm 1).

Algorithm 2 Right-looking block Cholesky, for in place factorization $A = LL^T$

```
  for $j = 1, nblk$ do
    $L_{jj} \leftarrow \text{factor}(A_{jj})$
    $L_{j(j+1:nblk)} \leftarrow A_{j(j+1:nblk)}L_{jj}^{-T}$
    for $k = j+1, nblk$ do
      $A_{(j+1:nblk)k} \leftarrow A_{(j+1:nblk)k} - L_{(j+1:nblk)j}L^T_{kj}$
    end for
  end for
```

This algorithm is called right-looking as the outer product updates are applied to the right-hand submatrix as they are generated. Algorithm 3 is a left-looking block Cholesky, applying updates to the current block column before we factorize it. Experiments [Anderson and Dongarra 1990] seem to show that there is little performance difference between these variants in the serial case although, conceivably, differences in cache read and write performance could result in one being more favourable than another on a particular machine.

Algorithm 3 Left-looking block Cholesky, for in place factorization $A = LL^T$

```
for $j = 1, nblk$ do
  for $k = 1, j-1$ do
    $A_{(j:nblk)j} \leftarrow A_{(j:nblk)j} - L_{(j:nblk)k} L_{jk}^T$
  end for
  $L_{jj} \leftarrow \text{factor}(A_{jj})$
  $L_{j(j+1:nblk)} \leftarrow A_{j(j+1:nblk)} L_{jj}^{-T}$
end for
```

These algorithms give performance very close to that of the level 3 BLAS they employ, and the standard interface to such routines is LAPACK [Anderson et al. 1999]. Cholesky factorization is performed by the routine `_potrf` that expects the lower or upper part of A to be entered in a full memory (ie $n \times n$) column-major format. Traditionally Cholesky factorizations using the packed format ($n(n+1)/2$ memory) have a lower performance due to the inability to exploit level 3 BLAS. Better performance in this case can be achieved through the rearrangement to a block-column [Anderson et al. 2005] or recursive [Andersen et al. 2001] format.

An alternative approach to optimizing such algorithms for serial execution is to encode the algorithms in a format which exposes the potential reorderings to a heuristic-driven compiler or library that can then formulate an optimal code for a given architecture. This approach is taken by the authors of FLAME [Zee et al. 2008], and seems to produce results comparable with the more traditional approaches.

2.1.2 Traditional parallel implementation

To expose parallelism while maintaining the cache friendly blocked form of the Cholesky factorization, we rewrite our blocked Cholesky factorization once more as Algorithm 4. This form has split the large panel updates of Algorithm 2 into operations involving only the blocks.

Algorithm 4 Fine-grained right-looking block Cholesky

```
for $j = 1, nblk$ do
  $L_{jj} \leftarrow \text{factor}(A_{jj})$
  for $i = j+1, nblk$ do
    $L_{ij} \leftarrow A_{ij} L_{jj}^{-T}$
    for $k = j+1, i$ do
      $A_{ik} \leftarrow A_{ik} - L_{ij} L_{kj}^T$
    end for
  end for
end for
```

There are three main block operations:

factorize(j) Factorize the block A_{jj}.

solve(i, j) Perform the triangular solve $L_{ij} \leftarrow L_{ij} A_{jj}^{-T}$.

update(i, j, k) Perform the outer product update to block (i, j) from column k, $A_{ij} \leftarrow A_{ij} - L_{ik} L_{jk}^T$.

Traditional approaches, such as that of ScaLAPACK [Blackford et al. 1997], merely paralellise the two inner loops of our algorithm. This results in a profile such as that in Figure 2.1. Clearly the major source of inefficiency is when processors are sitting idle waiting for the current loop to complete. Near the start this inefficiency is small. However, towards the end of the factorization (where the total number of tasks is small) this accounts for the majority of the time across all processors.

Various techniques have been developed to overcome this problem, the most successful has been the look-ahead method [Kurzak and Dongarra 2007; Strazdins 2001]. This method exploits

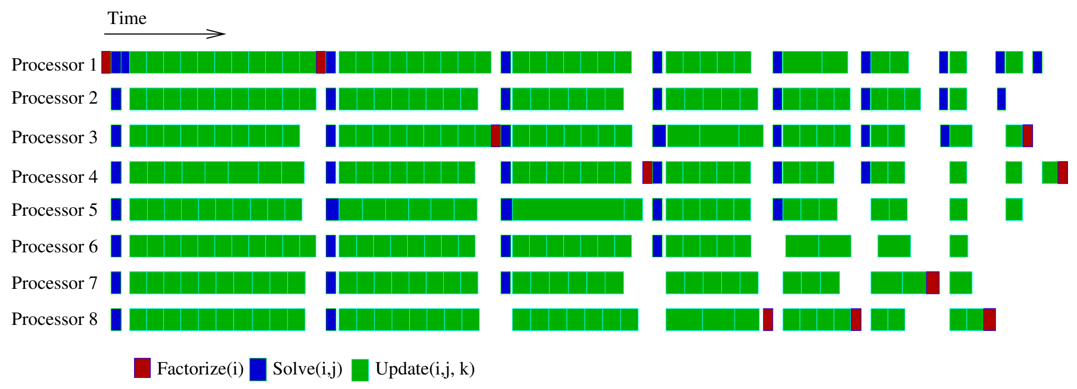


Figure 2.1: Parallel profile for right-looking Cholesky.

the left-looking formulation to overlap future update tasks for which the data is available with current computations.

2.1.3 DAG-based parallel implementation

An evolution of the look-ahead method uses a dependency graph to schedule block operations. Block operations (tasks) are represented as nodes, and order dependencies are represented as directed edges. The result is a directed acyclic graph (DAG). DAG-driven linear algebra uses either a static or dynamic schedule based on these graphs to allocate tasks to cores.

A Cholesky factorization is typically split into the three tasks of the previous section, with the following dependencies,

factorize(j) **depends on:** update(j,j,k) for all $k = 1,\dots,j-1$.

solve(i,j) **depends on:** update(i,j,k) for all $k = 1,\dots,j-1$, factorize(j).

update(i,j,k) **depends on:** solve(i,k), solve(j,k).

For a small 4×4 block matrix this results in the dependency DAG shown in Figure 2.2.

Dynamic scheduling uses the concept of a pool of ready tasks (those whose dependencies have already been met). Each thread executes one task at a time. Once the task is finished a global lock is acquired and dependent tasks are added to the ready pool if possible. The thread then picks a task from the pool and releases the global lock. This approach is used for example by the TBLAS [Song et al. 2009], though this does not account for data reuse.

Static scheduling allocates tasks to threads a priori and in order. Communication is then limited to checking if dependencies are complete, which can be done without global locking. This approach risks more stalls than the dynamic approach but also has much smaller communication overheads. This approach is currently used by PLASMA [Buttari et al. 2009; Buttari et al. 2006].

A recent comparison by Agullo et al. [2009] suggests that the static scheduling of PLASMA currently outperforms the dynamic scheduling of the TBLAS due to data reuse. They also present a comparison with traditional methods showing DAG-based codes outperforming them for Cholesky factorization. The competitiveness of DAG-based codes for other common LAPACK calls is limited by the introduction of additional operations to allow a task-based representation.

Alternative scheduling techniques for DAG-driven Cholesky factorizations are the subject of Chapter 6.

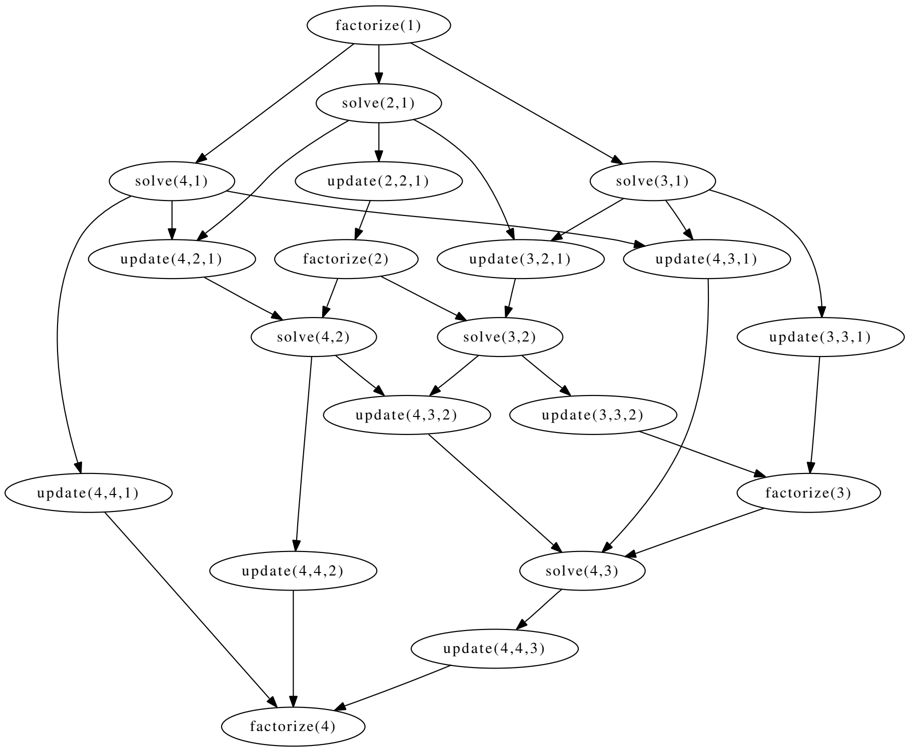


Figure 2.2: Dependency DAG for a 4×4 block matrix.

2.2 Symmetric indefinite factorization

In the indefinite case the $A = LL^T$ factorization does not exist. Consider attempting to factorize

$$\begin{pmatrix} -1 & 1 \\ 1 & 1 \end{pmatrix}.$$

Cholesky fails as we cannot take the square root of a negative number and remain real. Instead we consider an alternative factorization $PAP^T = LDL^T$ where D is block diagonal and P is a permutation matrix. The diagonal blocks of D are the pivots for the factorization. Restricting ourselves to 1×1 pivots is clearly insufficient: consider

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix},$$

where we cannot symmetrically permute a non-zero to the top-left corner. If A has full rank, it is instead sufficient to allow 2×2 pivots, as we can always symmetrically permute non-zero entries onto the subdiagonal. This approach works as we effectively relax the constraint of pivoting symmetrically in a very limited fashion. A naive implementation is shown as Algorithm 5.

This algorithm is naive as it does not take into account stability considerations. Consider the factorization

$$\begin{pmatrix} \epsilon & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & \\ 1/\epsilon & 1 \end{pmatrix} \begin{pmatrix} \epsilon & \\ & -1/\epsilon \end{pmatrix} \begin{pmatrix} 1 & 1/\epsilon \\ & 1 \end{pmatrix},$$

that has large entries in the factors. Note that the pivoted LU factorization is stable,

$$\begin{pmatrix} & 1 \\ 1 & \end{pmatrix} \begin{pmatrix} \epsilon & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & \\ & 1 \end{pmatrix} = \begin{pmatrix} 1 & \\ \epsilon & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ & 1 \end{pmatrix}.$$

We clearly need to perform some pivoting for numerical stability. The most common such tech-

Algorithm 5 Naive symmetric indefinite factorization, performing $PAP^T = LDL^T$ in place

```
$j = 1$
while $j \leq n$ do
  if $A_{jj} > 0$ then
    Choose a $1 \times 1$ pivot, $d_{jj} \leftarrow a_{jj}$.
    Set $s = 1$.
  else
    Find $k$ such that $A_{kj} \neq 0$.
    Symmetrically permute row/column $k$ to position $j + 1$.
    Choose a $2 \times 2$ pivot, $D_{(j:j+1)(j:j+1)} \leftarrow A_{(j:j+1)(j:j+1)}$.
    Set $s = 2$.
  end if
  $L_{(j+1:n)(j:j+s-1)} \leftarrow A_{(j+1:n)(j:j+s-1)} D^{-1}_{(j:j+s-1)(j:j+s-1)}$
  $A_{(j+s:n)(j+s-1:n)} \leftarrow A_{(j+s:n)(j+s-1:n)} - L_{(j+s:n)(j:j+s-1)} D_{(j:j+s-1)(j:j+s-1)} L^T_{(j+s:n)(j+s-1)}$
  $j = j + s$
end while
```

niques are Bunch-Parlett (full) [Bunch and Parlett 1971] and Bunch-Kaufman (rook) [Bunch and Kaufman 1977] pivoting. Bunch-Kaufmann pivoting replaces the pivoting step of our naive Algorithm 5 to obtain Algorithm 6, where the unusual value of μ derives from the stability analysis of the algorithm and is chosen to minimize the growth factors.

Full pivoting uses the column that minimizes the growth factor (that is the ratio of the largest element in the pivotal column before and after pivoting) as a pivot, requiring a full scan of the matrix. Rook pivoting instead determines a pivot with a limited growth factor by scanning only the row and column of the current candidate pivot.

2.3 Sparse matrices

A matrix is sparse if it is worth exploiting the presence of zero values, typically meaning that most of the entries are zero. Such zeroes are referred to as *structural zeros*, differentiated from *numerical zeros* that are handled explicitly (that is treated as if they were non-zero). Numerical zeros often result from cancellation, or are introduced deliberately to allow the exploitation of more efficient data structures. When there are only a handful of non-zeroes per column it is crucial for efficiency that this is exploited. However, for certain computations (such as symmetric factorizations) it can be advantageous to exploit sparsity when a small percentage of the matrix is non-zero. A typical data structure for storing a sparse matrix will store for each column:

- The number of non-zeros.
- The row indices of the non-zeros.
- The values of the non-zeros (corresponding to the row indices).

Typically the number of non-zeros is stored as an offset into the arrays containing the row indices and the numerical values.

Sparse matrix structures are equivalent to graphs. We define the graph of a sparse matrix by mapping columns to nodes and non-zeros to edges. If entry (i, j) in the matrix is non-zero, then there is an edge from node i to node j. Clearly if the matrix is symmetric we can replace the pairs of directed edges with an undirected edge. Diagonal entries are normally implicitly assumed to exist, and are omitted from the graph. Figure 2.3 shows a symmetric sparse matrix and its equivalent graph. Permutations of a matrix are equivalent to renumbering the nodes of the corresponding graph.

2.4 Sparse symmetric factorization

Algorithm 7 gives the obvious sparse variant of Algorithm 1.

Algorithm 6 Symmetric indefinite factorization with Bunch-Kaufmann pivoting, performing $PAP^T = LDL^T$ in place

```
$\mu = (1 + \sqrt{17})/8$
$j = 1$
**while** $j \leq n$ **do**
  Determine largest off-diagonal element in column $j$, $\lambda = |a_{rj}| = \max(|a_{(j+1)j}|, \ldots, |a_{nj}|)$.
  **if** $a_{jj} > \mu\lambda$ **then**
    Choose a $1 \times 1$ pivot, $d_{jj} \leftarrow \sqrt{a_{jj}}$.
    Set $s = 1$.
  **else**
    Determine largest off-diagonal element in row/column $r$,
        $\sigma = |a_{pr}| = \max(|a_{r1}|, \ldots, |a_{r(r-1)}|, |a_{(r+1)r}|, \ldots, |a_{nr}|)$.
    **if** $\sigma|a_{jj}| \geq \mu\lambda^2$ **then**
      Choose a $1 \times 1$ pivot, $d_{jj} \leftarrow \sqrt{a_{jj}}$.
      Set $s = 1$.
    **else if** $|a_{rr}| \geq \mu\sigma$ **then**
      Symmetrically permute row/column $r$ to position $j$.
      Choose a $1 \times 1$ pivot, $d_{jj} \leftarrow \sqrt{a_{jj}}$.
      Set $s = 1$.
    **else**
      Symmetrically permute rows/columns $r$ and $p$ to positions $j$ and $j + 1$ respectively.
      Choose a $2 \times 2$ pivot, $D_{(j:j+1)(j:j+1)} \leftarrow A_{(j:j+1)(j:j+1)}$.
      Set $s = 2$.
    **end if**
  **end if**
  $L_{(j+1:n)(j:j+s-1)} \leftarrow A_{(j+1:n)(j:j+s-1)} D^{-1}_{(j:j+s-1)(j:j+s-1)}$
  $A_{(j+s:n)(j+s-1:n)} \leftarrow A_{(j+s:n)(j+s-1:n)} - L_{(j+s:n)(j:j+s-1)} D_{(j:j+s-1)(j:j+s-1)} L^T_{(j+s:n)(j+s-1)}$
  $j = j + s$
**end while**
```

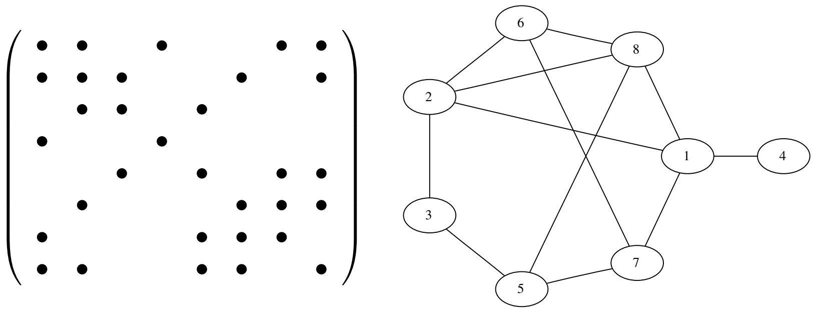


Figure 2.3: Sparse matrix structure and its equivalent graph.

Algorithm 7 Basic sparse Cholesky factorization

```
for $j = 1, n$ do
  $l_{jj} \leftarrow \sqrt{a_{jj}}$
    (factor).
  for $i \in \{i : l_{ij} \neq 0\}$ do
    $l_{ij} \leftarrow l_{ij}/a_{jj}$
      (solve).
  end for
  for $i \in \{i : l_{ij} \neq 0\}$ do
    for $k \in \{k > i : l_{kj} \neq 0\}$ do
      $a_{ki} \leftarrow a_{ki} - l_{ij}l_{kj}$
        (update).
    end for
  end for
end for
```

$$P = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 3 & 5 & 2 & 1 & 4 & 6 & 7 & 8 \end{pmatrix}$$

$$\underset{LL^T = A}{\begin{pmatrix}
\bullet & & & & & & & \\
\bullet & \bullet & & & & & & \\
 & \bullet & \bullet & & & & & \\
\bullet & \circ & \circ & \bullet & & & & \\
 & & \bullet & \circ & \bullet & & & \\
 & \bullet & \circ & \circ & \circ & \bullet & & \\
\bullet & \circ & \circ & \circ & \bullet & \bullet & \bullet & \\
\bullet & \bullet & \circ & \circ & \bullet & \bullet & \circ & \bullet
\end{pmatrix}}
\qquad
\underset{LL^T = PAP^T}{\begin{pmatrix}
\bullet & & & & & & & \\
 & \bullet & & & & & & \\
\bullet & & \bullet & & & & & \\
 & \bullet & & \bullet & & & & \\
 & \bullet & \bullet & \circ & \bullet & & & \\
 & & & & \bullet & \bullet & & \\
 & & \bullet & \bullet & \circ & \bullet & \bullet & \\
 & & \bullet & \bullet & \bullet & \bullet & \circ & \bullet
\end{pmatrix}}$$

Figure 2.4: Fill-in of the factors L with and without a fill-reducing permutation. Original entries drawn from A are represented as $\bullet$, and fill in by $\circ$.

Obviously a_{ki} may be a structural zero initially but takes a non-zero value due to the update operation; these new non-zeros generated during the factorization are known as *fill-in*. As an operation on a graph, each elimination corresponds to removing the corresponding node from the graph, and forming a clique from its neighbours. Any new edges are fill-in. At most $(r-1)^2$ fill-ins can be introduced, where r is the number of neighbours for that node (its degree). Typically the factors L are considerably denser than the original matrix A. The amount of fill-in can be dramatically reduced through the application of a symmetric permutation to the matrix before starting the factorization. Figure 2.4 shows the fill-in generated with two different permutations of the same matrix.

Looping over only the non-zeros and allowing sufficient space for fill-in to occur is a non-trivial problem. It is possible to predict the non-zero pattern of L before we begin the factorization to surmount this problem. This is typically known as the symbolic factorize or *analyse phase* of the sparse factorization.

There are typically four such phases in a sparse Cholesky factorization:

Preorder Permutes A symmetrically to reduce fill-in.

Analyse Determines the non-zero structure of the factors and plans the numerical computation. (Though some of these may in fact be numerical non-zeros due to cancellation).

Factorize Performs the actual numerical computation.

Solve Solves $LL^T\boldsymbol{x} = \boldsymbol{b}$ via forward and backward substitution.

Each phase is now addressed in more detail.

2.4.1 Preorder

Preordering is the process of determining a symmetric permutation of A such that the number of non-zeros of L is minimized. The problem of finding an optimal ordering is NP-complete [Yannakakis 1981], so we use heuristic methods. The most common approaches are usually variants either of Minimum Degree (eg MMD, AMD) or Nested Dissection (eg MeTiS).

Minimum degree attempts to minimize fill-in by choosing to eliminate the node with the smallest degree at each stage, thus minimizing $(r-1)^2$ (the maximum potential fill-in). Two main variants are Multiple Minimum Degree (MMD), where as many as possible non-adjacent nodes are removed at each stage, and Approximate Minimum Degree (AMD), where an expensive update step is replaced by a cheaper approximate update. The papers of George and Liu [1989] and Heggernes et al. [2001] give a more in-depth description of the algorithm and its variants, along with implementation details.

Nested dissection uses a recursive method which repeatedly identifies a small set of separators which partition the graph into two pieces and removes them from the graph. A fuller description can be found in the MeTiS paper [Karypis and Kumar 1995].

It is worth noting that all these algorithms can employ supernode detection in order to increase efficiency. These supernodes also aid in later stages of the factorization (supernodes are defined in section 2.4.2).

Some Cholesky codes will try several different ordering algorithms and then use the best one (in terms of fill-in) for the actual factorization. While this results in a more expensive preordering phase, the saving on the factorize phase is large enough to justify it. In cases where a matrix with the same non-zero pattern is factorized repeatedly, such as interior point methods, it is highly recommended.

Table 2.1 shows the fill-in of some matrices under different orderings and the resulting speed of the factorization phase. COLAMD and AMD are both taken from Tim Davis' Sparse Matrix programs, MeTiS uses the well known library of the same name, and NESDIS is a MeTiS-based nested dissection algorithm included in CHOLMOD. Matrices are drawn from the factorization performed in the first phase of an interior point method for problems drawn from the Netlib linear programming [Gay 1985] and Kennington [Carolan et al. 1990] test sets. All timings are for the factorize phase of CHOLMOD running on a single core of `seras` using the Goto BLAS (single threaded). Clearly the factorization time is directly affected by the number of non-zeros in the factors, as these require additional operations to compute and keep track of. There is no clear winning choice between the orderings on these problems, though on the denser PDS problems the nested dissection problems do better.

2.4.2 Analyse

At a minimum, the analyse phase determines storage requirements for each column of L. It often also pre-computes information that is useful in the factorize phase.

Traditionally the non-zero pattern of L is found using a scheme such as Algorithm 8. Here we use the calligraphic $\mathcal{A}_j = \{i : A_{ij} \neq 0, i \geq j\}$ and $\mathcal{L}_j = \{i : L_{ij} \neq 0\}$ to represent the lower triangular non-zero sets of the columns of the matrix and its factors (as A is symmetric the upper triangular structure is easily obtained from that of the lower triangle). The vector π is used to simplify this computation, but actually represents a structure known as the elimination tree that will be described presently.

This algorithm exploits an essential property of the sparse factorization: the non-zero pattern of a column is inherited by all columns that the update touches. A corollary of this is that the first such updated column will also update the same set of non-zeros (and hence columns), possibly with some additional entries. We can hence define a tree as the reduction of the data dependency graph of the factors. Each row/column becomes a node, whose parent is given by the first non-zero entry below the diagonal in the factor L. The remaining non-zeros are given by the path from a node to the root of the tree. This is known as the *elimination tree*, and is represented by the vector $\boldsymbol{\pi}$ (also known as the parent function) calculated in Algorithm 8. The

	25fv47		*cre-a*		*dcp1*		*ken-07*	
	nnz	time	nnz	time	nnz	time	nnz	time
COLAMD	39534	0.0059	37116	0.0072	228142	0.0495	15529	0.0039
AMD	34372	0.0056	35897	0.0073	257969	0.0496	15319	0.0039
MeTiS	33925	0.0049	37588	0.0068	231209	0.0485	16095	0.0039
NESDIS	32385	0.0046	37696	0.0076	226892	0.0465	15319	0.0036

	ken-11		*ken-13*		*ken-18*		*pds-02*	
	nnz	time	nnz	time	nnz	time	nnz	time
COLAMD	133221	0.0301	353474	0.0730	2.23e+06	0.4545	44486	0.0086
AMD	133087	0.0303	349175	0.0727	2.32e+06	0.4527	44267	0.0088
MeTiS	137043	0.0296	378309	0.0752	2.53e+06	0.4658	44788	0.0081
NESDIS	132958	0.0307	351710	0.0726	2.43e+06	0.4585	42551	0.0072

	pds-06		*pds-10*		*pds-20*	
	nnz	time	nnz	time	nnz	time
COLAMD	593816	0.1904	1.63e+06	0.5877	6.83e+06	3.8849
AMD	600597	0.1655	1.61e+06	0.5829	6.98e+06	3.9840
MeTiS	401383	0.0952	1.23e+06	0.3063	5.35e+06	2.0859
NESDIS	365900	0.0684	1.16e+06	0.2644	4.64e+06	1.5250

Table 2.1: Amount of fill-in and time required for factorization using CHOLMOD with the orderings: COLAMD [Davis et al. 2004], AMD [Amestoy et al. 2004], MeTiS [Karypis and Kumar 1998] and the NESDIS provided as part of CHOLMOD [Chen et al. 2008].

Algorithm 8 Normal (left-looking) analyse algorithm

```
Initialise $\pi(:) = n + 1$
**for** $i = 1$ to $n$ **do**
  $\mathcal{L}_i = \mathcal{A}_i$
  **for all** $j : \pi(j) = i$ **do**
    $\mathcal{L}_i = \mathcal{L}_i \cup \mathcal{L}_j$
  **end for**
  **if**$(L\backslash\{i\} \neq \phi)$ $\pi(j) = \min \mathcal{L}\backslash\{i\}$
**end for**
```

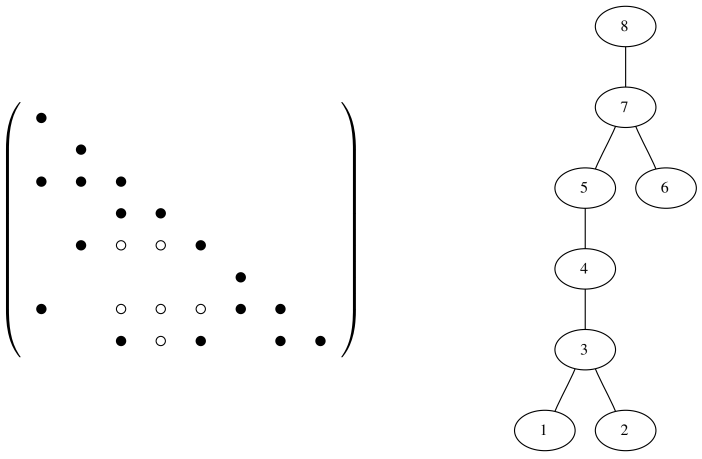


Figure 2.5: A matrix and its elimination tree.

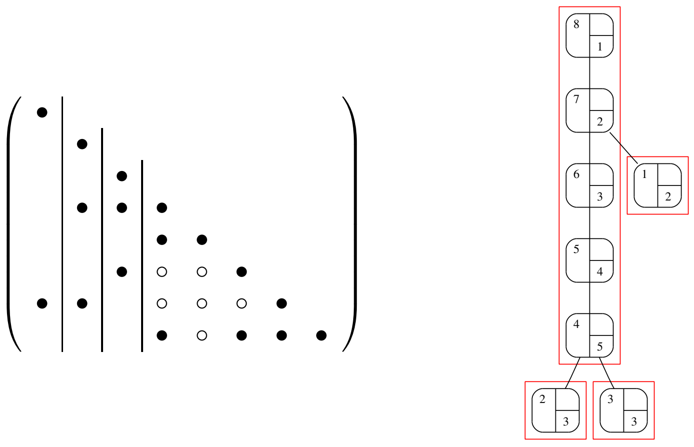


Figure 2.6: Matrix and its elimination tree, after some reordering. Numbers on the left are node numbers. Numbers in the bottom right are column non-zero counts.

entry $\pi(i)$ gives the parent of node i in the elimination tree, or has the value $n+1$ to indicate it is a root. Figure 2.5 demonstrates this concept.

Any ordering on the nodes of the elimination tree such that each node has a higher index than its children has a corresponding permutation of A that does not change the amount of fill-in. A postordering of the tree has this property and will place columns with similar non-zero patterns next to each other. (A postorder being a numbering that corresponds to a depth first search order).

The paper of Liu [1990] covers elimination trees and their properties in great detail.

Supernodes

We define a supernode as a set of consecutive columns with the same non-zero pattern (up to entries above the diagonal). These are characterised in the adjacency graph by a clique and may be exploited in the preordering phase. Detection in the elimination tree is easy if non-zero counts for each column are known: supernodes are direct lines of descent with an extra non-zero added at each level, and can be reordered to be consecutive. A full algorithm is given, for example, in Liu et al. [1993]. Figure 2.6 shows the previous matrix partitioned into supernodes, along with its elimination tree.

A supernode can be stored as a dense submatrix of L. We can hence exploit the level 3 BLAS for operations within and between such constructs. The reduction of the elimination tree which consists of supernodes is known as the *assembly tree*.

Unfortunately the number of supernodes in sparse matrices is typically small, limiting the scope for efficient operations. To increase the applicability of the BLAS we relax the requirement for matching non-zero patterns. We look for paths within the elimination tree with columns that have similar non-zero patterns (identified by a small difference in column counts) and combine these into artificial supernodes. This corresponds to introducing explicit zeros into our computations. Large performance increases are observed in most cases. This is due to the increased cache efficiency of the BLAS and the reduced number of sparse scatter operations involved in the updates. Ashcraft [1989] describes a number of algorithms for the identification of such supernodes.

Finding the elimination tree

In the next section we describe several algorithms which rely on having the elimination tree available. The elimination tree can be efficiently computed by Liu's algorithm [Liu 1986], shown here as Algorithm 9. Consider the construction of L on a row by row basis working from left to right. If an entry L_{ij} is non-zero then either:

- It is the first non-zero in column j. We set $\pi(j) = i$.
- There is an entry in column j above this one. Entry $L_{i\pi(j)}$ is non-zero.

This is equivalent to proceeding up the partially constructed elimination tree until we find the current root. We then add the current row at the top of the subtree. This algorithm is made more efficient through the use of path compression: as we ascend the tree we store a shortcut pointing to the new top of the tree, which we know will be i. These shortcuts are stored using a *virtual forest*, represented in Algorithm 9 by $\bar{\pi}(:)$.

Algorithm 9 Liu's algorithm with path compression for finding the elimination tree of A

```
for $i = 1$ to $n$ do
  for all $j \in \mathcal{A}_i$ do
    $k = j$
    while $\bar{\pi}(k) \neq 0$ do
      $l = \bar{\pi}(k)$
      $\bar{\pi}(k) = i$
      $k = l$
    end while
    $\pi(k) = \bar{\pi}(k) = i$
  end for
end for
```

Determining column counts

When using supernodes we only need the row entries for each supernode rather than each column. Further, we can generate the row indices on the fly in the factorize phase if we wish. It is hence useful if the column counts can be determined in fewer than $O(nz(L))$ operations (i.e. without simulating the factorization as in Algorithm 8). Such an algorithm is given by Gilbert et al. [1994]. Row and column counts are determined without explicitly determining the non-zero structure, using only a precalculated elimination tree and the non-zero structure of A. Once such column counts are known, supernode amalgamation may be performed allowing a much less demanding symbolic factorization to take place.

2.4.3 Factorize

Given the structural non-zero pattern, storage requirements and supernodes of L we can conduct the numerical factorization of A. There are two main schemes for implementing a sparse

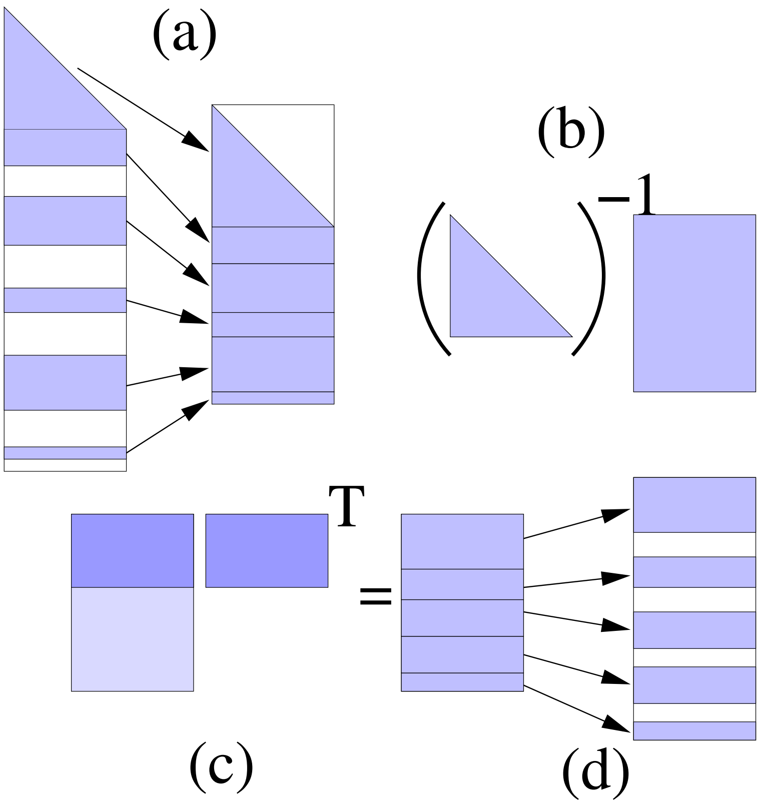


Figure 2.7: The various stages of a supernodal factorization.

factorization: supernodal and multifrontal. The differences relate to how updates between nodes are handled. Supernodal schemes are the direct translation of our dense algorithms to the sparse case, applying updates directly to the factor L in its final storage space. Multifrontal schemes instead pass a temporary partially-summed (square) submatrix up the assembly tree. The partially-summed submatrix contributions from each child are assembled into a single triangular matrix at each node and a partial dense Cholesky factorization is performed. In this section we shall only describe the supernodal variant. A fuller description of the multifrontal method can be found in the book of Duff, Erisman and Reid [1986].

The supernodal algorithm exists in both left- and right-looking flavours. Rothberg and Gupta [1993] claim that the particular casting makes little difference to performance; we shall hence focus only on the right-looking variant.

Consider Figure 2.7. Each supernode consists of one or more consecutive columns with the same non-zero pattern. We store only these non-zeros to achieve the trapezoidal matrix shown in (a). The upper part of the triangle may stored explicitly as zeros to allow the use of more efficient full format BLAS, instead of the slower but more memory efficient packed format BLAS. Once a given supernode is ready to be factorized we perform a standard dense factorization of the diagonal block (for example using the LAPACK routine `_potrf`). We then perform a triangular solve with these factors and the rectangular part of the supernode, shown as (b), for which we may use the level 3 BLAS routine `_trsm`. Alternatively, this factorization and solve may be carefully combined on some machines for a small performance gain.

We then iterate over ancestors of the node in the assembly tree. For each parent we identify the rows of the current supernode corresponding the parent's columns, and then form the outer product of those rows and the part of the supernode below those columns (shown as (c)). The resultant matrix, generated by the level 3 BLAS routine `_gemm`, is placed into a temporary buffer. The rows and columns of this buffer are then matched against elements of the ancestral node and are added to them in a sparse scatter operation. Demmel et al. [1999] recommend panelling these updates so that the temporary buffer matrix does not leave cache.

2.4.4 Solve

Solves are easily performed using the supernodal factors, as shown in Algorithm 10. If multiple right hand sides are given we can even exploit level 3 BLAS routines. Different routines are usually used for forward and backward solves to exploit the data structure. If the data is stored by columns this typically means a right-looking forward solve and a left-looking backward solve.

Algorithm 10 Supernodal solve

```
$\boldsymbol{x} = \boldsymbol{b}$
**for all** Supernodes $s$ in ascending order **do**
      ($S$ refers to the indices within the supernode. $N$ refers to the indices
      on the path between $s$ and the root of the elimination tree. $D$ refers to the diagonal
      block submatrix corresponding to the rows and columns S. $N$ refers to the dense
      matrix storing the entries of the supernode below $D$.)
  Triangular solve on supernode: $D\boldsymbol{x}_S = \boldsymbol{x}_S$ (Use _trsv).
  Modify non-supernode portion of $\boldsymbol{x}$: $\boldsymbol{x}_N = \boldsymbol{x}_N - R\boldsymbol{x}_S$ (Use _gemv).
**end for**
**for all** Supernodes $s$ in descending order **do**
  Modify non-supernode portion of $\boldsymbol{x}$: $\boldsymbol{x}_N = \boldsymbol{x}_N - R^T\boldsymbol{x}_S$ (Use _gemv).
  Triangular solve on supernode: $D^T\boldsymbol{x}_S = \boldsymbol{x}_S$ (Use _trsv).
**end for**
```

2.4.5 Parallel sparse Cholesky

Most parallel implementations of sparse Cholesky in the literature rely on the elimination tree to expose data dependencies that can be exploited. Children of any given node may be factorized independently, and any node of the elimination tree may be factored as soon as its children are ready. This is *tree-level parallelism.*

Often the majority of the work in a sparse factorization resides in the few nodes at the top of the tree, giving tree-level parallelism limited utility. We instead exploit techniques used for parallel dense Cholesky. This is known as *node-level parallelism.* Codes often switch from tree-level to node-level parallelism dynamically.

PARDISO [Schenk and Gärtner 2004; 2006], TAUCS [Irony et al. 2004; Rotkin and Toledo 2004] and PaStiX [Hénon et al. 2002] all exploit a third level of parallelism, which the authors of PARDISO term *pipelining parallelism.* This level of parallelism is obtained by allowing multiple updates from a node's descendants to occur in parallel, though the exact mechanisms involved vary between these codes.

PARDISO performs a left looking factorization (detailed in Schenk et al. [1999]). The external (inter-supernodal) update operations are parallelised. Once a supernode has been factorized, a relevant update task is added to queues associated with each of its ancestral supernodes. When all relevant descendants are completed and the ancestral supernode is scheduled on a thread, these updates are performed in a left-looking manner. The processor will wait on the task queue until all required update tasks have been completed; it will then perform the internal factorization. Large supernodes are broken into multiple smaller supernodes to generate sufficient parallelism near the root of the tree. These smaller supernodes are alternatively known as *panels* by Schenk, however in this thesis we call them *block columns* for reasons that will become apparent in Chapter 7. Further efforts to improve performance in PARDISO have resulted in a two-level factorization. Leaf subtrees are established that are processed on single nodes, eliminating any need to communicate via shared memory with other threads. After these small subtrees are completed the previously described parallel mechanism takes over.

TAUCS performs a multifrontal factorization using a recursive task-based parallel scheme through the programming language cilk. As this is a multifrontal code, expansion of updates into multiple ancestors is not a feasible parallel mechanism. Instead the update between a parent and child is broken into multiple operations corresponding to different target blocks in the parent, and these are executed in parallel. A recursive block data structure is used to ensure cache locality regardless of how fine grained the update operations become.

PaStiX is a supernodal factorization based on task graphs that is designed for distributed memory architectures (though threading is used to exploit SMP nodes). While the description of Hénon et al. [2002] is focused more on distributed memory and scheduling issues, so far as we can determine the following data structure is used. Given the supernode partition of the assembly tree, the matrix is divided into blocks using this partition for both rows and columns. Each block is stored densely, with zero rows as required, but omitting rows that would otherwise be at the start or end of a block. This approach appears to work surprisingly well. The processing of each block column is then either treated as a single task (for smaller block columns), or as a set of tasks, one for each block, and a task DAG is drawn between them. Much attention is paid to minimizing communication and a simulation of the factorization is performed during the analysis phase so that blocks can be assigned to processors in a near optimal fashion. The resulting fixed schedule is then used for the factorize phase. A more in depth explanation of PaStiX is provided in Section 7.6.

Other well known parallel sparse codes which support Cholesky/symmetric indefinite factorizations include: CHOLMOD [Chen et al. 2008], MUMPS [Amestoy et al. 2001; Amestoy et al. 2006], SuperLU_{MT,DIST} [Li and Demmel 2003; Li 2005], and WSMP [Gupta et al. 2001; Gupta et al. 1997]. These all work on similar principles to those already described. A comprehensive summary of sparse codes can be found in [Davis 2006], while an in depth serial comparison of the top symmetric indefinite codes is present by Gould et al. [2007].

2.5 Sparse symmetric indefinite factorization

We can adapt the sparse Cholesky factorization to produce a LDL^T factorization much as we did in the dense case. However, full Bunch-Kaufmann pivoting produces much fill in as it deviates from the fill-reducing ordering. Instead we use a threshold based approach, following Duff and Reid [1983; 1996]. We accept a 1×1 pivot if

$$a_{jj} \geq \mu \max_{i \neq j} a_{ij},$$

limiting the growth of off-diagonal entries to at most $1/\mu$; typically $\mu = 0.01$ or $\mu = 0.1$. If we cannot find a 1×1 pivot then we test for a 2×2 pivot which satisfies

$$\begin{pmatrix} a_{jj} & a_{(j+1)j} \\ a_{(j+1)j} & a_{(j+1)(j+1)} \end{pmatrix}^{-1} \begin{pmatrix} \max_{i \neq j,j+1} a_{ij} \\ \max_{i \neq j,j+1} a_{i(j+1)} \end{pmatrix}^{-1} \leq \begin{pmatrix} \mu^{-1} \\ \mu^{-1} \end{pmatrix}^{-1}.$$

If the pivot cannot be found it is *delayed*, meaning that the column is moved up to the parent supernode where we again attempt to find a pivot. This introduces additional non-zeros into the factors.

Variations on this concept include searching for large off-diagonal entries within the supernode (rather than just using entry $j + 1$), and the use of Bunch-Kaufmann pivoting restricted to pivots within the supernode [Schenk and Gärtner 2006].

Alternately, or in combination, static pivoting may be used. In this case a small perturbation is added to problematic pivots so as to limit the growth of the factors. We hence have a factorization of

$$PAP^T + E = LDL^T$$

where E is diagonal with small (mostly zero) entries. Similar approaches are mentioned multiple times in the literature [Gill and Murray 1974; Li and Demmel 1998; Duff and Pralet 2007]. Arioli et al. [2007] give theoretical and practical results on recovery of accurate solutions from such a perturbed factorization, and suggest that the perturbation should have the value $\sqrt{\epsilon}$ where ϵ is the machine precision.

2.5.1 Scaling

Under either real or static pivoting, encountering poor pivots is undesirable. In the static case they lead to numerical inaccuracies, but when real pivoting is done the delayed pivots introduce additional floating point operations (due to fill-in), additional data movement and other overheads. The use of scaling can reduce or eliminate such delayed pivots.

In the symmetric indefinite case a scaling is given by the diagonal matrix S, and we aim to factorize $SPAP^{-1}S = LDL^T$. A good scaling is one that minimizes the number of delayed pivots; no simple mathematical property has yet been found that achieves this. Many scalings aim for some form of equilibriated row, column or entry norms.

We shall now describe some common scalings. Readers are referred to Chapter 4 for an in depth practical comparison of sparse scalings.

`MC30`

`MC30` is the oldest scaling routine in the HSL subroutine library [HSL 2007] and is described as a symmetric adaption of the method described in the paper of Curtis and Reid [1972].

We scale $A = \{a_{ij}\}$ to the matrix $\hat{A} = \{\hat{a}_{ij}\} = SAS$ where

$$\hat{a}_{ij} = a_{ij} \exp(s_i + s_j).$$

We choose s_i and s_j to minimize the sum of the squares of logarithms of the absolute values of the entries,

$$\begin{aligned} & \min_{\boldsymbol{s}} \sum_{a_{ij} \neq 0} (\log |\hat{a}_{ij}|)^2 \\ = \; & \min_{\boldsymbol{s}} \sum_{a_{ij} \neq 0} (\log |a_{ij}| + s_i + s_j)^2 . \end{aligned}$$

This is achieved by a specialised conjugate gradient algorithm.

`MC64`

`MC64` finds a maximum matching of an unsymmetric matrix such that the largest entries are moved on to the diagonal [Duff and Koster 2001]; this leads to an unsymmetric scaling such that the scaled matrix has all ones on the diagonal and the off-diagonal entries are of modulus less than or equal to one. The approach can be symmetrized by the method of Duff and Pralet [2005], which essentially amounts to initially ignoring the symmetry of the matrix and then averaging the relevant row and column scalings from the unsymmetric permutation.

In recent years, `MC64` has been widely used in conjunction with both direct and iterative methods. It is used in the sparse direct solver SuperLU of Demmel and Li [Demmel et al. 1999], where it is particularly advantageous to put large entries on the diagonal because SuperLU implements a static pivoting strategy that does not allow pivots to be delayed but rather adheres to the data structures established by the analyse phase. Benzi, Haws and Tuma [2001] report on the beneficial effects of scalings to place large entries on the diagonal when computing incomplete factorization preconditioners for use with Krylov subspace methods.

The symmetrized version was developed following the success of `MC64` on unsymmetric systems and symmetrically permutes large entries onto the sub-diagonal (they can not be placed on the diagonal with a symmetric permutation) for inclusion in 2×2 pivots. This permutation can be combined with heuristics for a compressed or constrained ordering to yield pivot sequences that are often more stable than otherwise, but at a cost of additional fill-in [Duff and Pralet 2005].

`MC77`

`MC77` uses an iterative procedure [Ruiz 2001] to attempt to make all row and column norms of the matrix unity for a user-specified geometric norm $\|\cdot\|_p$. We shall consider the infinity and one norms in this paper. The infinity norm is the default within `MC77` due to good convergence properties. It produces a matrix whose rows and columns have maximum entry of exactly one. The one norm produces a matrix whose row and column sums are exactly one (a doubly stochastic matrix) and is, in some sense, an optimal scaling [Bauer 1963].

2.6 Out-of-core working

For some problems the factors L, and potentially the original matrix A, may not fit in the available memory, despite the computation time being feasible. In this case we may store the data we are not currently working with in files. Such an approach is said to be *out of core*.

The multifrontal method in particular is suited to such an approach since the working set is limited to the current frontal matrix and its contributions, which can be easily handled using stacks [Reid and Scott 2008; 2009b]. A virtual memory system, such as that of Reid and Scott [2009a], may be used to provide an easy, efficient yet robust interface to the filesystem.

Performing the factorization out of core does not result in a large performance decrease during factorization (as shown by Reid and Scott). However, the solve phase is considerably slower as the factors must be read from disk twice: once each for the forward and backward substitutions.

2.7 Refinement of direct solver results

While we will not cover pure iterative methods in this work, there is an important application of them relating to direct methods: refinement of a solution to reduce the backwards error. We discuss two such methods here, and note the implications for working in mixed precision.

2.7.1 Iterative refinement

Iterative refinement is a well known scheme that attempts to improve an approximate solution. Often several steps of iterative refinement are required to obtain an acceptable solution. At step k, we aim to reduce the residual,

$$\boldsymbol{r}^{(k)} = \boldsymbol{b} - A\boldsymbol{x}^{(k)}. \tag{2.3}$$

If $\hat{\boldsymbol{x}}$ is the exact solution satisfying $A\hat{\boldsymbol{x}} = \boldsymbol{b}$ we may write

$$\boldsymbol{x}^{(k)} = \hat{\boldsymbol{x}} - \boldsymbol{y}^{(k+1)},$$

and substitute into (2.3),

$$\begin{aligned} \boldsymbol{r}^{(k)} &= \boldsymbol{b} - A\hat{\boldsymbol{x}} + A\boldsymbol{y}^{(k+1)} \\ &= A\boldsymbol{y}^{(k+1)}. \end{aligned}$$

Here we have exploited $\hat{\boldsymbol{x}}$ being an exact solution. We can use our factorization to solve $A\boldsymbol{y}^{(k+1)} = \boldsymbol{r}^{(k)}$ to find the $(k+1)^{\text{st}}$ step update, $\boldsymbol{y}^{(k+1)}$. We then obtain a new $\boldsymbol{x}^{(k+1)}$ by

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \boldsymbol{y}^{(k+1)}.$$

Algorithm 11 Iterative refinement

```
Require: Matrix $A$, factorization of $A$, right-hand side $\boldsymbol{b}$, maximum number of iterations maxitr.
  Solve $A\boldsymbol{x}^{(1)} = \boldsymbol{b}$ (using factors).
  Compute $\boldsymbol{r}^{(1)} = \boldsymbol{b} - A\boldsymbol{x}^{(1)}$.
  Set $k = 1$.
  while residual large and $k <$ maxitr do
    Solve $A\boldsymbol{y}^{(k+1)} = \boldsymbol{r}^{(k)}$ (using factors).
    Set $\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \boldsymbol{y}^{(k+1)}$.
    Compute $\boldsymbol{r}^{(k+1)} = \boldsymbol{b} - A\boldsymbol{x}^{(k+1)}$.
    Check for stagnation, exit if found.
    Set $k = k + 1$.
  end while
  $\boldsymbol{x} = \boldsymbol{x}^{(k)}$
```

This scheme is implemented as Algorithm 11. It is typical to use a scaled residual norm such as

$$\frac{\|\boldsymbol{b} - A\boldsymbol{x}\|}{\|A\|\|\boldsymbol{x}\| + \|\boldsymbol{b}\|}$$

to measure progress. Stagnation can often be detected by requiring a minimum decrease in the residual norm at each stage, typically by at least a factor of two. For some very ill-conditioned matrices it is possible that the solution will grow very quickly while the scaled norm does not: we can detect these cases by monitoring the absolute norm $\|\boldsymbol{r}^{(k)}\|$.

This procedure can be rewritten as a recurrence:

$$\begin{aligned} \boldsymbol{x}^{(k+1)} &= \boldsymbol{x}^{(k)} + L^{-T}L^{-1}(\boldsymbol{b} - A\boldsymbol{x}^{(k)}) \\ &= L^{-T}L^{-1}\boldsymbol{b} + (I - L^{-T}L^{-1}A)\boldsymbol{x}^{(k)}. \end{aligned}$$

Taking the difference between the terms,

$$\begin{aligned} \boldsymbol{d}^{(k)} &= \boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)} \\ &= (I - L^{-T}L^{-1}A)(\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-1)}) \\ &= (I - L^{-T}L^{-1}A)\boldsymbol{d}^{(k-1)} \\ &= (I - L^{-T}L^{-1}A)^k\boldsymbol{d}^{(0)}, \end{aligned}$$

meaning the algorithm is convergent in exact arithmetic if the spectral radius of $(I - L^{-T}L^{-1}A)$ is less than one. Related results can be proved in both fixed precision and mixed precision arithmetic, see for example the book of Higham [2002].

Iterative refinement can be viewed as a standard iterative method preconditioned by a solve using the direct method. It is this view that leads logically to applying other iterative methods, such as in the next section.

2.7.2 FGMRES

The Flexible Generalised Minimum RESidual (FGMRES) method [Saad 1994; 2003] is a Krylov subspace technique for unsymmetric matrices that allows different preconditioners at each iteration. Practical experience [Arioli and Duff 2009; Arioli et al. 2007] shows that this method is better at refining solutions than symmetric methods such as conjugate gradients, or methods with a fixed preconditioner such as standard GMRES. Theoretical and practical results [Arioli and Duff 2009] show that stronger convergence is possible than with iterative refinement, though the iterations are considerably more expensive.

The seeming need for an unsymmetric method that allows for a non-constant preconditioner may be explained through the observation that, in finite arithmetic, solves from a direct method are unsymmetric and depend on the right-hand side.

The basic method is outlined as it applies to our problem in Algorithm 12. The maximum number of iterations, `restart`, differentiates a family of related algorithms; a particular member is referred to as FGMRES(`restart`). The reason for the naming of this parameter is that an outer algorithm is often used that effectively restarts the FGMRES algorithm every `restart` iterations for both numerical and practical reasons. First, while the restart parameter may be as large as desired, the data and number of iterations required grow as $O(\texttt{restart}^2 n)$ and the number of direct method solves grows as $O(\texttt{restart})$. Secondly, the surrogate measure of the residual norm, $\left\| \|\boldsymbol{r}\|\boldsymbol{e}^{(1)} - H^{(j)}\boldsymbol{y}^{(j)} \right\|$, may not detect convergence reliably, especially if the outer measure of convergence is not the 2-norm. Finally, in finite arithmetic, the orthogonality between vectors $\boldsymbol{v}^{(k)}$ will deteriorate as `restart` increases. However, a balance is necessary as a minimum value of `restart` must be achieved to allow optimization in a Krylov space of sufficiently large dimension to achieve fast convergence.

2.7.3 Mixed precision

The theoretical results for iterative refinement and FGMRES have been carefully constructed to have no assumption on the precision of the direct solve. This allows the possibility to perform

Algorithm 12 Basic FGMRES(`restart`) algorithm for refining a solution of $A\boldsymbol{x} = \boldsymbol{b}$

```
Require: $A$, factorization of $A$, right-hand side $\boldsymbol{x}$, restart parameter restart.
  Solve $A\boldsymbol{x} = \boldsymbol{b}$.
  Compute $\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}$.
  Initialise $\boldsymbol{v}^{(1)} = \boldsymbol{r}/\|\boldsymbol{r}\|,\ \ \boldsymbol{y}^{(0)} = \boldsymbol{0},\ \ j = 0$
  while $\big\|\|\boldsymbol{r}\|\boldsymbol{e}_1 - H^{(j)}\boldsymbol{y}^{(j)}\big\| \geq \gamma(\|A\|\|\boldsymbol{x}\| + \|\boldsymbol{b}\|)$ and $j <$restart do
    $j = j + 1$ (Increment iteration counter).
    Solve $A\boldsymbol{z}^{(j)} = \boldsymbol{v}^{(j)}$ and compute $\boldsymbol{w} = A\boldsymbol{z}^{(j)}$
      (Precondition by direct solve).
    Orthogonalize $\boldsymbol{w}$ against $\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(j)}$ to obtain a new $\boldsymbol{w}$. Set $\boldsymbol{v}^{(j+1)} = \boldsymbol{w}/\|\boldsymbol{w}\|$.
    Form $H^{(j)}$, a trapezoidal basis for the Krylov subspace spanned by $\boldsymbol{v}^{(1)}, \ldots, \boldsymbol{v}^{(j)}$
      (Using Arnoldi process, for full details see [Saad 2003]).
    $\boldsymbol{y}^{(j)} = \arg\min_{\boldsymbol{y}} \big\|\|\boldsymbol{r}\|\boldsymbol{e}_1 - H^{(j)}\boldsymbol{y}^{(j)}\big\|$
      (Minimize residual over the Krylov subspace).
  end while
  Set $Z^{(j)} = \big[\boldsymbol{z}^{(1)} \cdots \boldsymbol{z}^{(j)}\big]$.
  Compute $\boldsymbol{x} = \boldsymbol{x} + Z^{(j)}\boldsymbol{y}^{(j)},\ \ \boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}$.
```

the expensive direct method factorization in low precision (fast but inaccurate) and refine the solution in higher precision (slow but obtaining a more accurate result). A result of Skeel [1980] shows that this is possible for iterative refinement, while Arioli and Duff [2009] present the corresponding result for FGMRES. This has been further examined both theoretically and practically by many authors, including [Buttari et al. 2008; Buttari et al. 2007; Demmel et al. 2006; Demmel et al. 2009]. Chapter 5 describes the development of such a practical mixed precision code for sparse symmetric systems.

2.8 HSL solvers

There are two major sparse symmetric indefinite solvers in the HSL software library [HSL 2007] that we shall refer to. These are `MA57` and `HSL_MA77`, and are now briefly described.

2.8.1 `MA57`

`MA57` is a multifrontal solver is initially described by Duff [2004]. It has evolved to use the scaling and pivoting strategies of Duff and Pralet [2005; 2007].

In addition to the normal facilities (analysis, factorize and solve phases), `MA57` also offers options for the solution of multiple right-hand sides, computation of partial solutions, error analysis, matrix modification, and the facility to stop and restart. It also offers the ability to set control parameters affecting any part of the solution process, though it is normal to use the defaults which have been tuned by the authors.

Further, built-in scaling is available through a symmetrized version of the well-known package `MC64` [Duff and Koster 2001], and `MA57` is also capable of determining a preordering using variants of the minimum degree algorithm or multilevel nested dissection algorithms. It features an advanced heuristic [Duff and Scott 2005] to choose between these ordering algorithms.

2.8.2 `HSL_MA77`

`HSL_MA77` [Reid and Scott 2008; 2009b] is also a multifrontal solver that is designed to solve positive definite and indefinite sparse symmetric systems. A different inner kernel is used in each case to achieve the best performance. The fundamental difference between `MA57` and `HSL_MA77` is that the latter is an out-of-core solver capable of holding all the data out of core, enabling the solution of much larger problems. Further, care has been taken to allow the addressing of fronts with 64-bit integers. This is essential as some problems require the factorization of dense matrices containing this many elements.

It exploits the virtual memory system of Reid and Scott [2009a] to minimize overheads due to the out-of-core approach while remaining robust. The ability to work in core is also available, though some overhead from the out-of-core design remains. Despite this, on problems that `MA57` is able to solve, the performance of `HSL_MA77` is favourable in the factorization phase, though the solve phase can be comparatively slow.

While full control over the factorization is still available, the large range of support routines that come with `MA57` are not present. The user is expected to supply their own ordering and scaling. This reflects the limited availability of subroutines capable of performing these routines in an out-of-core fashion for large problems.

Chapter 3

Interior point methods

This chapter shows the use of symmetric matrix factorization in an important application: interior point methods. We briefly develop the theory of primal-dual interior point methods (IPMs) and address crucial numerical elements as they apply to matrix factorizations. Much of the theory presented here is covered in greater detail by the book *Primal-Dual Interior Point Methods* [Wright 1997].

We consider the general optimization problem

$$\begin{array}{rl} \max & f(\boldsymbol{x}) \\ \text{subject to} & \boldsymbol{g}(\boldsymbol{x}) = \boldsymbol{0} \\ & \boldsymbol{x} \geq \boldsymbol{0}, \end{array}$$

where $\boldsymbol{x} \in \mathbb{R}^n, f : \mathbb{R}^n \to \mathbb{R}$, and $g : \mathbb{R}^n \to \mathbb{R}^m$. When $f(\boldsymbol{x})$ and $g(\boldsymbol{x})$ are linear functions of $\boldsymbol{x}$, we have a *linear programming problem* (LP) that has the general form

$$\begin{array}{rl} \max & \boldsymbol{c}^T\boldsymbol{x} \\ \text{subject to} & A\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \geq 0, \end{array} \tag{3.1}$$

where $\boldsymbol{c} \in \mathbb{R}^n, \boldsymbol{b} \in \mathbb{R}^m$ and A is a $m \times n$ matrix. In this chapter we shall develop theory for only the linear case, however quadratic and general nonlinear programming involves the solution of similar systems, as described at the end of the chapter.

3.1 Karush-Kahn-Tucker conditions

The LP (3.1) is referred to as the *primal* form. We can define the associated *Lagrangian* function

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}) = \boldsymbol{c}^T\boldsymbol{x} - \boldsymbol{y}^T(A\boldsymbol{x} - \boldsymbol{b}) - \boldsymbol{z}^T(\boldsymbol{x} - \boldsymbol{0}),$$

with $\boldsymbol{y} \in \mathbb{R}^m$ and $\boldsymbol{z} \in \mathbb{R}^n$. We observe that the solution of (3.1) is equivalent to the unconstrained optimization

$$\max_{\boldsymbol{x} \geq 0} \min_{\substack{\boldsymbol{y} \\ \boldsymbol{z} \geq 0}} \mathcal{L}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}). \tag{3.2}$$

Finding the stationary point yields first order conditions for optimality

$$\left.\begin{array}{rcll} A^T\boldsymbol{y} + \boldsymbol{z} & = & \boldsymbol{c} & \text{(dual feasibility)} \\ A\boldsymbol{x} & = & \boldsymbol{b} & \text{(primal feasibility)} \\ x_i z_i & = & 0 \quad \forall\, i & \text{(complementarity)} \\ \boldsymbol{x}, \boldsymbol{z} & \geq & \boldsymbol{0}. & \text{(positivity)} \end{array}\right. \tag{3.3}$$

These are known as the *Karush-Kahn-Tucker*, or KKT, conditions.

Reversing the order of optimizations in (3.2) yields the *dual problem*, with general form

$$\begin{array}{rl} \min & \boldsymbol{b}^T\boldsymbol{y} \\ \text{subject to} & A^T\boldsymbol{y}+\boldsymbol{z}=\boldsymbol{c} \\ & \boldsymbol{z}\geq\boldsymbol{0}. \end{array}$$

In the linear case the primal and dual problems share the same optimal set.

3.2 Solution of the KKT systems

Interior point methods are essentially a variant of the Newton iteration. We solve the first three equations of (3.3) with all iterates satisfying the positivity condition. As the exact complementarity condition $x_iz_i = 0\ \forall\ i$ makes this problem very difficult to solve with the Newton iteration, we instead use a relaxed form, $x_iz_i = \sigma\mu\ \forall\ i$. The scalar parameters σ and μ will be explained later, but are chosen such that the products x_iz_i are driven to zero as the iteration progresses. This leads to the following equation for the *Newton direction*:

$$\begin{pmatrix} & A^T & I \\ A & & \\ Z & & X \end{pmatrix}\begin{pmatrix} \Delta\boldsymbol{x} \\ \Delta\boldsymbol{y} \\ \Delta\boldsymbol{z} \end{pmatrix} = \begin{pmatrix} \boldsymbol{c}-A^T\boldsymbol{y}-\boldsymbol{z} \\ \boldsymbol{b}-Ax \\ \sigma\mu\boldsymbol{e}-ZX\boldsymbol{e} \end{pmatrix}. \tag{3.4}$$

We use the convention that X is the diagonal matrix with the values of the vector $\boldsymbol{x}$ on the diagonal (likewise for Z and $\boldsymbol{z}$); $\boldsymbol{e}$ is the vector of all ones.

The basic primal-dual interior point method is shown as Algorithm 13. This form does not have any guarantees of convergence, but presents the main ideas. We find a descent direction that reduces the infeasibilities, and proceed along it such that $\boldsymbol{x}$ and $\boldsymbol{z}$ remain positive. The scalar μ is known as the complementarity gap, and is the average value of x_iz_i at each step. The complementarity gap is driven towards zero by the centring parameter σ, that aims to reduce μ at each iteration. In practice, the algorithm as stated will take many iterations to reach optimality (if it does so at all). Taking the full step to the boundary along each descent direction is likely to limit the length of the future steps. Additionally, getting too close to the boundary will result in an ill-conditioned matrix for (3.4) with accompanying errors.

Algorithm 13 The basic primal-dual interior point method

```
Require: Starting point $(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) : \boldsymbol{x},\boldsymbol{z}\geq\boldsymbol{0}$, centring parameter $\sigma$, convergence tolerance $\epsilon$.
  Calculate $\mu = \frac{\boldsymbol{x}^T\boldsymbol{z}}{n}$.
  while $(\|A^T\boldsymbol{y}+\boldsymbol{z}-\boldsymbol{c}\|, \|A\boldsymbol{x}-\boldsymbol{b}\|, \mu) \geq \epsilon$ do
    Determine the Newton direction by solving (3.4).
    Find step length $\alpha = \arg\max\{\alpha\in[0,1] : (\boldsymbol{x},\boldsymbol{z})+\alpha(\Delta\boldsymbol{x},\Delta\boldsymbol{z})\geq 0\}$.
    Take step $(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) \leftarrow (\boldsymbol{x},\boldsymbol{y},\boldsymbol{z})+\alpha(\Delta\boldsymbol{x},\Delta\boldsymbol{y},\Delta\boldsymbol{z})$.
    Calculate $\mu = \frac{\boldsymbol{x}^T\boldsymbol{z}}{n}$.
  end while
```

We define the *central path* for a problem as the unique curve parametrised by μ that satisfies the perturbed KKT system

$$\begin{aligned} A^T\boldsymbol{y}+\boldsymbol{z} &= \boldsymbol{c} \\ A\boldsymbol{x} &= \boldsymbol{b} \\ x_iz_i &= \mu \qquad \forall i=1,\ldots,n. \end{aligned}$$

This is an idealised trajectory where the complementarity products x_iz_i are always equal. A method where iterates remain within a neighbourhood of this path is said to be *path-following*.

We reproduce the path-following method given as Algorithm IPF by Wright [1997] as our

Algorithm 14. The particular neighbourhood used in this case is

$$\mathcal{N}_{-\infty}(\gamma,\beta) = \Bigg\{ (\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) : \frac{\|(\boldsymbol{b}-A\boldsymbol{x},\boldsymbol{c}-A^T\boldsymbol{y}-\boldsymbol{z})\|_\infty}{\mu} \leq \beta\frac{\|(\boldsymbol{b}^{(0)}-A\boldsymbol{x}^{(0)},\boldsymbol{c}-A^T\boldsymbol{y}^{(0)}-\boldsymbol{z}^{(0)})\|_\infty}{\mu^{(0)}}, \\ (\boldsymbol{x},\boldsymbol{z}) > \boldsymbol{0}, \quad \boldsymbol{x}_i\boldsymbol{z}_i \geq \gamma\mu \;\forall i = 1,\ldots,n \Bigg\}.$$

The neighbourhood is large when it is far from the optimum, but gets narrower as it approaches the optimal set. This is shown for a simple problem in Figure 3.1, with the shaded neighbourhood getting much smaller as the path converges to the optimum. The other condition that the neighbourhood imposes is that the primal and dual infeasibilities decrease at the same rate as, or faster than, the complementarity gap μ. This final condition is useful to ensure that reduction of all residuals in the solution of the KKT system occurs at the same rate.

Algorithm 14 Algorithm IPF from the book Primal-Dual Interior-Point Methods: A path following infeasible primal-dual interior point method

```
Require: Neighbourhood parameters $\gamma \in (0,1)$ and $\beta \geq 1$, minimum and maximum centring parameters $\sigma_{\min}$ and $\sigma_{\max}$, starting point $(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) : \boldsymbol{x},\boldsymbol{z} \geq \boldsymbol{0}$.
  Calculate $\mu = \frac{\boldsymbol{x}^T\boldsymbol{z}}{n}$.
  loop
    Choose $\sigma \in [\sigma_{\min},\sigma_{\max}]$ and solve system (3.4) to find the Newton direction.
    Determine the step length $\alpha$ as the largest value in the interval [0,1] that satisfies the conditions:
        $(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) + \alpha(\Delta\boldsymbol{x},\Delta\boldsymbol{y},\Delta\boldsymbol{z}) \in \mathcal{N}_{-\infty}(\gamma,\beta)$
    and
        $\frac{(\boldsymbol{x}+\alpha\Delta\boldsymbol{x})^T(\boldsymbol{z}+\alpha\Delta\boldsymbol{z})}{n} \leq (1-0.01\alpha)\mu.$
    Take step $(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) \leftarrow (\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) + \alpha(\Delta\boldsymbol{x},\Delta\boldsymbol{y},\Delta\boldsymbol{z})$.
    Calculate $\mu = \frac{\boldsymbol{x}^T\boldsymbol{z}}{n}$.
  end loop
```

A detailed proof of the convergence of Algorithm 14 is given in Chapter 6 of Wright [1997]. The main thrust of the proof is to show that at each stage the sufficient decrease (or Armijo condition),

$$\frac{(\boldsymbol{x}+\alpha\Delta\boldsymbol{x})^T(\boldsymbol{z}+\alpha\Delta\boldsymbol{z})}{n} \leq (1-0.01\alpha)\mu,$$

is satisfied. This follows largely from the final block of equations in (3.4) and the $\boldsymbol{x}_i\boldsymbol{z}_i \geq \gamma\mu$ condition from the neighbourhood. If μ satisfies this sufficient decrease condition, it must converge to 0; as the iterate is within the neighbourhood $\mathcal{N}_{-\infty}(\beta,\gamma)$, the primal and dual infeasibilities must also be converging to 0 at least as fast. Hence we may cease iterations once μ falls below our desired tolerance and we will be near the optimum. If an exact solution is required then we will need to use some form of simplex-like method to find a solution to the KKT system.

3.3 Higher order methods

The consideration of IPMs as modified Newton methods logically leads to the consideration of whether higher order non-linear equation solution methods can be adapted to provide yet better convergence. This has been explored by considering IPMs as trajectory-following solvers. A trajectory from the current iterate $(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z})$ to the solution set is defined such that the descent direction given by solution of (3.4) represents a first order Taylor approximation to the trajectory. Second order or higher approximations lead to the methods that will be described in the next section.

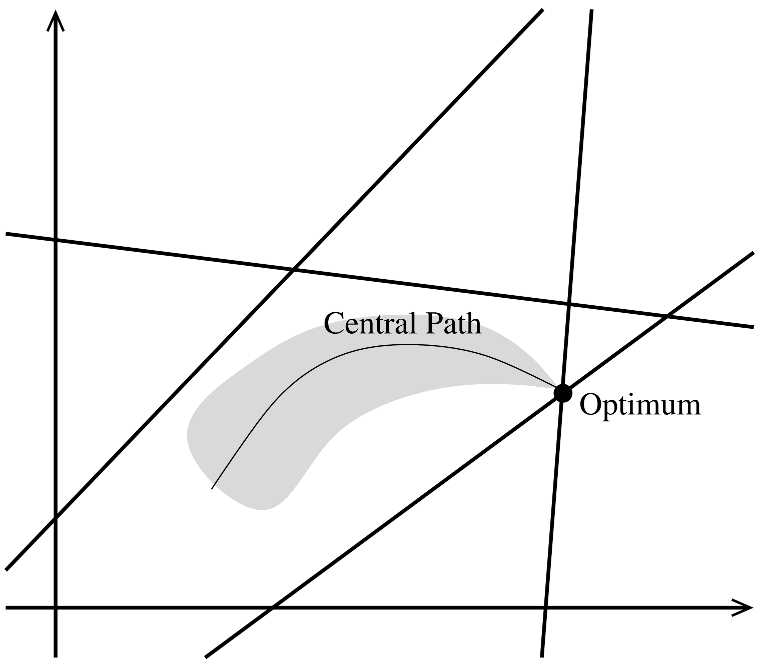


Figure 3.1: Showing the constraints of a simple LP problem, with the central path and neighbourhood marked.

3.3.1 Mehrotra's predictor-corrector algorithm

Mehrotra [1992] tied together several existing threads of research to produce his well known predictor-corrector algorithm, shown here as Algorithm 15. We essentially take an uncentred Newton step, referred to as the *predictor* step or *affine scaling direction*, followed by a *corrector* or *centring* step. The amount of centring is chosen by an adaptive heuristic such that strong centring is performed if a large reduction in the complementarity gap is offered by the predictor step. Another key feature of this method is that only a single matrix factorization is required per iteration, as the same matrix is involved in both the predictor and corrector equations.

3.3.2 Higher order correctors

The central path we are aiming to follow is described by Vavasis and Ye [1996] as having very tight turns that are not well described by a low order truncation of the Taylor series. This suggests higher than second order correction in a strict trajectory following sense is likely to be fruitless. Instead, Gondzio [Gondzio 1995; 1996; Colombo and Gondzio 2008] suggests using higher order corrections to increase the step length α; this is successfully demonstrated in his code HOPDM. As for the predictor-corrector algorithm, it requires multiple solves with the same matrix factorization.

3.4 Practical implementation

So far we have presented the theoretical side of interior point methods. However, the success or failure of an algorithm ultimately relies on practical performance. To this end a number of important tricks are required. Those relating to the solution of the linear systems are relevant to this thesis. These are addressed in the following sections.

Algorithm 15 The Mehrotra predictor-corrector algorithm

```
Require: $(\boldsymbol{x}^{(0)}, \boldsymbol{y}^{(0)}, \boldsymbol{z}^{(0)}) : \boldsymbol{x}^{(0)}, \boldsymbol{z}^{(0)} \geq \boldsymbol{0}$
  loop
    Solve
      $$\begin{pmatrix} & A^T & I \\ A & & \\ Z & & X \end{pmatrix} \begin{pmatrix} \Delta\boldsymbol{x}^{\text{aff}} \\ \Delta\boldsymbol{y}^{\text{aff}} \\ \Delta\boldsymbol{z}^{\text{aff}} \end{pmatrix} = \begin{pmatrix} \boldsymbol{c} - A^T\boldsymbol{y} - \boldsymbol{z} \\ \boldsymbol{b} - A\boldsymbol{x} \\ -XZe \end{pmatrix}$$
    to find the predictor direction $(\Delta\boldsymbol{x}^{\text{aff}}, \Delta\boldsymbol{y}^{\text{aff}}, \Delta\boldsymbol{z}^{\text{aff}})$.
    Determine $\alpha^{\text{aff}} = \arg\max\left\{\alpha \in [0,1] : (\boldsymbol{x}, \boldsymbol{z}) + \alpha(\Delta\boldsymbol{x}^{\text{aff}}, \Delta\boldsymbol{z}^{\text{aff}}) \geq 0\right\}$.
    Calculate $\mu^{\text{aff}} = \frac{(\boldsymbol{x} + \alpha\boldsymbol{x}^{\text{aff}})^T(\boldsymbol{z} + \alpha\boldsymbol{z}^{\text{aff}})}{n}$.
    Set $\sigma = (\mu^{\text{aff}}/\mu)^3$.
    Solve
      $$\begin{pmatrix} & A^T & I \\ A & & \\ Z & & X \end{pmatrix} \begin{pmatrix} \Delta\boldsymbol{x}^{\text{cor}} \\ \Delta\boldsymbol{y}^{\text{cor}} \\ \Delta\boldsymbol{z}^{\text{cor}} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \sigma\mu\boldsymbol{e} - \Delta X^{\text{aff}}\Delta Z^{\text{aff}}\boldsymbol{e} \end{pmatrix}$$
    to find the corrector direction $(\Delta\boldsymbol{x}^{\text{cor}}, \Delta\boldsymbol{y}^{\text{cor}}, \Delta\boldsymbol{z}^{\text{cor}})$.
    Define the full search direction $(\Delta\boldsymbol{x}, \Delta\boldsymbol{y}, \Delta\boldsymbol{z}) = (\Delta\boldsymbol{x}^{\text{aff}}, \Delta\boldsymbol{y}^{\text{aff}}, \Delta\boldsymbol{z}^{\text{aff}}) + (\Delta\boldsymbol{x}^{\text{cor}}, \Delta\boldsymbol{y}^{\text{cor}}, \Delta\boldsymbol{z}^{\text{cor}})$.
    Determine step to boundary $\alpha_{\max} = \arg\max\{\alpha \geq 0 : (\boldsymbol{x}, \boldsymbol{z}) + \alpha(\Delta\boldsymbol{x}, \Delta\boldsymbol{z}) \geq 0\}$.
    Set step length $\alpha = \min(0.99\alpha_{\max}, 1)$.
    Take step $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}) \leftarrow (\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}) + \alpha(\Delta\boldsymbol{x}, \Delta\boldsymbol{y}, \Delta\boldsymbol{y})$.
  end loop
```

3.4.1 Augmented and normal equations

The system (3.4) can rearranged to the following forms through substitution:

$$\begin{pmatrix} -X^{-1}Z & A^T \\ A & \end{pmatrix} \begin{pmatrix} \Delta\boldsymbol{x} \\ \Delta\boldsymbol{y} \end{pmatrix} = \begin{pmatrix} \boldsymbol{c} - A^T\boldsymbol{y} - \boldsymbol{z} - X^{-1}(\sigma\mu\boldsymbol{e} - XZ\boldsymbol{e}) \\ \boldsymbol{b} - A\boldsymbol{x} \end{pmatrix}, \tag{3.5}$$

$$AXZ^{-1}A^T\Delta\boldsymbol{y} = XZ^{-1}(\boldsymbol{c} - A^T\boldsymbol{y} - \boldsymbol{z}) - Z(\sigma\mu\boldsymbol{e} - XZ\boldsymbol{e}) + A(\boldsymbol{b} - A\boldsymbol{x}).$$

Both new forms are symmetric; the latter is also positive definite. The first is known as the *augmented system*, while the second is the *normal equations* form. We are not aware of any disadvantages in solving the augmented system in preference to the original system (3.4): the growth in the matrix factors is unavoidable whether we restrict ourselves to eliminating $\Delta\boldsymbol{z}$ first or not. The reduction from the augmented system to the normal equations corresponds to forcing the pivot order for the first n pivots to be $1, \ldots, n$. Numerical experience shows that this can result in a much poorer factorization than if full pivoting is allowed due to the extremal nature of some entries in XZ^{-1}. However, the positive definite nature of the normal equations allows the more efficient Cholesky factorization to be used rather than a symmetric indefinite factorization. Fourer and Mehrotra [1993] report that the augmented system approach was typically slower with their solver than the normal equations approach on all but very dense problems (where the cost of forming $AXZ^{-1}A^T$ dominates for normal equations), but is numerically better. While technology has moved on since these experiments, it would be logical to assume that similar results still hold as pivoting still requires additional scanning of the matrix and eliminates some parallelism inherent in the Cholesky algorithm. For some problem structures, the freedom of ordering for sparsity allowed by the augmented system is essential to avoid the factors becoming dense.

3.4.2 Numerical errors

As the interior point iteration approaches an optimum point it is expected that some entries of x_i/z_i will approach zero and others infinity. This would be expected to cause serious issues in the reliability of results. However, due to the nature of the direction being calculated (i.e. the nature of the right hand side) the solution can be shown to be sufficiently accurate to achieve convergence [Wright 1994]. Indeed, Wright [1999] demonstrates that for the normal equation the important result is that $\tilde{L}^T z - \tilde{L}^T \tilde{z}$ is small, where $\tilde{L}$ is the Cholesky factorization of a matrix near $AXZ^{-1}A^T$, $\tilde{z}$ denotes the solution with this factor and z is the exact solution. Wright goes on to show that a modified Cholesky factorization where small pivots are ignored produces a good descent direction.

3.5 Nonlinear optimization

Interior Point Methods can be employed in the solution of nonlinear problems, the main difference being the appearance of a Hessian term in the top left of system (3.4) to require the factorization of an augmented system matrix such as

$$\begin{pmatrix} H - X^{-1}Z & A^T \\ A & \end{pmatrix}.$$

This can clearly be tackled through similar algebra to the linear case, though the normal equations are not so easily available unless the Hessian is diagonal or otherwise easy to invert.

Part II

Improved algorithms

Improved algorithms: introduction

This part of the thesis describes novel algorithms and original work aimed at improving the solution of $A\boldsymbol{x} = \boldsymbol{b}$ through direct methods. The key resource on modern computers is memory bandwidth, but parallelism must often be exploited in order to become bound upon this resource. Algorithms are explored that seek to minimize the memory bandwidth used while aiming for strong scaling on modern multicore chips.

Chapter 4 attempts to reduce the number of delayed pivots generated during a factorization through scaling the rows and columns of a matrix. A large number of delayed pivots can lead to a substantial increase in the number of non-zeros that need to be stored, increasing the memory bandwidth required in the factorization. In addition the number of floating point operations is increased and the computational analysis performed in the analyse phase significantly deviated from. This can potentially lead to load balance problems when running in parallel.

Chapter 5 attempts to reduce the memory bandwidth used by performing the matrix factorization in single precision before using an iterative method to recover full double precision accuracy. A practical set of heuristics is developed to achieve this aim for sufficiently large matrices. These are implemented in a new library quality code, `HSL_MA79`.

Chapter 6 addresses the problem of attempting to improve scaling of dense Cholesky factorization on multicore chips. Existing task graph based schemes are explored and modest performance gains through the use of critical path based scheduling are demonstrated. This leads to the extension of such task based methods to the sparse case in Chapter 7. The resultant code is shown to be highly competitive with other shared memory solvers, especially on large problems. Attention is paid to minimizing memory traffic through modifications to the analysis phase and an improved cache-aware scheduling policy.

Finally Chapter 8 draws these separate threads together and explores their performance in an interior point context, a common application of direct methods. A mixed bag of results is presented, with some techniques showing promise, and current deficiencies of other approaches identified.

Chapter 4

Which scaling?

In this chapter we address the issue of matrix scalings. This description is based on our work in [Hogg and Scott 2008], that is a technical report the author produced in collaboration with Jennifer Scott while working at Rutherford Appleton Laboratory. The author wrote the test harness and performed most of the experiments and data analysis, while Scott provided guidance on background material, an in depth knowledge of sparse symmetric indefinite factorizations and helped with the data analysis and presentation. Useful discussions relating to this work were also held with John Reid, Iain Duff and Mario Arioli.

Our major scientific contribution is to carry out a systematic experimental study of several scalings applied to a large number of practical problems, including some highly challenging systems, and to form an opinion as to what work well.

A copy of the technical report is reproduced with kind permission of the copyright holders at the rear of this thesis.

4.1 Why scale?

As noted in Section 2.5.1, scaling can drastically affect the factorization time for symmetric indefinite systems. In this section we extend the work of Pralet [2004], presenting results of experiments performed to evaluate different scalings available within the HSL library [HSL 2007]. Our primary aim is to reduce the time taken for the factorization subject to sufficient accuracy being achieved. We demonstrate that this aim is highly correlated with reductions in the number of delayed pivots encountered (and hence how closely we stick to the predictions of the analyse phase).

However, for some problems time is not the primary concern. Accuracy and the ability to find a solution are.

4.2 The scalings

We will consider the following scalings:

No Scaling Many problems can be solved without the use of scaling and, as our numerical experiments will show, this often results in smaller residuals (before the application of refinement) than if a scaling were used. Further, without the computational overhead of scaling, total solution times can be faster.

`MC30` See description in Section 2.5.1.

`MC64` See description in Section 2.5.1.

`MC77` See description in Section 2.5.1. We will use `MC77` in the infinity and one norms, denoted by $\texttt{MC77}_\infty$ and $\texttt{MC77}_1$ respectively.

Hybrid scalings We run multiple scalings upon the matrix, one after another. If different scalings aim for different properties it may be that the combined scaling is better than either individually.

This choice represents a restriction to those options available within the HSL library, and hence does not consider other alternatives such as those others described in Section 2.5.1.

4.3 Methodology

In addition to the above scaling packages, we will use the following HSL codes, as shown in Algorithm 16.

`MA57` Multifrontal algorithm sparse symmetric solver. We use the default settings with the following exceptions: we force the MeTiS [Karypis and Kumar 1999] ordering and disable the internal call to the scaling routine `MC64`. All matrices are treated as indefinite (that is, we allow numerical pivoting with a threshold parameter of $\mu = 0.01$).

`MA60` Reverse communication code implementing iterative refinement. We use the default maximum iteration limit of 16. The termination condition requires the error estimate to be less than machine precision.

`MI15` Reverse communication code implementing FGMRES. We use the `MA57` factorization and solve as a right-preconditioner, as per Arioli et al. [2007]). The maximum iteration limit is 20, with a restart after 10 iterations (these parameters were chosen following numerical experimentation). We use default termination conditions, but additionally allow termination when the absolute value of the residual falls below 10^{-16}.

Algorithm 16 Algorithm for scaling test harness

```
Set $\boldsymbol{x} = (1, \dots, 1)$. Calculate $\boldsymbol{b} = A\boldsymbol{x}$.
Calculate scaling matrix $S$ using chosen algorithm.
Form $\hat{A} = SAS$.
Factorize $\hat{A}$ using MA57.
Perform a solve using MA57. Record the residual.
Use MA60 iterative refinement with $A$. Record the residual.
Use MI15 FGMRES refinement with $A$. Record the residual.
```

We work with a test set taken from the University of Florida Sparse Matrix Collection [Davis 2007]. All real symmetric problems of dimension less than 100,000 were chosen, however any problems where the predicted size of factors failed to fit in the memory of the test machine were removed. This gives us 367 problems, of which 158 are positive semi-definite and 21 are singular; we shall refer to this collection as ***Test Set 1*** (further details are given in Appendix A).

All the tests were conducted on `curtis` (recall Table 1.4) using the `g95` compiler with option `-O2` and the Goto BLAS [Goto and van de Geijn 2008]. All denormals were flushed to zero, and all computations were performed in double precision. Reported times are CPU timings acquired using the Fortran `system_clock()` intrinsic, and are stated in seconds.

For each run the following information was recorded.

Delayed pivots We use the number of delayed pivots reported by `MA57` as a predictor of speed and, to some extent, numerical stability. Given an initial pivot order, the time and the memory required to factorize and then solve the linear system will depend on the number of delayed pivots. A large number of delayed pivots can also, in some cases, be indicative of a numerically difficult factorization that would be unstable without pivoting.

Scaling time Assuming the required accuracy is attained with and without scaling, a scaling that results in a faster factorization need not be advantageous if the combined time of scaling, factorizing and solving exceeds the unscaled solution time. We are also interested in the relative speeds of the different scaling algorithms.

Factorization time This is the time taken for the numerical factorization phase. For a given pivot order and threshold parameter u, this will depend on the scaling used.

Total time This is total time taken for the scaling followed by the analyse, factorize and solve phases of `MA57` followed by refinement.

MA57 residual This is the scaled residual shown below evaluated using the original unscaled matrix directly after a single solve with `MA57`.

$$\beta = \frac{\|\boldsymbol{r}\|_\infty}{\|A\|_\infty \|\boldsymbol{x}\|_\infty + \|\boldsymbol{b}\|_\infty}, \tag{4.1}$$

where $\boldsymbol{r}$ is the residual given by $\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}$. The infinity norm is given by $\|\boldsymbol{x}\|_\infty = \max_i |x_i|$ and has an induced matrix norm $\|A\|_\infty = \max_i \sum_j |a_{ij}|$.

Iterative refinement residual This is the residual after `MA60` (iterative refinement).

FGMRES residual This is the residual after `MI15` (FGMRES(10)).

4.3.1 Performance profiles

Due to the large size of our test set, we present most of our results using performance profiles, as described in the paper of Dolan and Moré [2002].

If we have a set $\mathcal{S}$ of scalings, and a set $\mathcal{T}$ of problems, then we record a statistic $s_{ij} \geq 0, i \in \mathcal{S}, j \in \mathcal{T}$ on each run. It is assumed that the smaller this statistic the better the solver is considered to be. For example, s_{ij} might be the time for problem j using scaling i. We wish to compare each statistic with the best value for any scaling on that problem.

Let $\hat{s}_j = \min_{j\in\mathcal{T}}\{s_{ij}; i \in \mathcal{S}\}$. Then for $\alpha \geq 1$ and each $i \in \mathcal{S}$ we define the indicator function

$$k(s_{ij}, \hat{s}_j, \alpha) = \begin{cases} 1 & \text{if } s_{ij} \leq \alpha \hat{s}_j \\ 0 & \text{otherwise.} \end{cases}$$

The performance profile of solver i is then given by the function

$$p_i(\alpha) = \frac{\sum_{j\in\mathcal{T}} k(s_{ij}, \hat{s}_j, \alpha)}{|\mathcal{T}|}, \quad \alpha \geq 1. \tag{4.2}$$

As already noted, in this study, the statistics used are timings, the number of delayed pivots and residual sizes. The range of α illustrated is chosen in each case to highlight the dominant trends in the data.

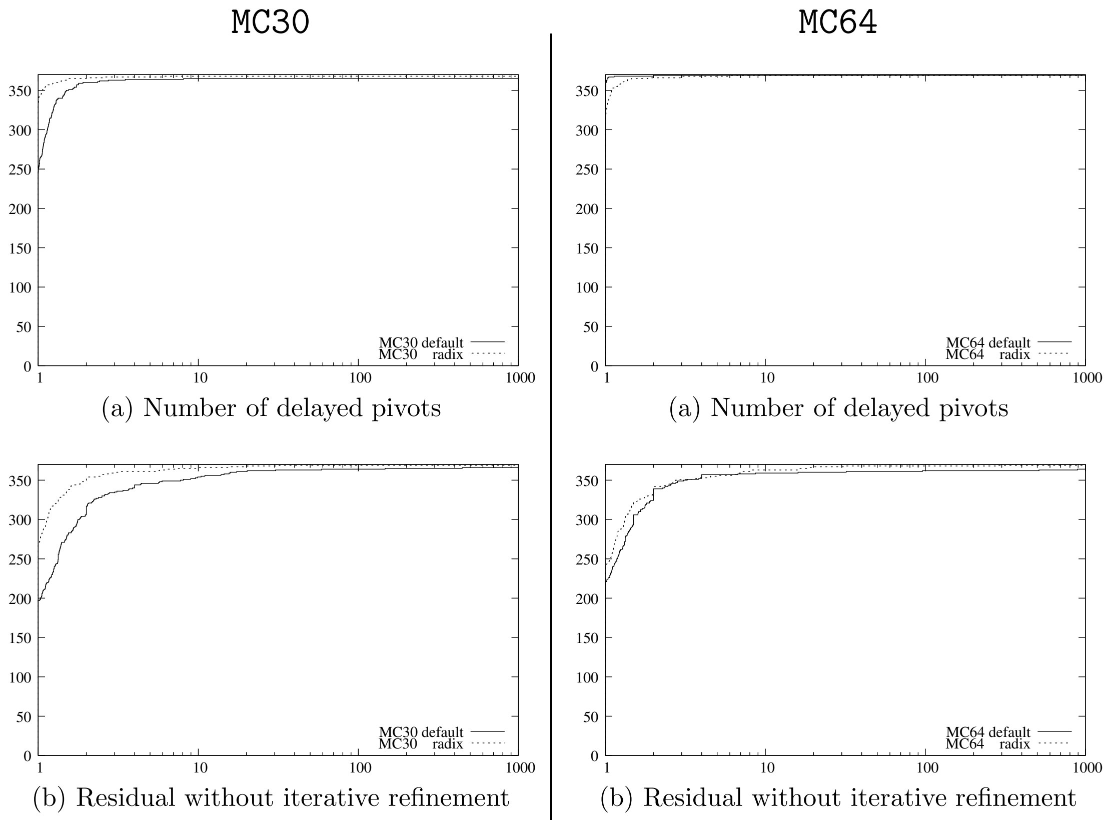
4.4 Numerical experiments

4.4.1 Scaling by the radix

It was common practice in the past [Curtis and Reid 1972] to scale by a power of the radix (on most modern machines, a power of two). When scaling a floating point value by a power of the radix we merely add the power to the floating point value's exponent, leaving the mantissa unchanged. This increased both the accuracy and the speed on older machines.

Modern processors can apply a scaling by any floating point value in the same number of cycles as scaling by a power of the radix, so the speed advantage no long applies. Further, some machines have a radix as large as sixteen (though this is not common), which make scaling by the radix a rather blunt instrument compared to pivot tolerances of `u` = 0.1. As a result radix-based scaling has fallen out of favour.

In Figure 4.1, we report on using `MC30`- and `MC64`-based radix and non-radix scalings. These results show that the radix scaling reduces the residual from the `MA57` solver in more cases than it does not, although in most cases the improvement is less than one order of magnitude. Though not shown here, these advantages were almost eliminated after iterative refinement was applied.



(a) Number of delayed pivots (a) Number of delayed pivots

(b) Residual without iterative refinement (b) Residual without iterative refinement

Figure 4.1: Performance profiles for radix and non-radix (default) scalings.

The number of delayed pivots shows an interesting property we have yet to explain. We would expect the non-radix scaling to produce fewer delayed pivots as it has improved the numerics of the problem; this is indeed the case with MC64, however with MC30 we see that the converse is true.

For the remainder of the experiments in this chapter, we use a non-radix based scaling.

4.4.2 Use of unscaled matrix for refinement

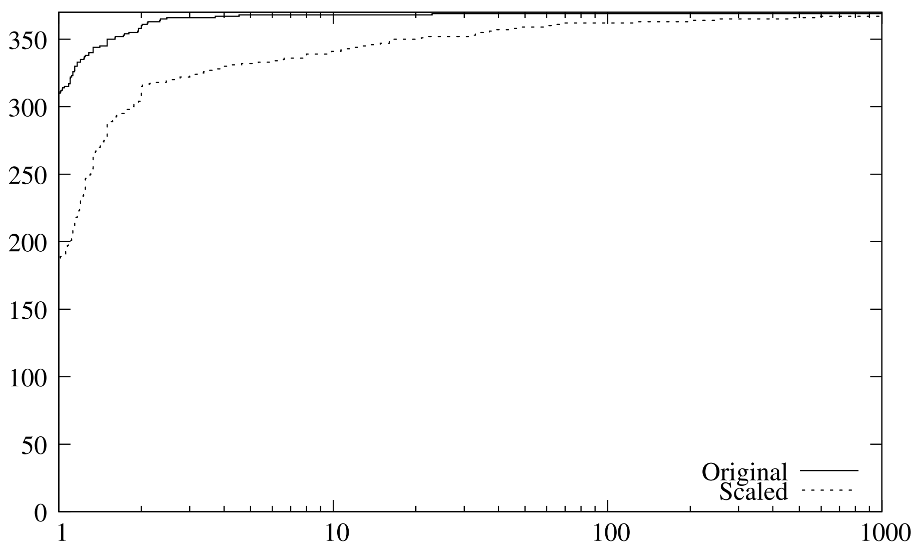
It is worth noting that the unscaled matrix A should be retained for any iterative method used to refine the solution computed by the direct solver. Figure 4.2 demonstrates that using the scaled matrix will generally result in a larger residual (as measured with the original matrix). This is easily explained because we are then, in effect, solving a perturbed system.

4.4.3 Comparison of hybrid scalings

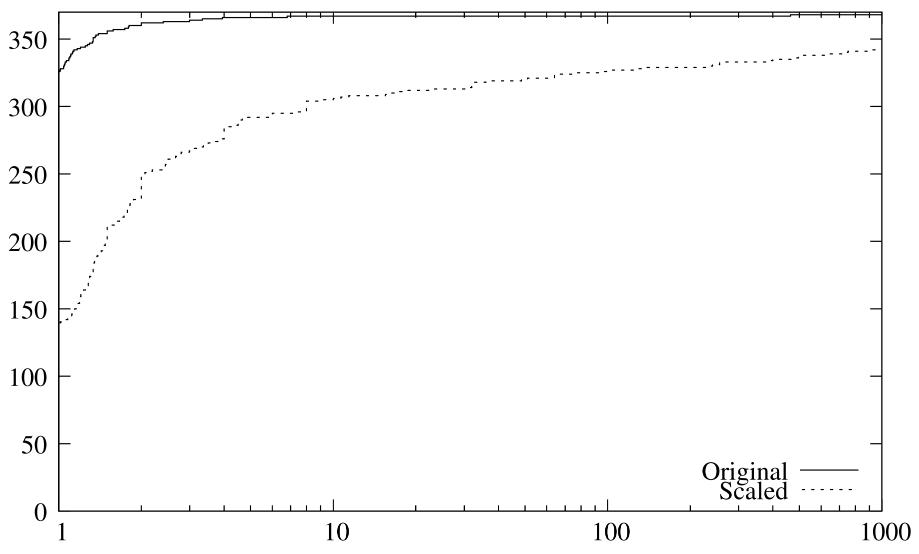
We first consider what the 'best' prescaling for each scaling is (ie MC30 by itself or after MC64) and then compare the performances of these best (combination) scalings before making our final recommendations.

We found in each case there was little to be gained from prescaling in compensation for the time overhead of doing two scalings rather than one. The results shown in Figure 4.3 for pre-scalings of $\texttt{MC77}_\infty$ are typical. They suggest we should avoid MC30 as a prescaling, and that MC64 and $\texttt{MC77}_1$ both slightly reduce the number of delayed pivots. However, use of $\texttt{MC77}_\infty$ by itself gives the smallest MA57 residuals, though this advantage is eliminated by the use of an iterative method.

In the remainder of this chapter we compare the performance of only straightforward scalings (ie with no prescaling).

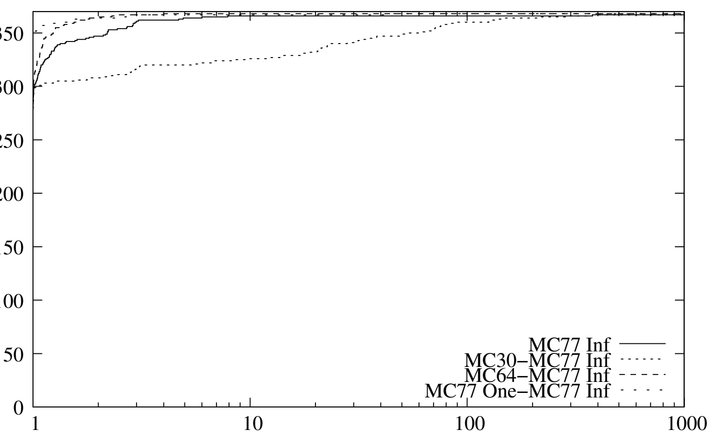


(a) Iterative refinement residual

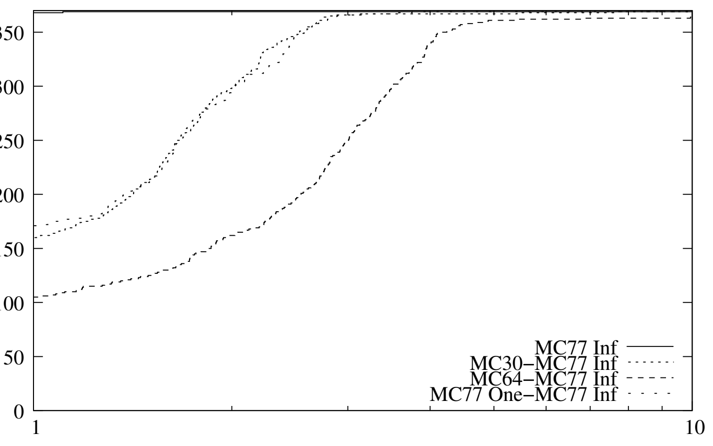


(b) FGMRES residual

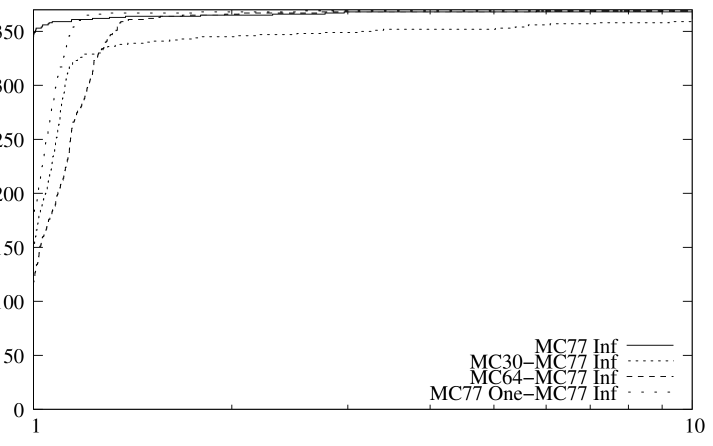
Figure 4.2: Performance profiles of the residuals after iterative refinement and after FGMRES for the original and the `MC64` scaled matrices.



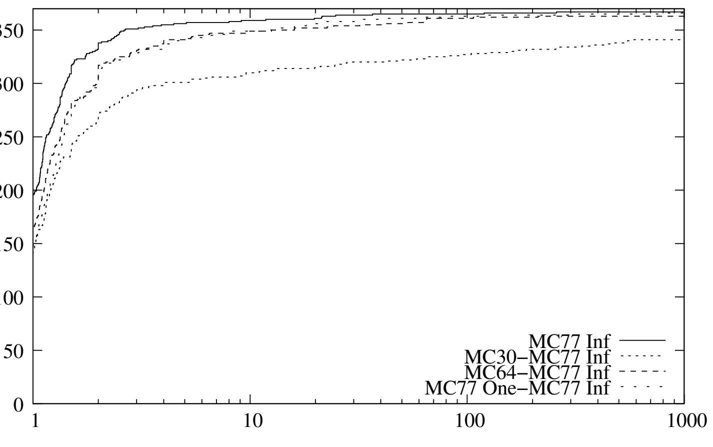
(a) Number of delayed pivots



(b) Time for scaling



(c) Total solution time



(d) MA57 residuals

Figure 4.3: Performance profiles comparing various prescalings for $\texttt{MC77}_\infty$.

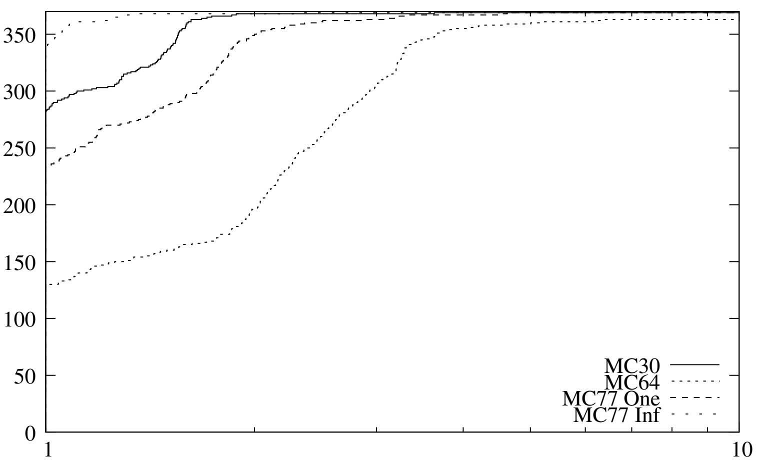


Figure 4.4: Performance profile for time to find and apply scaling only.

4.4.4 Overall results

It is worth noting that, for each scaling, there are some problems on which it is the best, and if it is important to reduce factorization time and/or the amount of fill-in, users may wish to try a variety of scalings on a representative sample of their problems before deciding which to use.

Figure 4.4, presents a performance profile of the scaling times. It is clear that $\texttt{MC77}_\infty$ is significantly faster than most other scalings, and that MC64 is significantly slower.

Straight forward performance profile comparisons of the scalings are presented in Figures 4.5–4.8. We note that after refinement (with iterative refinement or FGMRES) the residuals are comparable in quality. Further, none of the scalings needed more than four iterations of refinement on any problem.

Pralet [2004] reports that MC30 performs poorly on many indefinite problems, and our results support this finding. In terms of the number of delayed pivots, all the other scalings perform far better than no scaling, with comparable results for $\texttt{MC77}_1$ and MC64. With respect to the scaling times, MC64 is slower than the other scalings while $\texttt{MC77}_\infty$ is the fastest. The total time is lower for $\texttt{MC77}_1$ than for the other scalings but MC64 is slightly more robust (there are a small number of problems on which both the MC77 scalings do not perform well). If we look at the MA57 residuals, $\texttt{MC77}_\infty$ performs almost as well as no scaling, while MC30 generally performs worse (however the difference in the residual quality is eliminated once an iterative method is used).

To avoid distortions due to small problems (that are difficult to reliably time), we eliminate those problems taking less than 0.1 seconds to solve under any scaling. The resultant performance profile values, given by formula 4.2 at various values of α are shown in Table 4.1. A problem is regarded as 'not solved' if it runs out of memory or $\alpha_{ij} > 10$.

In order to see what effect the sparsity ordering has on the number of delayed pivots, data are shown for both Approximate Minimum Degree (AMD) [Amestoy et al. 1996; 2004] and MeTiS orderings. The AMD ordering is typically poorer than that generated by MeTiS for large problems; as a result fewer problems could fit their predicted factors in available memory and are not reported upon. It seems that general trends hold regardless of the ordering used.

If we require that a scaling enable the solution of a problem in a time no more than an order of magnitude longer than the fastest possible then only $\texttt{MC77}_1$ and MC64 would meet the requirement (though there is only one problem that $\texttt{MC77}_\infty$ does not succeed on). If we instead look for the scaling which solved the greatest number of problems without taking more than twice as long as the best possible then $\texttt{MC77}_1$ would slightly outperform MC64. It is not clear whether these conclusion will hold if, in the future, we were to consider even larger problems.

We shall demonstrate in the next section that we have a choice between a reliable but very slow MC64 scaling, and a fast but occasionally unreliable $\texttt{MC77}_1$ scaling.

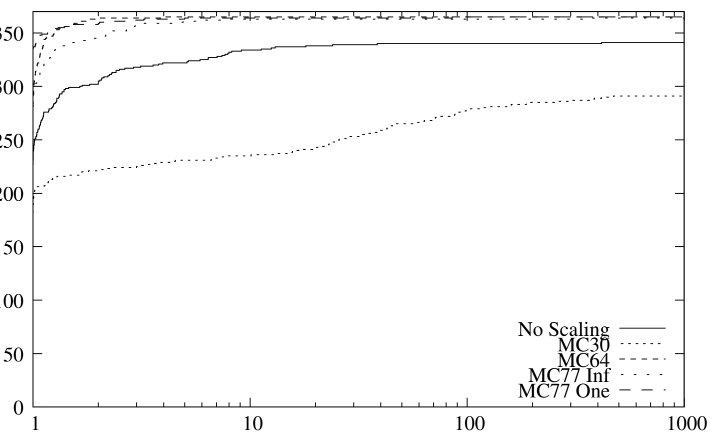


Figure 4.5: Number of delayed pivots for best variants.

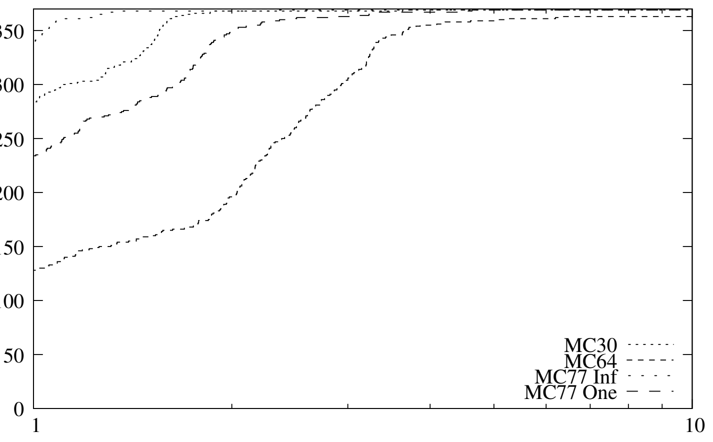


Figure 4.6: Scaling times for best variants.

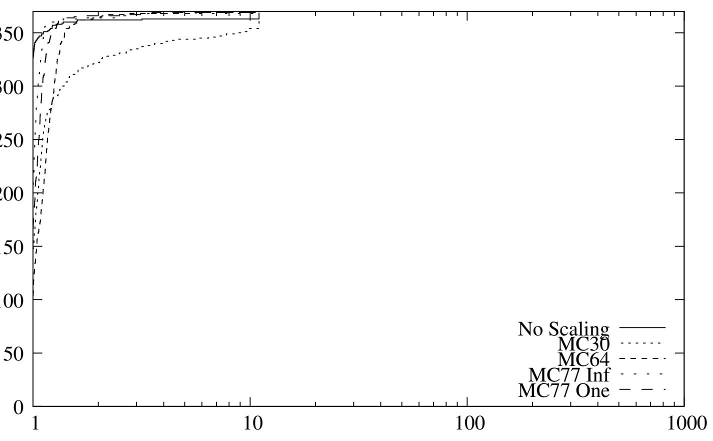


Figure 4.7: Total times for best variants.

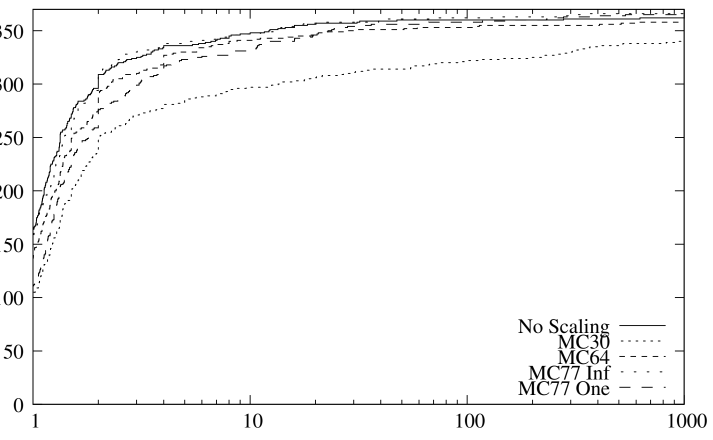


Figure 4.8: `MA57` residual for best variants.

α	No scale	MC30	MC64	MC77$_\infty$	MC77$_1$
1.0	**181**	5	5	30	21
1.1	**212**	110	64	184	133
1.2	214	153	133	**224**	204
1.3	220	165	180	**225**	218
1.4	223	169	208	**227**	**227**
1.5	223	177	220	**228**	**228**
1.6	223	179	224	228	**229**
1.7	225	183	226	228	**230**
1.8	225	185	226	228	**230**
1.9	225	187	228	228	**230**
2.0	225	187	228	228	**230**
Not Solved	6	15	**0**	1	**0**
$\alpha_{\max}$	-	-	**2.83**	-	3.16

(a) MeTiS

α	No scale	MC30	MC64	MC77$_\infty$	MC77$_1$
1.0	**151**	17	15	37	27
1.1	**180**	58	45	129	93
1.2	**183**	98	81	166	143
1.3	**187**	115	108	181	169
1.4	187	131	131	**189**	185
1.5	188	137	147	**190**	**190**
1.6	189	141	164	192	**193**
1.7	191	148	171	192	**194**
1.8	191	151	185	192	**195**
1.9	191	153	190	192	**195**
2.0	191	153	194	194	**195**
Not Solved	4	15	**0**	**0**	**0**
$\alpha_{\max}$	-	-	3.80	8.94	**3.26**

(b) AMD

Table 4.1: Performance data for the larger problems that require more than 0.1 seconds to solve. For each α interval the best results are highlighted in bold.

Name	m	Application	Properties
FIDAP/ex14	3973	Fluid Dynamics Finite Element	Indefinite
GHS_indef/bloweybl	30003	Materials Problem	Indefinite, Singular (rank 30002)
GHS_indef/copter2	55476	Fluid Dynamics Problem	Indefinite
GHS_indef/ncvxqp1	12111	Optimization Augmented System	Indefinite
GHS_indef/ncvxqp9	16554	Optimization Augmented System	Indefinite
Oberwolfach/LFAT5000	19994	Model Reduction	Positive Definite
Schenk_IBMNA/c-30	5321	Non-Linear Optimization	Indefinite
Schenk_IBMNA/c-52	23948	Non-Linear Optimization	Indefinite
Schenk_IBMNA/c-54	31793	Non-Linear Optimization	Indefinite
Schenk_IBMNA/c-62	41731	Non-Linear Optimization	Indefinite

Table 4.2: Ten interesting problems.

4.5 Interesting problems

We now examine ten problems more closely (see Table 4.2). These were chosen to illustrate where some of the scalings do poorly. We note that all the problems are part of our main test and the Schenk_IBMNA problems are from a larger set of such problems (see Appendix A).

Results for these problems are given in Table 4.3, and the relative timings for the different phases of `MA57` are shown in Graphs 4.9 and 4.10. We display results for both AMD and MeTiS orderings because, for our relatively small problems, the MeTiS time is a large proportion of the total solution time compared to that for AMD. We note that, because the analyse phase uses only the sparsity pattern, the analyse timings are independent of the scaling used.

These problems illustrate that, with the exception of `MC30`, there are problems for which each scaling is, by some measure, optimal. They also demonstrate each of the scalings can behave poorly. We note that, following iterative refinement, all residuals are comparable and hence are omitted. The optimization problems, which are characterised by having a mixture of extremely large and extremely small eigenvalues, provide the most challenging systems on which none of the scalings does consistently well.

Let us first consider using no scaling. This, in general, results in small `MA57` residuals (recall Figure 4.8) and, without the scaling overhead, the total solution time can be small (*copter2*, *c-30*, *c-52*). However there is a penalty in the large number of delayed pivots (*LFAT5000*, *ncvxqp1*) and, in the extreme case (*bloweybl*), this can lead to the problem not being solved because memory is exhausted.

While the results of the previous section tell us that `MC77`$_1$ is generally the best scaling in terms of delayed pivots, problems *ncvxqp1* and *c-62* illustrate that its behaviour is erratic and using `MC64` can lead to significantly fewer delayed pivots. On the extremal problem *c-62* the impact on the factorize time of the number of delayed pivots was almost a factor of four. On many of the problems, `MC64` results in slightly larger `MA57` residuals and problems *ncvxqp9*, *c-30* and *c-54* illustrate that `MC64` can be expensive, with the total solution time for these relatively small problems dominated by the scaling time. For larger problems, we anticipate that the time to scale will be a small fraction of the total time regardless of the scaling used.

`MC77`$_\infty$ is competitive when we compare the total solution times, and it leads to small `MA57` residuals. Furthermore, it appears to be the most robust for singular problems (although in our tests we have not applied the `MC64` modification suggested by Duff and Pralet [2005] where a preprocessing is applied to deal with rank deficiency before a weighted matching is attempted, designed to cope better with singular matrices).

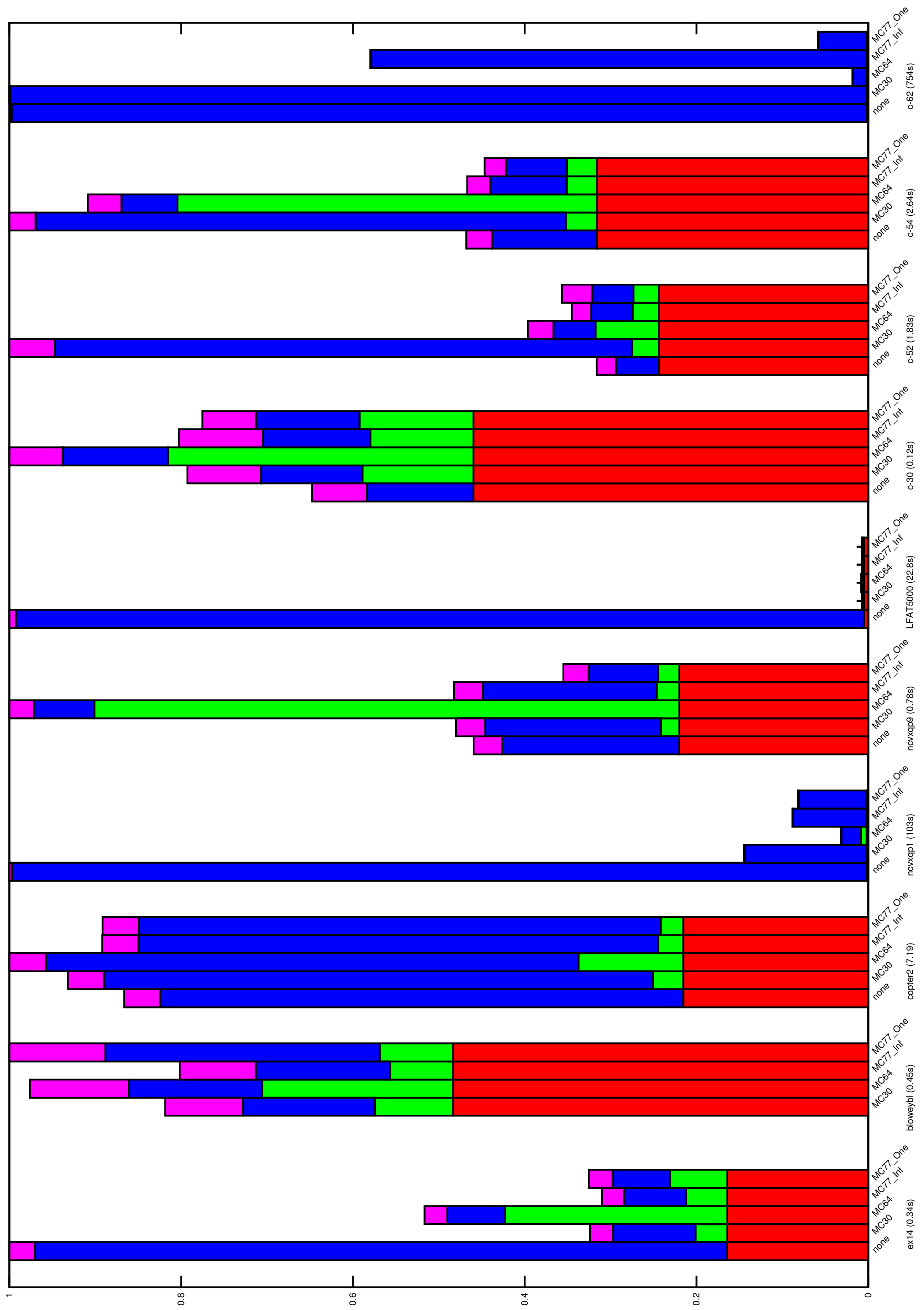
4.6 Conclusions

We conclude that, while the performance of `MC64` and `MC77`$_1$ is equally good for most problems, `MC64` is more consistent, with `MC77`$_1$ behaving poorly on a small minority of problems. This must, however, be set against the amount of time required to compute the scaling: `MC64` was

	Delayed pivots					MA57 Residual				
Problem	none	MC30	MC64	MC77$_\infty$	MC77$_1$	none	MC30	MC64	MC77$_\infty$	MC77$_1$
ex14	4825	839	586	735	692	1e-16	2e-13	5e-14	4e-14	3e-14
bloweybl	-	9652	9652	10435	19559	-	1e-12	3e-12	7e-16	3e-13
copter2	120	108	110	118	60	1e-12	2e-12	1e-12	1e-12	1e-12
ncvxqp1	1.7e5	59697	13184	40234	38788	3e-19	8e-18	1e-13	2e-17	4e-17
ncvxqp9	6217	6147	2808	6234	3275	1e-16	1e-16	4e-19	6e-17	1e-16
LFAT5000	46309	13	13	13	13	1e-16	2e-16	2e-16	2e-16	2e-16
c-30	24	0	1	6	0	6e-17	7e-16	2e-16	4e-17	7e-16
c-52	950	17116	825	880	738	6e-17	1e-16	8e-18	2e-18	1e-16
c-54	7135	19012	1881	5395	2681	8e-17	2e-15	5e-14	2e-16	8e-17
c-62	5.5e5	5.5e5	1333	5.0e5	1.1e5	2e-16	2e-16	2e-14	4e-16	6e-15

	Scaling Time					Factorize Time				
Problem	none	MC30	MC64	MC77$_\infty$	MC77$_1$	none	MC30	MC64	MC77$_\infty$	MC77$_1$
ex14	-	0.025	0.088	0.016	0.023	0.273	0.033	0.023	0.025	0.023
bloweybl	-	0.040	0.099	0.033	0.038	-	0.069	0.069	0.070	0.142
copter2	-	0.252	0.876	0.213	0.188	4.38	4.60	4.46	4.35	4.37
ncvxqp1	-	0.019	0.636	0.019	0.021	102.	14.5	2.34	8.75	8.09
ncvxqp9	-	0.017	0.532	0.020	0.019	0.161	0.160	0.552	0.158	0.063
LFAT5000	-	0.026	0.045	0.021	0.029	22.5	0.020	0.020	0.020	0.020
c-30	-	0.015	0.042	0.014	0.016	0.015	0.014	0.015	0.015	0.014
c-52	-	0.057	0.135	0.056	0.054	0.091	1.23	0.090	0.089	0.088
c-54	0	0.096	1.29	0.093	0.092	0.322	1.63	0.172	0.235	0.187
c-62	0	0.104	0.550	0.140	0.133	751.	752.	11.9	435.42	4.

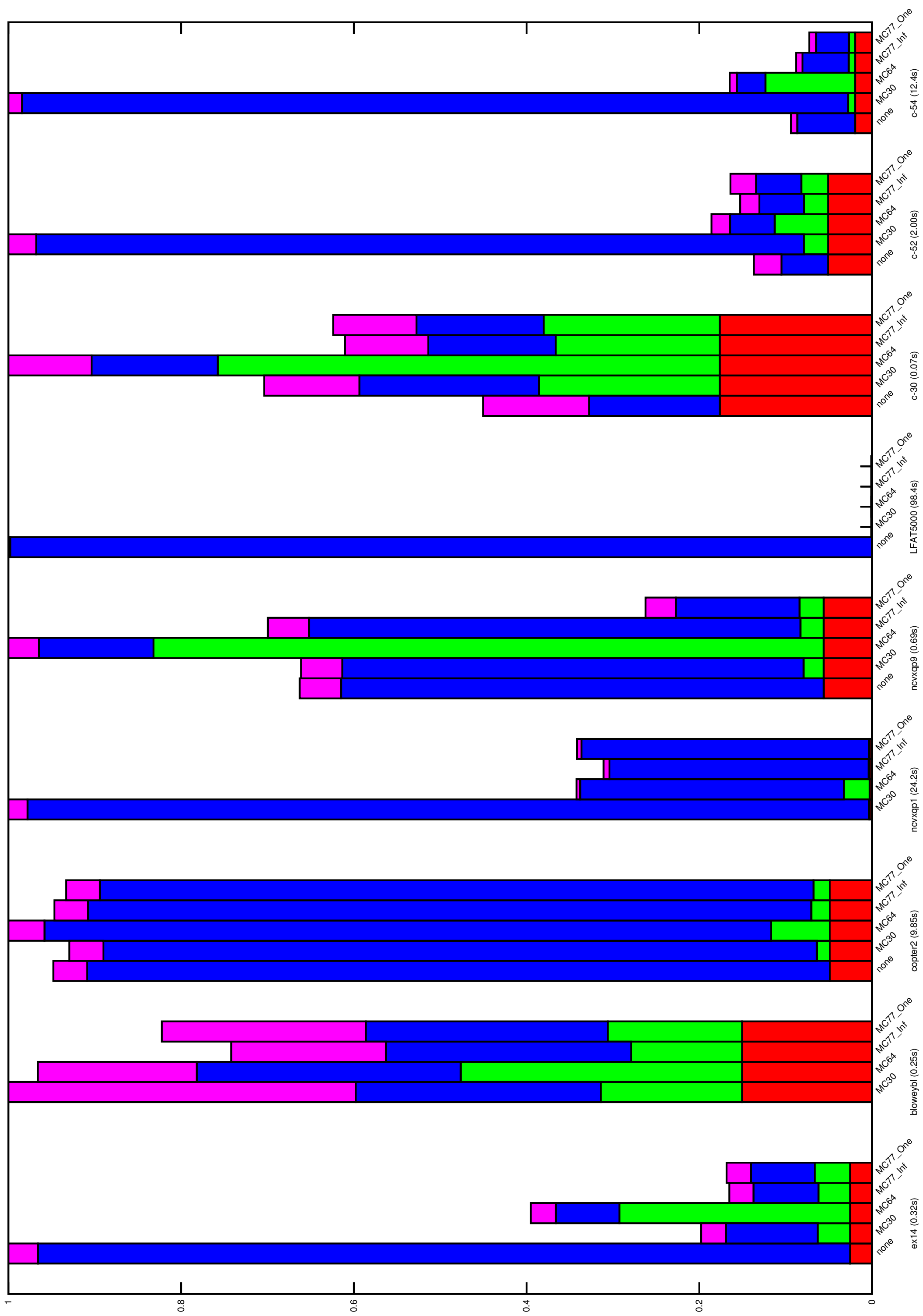
Table 4.3: Results for ten interesting problems.



Times are normalized by the longest time for each problem (given in parentheses).

From the bottom to the top: analyse,scaling,factorize,iterative refinement.

Figure 4.9: Showing relative timing dedicated to each part of the solve process, using MeTiS ordering.



Times are normalized by the longest time for each problem (given in parentheses).

From the bottom to the top: analyse,scaling,factorize,iterative refinement.
Note: *c-62* with the AMD ordering failed to factorize due to lack of memory.

Figure 4.10: Showing relative timing dedicated to each part of the solve process, using AMD ordering.

by far the most expensive to compute and, for the size of problems tested, sometimes led to large total solution times. MC64 is also observed to occasionally produce MA57 residuals that are two or three orders of magnitude larger than $\texttt{MC77}_1$, but we found these could be reduced by using $\texttt{MC77}_\infty$ as a prescaling. Either MC64 (possibly prescaled by $\texttt{MC77}_\infty$) or $\texttt{MC77}_1$ would make a good default approach for a direct solver. We would, however, recommend that the user perform some initial experiments if they have multiple related problems to solve. We note that for larger problems the scaling time may not be a significant proportion of the total solution time, so MC64 may prove better on this domain. MC30 is probably best avoided for indefinite problems.

A small number of steps of iterative refinement will, in most cases, remove any significant differences in the size of the residuals. If iterative refinement is undesirable, for example when using an out-of-core facility, $\texttt{MC77}_\infty$ may be the best choice (although it may suffer a very large number of delayed pivots for some problems).

Finally, we remark that, for our test matrices, it was often not necessary to do any scaling at all. However, without scaling the worst case behaviour was more extreme than for any of the scalings we tried and, of course, we do not know if the test data were prescaled before being included in the University of Florida sparse matrix collection.

Chapter 5

A mixed-precision solver

We consider the development of a fast and robust mixed precision sparse symmetric solver. This work described in this chapter has been accepted for publication in ACM Transactions on Mathematical Software as [Hogg and Scott 2010a]. The user interface design was jointly drafted by the author and Scott, but the majority of the implementation (that is the complex mixed precision wrapper around the existing codes) was done by the author, with quality control and additional testing performed by Scott. We are again grateful to John Reid, Iain Duff and Mario Arioli for useful discussions relating to this work. Additionally, helpful comments were received from three anonymous referees while the paper was under review. Our major scientific contribution is the engineering of a robust solver written in Fortran which realises the idea of a mixed precision solver suggested by theoretical papers, developing a methodology for combining iterative refinement and FGMRES in a tuned and robust fashion. The resultant software, `HSL_MA79`, is now part of the HSL software library [HSL 2007], which is freely available under license to academics.

A preprint of the paper is reproduced with kind permission of the copyright holders at the rear of this thesis. HSL software may be obtained from
`http://www.hsl.rl.ac.uk/`.

5.1 Mixed precision

A mixed precision solver is one that aims to achieve a high (double precision) accuracy solution through the use of a mixture of single and double precision operations. The use of single precision is motivated by the higher execution speeds of single precision operations, but at some stage must use double precision to improve the accuracy. In this section we describe how the use of a single precision symmetric indefinite (or Cholesky) factorization may be used as a preconditioner to a double precision iterative method.

Aiming for the same accuracy as performing the factorization in double precision is conventional: most direct solvers use double precision arithmetic. Double precision is normally the highest precision for which hardware is optimized, and so is a sensible choice. However, the accuracy required when solving a system of equations is application dependent, and may be lower than required. If the required accuracy is less than about 10^{-5} (which may be all that is appropriate if, for example, the problem data are not known to high accuracy), single precision may often be used. If a problem is very ill conditioned, high precision may be necessary to obtain a solution that is accurate to at least a modest number of significant figures. Regardless, common practice remains that double-precision accuracy is requested by users: either out of actual need, or just to be safe.

Motivated by recent studies and theoretical developments [Arioli and Duff 2009; Buttari et al. 2007; Buttari et al. 2008] demonstrating that simple high precision iterative methods can be effectively preconditioned by a lower precision factorization, we design and develop a library-quality mixed-precision sparse solver for symmetric linear systems.

5.2 Test environment

All reported experiments in this section are performed on a single core of `fox`, using the `gfortran-4.3` compiler with `-O1` optimization and a serial version of the Goto BLAS [Goto and van de Geijn 2008]. All timings are elapsed times measured using the Fortran intrinsic `system_clock` and are given in seconds. We set the option to flush denormal numbers to zero, based on poor performance of some problems in initial experiments if this was not done.

We work with two test sets; all but three of the problems are drawn from the University of Florida Sparse Matrix Collection [Davis 2007] and all are symmetric with either real or integer valued entries.

Test Set 2: Small to medium matrices with $n \geq 1000$ and at most 10^7 entries in the upper (or lower) triangular part. This set comprises 330 problems.

Test Set 3: Medium to large matrices with $n \geq 10000$. This set comprises 232 problems.

We note that 170 problems belong to both sets. The problems are held as two test sets because it is more practical to perform a lot of tests on the smaller test problems. Furthermore, `MA57` is not able to solve the largest problems in Test Set 3 (because of insufficient memory); for these, the out-of-core facilities offered by `HSL_MA77` are needed. In all our experiments, we use threshold partial pivoting with the threshold parameter set to $u = 0.01$ (thus, all the test problems are treated as indefinite, even though some are known to be positive definite). Furthermore, we scale the test problems using the HSL package `MC77` (the ∞-norm scaling is used)[1].

5.3 On single precision

As noted in Chapter 2, modern sparse direct solvers have two major numerical operations: the use of Level 3 BLAS, and a sparse scatter operation. The speed of the BLAS operations is limited largely by latency and the flop rate, but the scatter operations are limited more by memory bandwidth and latency. With the recent emergence of multicore processors and with future chips likely to have ever larger numbers of cores, the data transfer rate and memory latency are expected to become an ever tighter constraint [Graham et al. 2004]. In the context of these operations, using single-precision arithmetic in preference to double offers significant reductions in data movement as well as memory usage.

The storage requirements for a sparse direct solver are dominated by that required for the matrix factor L which, due to fill-in, generally contains many more entries than the original sparse matrix A. Using single precision gives storage savings of 50% for L, enabling the solver to be used to solve larger problems. Single precision has potential to be particularly advantageous for an out-of-core solver because the amount of disk access is also approximately halved and the in-core working set is doubled. We note that in the dense case the matrix factors are the same size as the original matrix and so, in the symmetric case, the memory saving using single precision is only 25% (when we keep a double-precision copy of A in addition to the factors so that refinement can be performed).

In addition to the advantages of memory savings and reduced data movement, single-precision arithmetic is currently more highly optimized (and hence faster) than double-precision computation on a number of architectures, such as commodity Intel chips, Cell processors and general-purpose computing on graphics cards (GPGPU). Buttari, Dongarra, and Kurzak [2007] report differences as great as a factor of ten in speed. Thus it is highly advantageous to carry out as much computation as possible on these chips using single precision arithmetic.

Figure 5.1 compares the performance of matrix-matrix multiplies (level 3 BLAS) in single (`sgemm`) and double (`dgemm`) precision, for square matrices up to order 1000. Since `_gemm` operations are used extensively within modern sparse direct solvers (including `MA57` and `HSL_MA77`), this demonstrates the potential advantage of performing the factorization using single precision. Figure 5.2 shows how this translates into a performance gain for the factorization phase of `MA57` (here the test set comprises the subset of Test Set 2 problems that take at least 0.01

[1] In our tests, the direct solvers were unable to solve problems *GHS_indef/boyd1* and *GHS_indef/aug3d* with this choice of scaling so, in these instances, we scale using `MC64` [Duff 2004; Duff and Koster 2001].

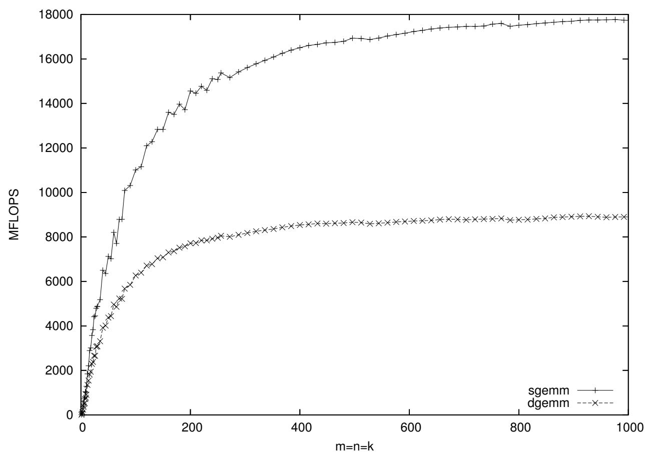


Figure 5.1: The performance of single (`sgemm`) and double (`dgemm`) precision matrix-matrix multiplication for a range of sizes of square matrices.

seconds to factorize in double precision on our test machine). While we do not see gains of quite the factor of two that is achieved for `sgemm`, we do see worthwhile improvements on the larger problems, that is, those taking longer than about 1 second to factorize. Here `_gemm` operations dominate the factorization time, whereas on the smaller problems, integer operations as well as book keeping are more dominant and for such problems there may be little reward in terms of computation time in pursuing a mixed-precision approach.

There was also one test matrix (not shown in Figure 5.2) for which the ratio of the single-precision to the double-precision factorization time was well in excess of two. This was an indefinite problem and although in both cases the sparsity ordering was the same, during the factorization the pivot sequence was modified to maintain numerical stability. In the double-precision factorization more delayed pivots were generated than in the single-precision case, resulting in a much higher operation count.

5.4 Basic algorithm

Our aim is to perform a single-precision factorization and then, if necessary, use double-precision post-processing to recover a solution to the desired precision. For maximum efficiency, we want to try the cheapest algorithm first and, only if this fails, resort to applying more computationally expensive alternatives. In the worst case, we fall back to performing a double-precision factorization.

We require that the computed solution $\boldsymbol{x}$ satisfies $\beta \leq \gamma$, where γ is a parameter chosen by the user and β is the scaled residual given by the equation (5.1).

$$\beta = \frac{\|\boldsymbol{r}\|_\infty}{\|A\|_\infty \|\boldsymbol{x}\|_\infty + \|\boldsymbol{b}\|_\infty}, \tag{5.1}$$

where $\boldsymbol{r}$ is the residual given by $\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}$ (the infinity norm is given by $\|\boldsymbol{x}\|_\infty = \max_i |x_i|$ and has an induced matrix norm $\|A\|_\infty = \max_i \sum_j |a_{ij}|$). If $\beta \leq \gamma$ is satisfied we say that the *requested accuracy* has been achieved. This provides a stopping criteria in our algorithms. We note that in the case where we scale A prior to the factorization, the unscaled matrix is retained for any iterative method used to refine the solution and β is calculated with the unscaled A for checking the accuracy (recall Section 4.4.2).

Algorithm 17 summarises the basic mixed-precision approach. The factorization is performed in single precision then, if the scaled residual β exceeds γ, iterative refinement in double precision is performed. If the requested accuracy has not yet been achieved, FGMRES (in double precision) is used and, finally, if β is still too large, a switch is made to double precision and the computation is restarted.

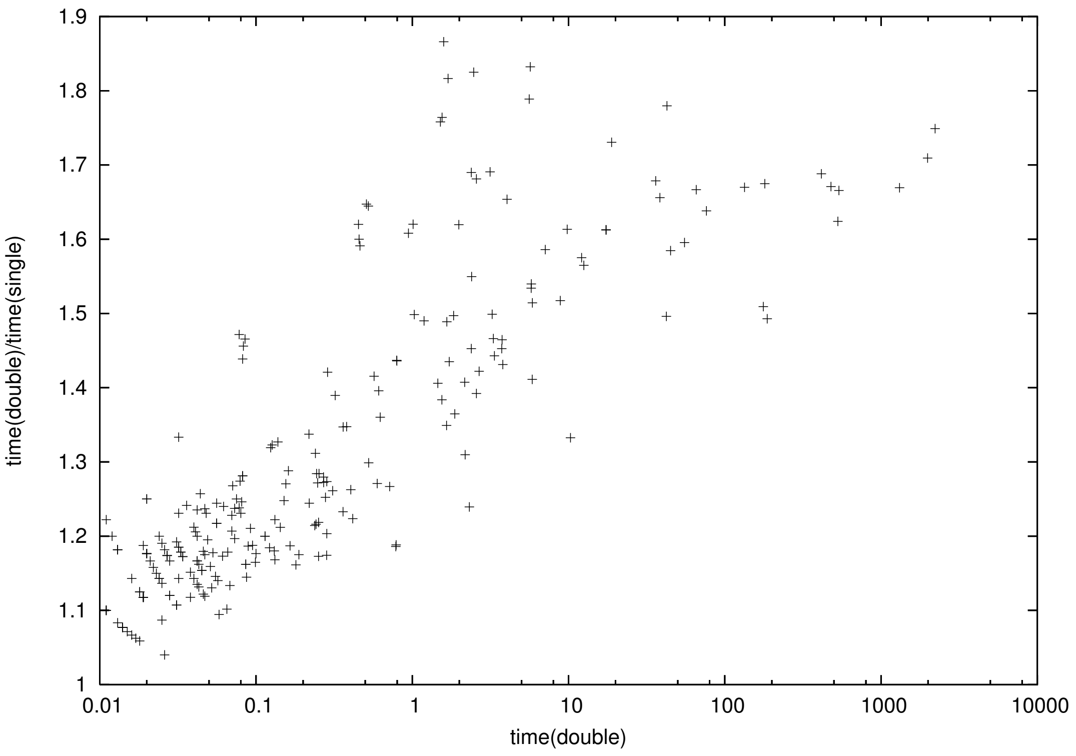


Figure 5.2: The ratios of the times required by the factorization phase of the sparse direct solver `MA57` when run in single and double precision (the problems are a subset of Test Set 2).

Algorithm 17 Mixed-precision solver

```
**Input:** Requested accuracy $\gamma$
Set $prec = single$.
**loop**
  Factorize $PAP^T$ as $LDL^T$ using precision *prec*.
  Solve $A\boldsymbol{x} = \boldsymbol{b}$ and compute $\beta$.
  **if** $\beta \le \gamma$ **then** exit.
  Perform iterative refinement (Algorithm 18).
  **if** $\beta \le \gamma$ **then** exit.
  Perform FGMRES (Algorithm 19).
  **if** $\beta \le \gamma$ **then** exit.
  **if** $prec = single$ **then**
    Set $prec = double$.
  **else**
    Set error flag and **exit**.
  **end if**
**end loop**
```

5.4.1 Solving in single

Clearly the solution using the single-precision factors could be implemented in a number of ways depending on what precision conversions are performed.

In this work we will use *"solve using single"* to mean that the input vector $\boldsymbol{b}$ is converted from double to single precision, a solve is performed in single, and then the result $\boldsymbol{x}$ is converted from single to double. The conversions are performed by a copy to a temporary variable. These transitory variables only exist for the duration of the solve where they are required.

An alternative, which we shall call *"solve using double"*, is to convert the single-precision factors L from single to double and then perform the solve completely in double precision. To avoid the overhead of storing the factors in double precision we have written a special solve subroutine which converts the factors from single to double as required during the solve. The double-precision versions are discarded after use.

One would expect solves using single precision to execute faster, and this is indeed the case. However the accuracy of solves using double precision is higher; that is to say, that we compute $\boldsymbol{x} = L^{-T}L^{-1}\boldsymbol{b}$ more accurately for the given L, which has the same accuracy in either precision. This may enable iterative methods preconditioned by the solve to converge faster. This is demonstrated for FGMRES in Section 5.6.

Unless otherwise stated we will always use the solve using single precision and store all vectors in double precision.

5.5 Iterative refinement

Algorithm 18 Mixed-precision iterative refinement

```
Input: Single precision factorization of $A$, requested accuracy $\gamma$,
       minimum reduction $\delta$ and i_maxitr.
Solve $A\boldsymbol{x}_1 = \boldsymbol{b}$ (using single precision).
Compute $\boldsymbol{r}_1 = \boldsymbol{b} - A\boldsymbol{x}_1$ and $\beta_1$ (using double precision).
Set $k = 1$.
while ($\beta_k > \gamma$ and $k <$ i_maxitr) do
  Solve $A\boldsymbol{y}_{k+1} = \boldsymbol{r}_k$ (using single precision).
  Set $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{y}_{k+1}$ (using double precision).
  Compute $\boldsymbol{r}_{k+1} = \boldsymbol{b} - A\boldsymbol{x}_{k+1}$ and $\beta_{k+1}$ (using double precision).
  if $\beta_{k+1} > \delta\beta_k$ or $\|\boldsymbol{r}_{k+1}\|_\infty \geq 2\|\boldsymbol{r}_k\|_\infty$ then Set error flag and exit (stagnation).
  $k = k + 1$
end while
$\boldsymbol{x} = \boldsymbol{x}_k$
```

Iterative refinement is a simple first order method used to improve a computed solution of $A\boldsymbol{x} = \boldsymbol{b}$; it is outlined in Algorithm 18. Here β_k is the norm of the scaled residual (5.1) on the k^{th} iteration and `i_maxitr` is the maximum number of iterations. The system $A\boldsymbol{x} = \boldsymbol{b}$ and the correction equation $A\boldsymbol{y}_{k+1} = \boldsymbol{r}_k$ are solved using the computed single-precision factors of A. Skeel [1980] proved that, to reduce the scaled residual to a given precision, it is sufficient to compute the residual and the correction in that precision. However, since we wish to obtain residuals with double-precision accuracy using factors computed in single precision, we perform the forward and back substitutions (which we refer to as the *solves* throughout the rest of this chapter) in single precision and compute the residuals and corrected solution $\boldsymbol{x}_{k+1}$ in double precision. This mixed-precision version of iterative refinement is also used by Buttari et al. [2007; 2008], while Demmel et al. [2009] use a similar scheme for overdetermined least squares problems, further developing bounds on the forward error.

Iterative refinement generally decreases the residual significantly for a number of iterations before stagnating (that is, reaching a point after which little further accuracy is achieved), although for some problems (including the test problems *HB/bcsstm27*, *Cylshell/s3rmq4m1*, and *GHS_psdef/s3dkq4m2* that were considered in [Arioli and Duff 2009]), a large number of iterations are needed before any substantial reduction in the residual is achieved. To detect

δ	0.001	0.01	0.05	0.07	0.08	0.1	0.2	0.3	0.4	0.5	∞
Converged	194	227	252	256	258	259	264	265	265	265	268
Failed	136	103	78	74	72	71	66	65	65	65	62

Table 5.1: The number of problems in Test Set 2 for which iterative refinement achieved the requested accuracy ($\gamma = 5 \times 10^{-15}$) using a range of values of the improvement parameter δ (`i_maxitr`= 10).

stagnation (and thus avoid performing unnecessary solves), we employ a minimum improvement parameter δ. A large δ allows the iterative refinement to continue until the maximum number of iterations has been performed. This increases the likelihood of convergence at the expense of carrying out additional iterations for problems that have stagnated before reaching the requested accuracy γ. The number of additional iterations can be reduced or eliminated by choosing a small δ. The condition $\|\boldsymbol{r}_{k+1}\|_\infty \geq 2\|\boldsymbol{r}_k\|_\infty$ detects numerical issues where $\boldsymbol{x}_{k+1}$ explodes but the residual β_{k+1} remains small due to scaling by $\boldsymbol{x}_{k+1}$. For different values of δ, Table 5.1 reports the number of problems belonging to Test Set 2 that achieve the requested accuracy when factorized using `MA57` in single precision and then corrected using mixed-precision iterative refinement. We see that values in the range $[0.05, 0.5]$ have a similar success rate of just under 80%. We choose as our default $\delta = 0.3$ as this provides a good compromise between the number of problems that converged (259) and minimizing the wasted iterations on the remainder — 62% of the problems that failed with this δ used the same number of solves as for $\delta = 0.001$, and only 4 of the 65 required more than 2 additional iterations before stagnation was recognised. We remark that the package `MA57` includes an option to perform iterative refinement (using the same precision as the factorization) and, by default, it uses $\delta = 0.5$ (see also [Demmel et al. 2006] and [Higham 1997]).

Our implementation of iterative refinement also offers an option to terminate once a chosen number `i_maxitr` of iterations has been performed. An upper limit on the maximum number of iterations can be established by considering the following example. Assume the initial scaled residual is $\beta = 10^{-7}$ and the default improvement parameter $\delta = 0.3$ is used. If stagnation has not occurred, after 13 iterations the scaled residual must be less than 1.6×10^{-14} and after 15 iterations, less than 4.8×10^{-15}. Based on our experiments, we set the default value to `i_maxitr` = 10 (note that for most of our test examples we found that either the requested accuracy was achieved or stagnation was recognised before this limit was reached).

5.6 Preconditioned FGMRES

For our examination of FGMRES, we use the 62 problems from Test Set 2 that failed to achieve the requested accuracy using iterative refinement with any δ. We call this the ***Reduced Test Set 2***.

In Algorithm 17, FGMRES [Saad 1994] refers to a right-preconditioned variant of FGMRES. Arioli, Duff, Gratton, and Pralet [2007] have shown that, in cases where iterative refinement fails, FGMRES may succeed and is more robust than either iterative refinement or GMRES. Arioli and Duff [2009] prove that a mixed-precision computation, where the matrix factorization is computed in single precision and the FGMRES iteration in double precision, gives a backward stable algorithm. We note that their result is given for sufficiently large values of the restart counter (k in Algorithm 19), and does not apply to our algorithm where this is small for practical reasons.

Our variant of FGMRES, shown as Algorithm 19, is essentially that given by Arioli and Duff [2009] but additionally uses an adaptive restart parameter. Here `f_maxitr` is the maximum number of iterations and $\boldsymbol{e}_1$ denotes the first column of the identity matrix. Algorithm 19 uses double precision throughout except for the solution of the systems involving A; for these systems we have the ability to perform the forward and backward substitutions in either single or double precision, as we detail below. Our adaptive restarting strategy relies on a similar concept to the minimum improvement parameter in iterative refinement. We expect to reduce the backward error in the outer iterations and, if the reduction is too small, we increase the restart parameter (up to a specified maximum `max_restart`). If there is no reduction in the

Algorithm 19 Mixed-precision FGMRES right preconditioned by a direct solver with adaptive restarting (Norms here are 2-norms)

```
**Input:** Single precision factors of $A$, $\gamma$, $\delta$, f_maxitr, restart, max_restart
Solve $A\boldsymbol{x} = \boldsymbol{b}$.
Compute $\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}$ and $\beta$.
Initialise $j = 0; \beta_{\text{old}} = \beta; \boldsymbol{x}_{\text{old}} = \boldsymbol{x}$.
while ($\beta > \gamma$ and $j <$ f_maxitr) do
  $\beta_{\text{old}} = \beta$
  Initialise $\boldsymbol{v}_1 = \boldsymbol{r}/\|\boldsymbol{r}\|$,  $\boldsymbol{y}_0 = \boldsymbol{0}$,  $k = 0$.
  while ($\|\|\boldsymbol{r}\|\boldsymbol{e}_1 - H_k\boldsymbol{y}_k\| \geq \gamma(\|A\|\|\boldsymbol{x}\| + \|\boldsymbol{b}\|)$  and  $k <$ restart do
    $k = k + 1$ (Increment restart counter).
    $j = j + 1$ (Increment iteration counter).
    Solve $A\boldsymbol{z}_k = \boldsymbol{v}_k$ and compute $\boldsymbol{w} = A\boldsymbol{z}_k$.
    Orthogonalize $\boldsymbol{w}$ against $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k$ to obtain a new $\boldsymbol{w}$. Set $\boldsymbol{v}_{k+1} = \boldsymbol{w}/\|\boldsymbol{w}\|$.
    Form $H_k$, a trapezoidal basis for the Krylov subspace spanned by $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k$
      (Full details of this step may be found in [Saad 2003]).
    $\boldsymbol{y}_k = \arg\min_{\boldsymbol{y}} \|\|\boldsymbol{r}\|\boldsymbol{e}_1 - H_k\boldsymbol{y}_k\|$
      (Minimize residual over the Krylov subspace).
  end while
  Set $Z_k = [\boldsymbol{z}_1 \cdots \boldsymbol{z}_k]$.
  Compute $\boldsymbol{x} = \boldsymbol{x} + Z_k\boldsymbol{y}_k$,  $\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}$ and compute new $\beta$.
  if $\beta \geq \delta\beta_{\text{old}}$ then
    restart $= 2 \times$ restart
    if restart $>$ max_restart then Set error flag and exit.
  end if
  if $\beta > \beta_{\text{old}}$ then
    $\boldsymbol{x} = \boldsymbol{x}_{\text{old}}$
  else
    $\boldsymbol{x}_{\text{old}} = \boldsymbol{x}; \beta_{\text{old}} = \beta$
  end if
end while
```

`restart` =	1	2	4	8	16
Problems failed for both single and double	23	23	23	23	23
Problems solved by double but not single	5	5	5	5	5
Average difference in number of solves (see (5.2))	14.9	15.6	16.2	12.9	12.9
Average ratio of number of solves (see (5.3))	2.8	2.5	3.1	2.1	2.1

Table 5.2: A comparison of the performance of FGMRES using single and double-precision solves following a single-precision factorization for a range of restart parameters on Reduced Test Set 2 ($\delta = 0.3$, `f_maxitr=128`, `max_restart=128`).

backward error β (although in exact arithmetic the convergence of FGMRES is non-increasing, in finite-precision arithmetic the backward error may increase), we restore the solution from the previous outer iteration before restarting. In our experiments, we compared adaptive restarting with the use of a fixed restart parameter. We found that a small initial value for the adaptive restart parameter (typically less than or equal to 4) reduced the number of iterations required to obtain the requested accuracy and enabled us to solve some problems that failed to converge with a fixed restart; the strategy had little effect for larger initial values. All FGMRES results in this chapter are hence based on this adaptive restarting algorithm.

As we noted previously in Section 5.4.1, we may alternatively perform the solves with the single precision factors of A using double precision. Our experiments using a range of values for the adaptive restart parameter show that, in all cases, using double precision reduces the total number of solves required compared to performing solves using single precision. Table 5.2 attempts to capture the extent to which the double-precision approach is better. Shown here are:

- The number of problems that fail to converge using either single- or double-precision solves.
- The number of problems that converge only using the double-precision solve.
- The arithmetic mean of the number of extra solves needed in single precision, that is,

$$\frac{1}{|\mathcal{P}|}\sum_{i\in\mathcal{P}}\left(\text{Solves}_i(\mathit{single}) - \text{Solves}_i(\mathit{double})\right), \tag{5.2}$$

  where $\text{Solves}_i(\mathit{double})$ ($\text{Solves}_i(\mathit{single})$) is the number of solves used for problem i when performed in double (single) precision, and $\mathcal{P}$ is the set of problems on which both single- and double-precision solves converge to the requested accuracy.

- The geometric mean of the ratio of the number of solves in single precision to the number in double precision, that is,

$$\left(\prod_{i\in\mathcal{P}}\frac{\text{Solves}_i(\mathit{single})}{\text{Solves}_i(\mathit{double})}\right)^{\frac{1}{|\mathcal{P}|}}. \tag{5.3}$$

A possible explanation of the good behaviour of solves using double precision in FGMRES, as opposed to the negligible gains for Iterative Refinement, is that FGMRES uses 2-norms rather than the ∞-norms used elsewhere in this chapter. The 2-norms are prone to larger error accumulation than the ∞-norms and double precision solves may help to counter this. Alternatively, the solves using double precision may provide a more consistent preconditioner than solves using single precision.

We comment that a constant number of failing problems regardless of the restart value is what we expect from our adaptive restarting procedure.

The reduction in the number of solves when using double precision has to be set against the increased time per solve compared to single precision. On our test computer the time for a solve in double precision is approximately twice that in single precision, hence if the ratio of the number of solves in single precision to those in double is more than two (as is the case in the last

Problem	`restart`				
	1	2	4	8	16
Boeing/bcsstk35	49	**48**	64	**48**	**48**
Boeing/bcsstk38	**3**	4	4	8	8
Boeing/crystk01	15	14	12	**8**	**8**
Boeing/crystk02	**3**	6	4	8	8
Boeing/crystk03	**3**	6	4	8	8
Boeing/msc01050	7	6	**4**	8	8
Cunningham/qa8fk	**3**	6	4	8	8
Cylshell/s3rmq4m1	11	10	**8**	**8**	**8**
Cylshell/s3rmt3m1	17	12	**4**	8	8
DNVS/ship_001	**48**	**48**	**48**	64	64
GHS_indef/cont-201	25	24	20	**16**	**16**
GHS_indef/cvxqp3	23	**22**	32	24	24
GHS_indef/ncvxbqp1	11	**8**	**8**	**8**	**8**
GHS_indef/ncvxqp5	10	10	**8**	**8**	**8**
GHS_indef/sparsine	14	14	**12**	16	16
GHS_indef/stokes128	**1**	2	4	8	8
GHS_psdef/oilpan	11	10	**8**	**8**	**8**
GHS_psdef/s3dkq4m2	40	16	16	**8**	**8**
GHS_psdef/s3dkt3m2	17	**16**	**16**	**16**	**16**
GHS_psdef/vanbody	31	30	28	**24**	**24**
Gset/G33	**2**	**2**	4	8	8
HB/bcsstm27	13	12	**8**	**8**	**8**
Koutsovasilis/F2	**3**	6	4	8	8
ND/nd3k	**2**	**2**	4	8	8
Oberwolfach/gyro	10	**8**	**8**	**8**	**8**
Oberwolfach/gyro_k	10	**8**	**8**	**8**	**8**
Oberwolfach/t2dah	7	6	**4**	8	8
Oberwolfach/t2dah_a	7	6	**4**	8	8
Oberwolfach/t2dal	7	6	**4**	8	8
Oberwolfach/t2dal_a	7	6	**4**	8	8
Oberwolfach/t2dal_bci	7	6	**4**	8	8
Oberwolfach/t3dh	**3**	6	4	8	8
Oberwolfach/t3dh_a	**3**	6	4	8	8
Oberwolfach/t3dl	**3**	6	4	8	8
Oberwolfach/t3dl_a	**3**	6	4	8	8
Pajek/Reuters911	15	12	12	**8**	**8**
Schenk_IBMNA/c-56	31	30	28	**16**	**16**
Schenk_IBMNA/c-62	31	30	28	**24**	**24**
Simon/olafu	16	12	16	**8**	**8**

Table 5.3: The number of solves performed within FGMRES for Reduced Test Set 2 using a range of values for `restart` following unsuccessful iterative refinement. For each problem, the best result is in bold. Results are shown for $\delta = 0.3$, `f_maxitr`$= 64$, and `max_restart=64`. The 23 problems that failed for all values of `restart` are not shown.

line of Table 5.2) then double precision is faster, while also allowing us to solve more problems. As a result, in the remainder of this chapter we use double-precision solves for FGMRES. Note that iterative refinement will still use single-precision solves as similar experiments showed there to be no significant benefit in using double precision. In particular, although more iterations were carried out, iterative refinement still eventually stagnates on the problems belonging to the Reduced Test Set 2.

In Table 5.3, we report the number of solves performed within FGMRES for a range of values for the adaptive restart parameter on the 39 problems in the Reduced Test Set 2 for which FGMRES was successful. The results indicate that `restart` $= 4, 8$ or 16 is generally the best choice; we use as our default `restart` $= 4$. We observe that for some problems a higher `restart` parameter results in a larger number of iterations. This is because the termination conditions are only tested when the algorithm is restarted; higher values of `restart` test for termination less frequently.

5.7 Design and implementation of the mixed-precision strategy

We designed our implementation of Algorithm 17, `HSL_MA79`, to be both robust and efficient, catering for both positive definite and indefinite systems. Where possible it uses existing HSL packages as its main building blocks to reduce maintenance overheads and increase robustness. In particular, it uses: the direct solvers `MA57` (in-core) and `HSL_MA77` (out-of-core), the `MI15` implementation of FGMRES, and the scalings `MC30`, `MC64`, and `MC77`.

While we include a range of options, it is not our intention to incorporate all possibilities available within the direct solvers `MA57` and `HSL_MA77`. If a user needs to use such an option we hope our code will provide a framework from which they may work. Our main aim was to design a general purpose package that is straightforward to use and, through the restriction on the number of parameters that must be set, does not require the user to have a detailed knowledge and understanding of all the different components of the algorithms used. As such we have strayed from the traditional analyse/factorize distinction and emphasise the mixed-precision nature by combining the factor and solve phases.

The following procedures are available to the user:

- `MA79_factor_solve` accepts the matrix A, the right-hand sides $\boldsymbol{b}$, and the requested accuracy. Based on the matrix, it selects whether to use `MA57` or `HSL_MA77`; by default, single precision is selected as the initial precision. The code then implements Algorithm 17. The matrix factorization is retained for further solves.

- `MA79_refactor_solve` uses information returned from a prior call to `MA79_factor_solve` to reduce the time required to factorize and solve $A\boldsymbol{x} = \boldsymbol{b}$. The sparsity pattern of A must be unchanged since the call to `MA79_factor_solve`; only the numerical values of the entries of A and $\boldsymbol{b}$ may have changed. By default, the precision for the factorization is chosen based on that used by `MA79_factor_solve`. The matrix factorization is retained for further solves.

- `MA79_solve` uses previously computed factors generated by `MA79_factor_solve` (or alternatively `MA79_refactor_solve`) to solve further systems $A\boldsymbol{x} = \boldsymbol{b}$. Multiple calls to `MA79_solve` may follow a call to `MA79_factor_solve` (or `MA79_refactor_solve`).

- `MA79_finalize` should be called after all other calls are complete for a problem. It deallocates the components of the derived data types and discards the matrix factors.

We now discuss the first three of the above procedures in more detail (the finalize routine needs no further explanation).

5.7.1 `MA79_factor_solve`

On the call to `MA79_factor_solve`, the user must supply the entries in the lower triangular part of A in compressed sparse column (CSC) format and may optionally supply a pivot order. If no pivot order is supplied, one is computed using the analyse phase of the in core solver MA57. Statistics on the forthcoming factorization (such as the maximum frontsize, the number of flops, and the number of entries in the factor) are also computed. These are exact for the factorization phase of MA57 if the problem is positive definite; otherwise, they are lower bounds for MA57 and estimates of lower bounds for HSL_MA77. By default, the statistics are used to choose the direct solver. The in-core solver (MA57) is selected unless one or more of the following holds:

1. It is not possible to allocate the arrays required for in-core factorization. (We allow the user to specify a maximum amount of memory, and if the predicted memory usage for MA57 exceeds this, we use HSL_MA77).
2. The matrix is positive definite with a maximum frontsize greater than 1500.
3. The matrix is not positive definite and the user-supplied pivot sequence includes 2×2 pivots (as these are only supported by the out-of-core solver HSL_MA77).
4. The user has chosen to use the out-of-core solver (HSL_MA77).

The choice in (2) was made on the basis of numerical experimentation by Reid and Scott [2009b]. The main motivation for selecting MA57 as the default solver is that HSL_MA77 is primarily designed as an out-of-core solver and this incurs an overhead (recall the solve phase is considerably slower than that of MA57). Furthermore, the process of refactorizing in double precision is also more expensive for HSL_MA77 because due to technical reasons it is necessary to reload the matrix data and repeat its analysis phase (this can be avoided for MA57).

At the start of the factorization with the in-core solver, HSL_MA79 allocates the required arrays based on the analyse statistics. If larger work arrays are later needed because of delayed pivots, HSL_MA79 attempts to use the stop and restart facility offered by the in-core solver but, if there is insufficient memory to allocate sufficiently large arrays, HSL_MA79 switches to using the out-of-core code. This may add a significant extra cost as MA77_analyse must be called and the factorization completely restarted.

By default, HSL_MA79 works in mixed precision following Algorithm 17 and its development through the preceding sections. The user may, however, choose to perform the whole computation in double precision. In this case, HSL_MA79 provides a convenient interface to MA57 and HSL_MA77 (albeit without the full flexibility and options offered by each of these packages individually). This facilitates comparisons between mixed and double precision. Working in double precision throughout may be advisable for very ill-conditioned systems or for very large problems for which repeated calls to the solve routine MA79_solve are expected.

HSL_MA79 includes a number of scaling options, provided by the HSL packages MC30 (Curtis and Reid's method minimizing the sum of logarithms of the entries [Curtis and Reid 1972]), MC64 (symmetrized scaling based on maximum matching by Duff and Koster [Duff and Koster 2001; Duff and Pralet 2005]), and MC77 using the 1 or ∞ norms (iterative process of simultaneous norm equilibration [Ruiz 2001]). The default is the ∞ norm equilibration scaling from MC77 based on the results of the previous chapter.

HSL_MA79 offers complete control of the parameters in Algorithms 18 and 19, in addition to the ability to disable any particular method of recovering precision in Algorithm 17 (for example, the user may specify that the use of iterative refinement is to be skipped). It also supports tuning of the major parameters affecting the performance of the factorization phase, such as the block size used by the dense linear algebra kernels that lie at the heart of the multifrontal algorithm.

By default, the requested accuracy is achieved when β given by (5.1) is less than a user-prescribed value γ. However, HSL_MA79 also allows the user to specify, using an optional subroutine argument on the call to MA79_factor_solve, an alternative measure of accuracy. If present, it must compute a function $\beta = f(A, \boldsymbol{x}, \boldsymbol{b}, \boldsymbol{r})$ that is compared to γ when testing for termination. For example, it may be used to test for the requested accuracy in the 2-norm or using a component-wise approach.

An important feature of MA79_factor_solve is that it returns detailed information on the solution process. This includes which solver was used and the precision, together with details of the matrix factorization (the number of entries in the factor, the maximum frontsize, the number of 2×2 pivots chosen, the numbers of negative and zero pivots) and information on the refinement (the number of steps of iterative refinement, the number of FGMRES iterations performed, and the total number of calls to the solution phase of MA57 or HSL_MA77). In addition, the full information type or array from the factorization code (MA57B or MA77_factor) is returned to the user.

We note that the user can pass any number of right-hand sides b to MA79_factor_solve. In particular, the user can set the number of right-hand sides to zero. In this case, the routine will only perform the matrix factorization in the requested (or default) precision.

5.7.2 MA79_refactor_solve

We envisage that a user may want to factorize and solve a series of problems with the same sparsity pattern as the original matrix A but different numerical values. In this case, HSL_MA79 takes advantage of both the pivot sequence used within MA79_factor_solve and the experience gained on the initial factorization. Part of the gain is that we do not expect to need to run the analysis phase of the direct solver a second time, with an additional gain through substantial avoidance of the stop and restart mechanism used by the in-core solver when it runs out of space.

On a call to MA79_refactor_solve, the user must input the values of the entries in both the lower and upper triangular parts of the new matrix in CSC format, with the entries in each column in order of increasing row index. This format (which is the format the original matrix is returned to the user in on exit from MA79_factor_solve) is required so that HSL_MA79 can avoid, before the factorization begins, taking and manipulating additional copies of the matrix (for large problems, this avoidance is important). Having the matrix in this form also has the side benefit of allowing a more efficient matrix-vector product.

5.7.3 MA79_solve

After a call to MA79_factor_solve, MA79_solve may be called to solve for additional right-hand sides. If an in-core solution has been performed, the cost of each additional solve is generally small but, if the factors are held on disk (out-of-core), the solve time can be significant (see [Reid and Scott 2009b]). If the solve is performed at the same time as the factorization, the entries of L are used to perform the forward substitution as they are generated, cutting the amount of data that must be read (and hence the time) for the solve approximately in half.

5.7.4 Errors and warnings

HSL_MA79 issues two levels of errors: fatal errors that cause the computation to terminate immediately and warnings that are intended to alert the user to what could be a problem but which will not prevent the computation from continuing. In either case, a flag is set (with a negative value for an error and a positive value for a warning) and a message is optionally printed (the user controls the level of diagnostic printing). Examples of fatal errors include a user-supplied pivot order that is not a permutation and calls to routines within the HSL_MA79 package that are out of sequence.

A warning is issued if the user-supplied matrix data contains out-of-range indices (these are ignored) or duplicated indices (the corresponding matrix entries are summed). A warning is also issued if the matrix is found to be singular or if in-core solution was requested but out-of-core solution is used because of insufficient main memory. Note that more than one warning may be issued. At the end of the computation a warning is given if the requested accuracy was not obtained after all allowable fall back options were attempted. In particular, if the factorization has been performed in single precision, the requested accuracy may not be achieved on a call to MA79_solve without resorting to a double precision factorization. In this case, because this cannot occur within MA79_solve, the user should call MA79_refactor_solve, explicitly specifying the factorization is to be performed in double precision.

Full details of errors and warnings and of the levels of diagnostic printing are given in the user documentation for HSL_MA79.

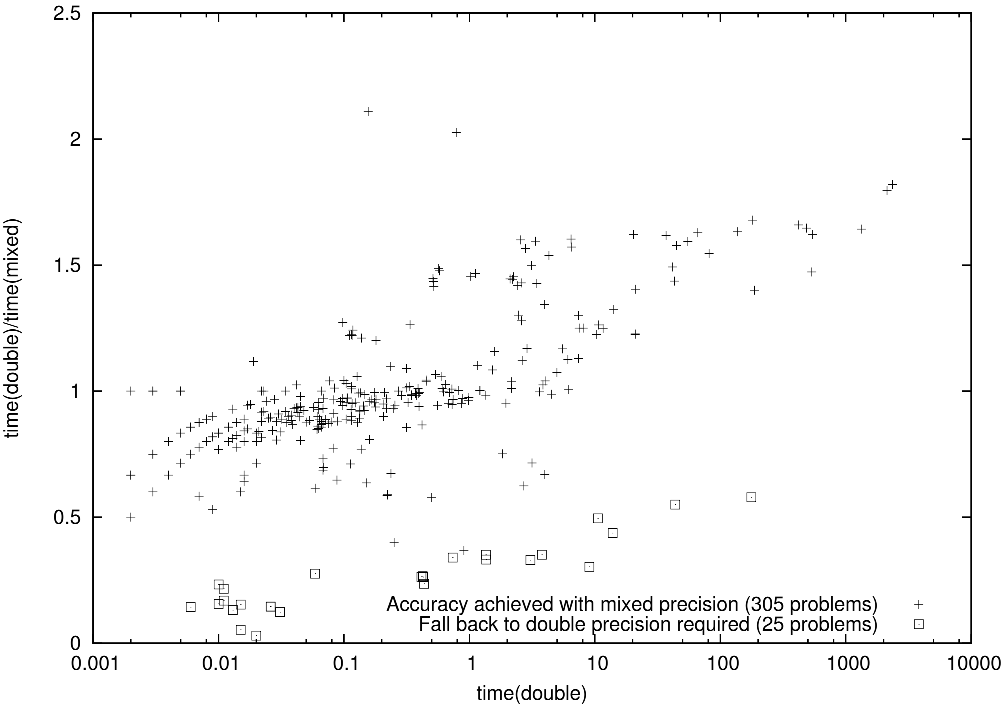


Figure 5.3: Ratio of total solution times with `HSL_MA79` in mixed-precision and double-precision modes on Test Set 2 using `MA57` as the solver, with accuracy $\gamma = 5 \times 10^{-15}$.

5.8 Numerical results

In this section, we present results obtained using Version 1.1.0 of `HSL_MA79`. This uses Version 3.2.0 of `MA57` and Version 4.0.0 of `HSL_MA77`. Our experiments are performed on the machine `fox` (see Section 1.9) using the test sets described previously. The requested accuracy is $\beta < 5 \times 10^{-15}$ and, unless stated otherwise, we use the default settings for all the `HSL_MA79` parameters (in particular, the parameters chosen in Sections 5.5 and 5.6 are used to control the solution recovery).

Figures 5.3 and 5.4 compare the performance of mixed precision and double precision for Test Sets 2 and 3, respectively, with the in-core option selected as the direct solver within `HSL_MA79` for Test Set 2 and out-of-core for Test Set 3. If the requested accuracy is only achieved by falling back on a double precision factorization, the mixed-precision time includes the double-precision factorization time. From Figure 5.3, we see that, if the time taken by `HSL_MA79` in double precision is less than about 1 second, there is generally little or no advantage in terms of computational time in using mixed precision (in fact, for a number of problems, running in double precision is almost twice as fast as using mixed precision). However, for the larger problems within Test Set 2, mixed precision outperforms double precision by more than 50%. For the problems in Test Set 3 with the out-of-core solver `HSL_MA77`, on our test machine mixed precision is only recommended if the double precision time is greater than about 10 seconds. For problems that run more rapidly than this, the savings from the single precision factorization are not large enough to offset the cost of the additional solves (which, in this case, involve reading data from disk). Of course, if the user is prepared to accept a less accurate solution (that is, the requested accuracy γ is chosen to be greater than 5×10^{-15}), this will effect the balance between the mixed-precision time (which will decrease as fewer refinement steps will be needed) and the double-precision time (which, in many instances, will be unchanged).

In Table 5.4, we report the number and percentage of problems in each test set for which the requested accuracy was achieved after iterative refinement both by itself and followed by FGMRES. We also report the number of problems that had to fall back to a double precision factorization to achieve the requested accuracy and the number that failed to achieve

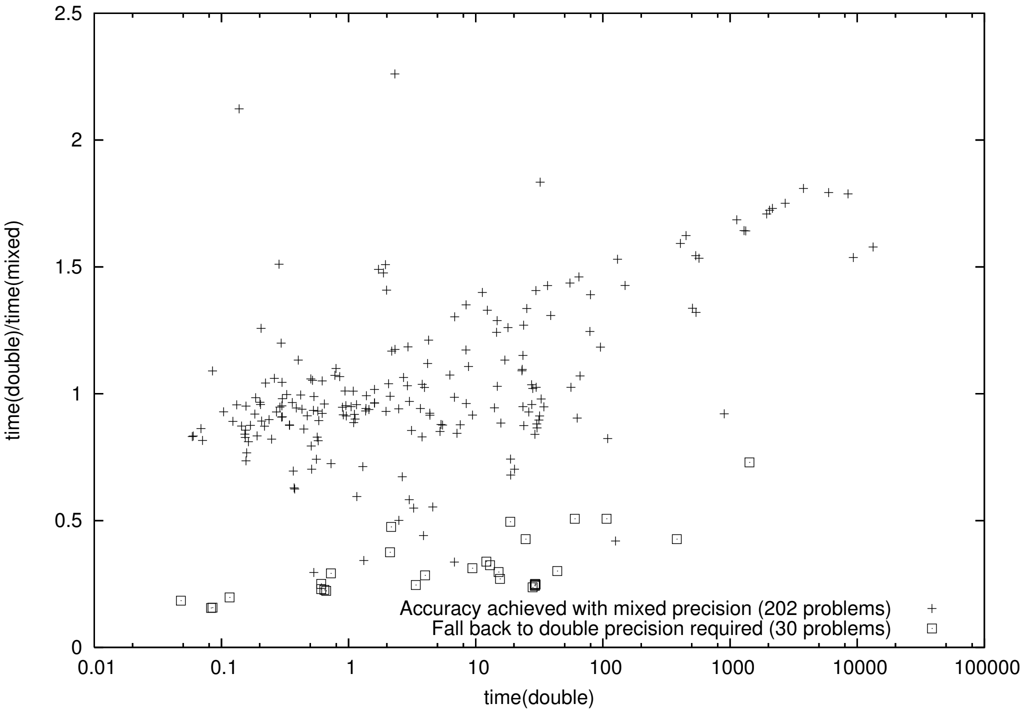


Figure 5.4: Ratio of total solution times with `HSL_MA79` in mixed-precision and double-precision modes on test set 3 using `HSL_MA77` as the solver, with accuracy $\gamma = 5 \times 10^{-15}$.

	Test Set 2		Test Set 3	
	`MA57`		`HSL_MA77`	
After iterative refinement	265	81%	157	68%
After FGMRES	40	12%	45	19%
After fall back to double precision	24	7%	28	12%
Failed	1	<1%	2	1%

Table 5.4: Number of problems that exited at each stage of Algorithm 17 implemented as `HSL_MA79`.

this even in double precision. There were just two such problems: *GHS_indef/boyd1* and *GHS_indef/blockqp1* (these had final scaled residuals of 6.2×10^{-14} and 2.9×10^{-14}, respectively).

As expected, when mixed precision fails to reach the requested accuracy, `HSL_MA79` spends longer establishing this fact than if double precision was used originally. Thus it is essential for a potential user to experiment to see whether the mixed-precision approach will be advantageous for his or her application and computing environment.

It is of interest to consider not only the total time taken to solve the system, but also the times for each phase of the solution process in mixed precision and in double precision. Table 5.5 reports timings for the various phases for a subset of problems of different sizes from Test Set 3. The problems are ordered by the total solution time required using double precision. The time for the analyse phase (which here includes the time to scale the matrix) is independent of the precision.

5.9 Conclusions and future directions

In this chapter, we have explored a mixed-precision strategy that is capable of outperforming a traditional double-precision approach for solving large sparse symmetric linear systems. Building on the recent work of Arioli and Duff [2009] and Buttari et al. [2008], we have designed and developed a practical and robust sparse mixed precision solver; the new package `HSL_MA79` is available within the HSL Library. Numerical experiments on a large number of problems have shown that, in about 90% of our test cases, it is possible to use a mixed-precision approach to get accuracy of 5×10^{-15}; in the remaining cases, it is necessary to resort to computing a double-precision factorization (or to accept a less accurate solution). `HSL_MA79` is designed to allow an automatic fall back to double precision and is tuned to minimize the work performed before this happens. However, although we have demonstrated robustness, our experience is that, in terms of computational time, the advantage of using mixed precision is limited to large problems (how large will depend on the direct solver used within `HSL_MA79`, on the computing platform, and also on the requested accuracy); the user is advised that experimentation with his or her problems will be necessary to decide whether or not to use mixed precision.

Future work on `HSL_MA79` will focus on more efficiently recovering double-precision accuracy in the case of multiple right-hand sides; this will lead to the replacement of the `MI15` implementation of FGMRES with a specially modified variant of FGMRES, which may require a different adaptive restarting strategy. We would also like to use `HSL_MA79` to solve problems that are so large that it is only possible to compute and store a single precision factorization.

Throughout this chapter, we have considered factorizing A in single precision combined with recovery using double precision. However, this does not have to be the case. Provided the condition number of the matrix is less than the reciprocal of the requested accuracy γ, the theory [Arioli and Duff 2009] supports recovery to arbitrary precision. In this case, the refinement must be carried out in extended precision. It is also possible to perform a factorization in double precision and then recover to higher precision. This would be the subject of a separate study.

	Total		Analyse	Factorize		Iterative refinement		FGMRES		β	
Problem	m	d		m	d	m	d	m	d	m	d
HB/bcsstk17	0.30	0.25	0.10	0.11	0.14	0.06 (5)	-	-	-	1.17e-15	1.24e-16
Boeing/crystm03	1.08	1.09	0.64	0.33	0.44	0.09 (3)	-	-	-	3.78e-16	4.80e-16
Boeing/bcsstm39	3.86	3.97	3.66	0.11	0.30	0.09 (2)	-	-	-	2.31e-15	1.28e-16
Rothberg/cfd1	6.22	8.41	2.98	2.48	5.43	0.69 (3)	-	-	-	1.41e-16	2.32e-16
Rothberg/cfd2	11.8	14.6	4.97	5.27	9.62	1.39 (3)	-	-	-	1.69e-16	2.74e-16
INPRO/msdoor	28.7	20.2	5.39	9.64	11.8	2.70 (3)	-	10.1 (8)	-	2.96e-16	2.64e-16
ND/nd6k	29.7	38.8	5.33	20.3	33.1	3.77 (8)	-	-	-	3.67e-15	1.68e-15
GHS_psdef/apache2	61.8	66.1	19.8	29.1	46.4	12.5 (6)	-	-	-	1.79e-16	1.43e-15
Koutsovasilis/F1	63.5	79.1	16.9	36.1	60.1	9.10 (4)	-	-	-	1.75e-15	2.16e-16
Lin/Lin	57.4	79.9	10.8	39.3	66.4	7.26 (5)	2.69 (1)	-	-	3.20e-16	2.03e-16
ND/nd12k	104	149	14.5	78.0	134	11.5 (8)	-	-	-	1.00e-15	2.07e-15
PARSEC/Ga3As3H12	348	537	21.6	302	511	24.30 (8)	5.86 (1)	-	-	3.27e-15	3.37e-16
ND/nd24k	372	570	36.5	297	532	36.5 (9)	-	-	-	1.02e-15	2.94e-15
GHS_indef/sparsine	409	540	18.4	314	517	5.28 (2)	4.92 (1)	70.5 (16)	-	4.20e-16	3.51e-16
PARSEC/Si34H36	783	1287	38.7	706	1236	37.7 (6)	12.1 (1)	-	-	4.91e-16	2.71e-16
GHS_psdef/audikw_1	810	1329	65.7	620	1262	120 (7)	-	-	-	4.67e-15	5.84e-17
PARSEC/Ga10As10H30	1187	2046	51.6	1082	1976	53.3 (6)	18.7 (1)	-	-	2.96e-16	1.96e-16
PARSEC/Si87H76	6058	9310	153	4703	8815	1202 (7)	344 (1)	-	-	6.64e-16	1.95e-16

Table 5.5: Times to solve $A\boldsymbol{x} = \boldsymbol{b}$ for a subset of problems from Test Set 3 with `HSL_MA79` using `HSL_MA77` as the solver. m denotes mixed precision and d denotes double precision. The numbers in parentheses are iteration counts. - indicates iterative refinement (or FGMRES) was not required.

Chapter 6

Scheduling priorities for dense DAG-based factorizations

In this chapter we consider scheduling priorities for a DAG-based Cholesky factorization, and present details of our own implementation. This work has been published as the technical report [Hogg 2008], and the text is partially drawn from that report. The major scientific contribution is to demonstrate that, for Cholesky factorization, complex scheduling schema that take into account the critical path offer only a small performance benefit over simpler methods. Additionally a library quality code, `HSL_MP54`, has been produced, and is now part of the HSL software library [HSL 2007]. It may be obtained from `http://www.hsl.rl.ac.uk/`.

We would like to thank Alfredo Buttari and Jack Dongarra for useful clarifications of their work. Additional helpful comments were received from anonymous referees of the original paper, that is reproduced with kind permission at the rear of this thesis.

6.1 Introduction

The reader is asked to recall the presentation of a parallel DAG-driven Cholesky factorization from Section 2.1.3. As a reminder, we have divided a blocked Cholesky factorization into the following tasks:

factorize(j) Factorize the block A_{jj}. Requires completion of all update(j, j, k) tasks for $k = 1, \ldots, j-1$.

solve(i, j) Perform the triangular solve $L_{ij} \leftarrow L_{ij} A_{jj}^{-T}$. Requires completion of update(i, j, k) tasks for all $k = 1, \ldots, j-1$ and additionally the factorize(j) task.

update(i, j, k) Perform the outer product update to block (i, j) from column k, $A_{ij} \leftarrow A_{ij} - L_{ik} L_{jk}^T$. Requires completion of the tasks solve(i, k) and solve(j, k) (which may be the same task if $i = j$).

In the remainder of this chapter, we shall address the task DAG using standard graph terminology. Each task is represented by a node, with dependencies represented by directed edges. If an edge from node a to node b exists then b is a child of a, with a as a parent of b. A node with no parents is a root and a node with no children is a leaf. A path is an ordered sequence of nodes connected by edges. Node b is a descendant of node a if there exists a path from node a to node b.

6.2 Task dispatch

Algorithm 20 shows pseudo-code for how we might execute a task DAG using a task pool. The task pool contains tasks, with subroutines get_task() and add_task() that allow us to retrieve a task or add a new one. The algorithm retrieves a task, does the necessary work and then

Algorithm 20 Task dispatch code for a simple execution of the task DAG

```
Initialise dep($i, j$) for $1 \leq i \leq j \leq nblk$ to number of parents of corresponding factorize or
solve nodes.
loop
   call get_task(task)
   select case(task)
   case(factorize)
      Perform Cholesky factorization of block $(j, j)$.
      for $i = j + 1, nblk$ do
         dep($i, j$) = dep($i, j$) $- 1$
         if dep($i, j$) $== 0$ then call add_task(solve($i, j$))
      end for
   case(solve)
      $L_{ij} = A_{ij} L_{jj}^{-T}$
      dep($i, j$) $= -2$
         (Flag that this solve has been performed).
      for $k = j + 1, i$ do
         if dep($k, j$) $== -2$ then call add_task(update($i, k, j$))
      end for
      for $k = i + 1, nblk$ do
         if dep($k, j$) $== -2$ then call add_task(update($k, i, j$))
      end for
   case(update)
      $A_{ik} = A_{ij} - L_{ik} L_{jk}^T$
      dep($i, j$) = dep($i, j$) $- 1$
      if dep($i, j$) $== 0$ then
         if $i \neq j$ then
            call add_task(solve($i, j$))
         else
            call add_task(factorize($j$))
         end if
      end if
   end select
end loop
```

updates dependency information of other tasks, adding any for which the dependencies have been satisfied.

The `dep(:,:)` array is initialised to contain the number of tasks that must be executed for a block before we can execute a factorize or solve task as appropriate. Each time a dependency (either an update from the left or the factorization of the diagonal block above) is met we decrement this count, adding the task when there are no more dependencies to satisfy (clearly the dependency count is equal to the number of edges entering the corresponding factorize or solve node in the task DAG).

Once the task solve(i, j) has been placed in the task pool, the element `dep`(i, j) can be reused as a flag to indicate whether that particular solve task has completed. After completion of the task solve(i, j) we scan through the elements of `dep(:,:)` for column j checking which update tasks may now be added to the task pool.

To run this algorithm in parallel (shared memory) we need to prevent data races. A data race occurs when two threads are trying to read or write to the same location at once. We avoid this in our code through the use of locks, as shown in Algorithm 21. We require locks to prevent multiple threads accessing the same `dep(:,:)` variable simultaneously; to minimize time spent waiting for locks we only perform the necessary operations while holding one. After the numerical work for a solve has been completed only one thread should attempt to add updates for any given column at once. Failure to do so could result in a task being generated multiple times. An additional set of locks, one for each off-diagonal block and held in the array update_lock(:, :), is required to prevent more than one thread performing an update operation on a block simultaneously. If a thread is unable to acquire the lock for its target block, it performs the update into a buffer T, that is then added to a list of delayed updates for later application. These updates are applied to the target block as part of the factorize or solve task for that block. A final lock is used in the get_task() and add_task() subroutines to prevent more than one thread modifying the task pool at once; while this could be a critical section, the lock allows the use of more complex control logic.

As the algorithm is described, we can clearly generate a large number of temporary T buffers. The following implementational tricks may be employed to reduce the memory used:

- Instead of generating a new T for each delayed update, attempt to add an existing one for that block. This requires an additional lock to be associated with each update buffer, but limits the number of buffers for any given block to be at most equal to the number of threads minus one.
- Once available memory for temporary buffers is exhausted, wait on the lock for the target block. However, this could result in poor performance if it is a common occurrence.

6.3 Prioritisation

We now consider associating a number with each task that we will refer to as its *schedule*. The subroutine get_task() is modified to always return the task in the pool with the lowest schedule. This allows us to prioritise tasks so as to keep the number of available tasks on the pool large, thus ensuring that processors are rarely idle waiting for tasks.

Buttari et al. [2009] recommend that tasks on the critical path (which they unusually define as the path connecting all nodes with the highest number of outgoing edges: presumably to maximize the number of available tasks) be executed with a high priority. We consider three variations of their scheduling strategy:

- All nodes of the same type have the same priority. Nodes that typically have a higher number of outgoing edges are scheduled first. (i.e. any factorize nodes are scheduled first, then solves, then updates)
- Nodes with a higher number of outgoing edges are scheduled first.
- Nodes with a higher number of descendants are scheduled first.

The first two options are easily realised; for the third option we can calculate the number of descendants using the following recurrence relations

$$\begin{aligned}
\text{descendants(factorize}(j)) &= \#\text{ solves in col } j + \#\text{ updates from col } j + \\
&\quad \text{descendants(factorize}(j+1)) + 1 \\
&= (nblk - j) + \tfrac{1}{2}(nblk - j)(nblk - j + 1) + \\
&\quad \text{descendants(factorize}(j+1)) + 1 \\
\text{descendants(solve}(i,j)) &= \#\text{ updates from block} + \text{descendants(solve}(i, j+1)) + 1 \\
&= (nblk - j) + \text{descendants(solve}(i, j+1)) + 1 \\
\text{descendants(update}(i,j,k)) &= \begin{cases} \text{descendants(factorize}(j)) + 1 & i = j \\ \text{descendants(solve}(i,j)) + 1 & i \neq j, \end{cases}
\end{aligned}$$

that have closed form solutions

$$\begin{aligned}
\text{descendants(factorize}(j)) &= \tfrac{1}{6}(nblk - j)(nblk - j + 1)(nblk - j + 5) + (nblk - j) \\
\text{descendants(solve}(i,j)) &= i\left(nblk - \tfrac{1}{2}i + \tfrac{3}{2}\right) - j\left(nblk - \tfrac{1}{2}j + \tfrac{3}{2}\right) + \\
&\quad \text{descendants(factorize}(i)) \\
\text{descendants(update}(i,j,k)) &= \begin{cases} \text{descendants(factorize}(j)) + 1 & i = j \\ \text{descendants(solve}(i,j)) + 1 & i \neq j. \end{cases}
\end{aligned}$$

Each of these approaches suggested by Buttari et al. potentially fail to prioritise long chains of tasks, possibly causing task starvation at some future point in the execution. For example,

Algorithm 21 Task dispatch code with synchronisations

```
loop
  call get_task(task)
  select case(task)
  case(factorize)
    Apply any updates from update list.
    Perform Cholesky factorization of block $(j, j)$.
    for $j = i + 1, nblk$ do
      Acquire lock$(i, j)$
      dep$(i, j)$ = dep$(i, j) - 1$
      if dep$(i, j) == 0$ then call add_task(solve$(i, j)$)
      Release lock$(i, j)$
    end for
  case(solve)
    Apply any updates from update list.
    $L_{ij} = A_{ij} L_{jj}^{-T}$
    dep$(i, j) = -2$
    Acquire lock$(j, j)$
    for $k = j + 1, i$ do
      if dep$(k, j) == -2$ then call add_task(update$(i, k, j)$)
    end for
    for $k = i + 1, nblk$ do
      if dep$(k, j) == -2$ then call add_task(update$(k, i, j)$)
    end for
    Release lock$(j, j)$
  case(update)
    if Can acquire update_lock$(i, j)$ then
      $A_{ij} = A_{ij} - L_{ik} L_{jk}^T$
      Release update_lock$(i, j)$
    else
      $T = -L_{ik} L_{jk}^T$
      Place $T$ upon update list for block $A_{ij}$.
    end if
    Acquire lock$(i, j)$
    dep$(i, j)$ = dep$(i, j) - 1$
    if dep$(i, j) == 0$ then
      if $i \neq j$ then
        call add_task(solve$(i, j)$)
      else
        call add_task(factorize$(j)$)
      end if
    end if
    Release lock$(i, j)$
  end select
end loop
```

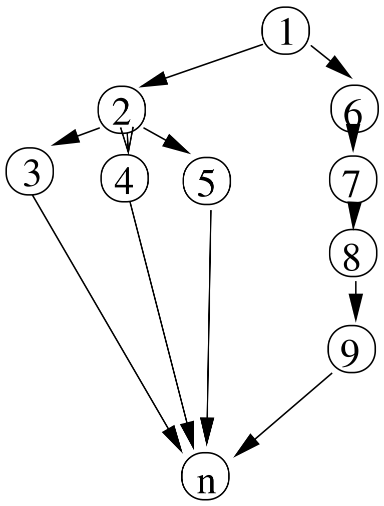


Figure 6.1: Difficult scheduling example

consider Figure 6.1 showing a somewhat artificial example of what could potentially go wrong: picking the task with the largest number of outgoing edges means picking task 2, rather than task 6 which is actually critical. Further, due to lack of tie breaking this could then be followed by execution of tasks 3,4 and 5 before returning to the chain 6, 7, 8, 9, ... n. This leads us to consider alternate strategies with a global view motivated by a more conventional definition of critical path.

In this thesis, we define the critical path as the longest path from a root node to a leaf node. This represents the shortest sequence of events that must be executed in serial if an infinite number of processors is available and all tasks take the same amount of time (though the latter of these assumptions is not generally the case and will later be relaxed). Figure 6.2 shows the critical path for our 4×4 example marked on the graph as the bold edges.

If we continue our assumptions that all tasks take an equal amount of time and that we have sufficient processors to complete the critical path scheduling on time, then we can consider each task to occupy a discrete time slice, numbered from 1. Our schedule number is derived as the latest time slice in which a task can be scheduled so that we can still meet the critical path. The numbering on Figure 6.2 shows such a scheduling for our 4×4 example.

In general the critical path is unique and consists of all the factorize tasks and the shortest path between them, that is, the tasks solve$(j, j-1)$ and update$(j, j, j-1)$ for $j = 1, \ldots, nblk$. We can thus derive the schedules for these tasks as

$$\begin{array}{lcll} \text{schedule}(\text{solve}(j, j-1)) & = & 3j-4 & j = 2, \ldots, nblk \\ \text{schedule}(\text{update}(j, j, j-1)) & = & 3j-3 & j = 2, \ldots, nblk \\ \text{schedule}(\text{factorize}(j)) & = & 3j-2 & j = 1, \ldots, nblk. \end{array}$$

Using these we are able to formulate a recurrence relation for non-critical tasks

$$\begin{array}{lcl} \text{schedule}(\text{update}(i, j, j)) & = & \begin{cases} \text{schedule}(\text{solve}(i, j)) - 1 & i \neq j \\ \text{schedule}(\text{factorize}(k)) - 1 & i = j \end{cases} \\ \text{schedule}(\text{solve}(i, j)) & = & \min\left(\min\limits_{k=j+1,i} \left[\text{schedule}(\text{update}(i, j, k))\right], \right. \\ & & \left. \qquad \min\limits_{k=i+1,nblk} \left[\text{schedule}(\text{update}(k, i, j))\right] \right) - 1. \end{array}$$

These equations are satisfied by the following closed form solutions:

$$\begin{array}{lcl} \text{schedule}(\text{factorize}(j)) & = & j + 2j - 2 \\ \text{schedule}(\text{solve}(i, j)) & = & i + 2j - 2 \\ \text{schedule}(\text{update}(i, j, k)) & = & i + 2j - 3. \end{array}$$

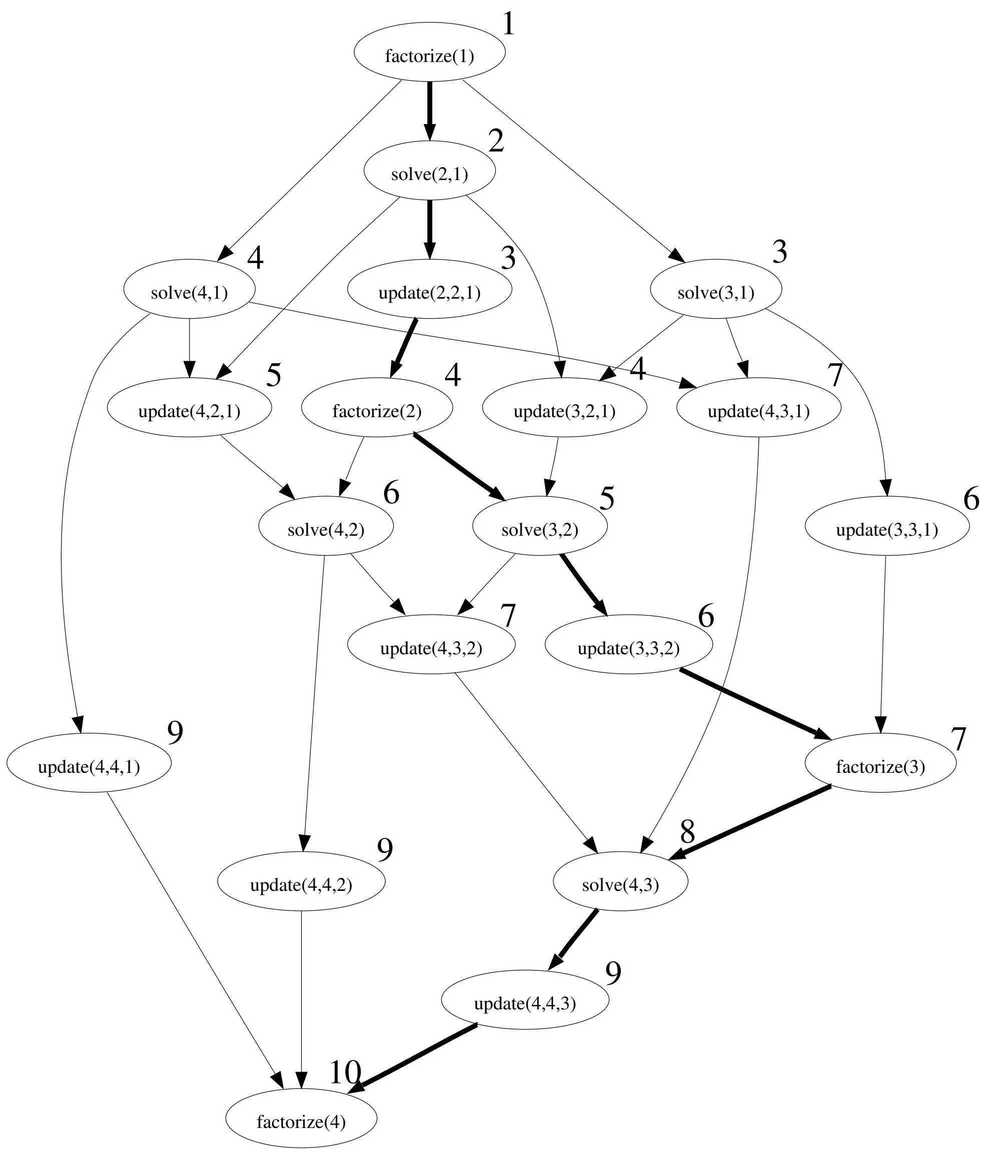


Figure 6.2: Scheduling for a 4 × 4 Cholesky factorization.

If we now consider the case where we have insufficient processors to meet the critical path (i.e., we cannot schedule tasks in such a way that the final task finishes in the time slot our analysis has given it), will this schedule give a good prioritisation? Not always. Consider a large collection of tasks that do not lie on the critical path, but require the completion of a number of dependencies before they become available. The dependencies for these tasks are not prioritised and, as a result, few tasks are available early in the execution sequence. However, due to the regular nature of a dense Cholesky factorization, we believe that the above schedule gives a near optimal scheduling provided that a sufficiently small block size is used.

We also consider relaxing the assumption that all tasks take the same amount of time. We assume all blocks are of size nb and consider the flop counts to be as follows:

$$\begin{array}{lcl} \text{flops(factorize}(j)) & = & \frac{1}{3}nb^3 + O(nb^2) \\ \text{flops(solve}(i,j)) & = & nb^3 \\ \text{flops(update}(i,j,k)) & = & \left\{ \begin{array}{cc} 2nb^3 & i \neq j \\ nb^3 & i = j. \end{array} \right. \end{array}$$

We hence give the factorize, solve and update tasks weights of $1, 3$ or 6 as appropriate. If we make no assumption on the critical path, this leads us to the recurrence relation

$$\begin{array}{lcl} \text{schedule(factorize}(j)) & = & \min\limits_{i=j+1,nblk} (\text{schedule(solve}(i,j))) - 1 \\ \text{schedule(solve}(i,j)) & = & \min \left(\min\limits_{k=j+1,i} [\text{schedule(update}(i,k,j))] \right., \\ & & \left. \min\limits_{k=i+1,nblk} [\text{schedule(update}(k,i,j))] \right) - 3 \\ \text{schedule(update}(i,j,k)) & = & \left\{ \begin{array}{ll} \text{schedule(solve}(i,j)) - 6 & i \neq j \\ \text{schedule(factorize}(j)) - 3 & i = j. \end{array} \right. \end{array}$$

With these new relations, we observe that there is no longer a single unique critical path. Instead, we identify the set of tasks belonging to the union of all critical paths as factorize(1), factorize($nblk$), all solves and updates of the form update($i, j+1, j$). This represents updating the whole of the next column as soon as possible. The reason this is so different from the critical path of the previous scheme is that the process of updating and factoring the diagonal is faster than that of updating the blocks below the diagonal required for the solve tasks. A consequence of this is that all solves are on the critical path as they are required for these updates. A special case is required when dealing with the final column as it has no solve tasks: instead the factorize task becomes critical.

The following formulae represent an explicit solution of the above

$$\begin{array}{lcl} \text{schedule(factorize}(j)) & = & \left\{ \begin{array}{ll} 9(j-1)+1 & j \neq nb \\ 9(j-1)-1 & j = nb \end{array} \right. \\ \text{schedule(solve}(i,j)) & = & 9(j-1)+2 \\ \text{schedule(update}(i,j,k)) & = & \left\{ \begin{array}{ll} 9(j-1)-4 & i \neq j \\ 9(j-1)-2 & i = j \neq nb \\ 9(j-1)-4 & i = j = nb. \end{array} \right. \end{array}$$

We observe that we may use the general formula in the special cases $j = nb$ as this does not affect the relative ordering, and thus the task scheduling.

6.4 Implementation

We implemented the algorithm described in the previous sections as the code `HSL_MP54` in the `HSL` software library [HSL 2007]. It is written in Fortran 95 and has been primarily designed to replace the current partial factorization kernel (`HSL_MA54`) used in that library that uses fork-join parallelism. The OpenMP 2.5 standard is used to implement our shared memory parallelism.

In addition to a standard (complete) Cholesky factorization, our code also supports a less common but related problem that arises as a dense subproblem in sparse multifrontal matrix factorizations. This is the partial factorization

$$A = \begin{pmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & \\ L_{21} & I \end{pmatrix} \begin{pmatrix} I & \\ & S_{22} \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T \\ & I \end{pmatrix}$$

where the columns of A have been partitioned into two sets, one of which is fully factorized and the other is updated by this factorization. The corresponding forward and backward substitutions are

$$\begin{pmatrix} L_{11} & \\ L_{21} & I \end{pmatrix} \boldsymbol{y} = \boldsymbol{b} \text{ and } \begin{pmatrix} L_{11}^T & L_{21}^T \\ & I \end{pmatrix} \boldsymbol{x} = \tilde{\boldsymbol{y}}.$$

Clearly these operations can be performed using a variant of the algorithm we have so far described: omission of the relevant factorize task upon reaching the second set of columns is sufficient. As our scheduling still prioritises factorizing the diagonal elements and solving we do not see any need to rework the schedules for this case, and indeed the surfeit of update tasks will allow us to smooth the tail of the computation.

Our code uses a job queue paradigm to allow multiple instances of `HSL_MP54` to be run using the same team of threads if multiple independent jobs are available. Each factorization or forward/back solve is considered as a job, which is itself composed of many tasks such as those described in the previous sections. We have subroutines that place jobs in the job queue, but do no work on it, and a work subroutine that actually performs these jobs. The current state of the job queue and associated task pool are stored in a shared variable of type `mp54_keep`. Each thread can join a pool of workers at any stage in a job, picking up the next available task when it joins. This ability allows easy load balancing to be achieved at a level above that of our code, for example in a tree-based parallel factorization of a sparse matrix.

We enforce the condition that for each job queue, no job may start before its predecessor has completed. This simplifies the implementation and prevents naive users from introducing data races.

Internally we use a blocked hybrid form similar to that of Anderson et al. [2005] for storing the data in a BLAS-compatible fashion while still using near minimal storage. The user's data is by default rearranged from a lower packed format to this format as an additional task within the factorization. Any columns that are not pivoted upon in a partial factorization are rearranged back to lower packed format, enabling the user to perform manipulations easily. For example, assembly in to other matrices in the case of sparse multifrontal factorizations.

Solves have also been written in a task DAG fashion and use the same blocking as the factorization. However, as the balance between computation and data movement is different these are not as efficient as if a smaller block size were used. We expect the overheads of a rearrangement to such a format would outweigh any performance gains from doing so. If a large number of solves are required then it may be worthwhile; we have not pursued this avenue of research.

Our code implements two simple performance enhancements over the Algorithm of Section 6.2. Firstly, a thread will keep a task for itself if it would otherwise become the task in the pool with the highest priority. This improves cache efficiency as the task will reuse some of the data from the current task. Secondly, as discussed in Section 6.2, we limit the amount of space required for the delayed update by looking for an existing update to the same block; if we find one we apply the update to it.

We note that for small ($n < 2000$) matrices, tuning of our code proved difficult due to context switching of our codes by the host operating system. Further, because of overheads inherent in communication, our code runs slower in parallel than in serial for very small matrices ($n < 200$ on our test machine). We were able to increase the performance in these circumstances by careful tuning of the get_task() routine so only one thread is ever spinning while waiting for tasks to appear in the queue. Other threads wait on a lock, making use of the far more efficient spinlocking mechanism provided by the OpenMP implementation. This seems to have helped reduce the number of cache misses on the threads doing actual work caused by use of the flush directive on idle threads.

n	Simple	Max Child		Max Descendants		Fixed-CP		Weighted-CP	
100	1.6	1.6	(0%)	1.6	(0%)	1.6	(0%)	1.6	(0%)
200	6.1	6.1	(0%)	5.9	(-4%)	6.5	(7%)	6.1	(0%)
300	10.0	10.1	(1%)	10.1	(1%)	10.6	(6%)	10.6	(6%)
400	13.7	13.7	(0%)	14.3	(4%)	14.5	(6%)	14.1	(3%)
500	16.9	17.0	(1%)	17.4	(3%)	18.1	(7%)	17.7	(5%)
1000	28.6	28.6	(0%)	28.7	(0%)	29.7	(4%)	29.1	(2%)
1500	35.9	35.8	(0%)	35.3	(-2%)	35.8	(0%)	35.2	(-2%)
2000	40.5	40.7	(0%)	39.7	(-2%)	39.6	(-2%)	40.2	(-1%)
2500	43.3	43.9	(1%)	42.6	(-2%)	42.5	(-2%)	43.5	(0%)
5000	53.4	53.5	(0%)	51.8	(-3%)	52.4	(-2%)	54.4	(2%)
10000	61.5	61.6	(0%)	59.1	(-4%)	60.4	(-2%)	61.9	(1%)
20000	64.8	64.3	(-1%)	63.7	(-2%)	65.0	(0%)	65.5	(1%)

Table 6.1: Comparison of different scheduling for complete factorization of matrices on 8 threads. Speed is measured in Gflop/s on `fox` (peak dgemm performance 72.8 Gflops/s), and percentages in brackets show performance gain over the Simple strategy.

6.5 Numerical results

We present results from two machines: `fox` and `HPCx` (recall Table 1.4). On `fox` we use the Intel Fortran compiler v10.1 with options `-fast -openmp`. On `HPCx` we use the IBM xlf Fortran compiler v10.1. Due to limited access, most of our results are reported on `fox`. For each set of parameters (code, scheduling variant, order of matrix `n`, and number of threads `nthread`) we experimented to find the optimal block size; the reported results are for these optimal choices. We note that the performance was not overly sensitive to this, see Section 6.5.3.

6.5.1 Choice of scheduling technique

We consider results for the five different scheduling methods discussed in Section 6.3. We shall refer to them by the following names

Simple All tasks of the same type have the same priority (method of Buttari et al.)

Max Child Tasks are prioritised by their number of children. The node with the largest number of children goes first.

Max Descendants Tasks are prioritised by their number of descendants. The node with the largest number of descendants goes first.

Fixed-CP Tasks are scheduled according to the critical path assuming all tasks take the same amount of time.

Weighted-CP Tasks are scheduled according to the critical path with weightings based on their operation counts.

Table 6.1 shows the average results for the complete factorization of a random diagonally-dominant matrix. We believe the averages to be reliable to within 0.2 Gflop/s. They were obtained by running factorizations until we had accumulated at least 1 second worth of computation and then repeating this until the average was determined. However, we note that performance for methods Simple and Max Child had a much higher variation that other methods. We believe this to be due to the lack of tie breaking for update tasks and the consequent highly random execution sequence.

While it is disappointing that there is little difference between results, Weighted-CP seems to give the best results on large problems; however Max Child and Fixed-CP do better in the range $n = 1500$ to $n = 2500$. Following profiling for problems in this range we have determined that Max Child has many fewer updates that are delayed, resulting in less work than for Weighted-CP. This seems to be caused by a more random ordering of updates for Max Child: Weighted-CP gives all updates to the same block the same schedule, resulting in more collisions

n	Factor memory	Simple	Max Child	Max Descendants	Fixed-CP	Weighted-CP
1000	3,911	864	720	1,775	2,028	3,211
2500	24,424	3,281	4,374	6,926	6,561	6,561
5000	97,676	9,032	9,935	12,644	11,741	12,644
10000	390,665	14,306	18,207	29,912	16,907	23,409

Table 6.2: Comparing factor storage with maximum observed memory used for delayed updates (`fox`, 8 threads, kilobytes).

n	1 thread	2 threads		4 threads		8 threads	
	Gflop/s	Gflop/s	Speedup	Gflop/s	Speedup	Gflop/s	Speedup
500	5.6	8.6	1.5	13.4	2.4	17.7	3.2
1000	6.8	11.3	1.7	20.1	3.0	29.1	4.3
2500	7.6	14.5	1.9	26.9	3.5	43.5	5.7
5000	8.3	15.2	1.8	31.1	3.7	54.4	6.6
10000	8.6	17.1	2.0	33.6	3.9	61.9	7.2
20000	8.8	17.7	2.0	35.1	4.0	65.5	7.4

Table 6.3: Scalability of `HSL_MP54` on a range of problem sizes using `fox` (max dgemm performance 9.3/72.8 on 1/8 threads).

that generate delayed updates. Fixed-CP does better than Weighted-CP on smaller problems, for which we offer the hypothesis that our choice of weighting is poor for small block sizes, where much of the time goes into the memory load rather than the floating-point operations. We note however that even on these small problems we saw evidence of Max Child causing task starvation mid-factorization due to a poor prioritisation of tasks. For larger problems the design assumptions of Weighted-CP are more accurate and the time spent in application of updates becomes negligible compared to other operations.

Table 6.2 shows the observed worst case memory usage for storage of our delayed updates. It confirms that in practice this does not become excessive. If we wished to minimize it, then we should choose either the Simple or Max Child schemes. As we are primarily concerned with speed for large matrices rather than memory usage, the comparisons in the remainder of this chapter shall use the Weighted-CP schedule (though on large machines it may be preferable to prefer memory efficiency instead!).

6.5.2 Scalability

Tables 6.3 and 6.4 demonstrate the scalability of `HSL_MP54`. We observe good scaling on large problems, however smaller problems do not scale quite so well. This is because the efficiency of the level 3 BLAS routines decreases with block size. In order to keep all threads fed for small problems on many threads a significantly smaller block size must be used than on a single thread, giving a lower efficiency than we would otherwise expect.

n	1 thread	2 threads		4 threads		8 threads		16 threads	
	Gflop/s	Gflop/s	Speedup	Gflop/s	Speedup	Gflop/s	Speedup	Gflop/s	Speedup
500	3.2	5.4	1.7	8.5	2.7	11.3	3.5	11.8	3.7
1000	4.0	7.2	1.8	12.9	3.2	19.6	4.9	25.7	6.4
2500	4.6	8.8	1.9	16.2	3.5	28.0	6.1	46.2	10.0
5000	4.8	9.4	2.0	18.1	3.8	32.0	6.7	53.4	11.1
10000	4.9	9.8	2.0	19.3	3.9	36.7	7.5	65.5	13.4
20000	4.9	-[1]	-	-[1]	-	39.3	8.0	69.9	14.2

[1] These results are not available due to time budget constraints

Table 6.4: Scalability of `HSL_MP54` on a range of problem sizes using `HPCx` (max dgemm performance 5/70.4 on 1/16 threads).

n	nb	$nblk$
100	100	1
200	32	7
300	40	8
400	40	10
500	56	9
1000	104	10
1500	120	13
2000	184	12
2500	216	15
5000	340	15
10000	408	25
20000	968	21

Table 6.5: Optimal block sizes used for Weighted-CP using 8 threads on `fox`.

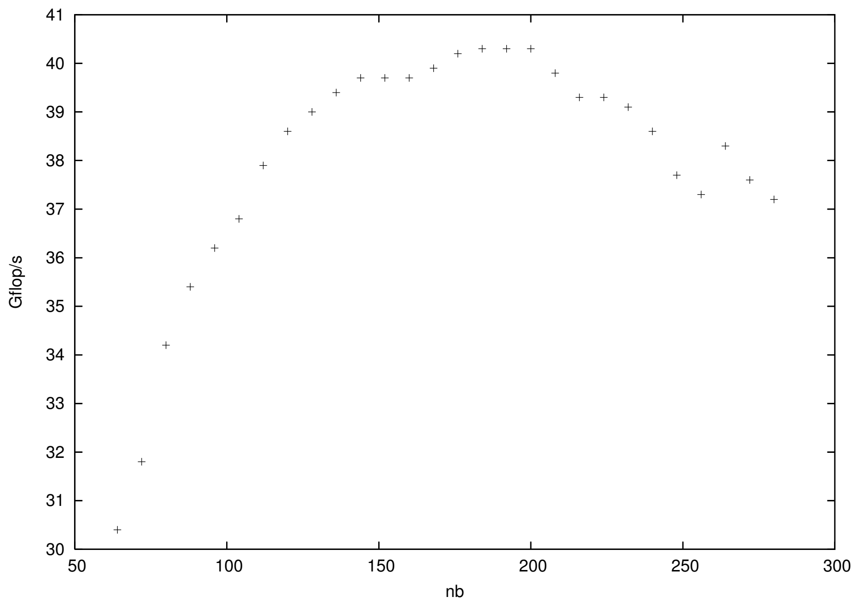


Figure 6.3: Performance varying with nb for $n = 2000$ on 8 threads on `fox`.

6.5.3 Block sizes

Table 6.5 shows the block sizes used on `fox` to achieve the results shown previously for the Weighted-CP schedule. We found that optimal block sizes are always multiples of 8, which we observe is the number of double precision numbers per cache line: this helps avoid *false sharing*, where a line of cache is being used by two different cores at the same time resulting in the cache line being repeatedly invalidated (when the other core writes to it) and reread from memory.

The optimal block size seemed not to vary between the different schedules, and was not very sensitive as long as nb was a multiple of 8, as is shown by Figure 6.3 for the case $n = 2000$.

Table 6.6 shows how the optimal nb varies with the number of threads in use. Based on these results we recommend that the user aims to have $nblk$ between 15 and 20 to estimate a near optimal value of nb for large problems on 8 threads. For a larger number of threads a slightly larger number of blocks should be used, as growth in the number of tasks is cubic in $nblk$.

6.5.4 Comparison with other codes

Figure 6.4 compares the performance of our code with the following codes:

HSL_MA54 Left-looking Cholesky factorization code with fork-join parallelism of loops (ver-

n	1 thread		2 threads		4 threads		8 threads	
500	200	(3)	104	(5)	72	(7)	56	(9)
1000	200	(5)	156	(7)	128	(8)	104	(10)
2500	400	(7)	340	(8)	264	(10)	216	(12)
5000	400	(13)	480	(11)	400	(13)	340	(15)
10000	400	(25)	480	(21)	400	(25)	408	(25)

Table 6.6: Optimal block sizes for varying number of threads and n, with *nblk* shown in brackets, using `fox`.

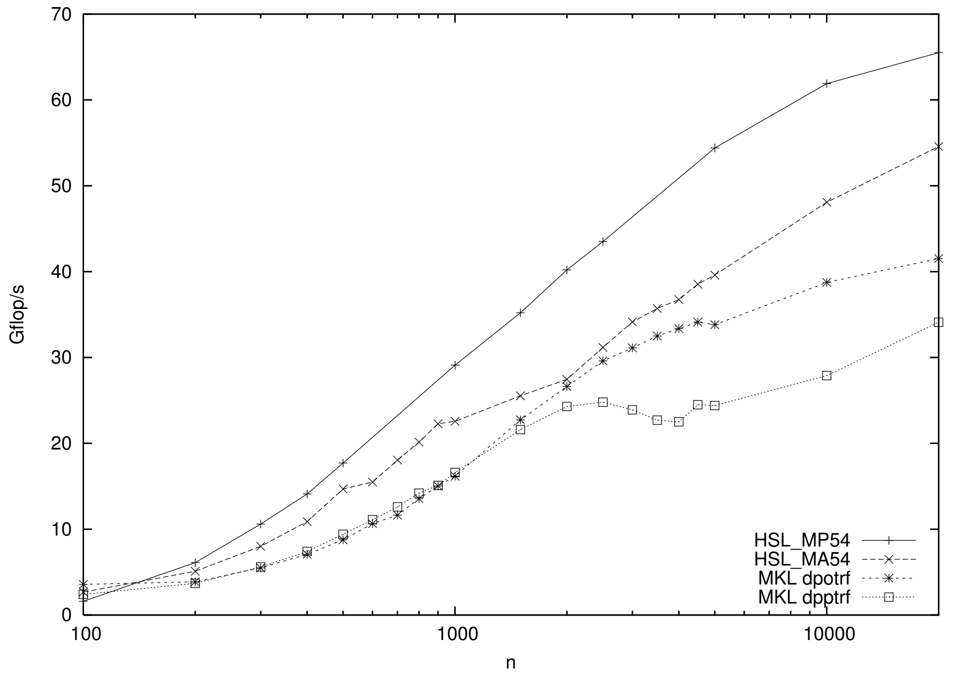


Figure 6.4: Performance varying with size of matrix n for complete factorizations, using 8 threads on `fox` (`dgemm` peak 72.8Gflop/s).

sion 1.3.0).

HSL_MP54 The DAG-based code described in this chapter.

MKL dpotrf Intel Math Kernel Library 10.0.1.014 full storage Cholesky factorization.

MKL dpptrf Intel Math Kernel Library 10.0.1.014 packed storage Cholesky factorization.

Clearly for matrices of nearly any size `HSL_MP54` offers the best performance on 8 threads. The exception to this is $n = 100$ when it is faster to factorize the matrix in serial due to caching issues and communication overheads.

Figure 6.5 shows a similar comparison on `HPCx`. Although we have not performed any tuning for this architecture beyond selecting good block sizes (which do not substantially differ from those on `fox`) we still get good performance, though the comparison with the vendor implementations of dpptrf and dpotrf are not quite so favourable except on small matrices, with all implementations close to peak performance on large matrices.

6.6 Conclusions

`HSL_MP54` is a Cholesky code that performs well on multicore machines, and has been found to be a good kernel for sparse multifrontal factorizations, if a different kernel is used for small front sizes. However, care must be taken with small front sizes, possibly using a different factorization kernel, or exploiting parallelism at a higher level.

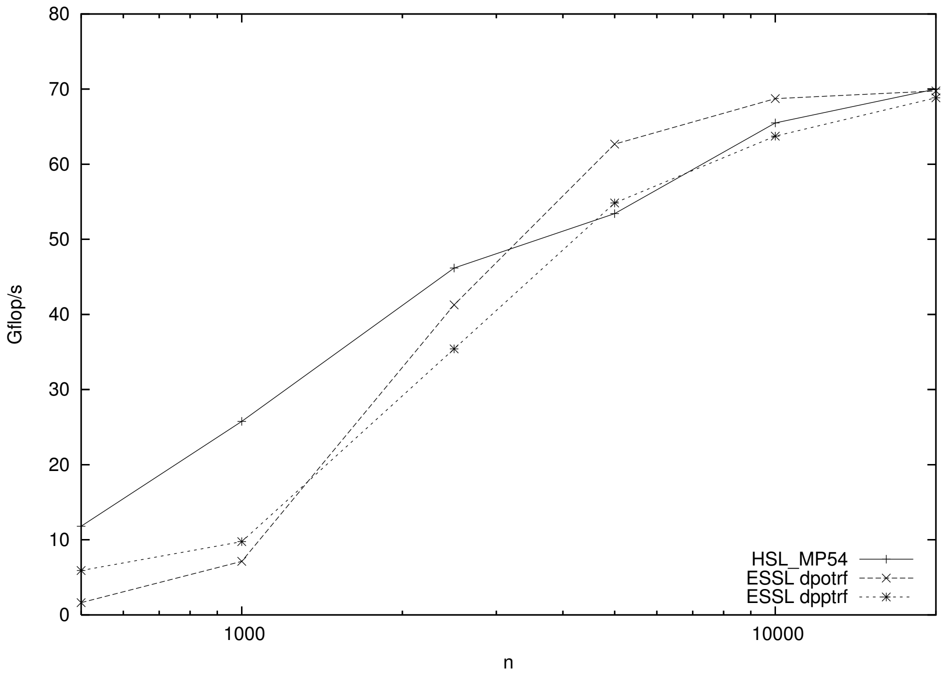


Figure 6.5: Performance varying with size of matrix n for complete factorizations, 16 threads on `HPCx` (`dgemm` peak 70.4Gflop/s).

We have shown that the effect of different scheduling schemes in DAG-based Cholesky factorizations makes only a small difference, at most 7%. However, if the extra performance is considered worthwhile, a scheme that takes the critical path should be used. Further, the comparative times for different tasks must be taken into account when determining the critical path. The use of a delayed updating mechanic leads to good performance for an additional memory overhead of only 5% in the best performing case.

Future improvements to the scheduling may aim to reduce the number of threads attempting to update a single target block at once during update tasks, and may focus more on being cache efficient. The meaning of cache efficient may however change if all or many cores are sharing the same on-chip cache at some level. See for example our modified task handling scheme of Chapter 7.

It may also be of interest to replace our task handling mechanism with that of OpenMP 3.0 tasks or Intel Thread Building Blocks.

Chapter 7

A DAG-based sparse solver

In this chapter we apply the DAG techniques from the dense Cholesky factorization to the equivalent sparse factorization. Achieving a worthwhile speed-up requires some exploitation of novel data structures and the careful specification of an additional task type.

The work has been published as a joint article with Jennifer Scott and John Reid [Hogg et al. 2010], and has been published in the SIAM Journal of Scientific Computing. The author was responsible for the core numerical factorization routines, while Scott and Reid provided a version of their analyse routine from `HSL_MA77` customised to use the new data structures. Scott developed a library-quality interface and provided extensive testing. Although the text of the paper was produced jointly, it was extensively revised by Reid for clarity. The major scientific contribution of this chapter is the extension of the dense DAG-based scheme to the sparse case and the careful implementation of this approach for multicore architectures. It is hoped that by such an explicit task-based casting developments and libraries used for dense linear algebra can be easily carried over into the sparse world. During the review process for this paper the similarities of our scheme to that of PaStiX were raised, and these have been addressed in Section 7.6.

The resultant article is reproduced at the rear of this thesis by permission of the copyright holders. The software produced is `HSL_MA87` that will be part of the HSL software library [HSL 2007]. It may be obtained by contacting the Numerical Analysis group at STFC (hsl@stfc.ac.uk).

We build on the work of Chapter 6, where a dense DAG-driven Cholesky code was developed. However the sparse DAG-based factorization code described in this chapter is a substantial rewrite and improvement of the dense code `HSL_MP54`. In particular we have made substantial changes to how tasks are scheduled and use a different data storage format that suits the new tasks from the sparse case.

7.1 Solver framework

The reader will recall from Section 2.4 that a sparse solver is split into four phases: preorder, analyse, factorize and solve. For the purposes of this chapter we will assume a suitable preordering has already been obtained. The following two subsections will describe the modifications to the analyse and solve phases, while the factorize phase is the subject of the remainder of this chapter.

7.1.1 Analyse

The analyse phase of `HSL_MA87` is a modification of that of `HSL_MA77`. The main purpose of the latter is to take the user-supplied pivot sequence and use it to determine the assembly tree, amalgamating nodes into supernodes to achieve good performance in the factorize phase (see Section 2.4.2).

Before modification the analyse phase of `HSL_MA77` was designed to run out of core. For our new solver this was modified to work in core and to use the new data structures described in

Section 7.2 when setting up the storage for L. For example, block dependency counts are easily determined at this stage. A restricted reordering was added to improve cache locality, and this is described in Section 7.5.

7.1.2 Solve

The solve phase takes one or more right-hand sides $\boldsymbol{b}$ and performs forward and back substitutions using the permutation P and factor L to solve the systems $L\boldsymbol{y} = P\boldsymbol{b}$ and $L^T P^{-1}\boldsymbol{x} = \boldsymbol{y}$. This phase is normally limited by how fast the factor data can be read from main memory.

We currently implement this phase in serial, although parallelisation could be achieved along similar lines to those used in Chapter 6. Our initial experiments along these lines revealed that a very limited speedup was possible due to saturation of the bandwidth to main memory. A full investigation of this phenomenon and possible solutions are beyond the scope of this thesis, however it is addressed in a recent technical report by Hogg and Scott [2010b]. They report that a single core is sufficient to saturate the memory bandwidth available to a full quad-core processor, and as solve is a memory-bound operation this limits the speedup to little more than the number of processor sockets in use. A range of approaches including compression are implemented, but offer little practical gain in performance.

7.2 Nodal data structures

Since a node of the assembly tree represents a set of contiguous columns of L with the same (or nearly the same) sparsity structure below a dense (or nearly dense) triangular submatrix, we can hold it in memory as a dense trapezoidal matrix, as illustrated in Figure 7.1(a). We refer to this matrix as the *nodal matrix*. We store this matrix using the row hybrid blocked structure of Anderson et al. [2005] with the modification that "full" storage is used for the blocks on the diagonal rather than storing only the actual entries (thus a rectangular array is used to store the trapezoidal matrix). Using the row hybrid scheme rather than the column hybrid scheme facilitates updates between nodes by removing any discontinuities at row block boundaries. Storing the blocks on the diagonal in full storage allows us to exploit efficient BLAS and LAPACK routines. This structure is illustrated in Figure 7.1(b). Note that the final block on the diagonal is often trapezoidal.

If the number of columns in the nodal matrix is large, we use the block size nb specified through a control parameter and most of the blocks will be of size $nb \times nb$. We divide the computation into tasks in which a single block is revised (details in Section 7.3). These tasks correspond to the vertices of our implicitly-held DAG.

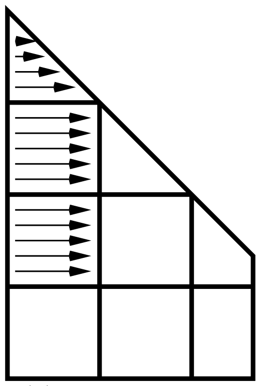
If the number of columns nc in the nodal matrix is less than nb but the number of rows is large, using the block size nb can lead to small tasks and inefficient execution. We therefore attempt to balance the number of entries in the blocks by basing the block size on the value nb^2/nc. We round the size up to a multiple of the cache line size since our experience is that this enhances performance. Having block sizes that differ from node to node is unusual, but our experience is that for some problems it gives a 5–10% performance improvement for little extra complication.

Finally it is worth commenting that row-wise storage can be problematic if there is a need to access data by columns, such as when pivoting. A further discussion on this is presented in Section 8.1.

7.3 Tasks

We depart from the $(row, column)$ block nomenclature used in the dense case of Chapter 6. Instead each block is given a unique number that can be readily determined from the triplet $(node, row, column)$. We recast the dense tasks, together with the new *update_between* task as follows (and illustrate them graphically in Figure 7.2):

factorize(`diag`) Computes the traditional dense Cholesky factor L_{diag} of the triangular part of a block `diag` that is on the diagonal using the LAPACK subroutine `_potrf`. If the


(a) Graphical view

$$\begin{array}{ccc|cc}
1 & & & & \\
4 & 5 & & & \\
7 & 8 & 9 & & \\
\cline{1-3}
10 & 11 & 12 & 25 & \\
13 & 14 & 15 & 27 & 28 \\
16 & 17 & 18 & 29 & 30 \\
\hline
19 & 20 & 21 & 31 & 32 \\
22 & 23 & 24 & 33 & 34
\end{array}$$

(b) Indices of entries

Figure 7.1: Row hybrid block structure for a nodal matrix.

block is trapezoidal, this is followed by a triangular solve of its rectangular part,

$$L_{rect} \Leftarrow L_{rect} L_{diag}^{-T},$$

using the BLAS subroutine `_trsm`.

solve(`dest, diag`**)** Performs a triangular solve of the off-diagonal block `dest` by the Cholesky factor L_{diag} of the block `diag` on its diagonal. i.e.

$$L_{dest} \Leftarrow L_{dest} L_{diag}^{-T},$$

using the BLAS subroutine `_trsm`.

update_internal(`dest, rsrc, csrc`**)** Within a nodal matrix, performs the update

$$L_{dest} \Leftarrow L_{dest} - L_{rsrc} L_{csrc}^{T},$$

where L_{dest} is the matrix of the block `dest`, L_{rsrc} is the matrix of the off-diagonal block `rsrc` with rows that correspond to the rows of L_{dest}, and L_{csrc} is the matrix of those rows of the off-diagonal block `csrc` that correspond to the columns of L_{dest}, see Figure 7.2(c). If `dest` is an off-diagonal block, we use the BLAS 3 kernel `_gemm` for this. If `dest` is a block on the diagonal, we use the BLAS 3 kernel `_syrk` for the triangular part and `_gemm` for the rectangular part, if any.

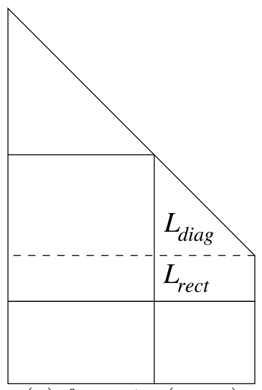
update_between(`dest, snode, scol`**)** Performs the update

$$L_{dest} \Leftarrow L_{dest} - L_{rsrc} L_{csrc}^{T},$$

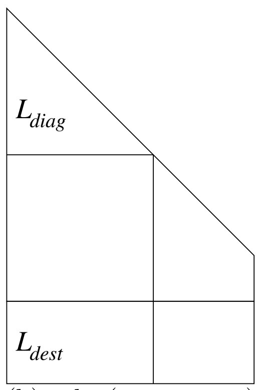
where L_{dest} is a submatrix of the block `dest` of an ancestor of the node `snode` and L_{rsrc} and L_{csrc} are submatrices of contiguous rows of the block column `scol` of the node `snode`. These blocks are uniquely determined by the destination block `dest`. Unless the number of entries updated is very small, we exploit the BLAS 3 kernel `_gemm` (and/or `_syrk` for a block that is on the diagonal) by placing its result in a buffer from which we add the update into the appropriate entries of the destination block `dest`, see Figure 7.2(d).

Observe that unlike in the dense case, there are multiple leaves to the task DAG (and potentially multiple roots): one for each leaf of the assembly tree.

We could have cast update_between as an operation from a pair of blocks, but this would cause the same destination block to be updated more than once from the same block column. This is undesirable since contested writes cause more cache misses than contested reads (a write may invalidate a cache line in another cache but a read cannot). As we are updating a single block, the number of operations is bounded by $2nb^3$, so we are not generating a disproportionate amount of work per task, but we do risk generating very little.

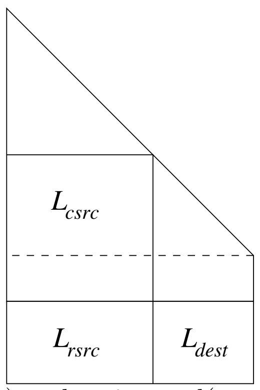


(a) factorize(`diag`)



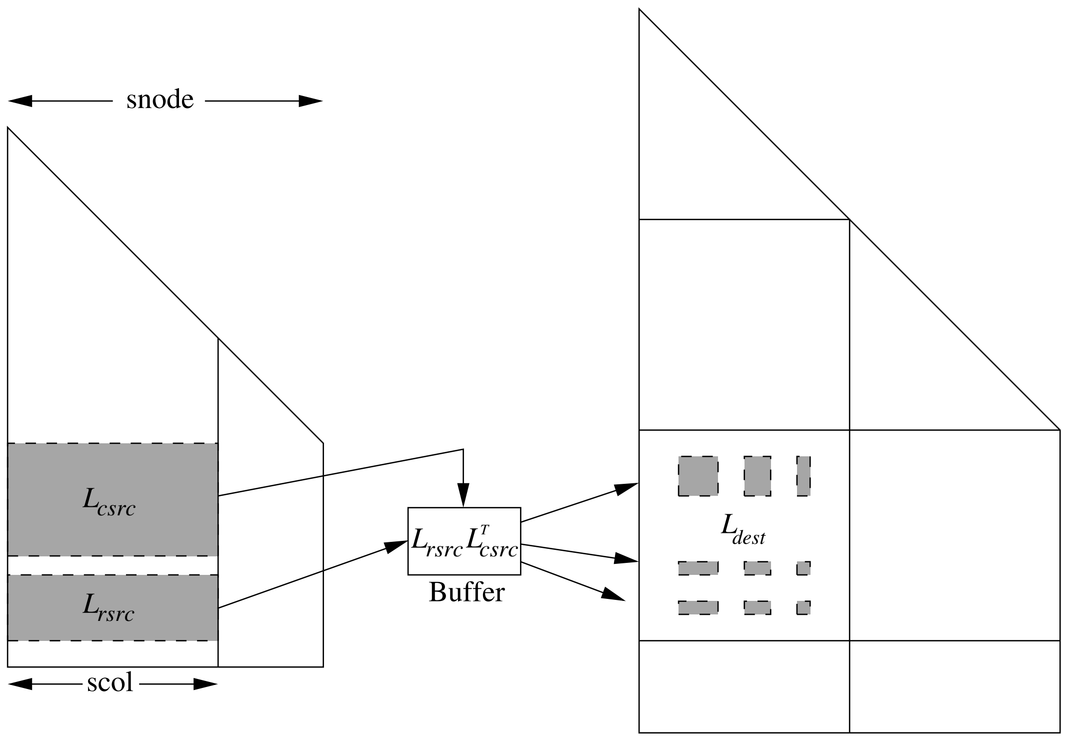
(b) solve(`dest`, `diag`)

$$L_{dest} \Leftarrow L_{dest} L_{diag}^{-T}$$



(c) update_internal(`dest`, `rsrc`, `csrc`)

$$L_{dest} \Leftarrow L_{dest} - L_{rsrc} L_{csrc}^{T}$$



(d) update_between(`dest`, `snode`, `scol`)

1. Form outer product $L_{rsrc} L_{csrc}^{T}$ into Buffer.
2. Distribute the results into the destination block L_{dest}.

Figure 7.2: Graphical interpretations of sparse DAG tasks.

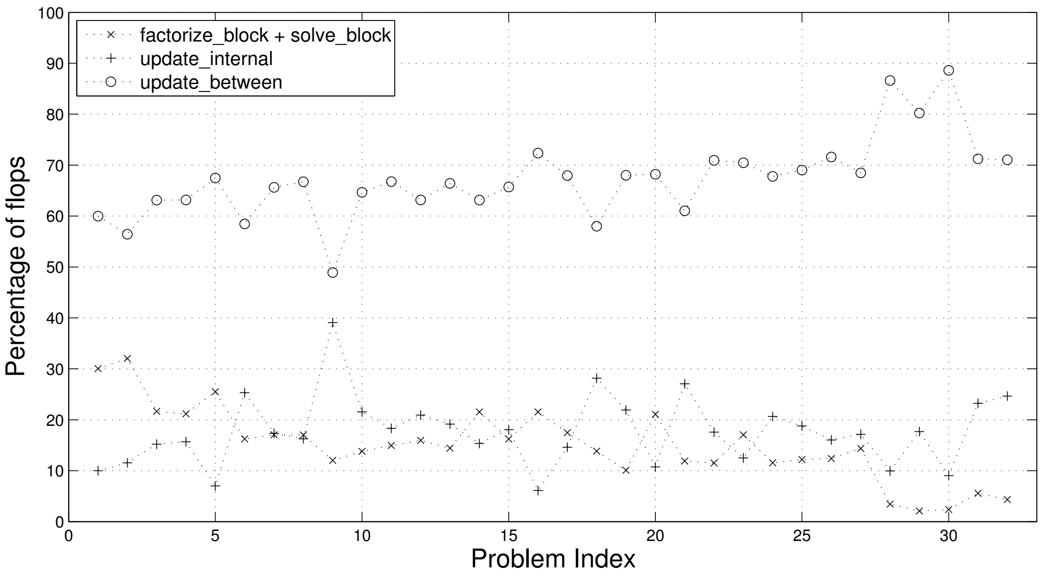


Figure 7.3: Percentage of flops in the different kinds of tasks.

In practice, it seems that the largest proportion of floating point operations are performed in update_between tasks. In Figure 7.3, we show the percentage of flops needed for the different kinds of tasks for the test problems of Section 7.7.

Following the techniques of Section 6.2, we do not store the entire task DAG explicitly. Instead a series of counts and flags is used in much the same way as the dense case. The only real difference is that the update_between tasks require slightly more complicated flag checking, potentially depending on the completion of up to four blocks in the source column.

7.4 Task dispatch engine

We have improved on the task dispatch engine of Chapter 6 by using cache-specific stacks to make the code more cache aware. In that chapter we found that complex prioritisation schemes that take into account critical paths offer very limited benefit. As a result, in `HSL_MA87` we have opted for a simpler prioritisation scheme, and instead favour cache awareness using a system of local task stacks together with a single task pool.

For each shared cache, there is a small local stack holding tasks that are intended for use by the threads sharing this cache. During the factorization, each thread adds or draws tasks from the top of its local stack. It is a stack rather than a more complicated data structure because this gives all the properties that are needed. Each stack has a lock to control access by its threads.

When a thread completes the last update for a diagonal block, it executes the factorize task for this block at once without putting it on its stack. This promotes both cache reuse and the generation of further tasks. When a thread completes a solve task, it first places any update_between tasks generated on its stack, then places any update_internal tasks generated. Both will be above any stacked solve tasks. Since some of the data needed for an update is likely to be in the local cache, cache reuse is encouraged naturally without the need for explicit management.

If a local stack becomes full, a global lock is acquired and the bottom half (that is, the tasks that have been in the stack the longest so that their data are unlikely still to be in the local cache) is moved to the task pool. We give the tasks in the task pool different priorities, in the descending order factorize, solve, update_internal, update_between, but our experience has been that this does not significantly improve the execution time over using a single stack for all the tasks in the task pool. To understand why this should be the case, note that, as

1	2	3	parent
4		4	4
5			5
6		6	6
	7		7
		8	8
		9	9

(a) original

1	2	3	parent
4		4	4
6		6	6
		8	8
		9	9
5			5
	7		7

(b) heuristic

1	2	3	parent
		8	8
		9	9
4		4	4
6		6	6
5			5
	7		7

(c) optimal

Figure 7.4: Column lists for a node and its three children under different reorderings.

already observed, a factorize task is executed as soon as it is generated and so never reaches the pool. When a factorize task completes, several solve tasks may be placed on the stack, one of which is probably executed immediately. When an update task completes for an off-diagonal block, a single solve task is placed on the stack and probably executed immediately. Few solve tasks therefore reach the pool. It follows that prioritization mainly affects the update tasks and update_internal tasks will have already been favoured by the order in which they are added to the local stacks.

When establishing update_between tasks, it is convenient to search the path from the node to the root through links to parents. This leads to the tasks being placed on the stack with those nearest the root uppermost, so that these will be executed first. This is the opposite to the order that will lead to early availability of further tasks. We tried reversing this order and indeed found that the number of available tasks increased. This led to a larger task pool being needed and to an increase in execution time. It seems that as long as there are sufficient tasks available to keep the threads active, this is sufficient. On the basis of numerical experimentation, we have set the default value for the initial size of the task pool to 25,000 (the size of the pool is increased whenever necessary during execution). For many of the test problems of Section 7.7, the task pool size did not exceed 10,000; the largest was approximately 22,000.

If a local task stack is empty, the thread tries to take a task from the task pool. Should this also be empty, the thread searches for the largest local stack belonging to another cache. If found, the tasks in the bottom half of this local stack are moved to the task pool (workstealing). The thread then takes the task of highest priority from the pool as its next task.

To check the effect of having a stack for each cache, we tried two tests while running the problems of Section 7.7. First, we ran with a stack for each thread. This led to a small loss of performance (around 1% to 2% for the larger problems). Second, we disabled processor affinity, which means that the threads are not required to share caches in the way the code expects. This led to a similar loss of performance.

We also tried the effect of using the global lock to control the local stacks as well as the global pool. We found that this had no noticeable effect on execution time on our test machine.

7.5 Increasing cache locality in update_between

We note that variable ordering within a supernode is unrestricted. In this section we identify a potential improvement that exploits this additional freedom to attempt to minimize cache misses in update_between tasks.

Using traditional variable ordering techniques during analyse can result in the update_between operation accessing non-sequential memory. Consider the third child in Figure 7.4(a). As children 1 and 2 were encountered before child 3 while building the parent's variable list, child 3's indices are non-contiguous when mapped into the parent's index list. If variables 4 to 9 are eliminated at the parent node then we are free to permute them as we wish, allowing us to remedy this situation based on the following argument.

The exact elimination order at the parent derives from the order in which the children are encountered. Consider the tree in Figure 7.5, with the children of a node encountered from left to right. We define the variable list i_j as those variables that are first encountered at node i

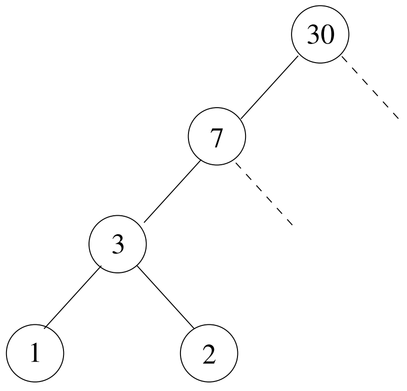


Figure 7.5: Partial assembly tree.

but are eliminated at node j. The list $i_j^{(k)}$ is the sublist of i_j containing those variables that are encountered at node k, a sibling of node i. Then we can represent the elimination-ordered variable lists as follows, using bold to indicate those variables eliminated at a node:

Node	Variable list
1	$\mathbf{1_1}1_3 1_7 1_{30}$
2	$\mathbf{2_2}1_3^{(2)}2_3 1_7^{(2)}2_7 1_{30}^{(2)}2_{30}$
3	$\mathbf{1_3 2_3 3_3}1_7 2_7 3_7 1_{30}2_{30}3_{30}$
7	$\mathbf{1_7 2_7 3_7}\ldots\mathbf{7_7}1_{30}2_{30}3_{30}\ldots 7_{30}$
30	$\mathbf{1_{30}2_{30}3_{30}}\ldots\mathbf{7_{30}}\ldots\mathbf{30_{30}}$,

As we can see, each of node 1's sublists is contiguous in its parent 3. However, this does not hold for its sibling node 2. The variables in 2's $1_j^{(2)}$ sublists are mapped non-contiguously into 1_j at nodes 3, 7 and 30. This behaviour can be generalised to an entire subtree's variables by relabelling the list $1_7 2_7 3_7$ to $\hat{3}_7$ and the list $1_{30}2_{30}3_{30}$ to $\hat{3}_{30}$. Thus we can state that the first child's lists have a contiguous map directly into its parent. Further, due to the recursive relabelling, we can express the entire assembly tree as a series of disjoint paths along which all operations are contiguous on these sublists. Each of these disjoint paths contains a node and all its first descendants.

This suggests a heuristic for maximizing the number of contiguous operations. That is to use a weighted postorder of the elimination tree such that the children of each node are ordered by the number of variables they pass up to the parent, with the largest first. The result of this is illustrated as Figure 7.4(b).

We may still have non-contiguous ordering with respect to the non-dominant children, however. We may have freedom to reorder the rows of the dominant child such that those rows that also come from the second child are ordered last within it. Such an ordering is shown as Figure 7.4(c). We note that this freedom may be more limited than is first apparent: the child's ordering will probably be limited by its own children.

A further improvement still would be to relax the elimination order requirement. We could then, for example, express the variable list of node 3 as $\mathbf{1_3 2_3 3_3}1_7 1_{30}2_7 2_{30}3_7 3_{30}$. However this would result in L_{csrc} being non-contiguous in update_between tasks involving non-first children.

To avoid excessive complications within our code we have only considered the simple heuristic strategy. Rather than using a weighted postorder, we have instead implemented an identical effect through a post-processing of the variable lists. Results are shown in Table 7.1 for a small number of representative problems. The five slowest problems have also been included (see Section 7.7 for a description of the problems). Though reasonably modest, of particular note is that the performance gains in parallel are often much more than merely an eighth of the serial gains, leading to better speedups.

Problem	Without reordering			With reordering		
	1	8	speedup	1	8	speedup
9	5.14	1.01	5.08	5.09	0.93	5.50
16	14.5	2.73	5.30	14.1	2.54	5.55
18	11.5	1.93	5.95	11.4	1.89	6.02
28	83.3	13.1	6.36	80.0	12.3	6.49
29	120.2	20.1	5.99	119.7	19.8	6.03
30	333.3	51.8	6.44	317.3	48.0	6.62
31	519.5	73.1	7.10	507.3	70.8	7.17
32	788.1	112.6	7.06	760.3	105.9	7.18

Table 7.1: Showing comparison of timings on 1 and 8 threads, plus the parallel speedup, with and without reordering the children of each node.

7.6 PaStiX

During the review process of the paper [Hogg et al. 2010] it was observed that our task scheme is similar to that used by PaStiX [Hénon et al. 2002]. In the following subsection we expand on the brief description of PaStiX from Section 2.4.5. We then highlight the major differences between that approach and our own. It is worth commenting that the results later in this chapter demonstrate a substantial performance advantage of our code over PaStiX on multicore machines.

7.6.1 The PaStiX algorithm

PaStiX follows the traditional sparse solver paradigm of using supernode identification and amalgamation on the elimination tree to obtain an assembly tree in the analyse phase. The analyse phase also computes a static scheduling for the parallel factorize phase, using a model of computation and communication time.

The factorize phase diverges from tradition by not using a compressed sparse column or related format. The structure used instead is illustrated in Figure 7.6. It divides the entire matrix into blocks using a partition of variables based on the node of the assembly tree at which they are eliminated. As the code is closely tied to a nested dissection ordering it can be easily seen that many of these blocks will be zero; the resultant block sparsity pattern is stored. Each non-zero block has its entries stored densely, possibly introducing explicit zero rows. Leading and trailing zero rows are not stored.

The similarities with our approach come in the definition of tasks on this data structure. Using the notation (j, k) to refer to block j in column k, PaStiX defines 4 such tasks:

pastix_comp1d(k) factorizes the entire block column k and computes all update contributions. *(There is no comparable task in our scheme.)*

pastix_factor(k) factorizes the diagonal block (k, k). *(The same as our factorize task.)*

pastix_bdiv(j, k) performs the triangular solve of block (j, k) in column with the column's diagonal block (k, k). *(The same as our solve task.)*

pastix_bmod(i, j, k) computes the update contribution to block (i, j) from the outer product of blocks (i, k) and (j, k). *(Similar to our update_internal and update_between tasks. Due to the block data structure used there is no need to distinguish between them.)*

The static scheduling phase works on the corresponding task DAG. There are two stages. First each node is assigned to a set of candidate processes. Next a static schedule is determined through simulation to select a single process for each task.

The candidate processes are determined by repeated bisection of the tree from the top down such that there are similar amounts of work in each partition. Hence the root node's set of candidate processes is the set of all available processes. If the tree is balanced then at the next level each child has half of the processes in its candidate set. Typically leaf subtrees with a single candidate result.

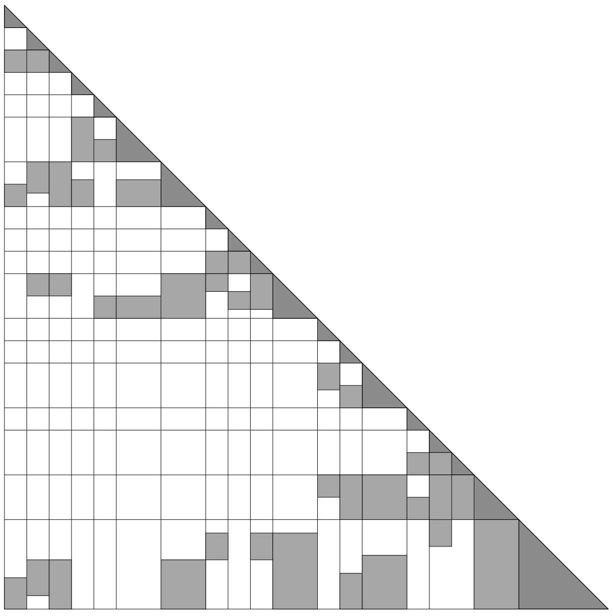


Figure 7.6: Storage format used by PaStiX. Each non-zero block is stored densely.

Static scheduling is then performed by simulating runtime for each task. A task can be executed by any process that is a candidate for its destination block's node. Tasks are scheduled one at a time on the candidate process where they can start executing earliest (after allowance for communication time). The task ordering is given by a postordering of the assembly tree. Once a task is scheduled the system model is updated and the next task is then scheduled

The factorize phase of PaStiX then executes this static scheduling to perform the factorization.

7.6.2 Major differences

There are many differences between our approach and that of PaStiX. Major ones are:

- **Shared memory:** our approach is written for shared memory multicore processors, and is able to take into account the shared caches that are often present. PaStiX is fundamentally a distributed memory code that exploits shared memory parallelism in SMP nodes. As a result different design choices have been made.
- **Row storage:** in our approach the entire block column is stored row-wise, rather than in small blocks determined by the assembly tree. Beyond merely being an implementational detail, this ensures that all possible divisions of the column into multiple blocks may be used without penalty. This is exploited by using different row blocking in update_between than for the other tasks, allowing the use of the optimal block sizes when calling the BLAS: in contrast, PaStiX is constrained to the block sizes given by its supernode partition. A corollary to this is that we are able to generate fewer, bigger, tasks than PaStiX, reducing overheads and increasing performance.
- **Reuse of data in cache:** through the use of local cache stacks, we achieve a good reuse of data that is in cache. For the majority of tasks at least one of the blocks involved has been recently used in a previous task. In contrast, PaStiX uses a strategy more concerned with the earliest start time; as our strategy allows any thread to tackle any task we are less constrained in seeking load balance.
- **Reduced number of operations on zero:** our approach avoids the extra computations on zero rows that the PaStiX approach requires by using dense blocks rather than non-zero rows. The update_between tasks are as a result less efficient as they are performing a sparse update rather than the dense one of PaStiX, though less data movement is typically

Identifier	n	$nz(A)$	$nz(L)$	Gflops	Application/description
1. *CEMW/tmt_sym*	726.7	2.9	30.0	9.38	Electromagnetics
2. *Schmid/thermal2*	1228.0	4.9	51.6	14.6	Unstructured thermal FEM
3. *Rothberg/gearbox**	153.7	4.6	37.1	20.6	Aircraft flap actuator
4. *DNVS/m_t1*	97.6	4.9	34.2	21.9	Tubular joint
5. *Boeing/pwtk*	217.9	5.9	48.6	22.4	Pressurised wind tunnel
6. *Chen/pkustk13**	94.9	3.4	30.4	25.9	Machine element, 21 noded solid
7. *GHS_psdef/crankseg_1*	52.8	5.3	33.4	32.3	Linear static analysis
8. *Rothberg/cfd2*	123.4	1.6	38.3	32.7	CFD pressure matrix
9. *DNVS/thread*	29.7	2.2	24.1	34.9	Threaded connector/contact
10. *DNVS/shipsec8*	114.9	3.4	35.9	38.1	Ship section
11. *DNVS/shipsec1*	140.9	4.0	39.4	38.1	Ship section
12. *GHS_psdef/crankseg_2*	63.8	7.1	43.8	46.7	Linear static analysis
13. *DNVS/fcondp2**	201.8	5.7	52.0	48.2	Oil production platform
14. *Schenk_AFE/af_shell3*	504.9	9.0	93.6	52.2	Sheet metal forming matrix
15. *DNVS/troll**	213.5	6.1	64.2	55.9	Structural analysis
16. *AMD/G3_circuit*	1585.5	4.6	97.8	57.0	Circuit simulation
17. *GHS_psdef/bmwcra_1*	148.8	5.4	69.8	60.8	Automotive crankshaft model
18. *DNVS/halfb**	224.6	6.3	65.9	70.4	Half-breadth barge
19. *Um/2cubes_sphere*	101.5	0.9	45.0	74.9	Electromagnetics
20. *GHS_psdef/ldoor*	952.2	23.7	144.6	78.3	Large door
21. *DNVS/ship_003*	121.7	4.1	60.2	81.0	Ship structure—production
22. *DNVS/fullb**	199.2	6.0	74.5	100	Full-breadth barge
23. *GHS_psdef/inline_1*	503.7	18.7	172.9	144	Inline skater
24. *Chen/pkustk14**	151.9	7.5	106.8	146	Civil engineering. Tall building
25. *GHS_psdef/apache2*	715.2	2.8	134.7	174	3D structural problem
26. *Koutsovasilis/F1*	343.8	13.6	173.7	219	AUDI engine crankshaft
27. *Oberwolfach/boneS10*	914.9	28.2	277.8	282	Bone micro-finite element model
28. *ND/nd12k*	36.0	7.1	116.5	505	3D mesh problem
29. *JGD_Trefethen/ Trefethen_20000*	20.0	0.3	90.7	652	Integer matrix
30. *ND/nd24k*	72.0	14.4	320.6	2054	3D mesh problem
31. *bone010*	986.7	36.3	1076.4	3876	Bone micro-finite element model
32. *audikw_1*	943.7	39.3	1242.3	5804	Automotive crankshaft model

Table 7.2: Test matrices and their characteristics. n denotes the order of A in thousands; $nz(A)$ and $nz(L)$ are the number of entries in the lower triangular part of A and in L (without node amalgamation), respectively, in millions; * indicates only the sparsity pattern is provided.

required. In particular, the cache friendly limited reordering of Section 7.5 ensures that for our largest update operations the data is contiguous.

7.7 Numerical results

7.7.1 Test environment

The experiments we report on in this chapter were all performed on our multicore test machine `fox`, with additional tests on a newer architecture machine `jhogg`. Details of both are given in Table 1.4. The Intel Fortran compiler v11.0 was used with the options `-fast -openmp`, and the BLAS library was the Intel MKL v10.1. The sparse matrices used are listed in Table 7.2. This set comprises 32 examples that arise from a range of practical applications. In selecting the test set, our aim was to choose a wide variety of large-scale problems. Each problem is available from the University of Florida sparse matrix collection [Davis 2007].

For all tests, the right-hand side $\boldsymbol{b}$ is generated so that the required solution $\boldsymbol{x}$ is the vector of ones. When only the sparsity pattern of a matrix was available, reproducible pseudo-random off-diagonal entries in the range $(0, 1)$ were generated.

Unless stated otherwise, runs are performed using all 8 cores on our test machine `fox` and all control parameters used by `HSL_MA87` are given their default settings. All times are elapsed

n	`HSL_MA87`(a)	`HSL_MA87`(b)	`HSL_MP54`	`mkl`
100	0.71	0.74	1.63	3.25
500	15.3	15.0	17.7	20.7
1000	31.3	30.3	29.3	35.1
1500	40.8	40.0	35.7	42.6
2000	48.8	47.2	40.8	47.3
2500	51.8	50.9	44.3	51.4
5000	62.2	61.9	55.8	57.3
10000	66.3	65.1	63.7	64.4
20000	69.6	68.9	67.9	67.1

Table 7.3: A comparison of dense Cholesky implementations. Speeds in Gflop/s are reported.

times, in seconds, measured using the system clock. Unfortunately, we found that when the elapsed time on 8 cores was less than a second, it could vary by 20% to 30% between runs. Occasionally, a time would be greater by much more than this. We believe that the occasional very slow runs were caused by an executing thread being asked to perform a system task. In each experiment we therefore averaged over ten complete runs of each of the problems except 28, 29, 30, 31, 32, which are averaged over three. In the following subsection, we refer to these five problems, that each require more that 500 Gflops, as the *slow subset*.

Ordering is performed before the analyse phase is called by calling the MeTiS routine `METIS_NodeND` [Karypis and Kumar 1998; 1999]. In Table 7.2, we include the number of millions of entries in the matrix factor (denoted by $nz(L)$) with this pivot sequence before node amalgamation is performed.

7.7.2 Dense comparison

We can trivially represent a dense problem as a sparse one with a single node that contains all n variables. This allows us to compare the efficiency of the dense tasks within `HSL_MA87` with other dense Cholesky factorization implementations (although it should be noted that `HSL_MA87` accepts a different data format and hence avoids the reordering that they may require).

In Table 7.3 comparisons are given for randomly generated dense matrices of order up to 20000. Results for the following codes are presented:

`HSL_MA87(a)`: `HSL_MA87` **with** processor affinity enabled. Processor affinity refers to whether threads are tied to specific processors, or the operating system is allowed to move them. For our cache management techniques to reflect the actual physical circumstance, processor affinity needs to be enabled. On our test machine with the Intel compiler we do this by setting the environment variable `KMP_AFFINITY` to `compact`.

`HSL_MA87(b)`: `HSL_MA87` **without** processor affinity enabled.

`HSL_MP54`: dense DAG code described in Chapter 6. It is designed to perform both partial and complete Cholesky factorizations.

`mkl`: LAPACK Cholesky factorization subroutine (`dpotrf`) supplied by Intel MKL 10.1, which we believe uses a DAG-based algorithm.

The test results were averaged over ten runs. For each run, the factorization times for different problems of the same size were accumulated until the total elapsed time was at least one second; the average speed in Gflop/s was then calculated.

Results are reported in Table 7.3: comparing `HSL_MA87` with `HSL_MP54` and `dpotrf`, we see that `HSL_MA87` is competitive for sufficiently large n (n greater than about 2000) and its advantage over the other codes increases with n. However, its performance compares less favourably for small problems. This is due to workstealing with the relatively small blocks used for optimal performance on these problems causing an excessive number of blocks to be switched between caches. In the sparse case, we do not anticipate this will cause problems since

Problem	nz(L) (millions)				
	`1`	`8`	`16`	`32`	`64`
3	37.	39.	41.	44.	49.
16	98.	119.	139.	172.	228.
22	74.	76.	79.	86.	101.
28	117.	117.	118.	119.	121.
29	91.	92.	94.	96.	99.
30	321.	322.	323.	326.	331.
31	1076.	1090.	1108.	1135.	1183.
32	1242.	1257.	1275.	1303.	1359.

Problem	Factorize times					Solve times				
	`1`	`8`	`16`	`32`	`64`	`1`	`8`	`16`	`32`	`64`
3	0.91	**0.83**	0.86	**0.83**	0.89	0.27	0.27	0.29	0.31	0.34
16	5.00	2.77	**2.59**	**2.61**	3.04	1.52	1.09	1.24	1.49	1.80
22	2.70	**2.57**	**2.59**	**2.64**	2.89	0.50	0.51	0.53	0.58	0.66
28	22.5	15.3	13.7	12.3	**11.1**	0.75	0.74	0.74	0.76	0.74
29	119.9	40.8	27.5	19.9	**15.7**	0.62	0.58	0.59	0.65	0.61
30	80.4	58.4	53.1	48.0	**44.1**	2.05	2.02	2.03	2.08	2.03
31	**72.0**	**71.0**	**70.7**	**70.8**	**71.4**	6.82	6.86	7.00	7.19	7.39
32	**109.**	**107.**	**106.**	**106.**	**106.**	7.82	7.85	8.01	8.23	8.43

Table 7.4: Comparison of the number of entries in L (in millions) and the factorize and solve times for values of the node amalgamation parameter `nemin` in the range `1` to `64`. The fastest factorize times (and those within 3 per cent of the fastest) are in bold.

there are many more tasks and more than one initial task. Workstealing should only play a real role near the start and end of the factorization.

Processor affinity seems to marginally enhance the performance of `HSL_MA87` unless n is small. Tests on sparse problems also showed a very moderate gain from use of affinity (about 2%). For all future tests involving `HSL_MA87` we do use processor affinity, but for tests with other codes we do not (if a code does not exploit processor affinity it typically results in poor performance).

7.7.3 Effect of node amalgamation

The `HSL_MA87` strategy of amalgamating nodes of the tree is taken from `HSL_MA77` and explained in [Reid and Scott 2009b]. A child is amalgamated with its parent if both involve fewer than a given number, `nemin`, of eliminations. This is very simple heuristics and offers little control over increases in the amount of data stored and amount of computation required. In particular, the heuristic can be quite poor for small matrices where the relative increase in density is high.

In Table 7.4, we show the factorize times for the slow subset (problems 28 to 32), where `nemin` value 64 gave the best performance,and for three others to represent the three kinds of behaviour that we saw in the rest: flat (3), U-shaped (22) and `nemin`=1 much slower (16). Apart from the slow subset, `nemin`=32 was almost always within 3% of the best.

Also shown are the number of entries in L and the solve times. The examples were chosen to also illustrate the three kinds of behaviour shown in solve times: rising slowly (3,22,31,32), flat (28, 29, 30), and `nemin=1` much slower (26). For some problems the number of entries in L grows rapidly with `nemin` and this leads to rising solve times (16, 22).

These considerations led us to use 32 as our default `nemin` value. However, if the number of entries in L increases slowly with `nemin`, it can be advantageous to use an even larger value. It may be advantageous to run with `nemin`=8 if a large number of solves are to be performed (for example in iterative refinement).

Problem	Single core factorize times						8-core factorize times				
	128	192	256	320	384	448	128	192	256	320	384
3	4.30	**4.13**	**4.09**	**4.08**	**4.06**	**4.06**	**0.77**	0.80	0.82	0.84	0.94
15	10.07	9.58	**9.36**	**9.33**	**9.30**	**9.25**	**1.67**	**1.63**	**1.63**	**1.67**	1.77
22	17.5	16.3	**16.1**	**15.9**	**15.8**	**15.8**	2.79	**2.60**	**2.61**	2.69	2.69
28	90.9	82.6	**80.0**	**79.0**	**79.2**	**80.1**	14.5	**12.6**	**12.3**	**12.4**	13.1
29	148.	125.	120.	**117.**	**117.**	**116.**	23.8	20.5	**19.8**	**19.7**	20.6
30	394.	331.	**318.**	**313.**	**312.**	**316.**	63.8	50.3	**48.0**	**48.1**	50.1
31	580.	528.	508.	**499.**	**492.**	**488.**	82.9	73.4	**70.9**	**69.5**	**69.6**
32	884.	794.	761.	**747.**	**735.**	**729.**	128.	110.	**106.**	**104.**	**103.**

Table 7.5: Comparison of the factorize times for different block sizes *nb*. The fastest times (and those within 3 per cent of the fastest) are in bold.

Problem	No local stack	Stack size				
		10	50	100	200	300
3	0.92	0.86	0.86	**0.82**	**0.84**	0.85
15	1.80	1.74	**1.63**	**1.63**	**1.59**	**1.61**
16	3.14	2.80	**2.60**	**2.58**	**2.58**	**2.61**
28	12.8	12.7	**12.4**	**12.3**	**12.2**	**12.1**
29	20.3	20.3	**20.0**	**19.8**	**19.7**	**19.5**
30	49.9	49.5	48.7	**48.0**	**47.4**	**47.2**
31	74.0	73.6	**71.3**	**70.9**	**70.7**	**70.8**
32	110.	110.	**107.**	**106.**	**106.**	**106.**

Table 7.6: Comparison of the factorize times for different local task stack sizes. The fastest times (and those within 3 per cent of the fastest) are in bold.

7.7.4 Block size

The block size *nb* was discussed in Section 7.2. In Table 7.5, we report the factorize time for a range of block sizes on a single core and on 8 cores. As before, we show results for three cases with longer factorization times, and three cases representing the behaviour of the others. The fastest times (and those within 3 per cent of the fastest) are in bold. We see that, on a single core, the best times are obtained using a larger block size than on 8 cores, but in our tests the reductions in time using $nb > 256$ are less than 5 per cent. On eight cores, 192 and 256 almost always gave good times; hence we have chosen a default block size of 256.

7.7.5 Local task stack size

In Section 7.4, we discussed the use of local task stacks. In Table 7.6, we report the factorization times for a range of local task stack sizes and also for when there is no local task stack. Improvements over having no local task stack are usually in the range of 10% to 20%. The exceptions were in the slowest cases, where the gain was only about 5%. It is never disadvantageous to use a local stack. These results have led us to choose 100 as our default stack size.

For three of the slow problems, we show in Table 7.7 the number of leaf nodes (initially a factorize_block task is put into the global task pool for each leaf), the total number of tasks, the number of tasks taken directly from the local stacks, the number of tasks sent to the global task pool because a local stack became full, and the number of tasks moved to the global task pool by workstealing. We see that the number of tasks moved by workstealing is small. Provided the stack size is at least 100, a good proportion of the tasks are executed directly.

For the smaller problems, the local stacks became full for only eight cases when the stack size was 100. This happened most often for problem 19, also shown in Table 7.7. Next was problem 26, where 750 tasks were moved to the pool because of a full stack. For a further 6 problems, fewer than 400 tasks were moved to the pool because of a full stack. For the remaining smaller

Problem	Leaves (10^3)	Tasks (10^3)	Stack size	Direct (10^3)	Full (10^3)	Workstealing (10^3)
19	0.88	48	10	19	28	0.14
			50	40	7	0.30
			100	45	1	0.43
			200	46	0	0.54
			300	46	0	0.47
28	0.06	124	10	19	105	0.16
			50	45	78	0.47
			100	66	56	0.81
			200	92	30	1.61
			300	108	13	1.83
29	0.06	241	10	21	221	0.13
			50	46	195	0.39
			100	63	177	1.25
			200	97	144	0.89
			300	119	121	1.76
32	5.58	772	10	232	534	0.16
			50	573	192	0.92
			100	703	60	2.84
			200	751	11	3.57
			300	759	4	3.95

Table 7.7: Tasks taken directly from local stacks, moved to pool because of a full stack, and moved to pool because of workstealing on 8 cores.

Problem	Stack size 10		Stack size 100	
	with	without	with	without
2	1.24	1.30	**1.23**	1.28
17	1.86	1.96	**1.71**	2.05
27	7.54	7.67	**7.10**	11.08
28	12.7	12.8	**12.3**	16.3
29	**20.3**	**20.2**	**19.8**	21.9
30	49.5	49.8	**48.0**	49.7
31	73.6	73.7	**70.8**	85.9
32	109.7	110.0	**105.8**	139.3

Table 7.8: Factorize times with and without workstealing on 8 cores. Default values for other parameters. The times within 3% of the fastest are in bold.

problems, none of the local stacks became full. With a stack size of 200, a local stack became full only for problem 22 and only 100 tasks were moved to the task pool because of this.

Although the workstealing figures in Table 7.7 are small, workstealing is important for load balance and its importance increases with the local stack size. To illustrate this, in Table 7.8 we present times on 8 cores with and without workstealing using local stack sizes of 10 and 100.

7.7.6 Speedups and speed for `HSL_MA87`

One of our concerns is the speedup achieved by `HSL_MA87` as the number of cores increases. In Figure 7.7, we plot the speedups in the factorize times when 2, 4, and 8 cores are used. We see that the speedup on 2 cores is close to 2, on 4 cores it is generally more than 3 and for the largest problems (in terms of $nz(L)$) it exceeds 3.6. On 8 cores, `HSL_MA87` achieves speedups of more than 6 for many of the largest problems and for all but three test problems, the speedup exceeds 5. The two largest problems achieve a speedup of almost 7.2 on 8 cores. This deterioration in speedup is to be expected due to shared resources in a multicore environment. Our machine has two processor packages, each with its own memory bus, and thus we expect good speedup

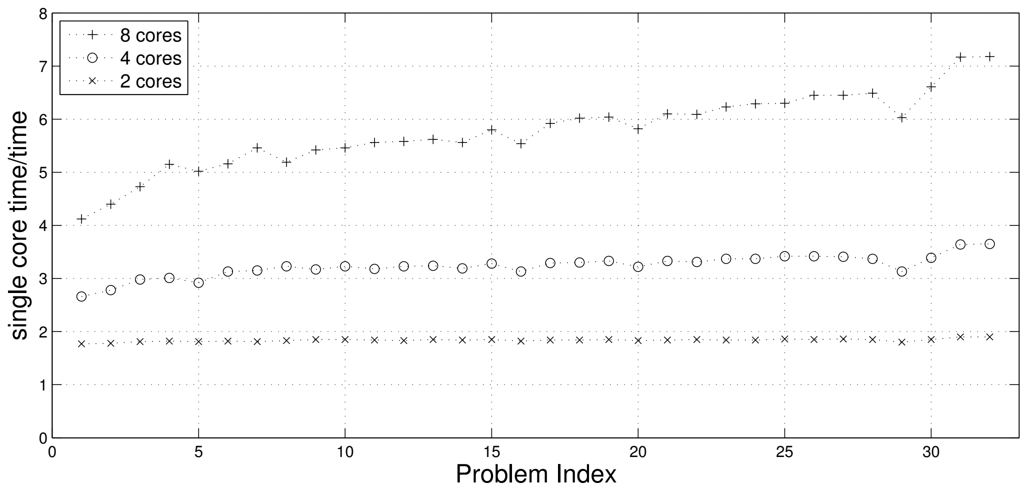


Figure 7.7: The ratios of the factorize times on 2, 4 and 8 cores to the factorize time on 1 core.

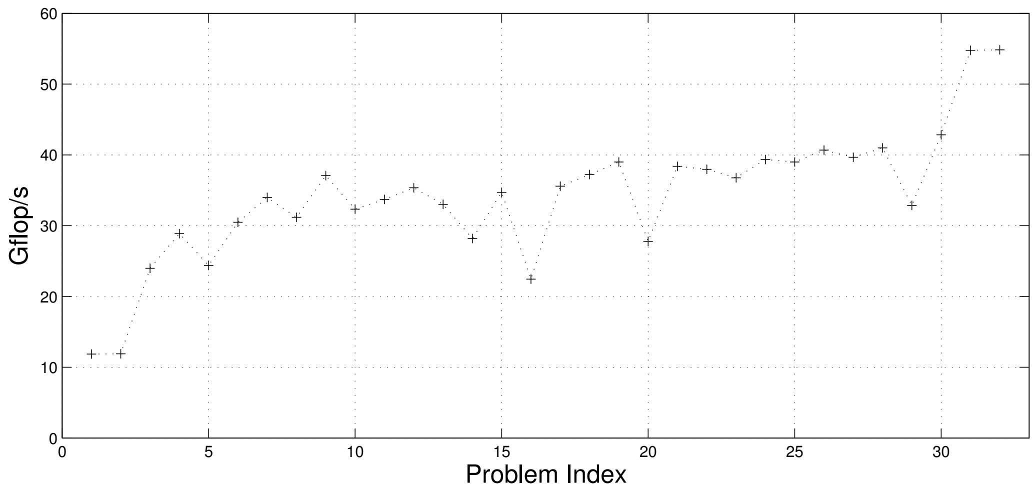


Figure 7.8: The speeds in Gflop/s on 8 cores.

on 2 out of the 8 cores. When we switch to using 4 cores, we are sharing a memory link between each pair of threads. Finally, going to all 8 cores shares level 2 cache pairwise and a memory link between each set of 4 threads.

Of course, our primary concern is the actual speed achieved. We show the speeds in Gflop/s on 8 cores in Figure 7.8. Here we compute the flop count from a run with the node amalgamation parameter `nemin` having the value 1. We note that for 14 of our 30 test problems, the speed exceeds 36.4 Gflop/s, which is half the maximum `dgemm` speed (recall Table 1.4). Furthermore, on all but four of the problems, it is greater than 24.3 Gflop/s, which is a third of the `dgemm` maximum. The top speed achieved is 54.8 Gflop/s.

7.8 Comparisons with other solvers

We wish to compare the performance of `HSL_MA87` with some of the other readily available solvers. These have all been described previously in Section 2.4.5:

PARDISO Version 4.0.0, ifort 10.1 version. The same MeTiS ordering used for `HSL_MA87` was supplied to PARDISO.

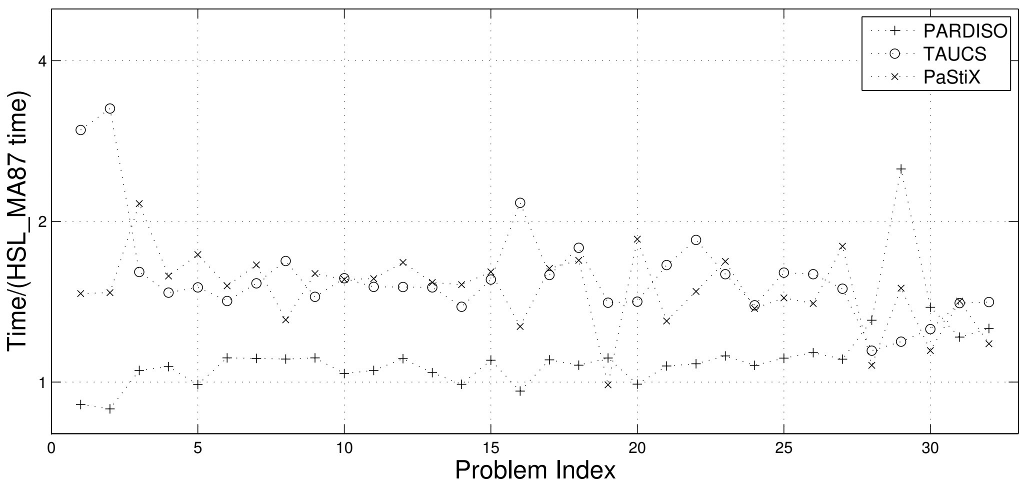


Figure 7.9: The ratios of the PARDISO, TAUCS and PaStiX factorize times to the `HSL_MA87` factorize time (single core).

TAUCS Version 2.2, compiled with Cilk 5.4.6 under gcc. (Will not compile with icc, so potentially unfair to compare against codes compiled under ifort. However, experiments indicate `HSL_MA87` performance is similar under gfortran to ifort.) No option for supplying an ordering seems to exist, so TAUCS was allowed to use its own MeTiS routine. The solver was explicitly told to use its multifrontal cilk factorization option.

PaStiX Version 5.1.2, release 2200, compiled with icc 10.1 and ifort 11.0. Runs were performed with a single MPI process, but multiple threads, as this gave the best performance. Rather than passing in an ordering, PaStiX was allowed to use its built-in SCOTCH routine, as the results with a supplied ordering were very poor.

Comparisons on one and eight cores

We first compare the performance of the above packages with that of `HSL_MA87` when run on a single core of our test machine. The ratios of the factorize times for each package to the factorize time for `HSL_MA87` are given in Figure 7.9. Values above one indicate a code was slower than `HSL_MA87`, and below indicate it was faster. For the majority of problems `HSL_MA87` is marginally faster than PARDISO, and considerably faster than TAUCS and PaStiX. For the largest problems we are consistently better than PARDISO, by a good margin. As PARDISO was found to have excellent serial performance by Gould, Hu and Scott [2007], this shows that we are very competitive even in serial.

In Figure 7.10 the ratios of the factorize times for PARDISO, TAUCS and PaStiX to the factorize time for `HSL_MA87` on 8 cores are given. Some ratios are omitted as they are outside the plotted range. We see that for most problems TAUCS and PaStiX are generally uncompetitive, and `HSL_MA87` outperforms PARDISO, significantly so as the number of floating point operations becomes large. On the largest problems TAUCS and PaStiX are much more competitive, but PARDISO is slower than either. We are at least 50% faster than all the other codes on the two biggest problems.

Finally in Figure 7.11 we show a comparison of achieved speedups in an attempt to illustrate the potential, rather than implementation, of the different strategies. This picture is very similar to that for the speed on 8 processors, so we will not comment on it further.

We believe the relatively poor performance of PARDISO on the largest problems is due to its lack of a 2 dimensional blocking to exploit parallelism in update and factorization of large nodes near the root of the tree.

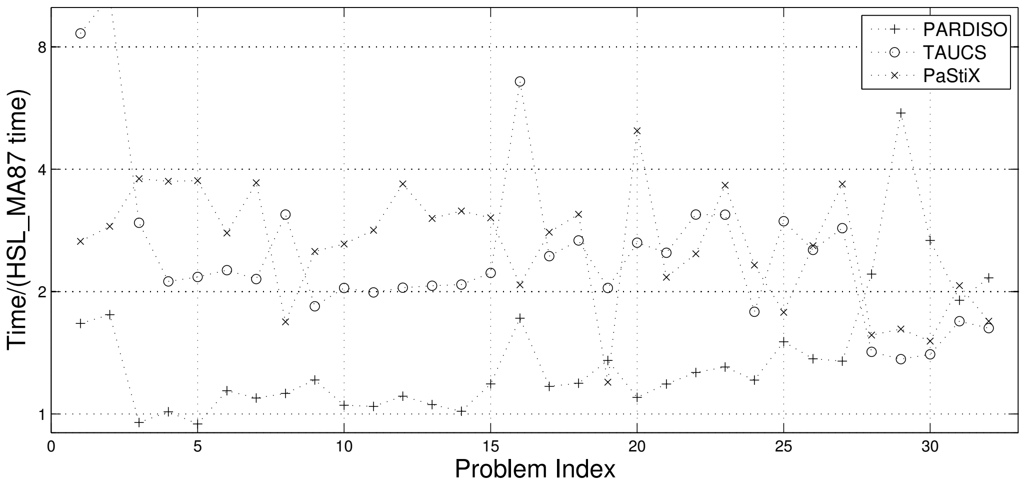


Figure 7.10: The ratios of the PARDISO, TAUCS and PaStiX factorize times to the `HSL_MA87` factorize time (8 cores).

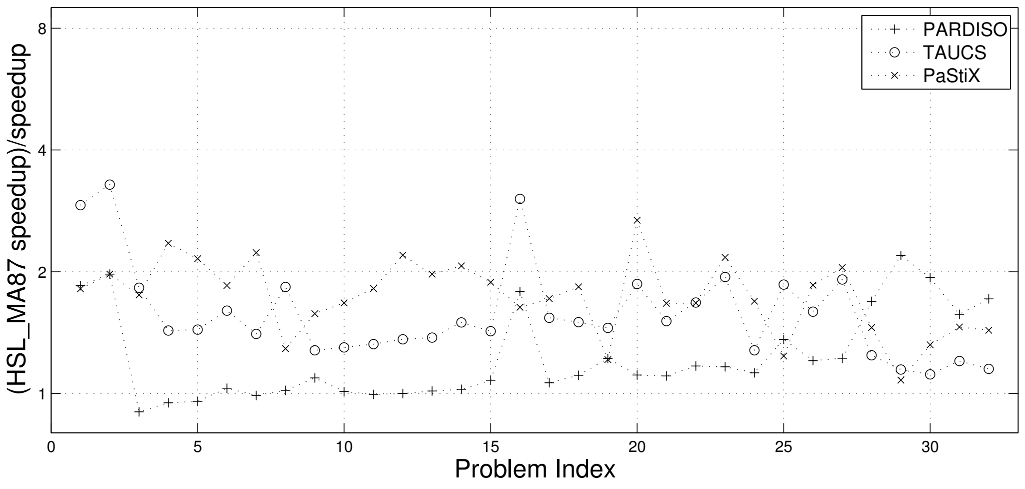


Figure 7.11: The ratios of `HSL_MA87` speed-up to those of PARDISO, TAUCS and PaStiX (8 cores).

	Intel Nehalem	AMD Shanghai
Architecture	Intel(R) Xeon(R) CPU E5540	AMD Opteron 2376
Clock	2.53 GHz	2.30 GHz
Cores	2×4	2×4
Level-1 cache	128 K on each core	128 K on each core
Level-2 cache	128 K on each core	512 K on each core
Level-3 cache	8192 K shared by 4 cores	6144 K shared by 4 cores
Memory	24 GB for all cores	16 GB for all cores
BLAS	Intel MKL 11.0	Intel MKL 11.0
Compiler	Intel 11.0 with option -fast	Intel 11.0 with options -ipo -O3 -no-prec-div -static -msse3

Table 7.9: Specifications of two further test machines.

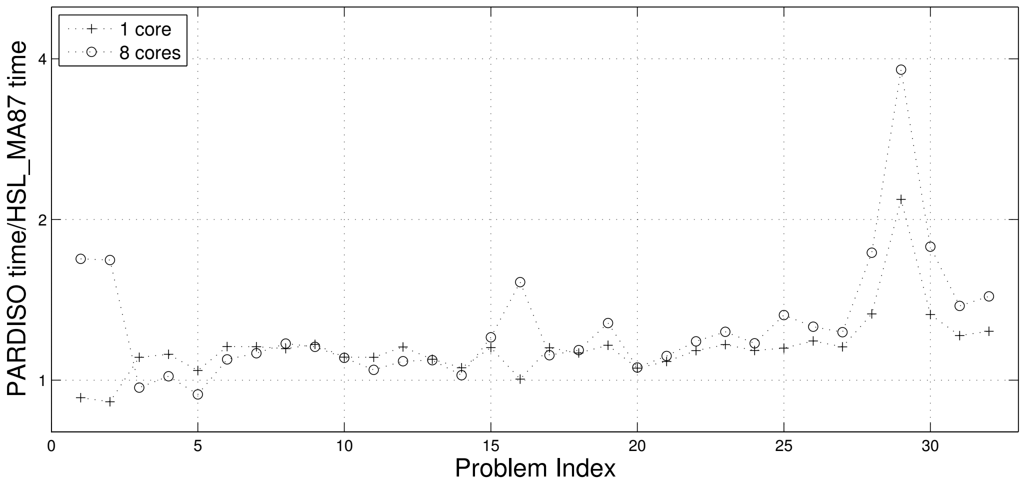


Figure 7.12: Comparison of HSL_MA87 and PARDISO factorize times on the Intel Nehalem architecture.

7.9 Comparisons on other architectures

So far all results have been reported on `fox`. We end this section by presenting runs on two further multicore machines, briefly summarised in Table 7.9. Since the results already reported indicate that PARDISO is the most competitive, we give results in Figures 7.12 and 7.13 comparing only it and HSL_MA87. Due to licencing issues we use the MKL 11.0 version of PARDISO to obtain these results. As on `fox`, we compare favourably with PARDISO, again significantly outperforming it on the largest problems.

7.10 Future work

We believe that our cache-aware scheduler can be improved to more accurately model the caching arrangements of whatever machine we are using, rather than just the shared level-2 caches on our current system. A recursive layer based approach would even allow us to model MPI and out-of-core storage as further levels of cache. Tasks would filter up and down through different levels of cache to reflect the location of relevant data. A queue containing the next few scheduled tasks would allow data to be prefetched from lower layers before it is required, thus enabling the latency to be effectively hidden. A good heuristic for selecting tasks may be able to match current approaches in minimizing communication volume. These modifications are, however, outside the scope of this thesis.

Buttari et. al. [2009] discuss a version of their dense DAG code for *LU* factorization that incorporates a pairwise partial pivoting strategy. It may be possible to build on this approach

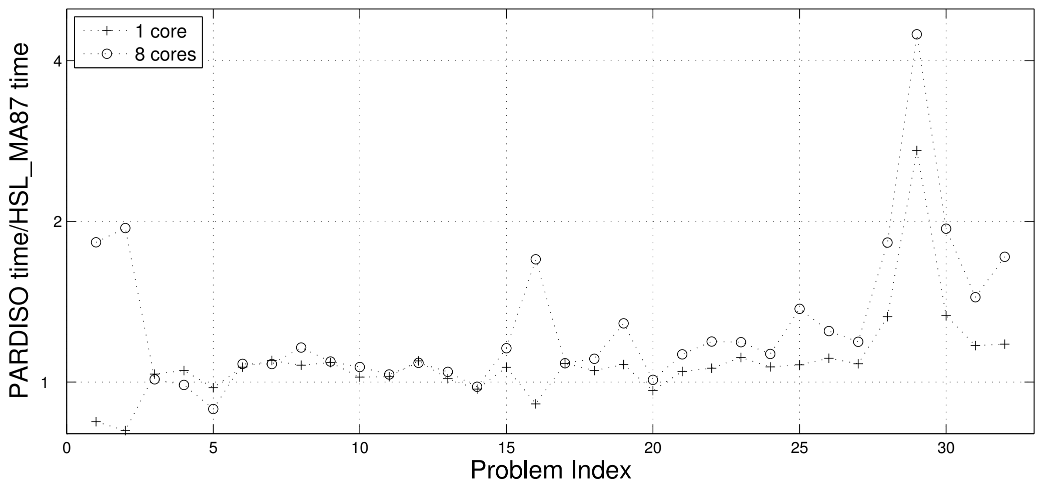


Figure 7.13: Comparison of `HSL_MA87` and PARDISO factorize times on the AMD Shanghai architecture.

in the sparse symmetric indefinite case. We present a preliminary variant that attempts to do this in the next chapter. It follows the work of Reid and Scott [Reid and Scott 2009b]: if a pivot is rejected at a given node for failing to meet stability criteria, the corresponding column is pushed up the assembly tree to become part of the parent node (the pivot is *delayed*).

Chapter 8

Improved algorithms applied to interior point methods

Interior point methods, as described in Chapter 3, spend the vast majority of their computational time in the determination of a descent direction through the solution of a linear system. In this chapter we give results for our new solvers in this context. The open source non-linear interior point code `Ipopt` [Wächter and Biegler 2006] is used to embed the linear solvers in a real world context.

As Ipopt requires the solution of indefinite rather than positive-definite systems, Section 8.1 describes the modifications of `HSL_MA87` necessary to handle a LDL^T factorization. The technicalities of Ipopt interfaces are then described in Section 8.2, with results on linear programming problems in Section 8.3. Finally, some brief conclusions are presented in Section 8.4.

8.1 DAG-driven symmetric-indefinite solver

Our code `HSL_MA87`, described in Chapter 7, is a competitive parallel Cholesky factorization. To convert this code into an indefinite factorization we must do the following:

Compute $PAP^T = LDL^T$, requiring at the very least that D is diagonal, with diagonal entries of plus or minus one. More stably D is block-diagonal, with 1×1 and 2×2 blocks. All factorization tasks and the solve phase must be updated to work with LD and L rather than just with L.

Handle pivoting, requiring minimally that we detect and ignore or modify poor pivots. For a stable factorization supernodal or Bunch-Kaufmann pivoting will be necessary. This will require access to the fully updated column upon which we pivot, resulting in a reorganisation of the task DAG.

The most basic rearrangement of our code is to just store the sign of the pivot and to form LD on the fly. If the pivot is very small (has an absolute value less than some threshold `small`, or relative to the largest entry in the rest of the column) then we could replace the diagonal with a very large number and zero the rest of the column. Doing so effectively treats the column as being empty and produces a zero for the corresponding part of a solution vector. This modification was tested with Ipopt, but it was found that the numerical instability inherent in the approach led to an excess number of interior point iterations.

Instead we have adapted `HSL_MA87` into a fully featured indefinite factorization code comparable to `MA57` and `HSL_MA77`. We will henceforth refer to this variant as DSIS (DAG-driven Symmetric-Indefinite Solver). The version described in this section is a preliminary effort; a full version that has been more fully developed will be completed in the future. This development version has been designed and written with the assistance of Jennifer Scott and John Reid.

8.1.1 Modifications

The new version combines the factor and solve tasks to allow the full column to be used in pivoting, that is performed using a modified `HSL_MA64` kernel [Reid and Scott 2009c] to provide a pivoted partial factorization on each block column. We tested both storing LD and calculating it from L and D on the fly. The first approached worked best in serial, while the latter worked better in parallel when memory bandwidth was more limited. The code uses the calculation approach as we are primarily interested in parallel performance.

The combination of the factor and solve tasks for a given column allows some simplifications in the code for adding updates, as the entire column is handled by a single thread. Unfortunately the relative sizes of the factor-solve and various update tasks are now heavily imbalanced. To ensure sufficient work is always available when running in parallel we reduce the blocking factor from that used in the positive-definite code, but still suffer from a long tail of (near) serial execution at the root of the factorization.

The modifications of the `HSL_MA64` code restrict the factorization to a single block column (the original performs a partial factorization on packed block columns). Both row-wise and column-wise storage variants have been tried, and it was found that a hybrid variant that uses the `HSL_MA87` row-wise storage for the update tasks but converts to and from column-wise storage for the factor-solve task gave best performance. This is due to the need to repeatedly scan data by columns during pivoting.

A further issue was dealing with delayed pivots. We have taken the simplistic approach of merely passing any columns which have been rejected for pivoting to the next block column, or the parent if there are no further block columns in the supernode. This can lead to some block columns containing substantially fewer or greater numbers of actual columns than predicted and can lead to highly unbalanced tasks.

The following pseudocode describes the current algorithm used for the factor_solve task:

```
if this is the first column of a supernode then
  ndelay = sum of delays from final column of child supernodes
else
  ndelay = delays from preceding column of supernode
end if
Allocate a new array of sufficient size to store this block column's data, plus an additional
ndelay columns
Copy this block column's data into the first part of array, converting from row to column
storage in the process
Copy any delayed columns from child supernodes or preceding column as appropriate
Call block column pivoting algorithm (modified HSL_MA64)
Convert eliminated columns back to row storage within the recently allocated array
Leave any delays in column storage ready for passing to parent or next column
```

8.1.2 Results on general problems

In an effort to judge how successful the version of the code used here is, we show results on some general problems in Table 8.1. This compares our DSIS code with `MA57`, `HSL_MA77` run in-core, and the Intel MKL version of PARDISO. All codes use default settings, except an ordering (obtained through `MA57`) is supplied, internal scaling is disabled and the matrix is externally scaled by $\texttt{MC77}_1$. This shows that our DSIS code is competitive on these problems, though is rarely the best serial code due to design decisions optimizing parallel performance. PARDISO has more limited pivoting options than the other codes: as a result of taking numerical short cuts it is by far the fastest on the *c-62* and *aug3d* problems. Similar performance can be obtained from other codes by changing their control settings. However, doing so can lead to failure to obtain an accurate solution.

Problem	`MA57`	`HSL_MA77`	PARDISO	DSIS
Boeing/msc01050	0.010	**0.004**	0.006	**0.004**
Cylshell/s3rmt3m1	0.064	**0.042**	0.050	**0.043**
Boeing/bcsstk38	0.152	**0.076**	0.089	0.081
Boeing/crystk01	0.144	**0.089**	0.109	**0.092**
Schenk_IBMNA/c-56	0.404	0.130	**0.072**	0.158
Simon/olafu	0.559	**0.234**	0.287	**0.233**
GHS_indef/stokes128	0.713	0.250	**0.225**	0.254
GHS_psdef/oilpan	1.45	0.986	**0.917**	1.02
GHS_psdef/s3dkq4m2	2.94	**1.80**	2.12	**1.89**
DNVS/ship_001	3.54	**2.14**	2.39	**2.20**
Koutsovasilis/F2	4.48	**2.42**	2.88	**2.54**
Cunningham/qa8fk	7.00	**4.13**	4.68	**4.29**
ND/nd3k	8.94	5.13	5.43	**4.80**
Schenk_IBMNA/c-62	19.7	7.26	**1.77**	9.85
Oberwolfach/t3dh	20.2	**11.7**	13.5	12.4
ND/nd6k	40.5	24.5	25.5	**22.8**
GHS_indef/aug3d	OOM	38.2	**0.105**	62.8
ND/nd12k	164.0	**104.8**	109.7	**99.9**
PARSEC/GaAsH6	482.7	**325.9**	375.8	**312.9**
GHS_indef/sparsine	537.0	376.5	499.6	**319.6**

Table 8.1: Comparison results for DSIS on general problems, ordered by DSIS factorize time. Results within 5% of the fastest are shown in bold. OOM indicates a problem for which a code ran out of memory.

Problem	ndelay	DSIS Time (Speedup)							PARDISO Time (Speedup)	
		1	2		4		8			
Boeing/msc01050	0	0.004	0.013	(0.30)	0.095	(0.04)	0.064	(0.06)	0.091	(0.07)
Cylshell/s3rmt3m1	0	0.043	0.030	(1.43)	0.073	(0.59)	0.166	(0.26)	0.122	(0.41)
Boeing/bcsstk38	48	0.081	0.051	(1.59)	0.134	(0.60)	0.092	(0.88)	0.110	(0.81)
Boeing/crystk01	0	0.092	0.070	(1.32)	0.137	(0.68)	0.128	(0.72)	0.038	(2.83)
Schenk_IBMNA/c-56	287	0.158	0.113	(1.40)	0.162	(0.98)	0.168	(0.94)	0.083	(0.87)
Simon/olafu	6	0.233	0.148	(1.57)	0.088	(2.66)	0.184	(1.26)	0.110	(2.62)
GHS_indef/stokes128	3972	0.254	0.154	(1.65)	0.161	(1.58)	0.167	(1.52)	0.109	(2.06)
GHS_psdef/oilpan	0	1.02	0.563	(1.81)	0.365	(2.79)	0.269	(3.79)	0.203	(4.53)
GHS_psdef/s3dkq4m2	0	1.89	1.05	(1.80)	0.655	(2.88)	0.453	(4.17)	0.377	(5.63)
DNVS/ship_001	2	2.20	1.24	(1.78)	0.728	(3.03)	0.447	(4.93)	0.442	(5.41)
Koutsovasilis/F2	0	2.54	1.40	(1.81)	0.757	(3.36)	0.540	(4.70)	0.533	(5.40)
Cunningham/qa8fk	0	4.29	2.39	(1.80)	1.43	(3.00)	0.864	(4.96)	0.848	(5.52)
ND/nd3k	0	4.80	2.74	(1.75)	1.72	(2.78)	1.09	(4.41)	1.10	(4.95)
Schenk_IBMNA/c-62	46459	9.85	7.82	(1.26)	6.44	(1.53)	5.57	(1.77)	0.414	(4.27)
Oberwolfach/t3dh	0	12.4	6.83	(1.81)	3.86	(3.21)	2.30	(5.40)	2.54	(5.29)
ND/nd6k	0	22.8	12.7	(1.80)	7.52	(3.03)	4.24	(5.38)	5.61	(4.54)
GHS_indef/aug3d	232943	62.8	56.7	(1.11)	53.1	(1.18)	49.1	(1.28)	0.074	(1.41)
ND/nd12k	0	99.9	55.2	(1.81)	32.0	(3.13)	17.4	(5.74)	28.2	(3.89)
PARSEC/GaAsH6	0	312.9	174.6	(1.79)	103.2	(3.03)	56.9	(5.50)	110.8	(3.39)
GHS_indef/sparsine	30	319.6	179.6	(1.78)	109.8	(2.91)	60.9	(5.25)	174.0	(2.87)

Table 8.2: Parallel scaling results for the factorize phase of DSIS on 2, 4 and 8 cores. The ndelay column gives the number of delayed pivots reported by DSIS. Factorization time and speedup is reported for PARDISO on 8 cores for comparison purposes.

We examine parallel performance in Table 8.2. This shows consistent scaling behaviour, with an increase in the speedup as the problem size increases. For small and mid-sized problems we are not as efficient as PARDISO, however for larger problems we are significantly faster. As in the positive-definite case, this is probably caused by the choice between one and two dimensional task decompositions.

The exception to the scaling behaviour occurs when large numbers of delayed pivots occur. This results in much larger factor-solve tasks, giving effectively serial factorization processes. Modifications to the code that would reduce these effects are beyond the scope of this chapter, but are planned for a full release of this code.

8.2 Ipopt interfaces

Ipopt is a non-linear interior point solver that uses the augmented system approach. It is written in C++ and implements interfaces to a number of solvers. We have written complementary interfaces to the mixed precision solver `HSL_MA79` (Chapter 5) and the DSIS code described in the previous section. We have additionally written an interface for `HSL_MA77` to allow comparisons with that code.

The interface must support the following features:

- Perform the analyse phase once and reuse it for multiple factorizations and solves.
- Perform a factorization and allow multiple solves. This allows Ipopt to implement its own iterative refinement.
- Detect deviations from expected inertia; indicate if the matrix is singular.

All these features are easily implemented for the HSL solvers.
We will use the following matrix factorization interfaces within Ipopt:

`MA57` used with the standard Ipopt interface, but with internal `MC64` scaling disabled (problems are already scaled and this slowed the code down substantially on most problems). A substantial difference from default `MA57` settings is the choice of pivoting parameter $u = 1 \times 10^{-8}$ (default in the Ipopt interface).

`HSL_MA77` used with a custom interface. `maxstore` is set to `huge(0)/8` and `bits=32`. The analyse phase of `MA57` is used to obtain the ordering, and otherwise default parameters are used except `nemin=16` (this was found to be better on modern machines) and $u = 1 \times 10^{-8}$ (in line with existing `MA57` interface).

`HSL_MA79` used with a custom interface and default parameters, except that the value of the parameters `small_sp` and `small_dp` are set to 1×10^{-20} as this seems to give better inertia detection in Ipopt. Again, $u = 1 \times 10^{-8}$ was chosen in line with existing Ipopt practice. As Ipopt does not currently support iterative methods we have opted to just require that `HSL_MA79` achieves double precision accuracy of $\beta = 2 \times 10^{-15}$. If a fall back to double precision factorization occurs then all future iterations will use a double rather than mixed precision factorization as the condition of the matrix is likely to get worse rather than better. While it may be possible to use reduced accuracy in early iterations we have chosen not to investigate this.

DSIS as described in the Section 8.1, used with a custom interface and pivoting parameter $u = 1 \times 10^{-8}$.

PARDISO used with the standard Ipopt interface.

The use of such a small pivoting parameter is considered unusual to general linear algebra practitioners, but is an established practice in optimization. This follows from practical experience in the optimization community and reflects the ability to correct numerical errors through perturbations of the problem at a higher algorithmic level.

MA57	MA77	MA79	DSIS	PARDISO
greenbea	*greenbea*	*greenbea*	*greenbea*	*d6cube*
pilot4	*pilot4*	*pilot4*	*pilot4*	*dfl001*
		stocfor3		*greenbea*
				pilot4

Table 8.3: Problems from the Netlib test set that failed to converge to an optimal solution with each code.

8.3 Numerical results

We performed tests on `fox` using g++ 4.1.2 (default Ipopt optimization flags), ifort 11.1 (with `-g -xT -O3`) and BLAS/LAPACK from the Intel MKL 10.1. The version of PARDISO used was *not* from the MKL, but instead version 4.0 from the pardiso-project website. This was due to advice that the MKL version did not support good inertia detection [Ipopt 2009; Schenk 2009], a required ability for Ipopt. Aside from controls on the linear solvers, default options were used to run Ipopt. This results in poor performance on linear problems as design choice have been made that favour robustness in the solution of non-linear problems. As some of our test problems are non-linear we choose to use the same options for all problems (we are comparing the linear algebra component rather than optimization algorithms after all).

As the test collections employed are large, full results have been placed in Appendix B, and only summary results are presented here. We ran Ipopt using our solvers on both the historical Netlib LP test set [Gay 1985] (to assess numerical reliability) and a selection from Mittleman's collection of benchmark problems [Mittelmann 2009] (to assess performance on larger systems).

8.3.1 Netlib results

As Netlib problems are relatively small by today's standards we consider them mainly in the context of numerical accuracy and the ability to solve difficult problems. All codes were able to solve most problems within 3000 iterations, though PARDISO failed on several where others succeeded. The failed problems are listed in Table 8.3.

Considering the 90 problems for which an optimum was obtained, the various codes produced iteration counts within 10% of each other on 63 problems. The remaining 27 are tabulated in Table 8.4. The main reason for these vastly differing results is the action taken when a matrix is found to have the wrong inertia (that is the matrix is determined to be singular or have the wrong number of negative pivots). In this case Ipopt will add a small multiple of the identity matrix to the upper left portion of the augmented system matrix (3.5). This reduces the condition number, but gives a poorer Newton direction, normally resulting in a larger iteration count. This is done in the linear case because obtaining an incorrect inertia is normally indicative of poor conditioning in the factorized matrix having lead to inaccurate signs of pivots (equivalent in a Cholesky factorization to finding negative pivots). In the non-linear case inertia correction is additionally used to detect and overcome non-convexity. An indication that this has occurred often is a large difference between iteration and factorization counts in Table 8.4.

The other reason some solvers require more iterations is that the iterates $(\boldsymbol{x}, \boldsymbol{z})$ are perturbed away from their boundaries if iterative refinement is unable to produce a sufficiently small residual. This perturbation often causes significantly more iterations to be required before convergence is achieved.

We can now suggest a reason for the significant differences between the performance of PARDISO and the other codes. PARDISO doesn't allow pivots to be delayed between supernodes, instead it uses a weighted-matching based permutation to move large pivots close to the diagonal. This permutation often pays off and results in the smaller iteration counts observed, however it is sometimes insufficient to overcome the ill conditioning, giving significantly larger iteration counts or failing to converge at all.

Full timing results are again included in Appendix B. We shall only comment to say that on these (typically very small) problems `MA57` substantially outperformed the other codes on almost all problems. The performance of `HSL_MA77` and PARDISO was similar on problems for which

Problem	Iteration (Factorization) count				
	MA57	HSL_MA77	HSL_MA79	DSIS	PARDISO
25fv47	**203** (211)	**205** (220)	257 (308)	**206** (218)	**203** (204)
bnl2	**287** (344)	**277** (304)	**289** (362)	**279** (307)	306 (330)
boeing2	**62** (73)	63 (70)	**57** (61)	**60** (62)	**57** (58)
bore3d	194 (202)	195 (202)	194 (239)	206 (211)	**137** (140)
cycle	171 (344)	195 (400)	171 (346)	187 (375)	**110** (125)
degen2	78 (141)	**62** (107)	102 (217)	96 (158)	**64** (65)
degen3	276 (283)	**62** (118)	84 (169)	**63** (120)	353 (374)
etamacro	113 (117)	113 (120)	113 (117)	114 (120)	**80** (81)
finnis	237 (496)	**153** (173)	**149** (177)	**154** (174)	**147** (148)
greenbeb	2718 (3745)	2319 (2578)	2295 (2512)	2729 (3808)	**2081** (2136)
lotfi	**58** (71)	**55** (57)	**55** (62)	61 (69)	**55** (57)
maros	783 (1245)	830 (1204)	813 (1351)	819 (1176)	**508** (509)
perold	1317 (1323)	**1152** (1156)	1278 (1285)	1346 (1350)	1344 (1345)
pilot_we	**229** (230)	**229** (231)	271 (306)	**229** (230)	**229** (230)
pilot87	363 (519)	337 (367)	**317** (372)	360 (419)	**306** (307)
pilotnov	340 (431)	340 (424)	340 (430)	340 (421)	**290** (291)
recipe	50 (92)	45 (52)	48 (57)	45 (78)	**33** (34)
scorpion	**35** (43)	**35** (43)	**35** (42)	45 (90)	**37** (38)
scrs8	**85** (120)	**85** (100)	116 (241)	**83** (101)	**83** (84)
share1b	**356** (357)	**357** (359)	**325** (337)	**355** (356)	359 (360)
ship04s	59 (63)	59 (66)	59 (66)	59 (63)	**48** (49)
ship08l	**85** (89)	**85** (92)	**83** (88)	**85** (89)	94 (95)
ship08s	158 (163)	156 (163)	162 (173)	157 (161)	**71** (72)
ship12l	99 (103)	**66** (74)	**66** (80)	**67** (72)	**69** (70)
ship12s	100 (104)	100 (107)	105 (118)	100 (104)	**70** (71)
stocfor3	159 (161)	**140** (147)	158* (163)	159 (160)	159 (160)
wood1p	**72** (118)	**73** (123)	**72** (117)	**72** (113)	81 (82)

Table 8.4: The 27 problems in Netlib on which iteration counts varied by more than ten per cent for problems that converged. Bold font indicates iteration counts within 10% of the best, and numbers in brackets indicate the actual number of factorizations performed. A star indicates that the problem did not reach an optimal solution.

they had similar factorization counts, and DSIS was typically slightly slower as in Section 8.1. The performance of `HSL_MA79` was highly erratic, but when performing well typically took longer than `MA57` but less time than `HSL_MA77`.

Problem	Run time		Iterations (Factorizations)		Time per factorization	
Solver:	MA57	HSL_MA79	MA57	HSL_MA79	MA57	HSL_MA79
Test Set 4						
cont11	246.9	**235.2**	64 (66)	62 (65)	3.74	3.62
cont1	**57.5**	80.2	50 (52)	48 (50)	1.11	1.60
cont4	**80.6**	116.3	84 (86)	81 (83)	0.94	1.40
neos1	378.6	**193.5**	269 (271)	269 (271)	1.40	0.71
neos2	378.5	**327.1**	309 (313)	309 (313)	0.93	1.05
neos3	1189.9	**325.7**	22 (29)	18 (20)	41.03	16.28
Test Set 5						
bearing_400	**66.6**	90.5	16 (16)	16 (16)	4.16	5.65
cont5_1	**96.8**	149.1	16 (26)	17 (23)	3.72	6.48
cont5_2_1	**669.8**	1560.7	69 (178)	95 (209)	3.76	7.47
cont5_2_2	**431.8**	480.4	62 (121)	50 (73)	3.57	6.58
cont5_2_3	**505.9**	596.8	76 (144)	59 (91)	3.51	6.56
cont5_2_4	221.4	**181.3**	14 (25)	12 (18)	8.85	10.07
dirichlet_120	**569.0**	1824.9	56 (119)	56 (119)	4.78	15.34
ex1_160	**43.6**	53.1	10 (11)	10 (11)	3.96	4.83
ex1_320	**41.9**	53.2	9 (10)	9 (10)	4.19	5.31
ex4_2_320	**43.8**	53.8	10 (11)	10 (11)	3.98	4.80
NARX_CFy	1593.8	**292.4**	352 (878)	241 (595)	1.82	0.49

Table 8.5: Showing comparative performance of the mixed precision solver HSL_MA79 against its base solver for some larger problems on Test Sets 4 and 5. The fastest solution time is in bold.

8.3.2 Larger problem results

We have drawn two sets of problems from those collected by Mittlemann [2009]:

Test Set 4 Linear problems with factorization times greater than one second under MA57.

Test Set 5 Nonlinear problems with factorization times greater than one second under MA57.

All tests on these problems were limited to one hour of run time and 32GiB of virtual memory. We give full details of the test sets in Appendix A, with full results appearing in Appendix B. When run in parallel both solvers could give slightly different solutions to the serial case due to changes in the order of operations. This sometimes resulted in wildly different iteration and factorization counts. To get results meaningful in comparison to the serial case we ran each problem 3 times and give the results from the problem with the closest iteration count to the serial run, or an average if there is a tie.

We will now consider separately the effectiveness of the mixed precision solver HSL_MA79 and the parallel solver DSIS.

Mixed precision

Table 8.5 shows comparative results for MA57 and HSL_MA79; these are compared as in all the listed problems MA57 was chosen as the base solver for HSL_MA79. The wall clock time, iteration and factorization counts are presented, together with the computed average time per factorization. This computed quantity gives an idea of the performance gained, but is not an ideal measure for comparison as different paths are taken through the problem, resulting in different matrices to be factored (possibly with significantly different numbers of delayed pivots). Further, it does not attempt to remove time spent in function evaluations and Hessian computations, which may represent non-trivial amounts of time for the nonlinear problems.

Of these problems 6 out of the 17 ran faster with HSL_MA79 than with MA57, however most of the rest ran considerably slower with the mixed precision approach. This was likely due to requiring too many steps of iterative refinement to recover a good descent direction. However,

Problem	DSIS				PARDISO
threads=	1	2	4	8	8
Test Set 4					
cont1	115.5	88.9 (1.30)	73.1 (1.58)	62.6 (1.85)	31.2 (1.69)
cont4	166.3	138.7 (1.20)	110.8 (1.50)	96.5 (1.72)	95.0 (1.42)
neos1	802.5	728.9 (1.10)	508.9 (1.58)	401.6 (2.00)	422.6 (3.00)
neos2	898.6	713.0 (1.26)	554.1 (1.62)	430.3 (2.09)	395.4 (2.73)
neos3	550.6	416.9 (1.32)	291.0 (1.89)	244.0 (2.26)	1411.3 (2.53)
ns1687037	404.9	345.3 (1.17)	303.9 (1.33)	241.8 (1.67)	193.3 (2.56)
Test Set 5					
bearing_400	70.6	65.6 (1.08)	62.2 (1.14)	61.4 (1.15)	57.9 (1.16)
cont5_1	65.7	48.3 (1.36)	40.2 (1.63)	33.6 (1.96)	9.6 (2.88)
cont5_2_1	307.4	237.2 (1.30)	143.5 (2.14)	114.2 (2.69)	19.0 (3.25)
cont5_2_4	130.2	87.2 (1.49)	76.1 (1.71)	72.6 (1.79)	33.25 (1.65)
dirichlet_120	441.7	315.0 (1.40)	234.7 (1.88)	195.3 (2.26)	205.2 (2.82)
ex1_160	52.5	41.8 (1.26)	34.5 (1.52)	32.3 (1.63)	15.3 (1.80)
ex1_320	44.3	35.0 (1.27)	29.6 (1.50)	26.7 (1.66)	14.4 (1.78)
ex4_2_320	52.5	41.8 (1.26)	34.4 (1.53)	31.3 (1.68)	15.3 (1.80)

Table 8.6: Showing parallel performance of the solver DSIS inside Ipopt. Results for PARDISO (on 8 threads only) are supplied for comparison. All times are wall clock times, and parallel speedups are shown in brackets.

we achieve impressive speedups on some problems (*neos3*, *NARX_CFy*) and moderate speedups on others. Overall this seems to mimic the results for mixed precision on general matrices: for some problems it works really well, but for others it is considerably worse.

The interface could be improved to run one iteration in double precision and one in mixed precision and then choose the faster factorization method. This would hopefully give the best performance with little penalty.

Parallel

Table 8.6 shows the performance of DSIS when used in parallel. While it does not perform as well as PARDISO, it still produces comparable speedups on most problems. The performance gap is largely due to encountering large numbers of delayed pivots in these problems. As we recall from our tests on general indefinite problems this causes the DSIS factorization to serialise. Despite this, it is of particular interest to observe that on 8 threads it significantly beats the serial performance of all the HSL solvers (that is `MA57`, `HSL_MA77` and `HSL_MA79`).

The relatively low parallel efficiencies given by both PARDISO and DSIS are caused by several factors, including serial overheads inside Ipopt, the repeated stop-starting of the parallel regions and the execution of a large number of forward and back solves that only run in serial. If parallelism could be induced throughout the interior point code then better speedups would be expected (though may be limited by memory bandwidth in the solve phase).

8.4 Conclusions

We have shown that significant gains in the performance of interior point methods can be achieved through the use of the methods we have developed. In particular for certain problems the mixed precision approach can achieve a solution in roughly half the time.

While the performance of DSIS is sometimes disappointing compared to PARDISO at present, the results of `HSL_MA87` for positive-definite problems suggest that with further development it could become competitive. Even as is, it offers worthwhile gains over a serial code if a multicore machine is available, especially on larger problems where existing one dimensional parallel dissections fail to exploit the full extent of parallelism available in the problem.

Chapter 9

Conclusions and Future Work

This thesis has presented a number of techniques and investigations aimed at accelerating direct methods for the solution of $A\boldsymbol{x} = \boldsymbol{b}$ for symmetric A on modern multicore machines.

Particular success has been demonstrated for the solution of large sparse symmetric systems with the development of library quality codes that exploit mixed precision and DAG-based techniques. However, limited success was obtained when applying these techniques to interior point methods.

Future work would logically proceed down several paths.

The simplest of these is to change the way delayed pivots are propagated within the calculation. The current technique of passing them to the next block column in the supernode or tree often results in the same poor pivot candidates being scanned repeatedly and additionally causes substantial load imbalance. Modifying the task representation so that block columns with insufficient pivots are back filled when using subsequent columns is one obvious method to examine; this would result in similar behavior with respect to the number of column scans and delayed pivots exhibited by existing codes, such as the out-of-core multifrontal code `HSL_MA77`. Other methods could also be experimented with, for example each dense submatrix factorization could use use recursive bisection to build a binary tree of block columns, and failed pivots could then flow up the tree, reducing the number of examination from the number of block columns to the log there of.

Another obvious strand to pursue is to explore alternative pivoting techniques to avoid or limit the need for columnwise scanning and/or delayed pivots. The constrained or compressed weighted matching based orderings of Duff and Pralet [2007] could be pursued, possibly with a block diagonal variant of the random butterfly matrix scaling described by Parker [1995]. Alternatively, or in conjunction, an a posteri approach to pivoting could be taken. Bunch-Kaufamann pivoting could be performed restricted to the diagonal block of each block column, and growth measured in the blocks below the diagonal with each triangular solve task. If a pivot is found to have been poor then the block column is reverted to a previously version and full pivoting performed. Such a technique would accelerate factorizations that do not encounter more than a handful of delayed pivots by removing the need to scan and update entire columns during the factor or factor_solve task. In our experience many problems exhibit this characteristic after scaling has been applied.

The scheduler is a rich source of research problems, especially when considered in combination with supernode amalgamation. It may even be profitable from a minimizing communication point of view to use different supernode amalgamations at different points in the calculation. The design of the sparse DAG code developed here allows the scheduler to be easily swapped out, allowing easy experimentation with this technique. It is unclear how far from optimal current supernode amalgamation techniques are and the development of an accurate computational model could aid in the study of such. Finally, to many users the repeatability of results is more critical than performance. Our current scheduling algorithm gets different (but equally numerically valid) answers on each run due to differing orders in which update tasks are performed. It is hence of interest to develop an algorithm capable of delivering repeatable results for minimal loss of performance.

With respect to improving performance with interior point problems, one reason for the relatively poor performance described is the heavy reliance of the tested interior point code on aggressive iterative refinement. This results in much of the time being spent in the solve rather than factorize phase. As the solve phase is strongly memory-bound, scaling on multicore systems is extremely limited. Further work will be to examine the solve phase in detail and determine what techniques can be used to reduce the memory bandwidth used.

Finally, the manycore future may well feature GPU or GPU-like architectures. It would be worthwhile to explore how direct solvers can be adapted to run on such chips that are extremely efficient with dense vector operations but relatively inefficient with sparse operations. Memory bandwidth is higher with these chips, but is also more critical than ever for performance.

Bibliography

Agullo, E., Hadri, B., Ltaief, H., and Dongarra, J. 2009. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 1–12. Also appears as UT-CS-09-640.

Amestoy, P. R., Davis, T. A., and Duff, I. S. 1996. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications 17*, 886–905.

Amestoy, P. R., Davis, T. A., and Duff, I. S. 2004. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software 30,* 3, 381–388.

Amestoy, P. R., Duff, I. S., L'Excellent, J.-Y., and Koster, J. 2001. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications 23,* 1, 15–41.

Amestoy, P. R., Guermouche, A., L'Excellent, J.-Y., and Pralet, S. 2006. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing 32,* 2, 136–156.

Andersen, B. S., Gustavson, F., and Wasniewski, J. 2001. A recursive formulation of Cholesky factorization of a matrix in packed storage. *ACM Transactions on Mathematical Software 27,* 2, 214–244. Also appears as LAPACK Working Note 146.

Anderson, B. S., Gunnels, Gustavson, F., Reid, J. K., and Wasniewski, J. 2005. A fully portable high performance minimal storage hybrid format Cholesky algorithm. *ACM Transactions on Mathematical Software 31,* 2, 201–227.

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. 1999. *LAPACK Users' Guide*, Third ed. Society for Industrial and Applied Mathematics, Philadelphia, PA.

Anderson, E. and Dongarra, J. J. 1990. Evaluating block algorithm variants in LAPACK. In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing.* Society for Industrial and Applied Mathematics, 3–8. Also appears as LAPACK Working Note 19.

Arioli, M. and Duff, I. S. 2009. Using FGMRES to obtain backward stability in mixed precision. *Electronic Transaction on Numerical Analysis 33*, 31–44. Also appears as RAL-TR-2008-006.

Arioli, M., Duff, I. S., Gratton, S., and Pralet, S. 2007. A note on GMRES preconditioned by a perturbed LDL^T decomposition with static pivoting. *SIAM Journal on Scientific Computing 29,* 5, 2024–2044.

Ashcraft, C. and Grimes, R. 1989. The influence of relaxed supernode partitions on the multifrontal method. *ACM Transactions on Mathematical Software 15,* 4, 291–309.

Bauer, F. L. 1963. Optimally scaled matrices. *Numerische Mathematik 5,* 1 (12), 73–87.

Benzi, M., Haws, J. C., and Tuma, M. 2001. Preconditioning highly indefinite and nonsymmetric matrices. *SIAM Journal on Scientific Computing 22,* 4, 1333–1353.

Biggar, P. and Gregg, D. 2005. Sorting in the presence of caches and branch predictors. Technical Report TR-2005-57, Department of Computer Science, Trinity College Dublin. August.

Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., and Whaley, R. C. 1997. *ScaLAPACK Users' Guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA.

Bunch, J. R. and Kaufman, L. 1977. Some stable methods for calculating inertia and solving symmetric linear systems. *Mathematics of Computation 31,* 137, 163–179.

Bunch, J. R. and Parlett, B. N. 1971. Direct methods for solving symmetric indefinite systems of linear equations. *SIAM Journal on Numerical Analysis 8,* 4, 639–655.

Buttari, A., Dongarra, J., and Kurzak, J. 2007. Limitations of the playstation 3 for high performance cluster computing. Tech. Rep. CS-07-594, University of Tennessee Computer Science.

Buttari, A., Dongarra, J., Kurzak, J., Langou, J., Luszczek, P., and Tomov, S. 2006. The impact of multicore on math software. In *Proceedings of Workshop on State-of-the-art in Scientific and Parallel Computing (Para06).*

Buttari, A., Dongarra, J., Kurzak, J., Luszczek, P., and Tomov, S. 2008. Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *ACM Transactions on Mathematical Software 34*, 17:1–17:22.

Buttari, A., Dongarra, J., Langou, J., Luszczek, P., and Kurzak, J. 2007. Mixed precision iterative refinement techniques for the solution of dense linear systems. *Int. Journal of High Performance Computing Applications 21,* 4, 457–466.

Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. 2009. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing 35,* 1, 38–53. also appears as LAPACK Working Note 191.

Carolan, W. J., Hill, J. E., Kennington, J. L., Niemi, S., and Wichmann, S. J. 1990. An empirical evaluation of the KORBX algorithms for military airlift applications. *Operations Research 38,* 2, 240–248.

Chapman, B., Jost, G., and Pas, R. V. D. 2008. *Using OpenMP.* The MIT Press.

Chen, Y., Davis, T. A., Hager, W. W., and Rajamanickam, S. 2008. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software 35.* Article 22 (14 pages).

Colombo, M. and Gondzio, J. 2008. Further development of multiple centrality correctors for interior point methods. *Computational Optimization and Applications 41,* 3, 277–305.

Curtis, A. R. and Reid, J. K. 1972. On the automatic scaling of matrices for Gaussian elimination. *IMA Journal of Applied Mathematics 10,* 1, 118–124.

Davis, T. 2007. The University of Florida sparse matrix collection. Technical report, University of Florida. http://www.cise.ufl.edu/~davis/techreports/matrices.pdf.

Davis, T. A. 2006. *Direct Methods for Sparse Linear Systems.* SIAM.

Demmel, J., Hida, Y., Kahan, W., Li, X. S., Mukherjeek, S., and Riedy, E. J. 2006. Error bounds from extra precise iterative refinement. *ACM Transactions on Mathematical Software 32,* 2, 325–351.

Demmel, J., Hida, Y., Li, X. S., and Riedy, E. J. 2009. Extra-precise iterative refinement for overdetermined least squares problems. *ACM Transactions on Mathematical Software 35,* 4. Also LAPACK Working Note 188.

Demmel, J. W., Eisenstat, S. C., Gilbert, J. R., Li, X. S., and Liu, J. W. H. 1999. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications 20,* 3, 720–755.

Dijkstra, E. W. 1965. Cooperating sequential processes. Tech. rep., Technological University, Eindhoven. September.

Dolan, E. D. and Moré, J. J. 2002. Benchmarking optimization software with performance profiles. *Mathematical Programming 91,* 2, 201–213.

Dolan, E. D., Moré, J. J., and Munson, T. S. 2004. Benchmarking optimization software with COPS 3.0. Technical Report ANL/MCS-273, Mathematics and Computer Science Division, Argonne National Laboratory. 2.

Dongarra, J. J., Croz, J. D., Hammarling, S., and Duff, I. S. 1990. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software 16,* 1, 1–17.

Dongarra, J. J., Croz, J. D., Hammarling, S., and Hanson, R. J. 1986. An extended set of Fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software 14*, 1–17.

Duff, I. S. 2004. MA57 — a new code for the solution of sparse symmetric definite and indefinite systems. *ACM Transactions on Mathematical Software 30*, 118–154.

Duff, I. S., Erisman, A. M., and Reid, J. K. 1986. *Direct methods for sparse matrices.* Oxford Science Publications, Oxford.

Duff, I. S. and Koster, J. 2001. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications 22,* 4, 973–996.

Duff, I. S. and Pralet, S. 2005. Strategies for scaling and pivoting sparse symmetric indefinite problems. *SIAM Journal on Matrix Analysis and Applications 27,* 2, 313–340. Also appears as RAL-TR-2004-020.

Duff, I. S. and Pralet, S. 2007. Towards a stable static pivoting strategy for the sequential and parallel solution of sparse symmetric indefinite systems. *SIAM Journal on Matrix Analysis and Applications 29*, 1007–1024.

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

Duff, I. S. and Reid, J. K. 1996. Exploiting zeros on the diagonal in the direct solution of indefnite sparse symmetric linear systems. *ACM Transactions on Mathematical Software 22,* 2, 227–257.

Duff, I. S. and Scott, J. A. 2005. Towards an automatic ordering for a symmetric sparse direct solver. Technical Report RAL-TR-2006-001, Rutherford Appleton Laboratory.

Fourer, R. and Mehrotra, S. 1993. Solving symmetric indefinite systems in an interior-point method for linear programming. *Mathematical Programming 62,* 1, 15–39.

Gay, D. M. 1985. Electronic mail distribution of linear programming test problems. *Mathematical Programming Society COAL Newsletter 13*, 10–12.

George, A. and Liu, W. H. 1989. The evolution of the minimum degree ordering algorithm. *SIAM Review 31,* 1, 1–19.

Gilbert, J. R., Ng, E. G., and Peyton, B. W. 1994. An efficient algorithm to compute row and column counts for sparse Cholesky factorization. *SIAM Journal on Matrix Analysis and Applications 15,* 4, 1075–1091.

Gill, P. E. and Murray, W. 1974. Newton-type methods for unconstrained and linearly constrained optimization. *Mathematical Programming 7,* 1, 311–350.

Gondzio, J. 1995. HOPDM (version 2.12) — a fast LP solver based on a primal-dual interior point method. *European Journal of Operational Research 85*, 221–225.

Gondzio, J. 1996. Multiple centrality corrections in a primal-dual method for linear programming. *Computational Optimization and Applications 6,* 2, 137–156.

Goto, K. and van de Geijn, R. 2008. High performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software 35,* 1, 1–14.

Gould, N. I. M., Scott, J. A., and Hu, Y. 2007. A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations. *ACM Transactions on Mathematical Software 33,* 2, 10:1 – 10:32.

Graham, S. L., Snir, M., and Patterson, C. A. 2004. *Getting Up to Speed: The Future of Supercomputing.* National Academies Press.

Gropp, W., Lusk, E., and Thakur, R. 1999. *Using MPI-2, Advanced Features of the Message-Passing Interface.* The MIT Press.

Gupta, A., Joshi, M., and Kumar, V. 2001. WSMP: A high-performance serial and parallel sparse linear solver. Technical Report RC 22038 (98932), IBM T.J. Watson Reserach Center. www.cs.umn.edu/˜agupta/doc/wssmp-paper.ps.

Gupta, A., Karypis, G., and Kumar, V. 1997. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Trans. Parallel and Distributed Systems 8,* 5, 502–520.

Heggernes, P., Eisenstat, S., Kumfert, G., and Pothen, A. 2001. The computational complexity of the minimum degree algorithm.

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

Higham, N. J. 1997. Iterative refinement for linear systems and LAPACK. *IMA Journal of Numerical Analysis 17*, 495–509.

Higham, N. J. 2002. *Accuracy and Stability of Numerical Algorithms.* SIAM.

Hogg, J. D. 2008. A DAG-based parallel Cholesky factorization for multicore systems. Technical Report RAL-TR-2008-029, Rutherford Appleton Laboratory.

Hogg, J. D., Reid, J. K., and Scott, J. A. 2010. Design of a multicore sparse Cholesky solver using DAGs. *SIAM Journal on Scientific Computing 32,* 6, 3627–3649.

Hogg, J. D. and Scott, J. A. 2008. The effects of scalings on the performance of a sparse symmetric indefinite solver. Technical Report RAL-TR-2008-007, Rutherford Appleton Laboratory.

Hogg, J. D. and Scott, J. A. 2010a. A fast and robust mixed-precision solver for the solution of sparse symmetric linear systems. *ACM Transactions on Mathematical Software 37,* 2. to appear. Preprint as RAL-TR-2008-023.

Hogg, J. D. and Scott, J. A. 2010b. A note on the solve phase of a multicore solver. RAL-TR-2010-007.

HSL. 2007. A collection of Fortran codes for large-scale scientific computation. See http://www.cse.clrc.ac.uk/nag/hsl/.

IEEE. 2008. IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, 1–58.

Intel Corporation. 1996. *Cluster OpenMP User's Guide.* Intel Corporation.

Intel Corporation. 2006. *IA-32 Intel architecture optimization: reference manual.* Intel Corporation. URL: `http://www.intel.com/design/pentium4/manuals/index_new.htm`.

Intel Corporation. 2007. *Intel 64 and IA-32 Intel Architectures Optimization Reference Manual.* Intel Corporation.

Ipopt 2009. Ticket #88: Working interface for Intel MKL Pardiso. https://projects.coin-or.org/Ipopt/ticket/88.

Irony, D., Shklarski, G., and Toledo, S. 2004. Parallel and fully recursive multifrontal cholesky. *Journal of Future Generation Computer Systems 20,* 3, 425–440.

ISO 1997. *ISO 1539-1997.* The Fortran 95 standard.

ISO 2004. *ISO/IEC 1539-1:2004.* The Fortran 2003 standard.

Karypis, G. and Kumar, V. 1995. A fast and high quality multilevel scheme for partitioning irregular graphs. In *International Conference on Parallel Processing.* 113–122.

Karypis, G. and Kumar, V. 1998. METIS - family of multilevel partitioning algorithms. See http://glaros.dtc.umn.edu/gkhome/views/metis.

Karypis, G. and Kumar, V. 1999. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing 20*, 359–392.

Kurzak, J. and Dongarra, J. 2009. Fully dynamic scheduler for numerical computing on multicore processors. Tech. Rep. 220, LAPACK Working Note. June. Also appears as UT-CS-09-643.

Kurzak, J. and Dongarra, J. J. 2007. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. In *Applied Parallel Computing: State of the Art in Scientific Computing*, B. Kågström, E. Elmoth, J. Dongarra, and J. Wasniewski, Eds. 147–156. Also appears as LAPACK Working Note 178.

Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software 5,* 3, 308–323.

Li, X. S. 2005. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Transactions on Mathematical Software 31,* 3, 302–325.

Li, X. S. and Demmel, J. W. 1998. Making sparse Gaussian elimination scalable by static pivoting. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM).*

Li, X. S. and Demmel, J. W. 2003. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software 29,* 2.

Liu, J. W. 1986. A compact row storage scheme for Cholesky factors using elimination trees. *ACM Transactions on Mathematical Software 12,* 2, 127–148.

Liu, J. W. 1990. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications 11,* 1, 134–172.

Liu, J. W. H., Ng, E. G., and Peyton, B. W. 1993. On finding supernodes for sparse matrix computations. *SIAM Journal on Matrix Analysis and Applications 14,* 1, 242–252.

Maurer, H. and Mittelmann, H. D. 2000. Optimization techniques for solving elliptic control problems with control and state constraints. part 1: Boundary control. *Computational Optimization and Applications 16*, 29–55.

Mehrotra, S. 1992. On the implementation of a primal-dual interior point method. *SIAM Journal on Optimization 2,* 4, 575–601.

Message Passing Interface Forum 2008a. *MPI: A Message-Passing Interface Standard, version 1.3.* Message Passing Interface Forum.

Message Passing Interface Forum 2008b. *MPI: A Message-Passing Interface Standard, version 2.1.* Message Passing Interface Forum.

Mittelmann, H. D. 2001. Sufficient optimality for discretized parabolic and elliptic control problems. In *Fast solution of discretized optimization problems*, K.-H. Hoffmann, R. Hoppe, , and V. Schulz, Eds. Birkhaeuser.

Mittelmann, H. D. 2009. Decision tree for optimization software (benchmarks). http://plato.asu.edu/bench.html.

Mittelmann, H. D., Pendse, G., Rivera, D. E., and Lee, H. 2007. Optimization-based design of plant-friendly multisine signals using geometric discrepancy criteria. *Computational Optimization and Applications 38*, 173–190.

Mittelmann, H. D. and Troeltzsch, F. 2001. Sufficient optimality in a parabolic control problem. In *Proceedings of the first International Conference on Industrial and Applied Mathematics in Indian Subcontinent*, P. Manchanda, A. Siddiqi, , and M. Kocvara, Eds. Kluwer.

OpenMP Architecture Review Board 2008. *OpenMP Application Programming Interface, v3.0.* OpenMP Architecture Review Board.

Parker, D. 1995. Random butterfly transformations with applications in computational linear algebra. Tech. rep., University of California.

Perez, J. M., Badia, R. M., and Labarta, J. 2008. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing.* 142–151.

Pralet, S. 2004. Constrained orderings and scheduling for parallel sparse linear algebra. Ph.D. thesis, CERFACS.

Randall, K. H. 1998. Cilk: Efficient multithreaded computing. Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Reid, J. K. and Scott, J. A. 2008. An efficient out-of-core sparse symmetric indefinite direct solver. Technical Report RAL-TR-2008-024, Rutherford Appleton Laboratory.

Reid, J. K. and Scott, J. A. 2009a. Algorithm 891: A Fortran virtual memory system. *ACM Transactions on Mathematical Software 36,* 1, 1–12.

Reid, J. K. and Scott, J. A. 2009b. An out-of-core sparse Cholesky solver. *ACM Transactions on Mathematical Software 36,* 2, 1–33.

Reid, J. K. and Scott, J. A. 2009c. Partial factorization of a dense symmetric indefinite matrix. Technical Report RAL-TR-2009-015, Rutherford Appleton Laboratory.

Rothberg, E. and Gupta, A. 1993. An evaluation of left-looking, right-looking and multifrontal approaches to sparse Cholesky factorization on hierarchical memory machines. *Int. Journal of High Speed Computing 5,* 4, 537–593. Also appears as STAN-CS-91-1377.

Rotkin, V. and Toledo, S. 2004. The design and implementation of a new out-of-core sparse Cholesky factorization method. *ACM Transactions on Mathematical Software 30,* 1, 19–46.

Ruiz, D. 2001. A scaling algorithm to equilibrate both rows and columns norms in matrices. Tech. Rep. RAL-TR-2001-034, Rutherford Appleton Laboratory.

Saad, Y. 1994. A flexible inner-outer preconditioned GMRES algorithm. *SIAM Journal on Scientific and Statistical Computing 14*, 461–469.

SAAD, Y. 2003. *Iterative Methods for Sparse Linear Systems*, 2 ed. SIAM, 273–275.

SCHENK, O. 2009. [Ipopt] IPOPT and PARDISO. Message to Ipopt mailing list. http://list.coin-or.org/pipermail/ipopt/2009-June/001583.html.

SCHENK, O. AND GÄRTNER, K. 2004. Solving unsymmetric sparse systems of linear equations with PARDISO. *Journal of Future Generation Computer Systems 20*, 475–487.

SCHENK, O. AND GÄRTNER, K. 2006. On fast factorization pivoting methods for sparse symmetric indefinite systems. *Electronic Transaction on Numerical Analysis 23*, 158–179.

SCHENK, O., GÄRTNER, K., AND FICHTNER, W. 1999. Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors. *BIT Numerical Mathematics 30,* 1, 158–176.

SKEEL, R. D. 1980. Iterative refinement implies numerical stability for Gaussian elimination. *Mathematics of Computation 35*, 817–832.

SNIR, M., OTTO, S. W., HUSS-LEDERMAN, S., WALKER, D. W., AND DONGARRA, J. 1996. *MPI, The Complete Reference.* The MIT Press.

SONG, F., YARKHAN, A., AND DONGARRA, J. 2009. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. Tech. Rep. UT-CS-09-638, University of Tennessee. Apr.

STRAZDINS, P. E. 2001. A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. *Int. Journal of Parallel and Distributed Systems and Networks 4,* 1 (June), 26–35.

TREFETHEN, L. N. AND BAU III, D. 1997. *Numerical Linear Algebra.* SIAM.

VAVASIS, S. A. AND YE, Y. 1996. A primal-dual interior point method whose running time depends only on the constraint matrix. *Mathematical Programming 74,* 1, 79–120.

WÄCHTER, A. AND BIEGLER, L. 2006. On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. *Mathematical Programming 106,* 1, 25–57.

WILKINSON, J. H. 1968. Á priori error analysis of algebraic processes. In *Proceedings International Conference of Mathematicians.* 629–639.

WRIGHT, S. J. 1994. Stability of linear equations solvers in interior-point methods. *SIAM Journal on Matrix Analysis and Applications 16*, 1287–1307.

WRIGHT, S. J. 1997. *Primal-Dual Interior-Point Methods.* SIAM.

WRIGHT, S. J. 1999. Modified cholesky factorizations in interior-point algorithms for linear programming. *SIAM Journal on Optimization 9,* 4, 1159–1191.

YANNAKAKIS, M. 1981. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic and Discrete Methods 2,* 12, 77–79.

ZEE, F. G. V., BIENTINESI, P., LOW, T. M., AND VAN DE GEIJN, R. A. 2008. Scalable parallelization of flame code via the workqueuing model. *ACM Transactions on Mathematical Software 34,* 2 (Mar.).

Appendix A

Test set details

Details of test problems drawn from the University of Florida sparse matrix collection [Davis 2007]. Summary details taken from its index:

m is the number of rows and columns in the matrix.

nnz(A) is the number of non-zeroes in the matrix (both upper and lower triangles).

Pos Def indicates whether the matrix is positive definite.

Singular indicates whether a matrix is structurally singular.

Test Set X indicates whether a matrix is a member of that test set.

Name	*m*	*nnz(A)*	*Pos Def*	*Singular*	*Test Set 1*	*Test Set 2*	*Test Set 3*
ACUSIM							
Pres_Poisson	14822	715804	Yes		•	•	•
Alemdar							
Alemdar	6245	42581			•		
AMD							
G2_curcuit	150102	726674	Yes				•
G3_curcuit	1585478	7660826	Yes				•
Andrews							
Andrews	60000	760154	Yes		•		•
Andrianov							
ex3sta1	16782	678998					•
fxm4_6	18892	497844					•
ins2	309412	2751484					•
lp1	534388	1643420					•
lpl1	32460	328036					•
mip1	66463	10352819					•
net4-1	88343	2441777					•
net50	16320	945200					•
net75	23120	1489200					•
net100	29920	2033200					•
net125	36720	2557200					•
net150	43520	3121200					•
pattern1	19242	9323432					•

Bai

Name	*m*	*nnz(A)*	*Pos Def*	*Singular*	*Test Set 1*	*Test Set 2*	*Test Set 3*
bfwb62	62	342			●		
bfwb398	398	2910			●		
bfwb782	782	5982			●		
mhd3200b	3200	18316	Yes		●	●	
mhd4800b	4800	27520	Yes		●	●	
mhdb416	416	2312	Yes		●		
odepb400	400	399		Yes	●		

Bates

Name	*m*	*nnz(A)*	*Pos Def*	*Singular*	*Test Set 1*	*Test Set 2*	*Test Set 3*
Chem97ZtZ	2541	7361	Yes		●	●	

BenElechi

Name	*m*	*nnz(A)*	*Pos Def*	*Singular*	*Test Set 1*	*Test Set 2*	*Test Set 3*
BenElechi1	245874	13150496	Yes				●

Bindel

Name	*m*	*nnz(A)*	*Pos Def*	*Singular*	*Test Set 1*	*Test Set 2*	*Test Set 3*
ted_B	10605	144759	Yes		●	●	●
ted_B_unscaled	10605	144759	Yes		●	●	●

Boeing

Name	*m*	*nnz(A)*	*Pos Def*	*Singular*	*Test Set 1*	*Test Set 2*	*Test Set 3*
bcsstk34	588	21418	Yes		●		
bcsstk35	30237	1450163			●	●	●
bcsstk36	23052	1143430	Yes		●	●	●
bcsstk37	25503	140977			●	●	●
bcsstk38	8032	355460	Yes		●	●	
bcsstk39	46772	2060662			●	●	●
bcsstm34	588	24270			●		
bcsstm35	30237	20619		Yes	●	●	●
bcsstm36	23052	320606		Yes	●	●	●
bcsstm37	25503	15525		Yes	●	●	●
bcsstm38	8032	10485		Yes	●	●	
bcsstm39	46772	46772	Yes		●	●	●
crystk01	4875	315891			●	●	
crystk02	13965	968583				●	●
crystk03	24696	1751178				●	●
crystm01	4875	105339	Yes		●	●	
crystm02	13965	322905	Yes		●	●	●
crystm03	24696	583770	Yes			●	●
ct20stif	52329	2600295	Yes		●	●	●
msc00726	726	34518	Yes		●		
msc01050	1050	26198	Yes		●	●	
msc01440	1440	44998	Yes		●	●	
msc04515	4515	97707	Yes		●	●	
msc10848	10848	1229776	Yes		●	●	●
msc23052	23052	1142686	Yes		●	●	●
nasa1824	1824	39208			●	●	
pcrystk02	13965	968583					●
pcrystk03	24969	1751178					●
pct20stif	52329	2698463					●
pwtk	217918	11524432	Yes				●

Name	*m*	*nnz(A)*	*Pos Def*	*Singular*	*Test Set 1*	*Test Set 2*	*Test Set 3*
Cannizzo							
sts4098	4098	73256	Yes		●	●	
Chen							
pkustk01	22044	979380					●
pkustk02	10800	810000					●
pkustk03	63336	3130416					●
pkustk04	55590	4218660					●
pkustk05	37164	2205144					●
pkustk06	43164	2571768					●
pkustk07	16860	2418660					●
pkustk08	22209	3226671					●
pkustk09	33960	1583640					●
pkustk10	80676	4308984					●
pkustk11	87804	5217912					●
pkustk12	94653	7512317					●
pkustk13	94893	6616827					●
pkustk14	151926	14836504					●
Cote							
vibrobox	12328	301700			●	●	●
Cunningham							
m3plates	11107	6639		Yes	●	●	●
qa8fk	66127	1660579			●	●	●
qa8fm	66127	1660579	Yes		●	●	●
Clyshell							
s1rmq4m1	5489	262411	Yes		●	●	
s1rmt3m1	5489	217615	Yes		●	●	
s2rmq4m1	5489	263351	Yes		●	●	
s2rmt3m1	5489	217681	Yes		●	●	
s3rmq4m1	5489	262943	Yes		●	●	
s3rmt3m1	5489	217669	Yes		●	●	
s3rmt3m3	5357	207123	Yes		●	●	
DNVS							
crplat2	18010	960946					●
fcondp2	201822	11294316					●
fullb	199187	11708077					●
halfb	224617	12387821					●
m_t1	97578	9573570	Yes			●	●
ship_001	34920	3896496	Yes			●	●
ship_003	121728	3777036	Yes				●
shipsec1	140874	3568176	Yes				●
shipsec5	179860	4598604	Yes				●
shipsec8	114919	3303553	Yes				●
thread	29736	4444880	Yes			●	●
trdheim	22098	1935324					●
troll	213453	11985111					●
tsyl201	20685	2454957					●
x104	108384	8713602	Yes				●

Name	*m*	*nnz(A)*	*Pos Def*	*Singular*	*Test Set 1*	*Test Set 2*	*Test Set 3*
FIDAP							
ex2	441	26839			●		
ex3	1821	52685	Yes		●	●	
ex4	1601	31849			●	●	
ex5	27	279	Yes		●		
ex9	3363	99471	Yes		●	●	
ex10	2410	54840	Yes		●	●	
ex10hs	2548	57308	Yes		●	●	
ex12	3973	79077			●	●	
ex13	2568	75628	Yes		●	●	
ex14	3251	65875			●	●	
ex15	6867	98671	Yes		●	●	
ex32	1159	11047			●	●	
ex33	1733	22189	Yes		●	●	
Gaertner							
nopoly	10774	70842			●	●	●
GHS_indef							
a0nsdsil	80016	355034			●	●	●
a2nnsnsl	80016	347222				●	●
a5esindl	60008	255004			●	●	●
aug2d	29008	76832		Yes		●	●
aug2dc	30200	80000		Yes		●	●
aug3d	24300	69984		Yes		●	●
aug3dcqp	35543	1228115			●	●	●
blockqp1	60012	640033			●	●	●
bloweya	30004	150009			●	●	●
bloweybl	30003	109999		Yes	●	●	●
bloweybq	10001	49999	Yes		●	●	●
bmw3_2	227362	11288630					●
boyd1	93279	1211231				●	●
boyd2	466316	1500397					●
brainpc2	27607	179395			●	●	●
bratu3d	27792	173796			●	●	●
c-55	32780	403450			●	●	●
c-58	37595	552551			●	●	●
c-59	41282	480536			●	●	●
c-62ghs	41731	559339			●	●	●
c-63	44234	434704			●	●	●
c-68	64810	565996			●	●	●
c-69	67458	623914			●	●	●
c-70	68924	658986			●	●	●
c-71	76638	859520				●	●
c-72	84064	707546			●	●	●
cont-201	80595	438795			●	●	●
cont-300	180895	985195					●
copter2	55476	759952			●	●	●
cvxqp3	17500	114962				●	●
darcy003	389874	2097566				●	●
dawson5	51537	1010777			●	●	●

Name	*m*	*nnz(A)*	*Pos Def*	*Singular*	*Test Set 1*	*Test Set 2*	*Test Set 3*
GHS_indef *(cont.)*							
dixmaanl	60000	299998			●	●	●
d_pretok	182730	1641672				●	●
dtoc	24993	69972		Yes	●	●	●
exdata_1	6001	2269500			●	●	
helm2d03	392257	2741935				●	●
helm3d01	32226	428444			●	●	●
k1_san	67759	559774		Yes	●	●	●
laser	3002	9000			●	●	
linverse	11999	95977			●	●	●
mario001	38434	240912			●	●	●
mario002	389874	2097566				●	●
ncvxbqp1	50000	349968			●	●	●
ncvxqp1	12111	73963			●	●	●
ncvxqp3	75000	499964				●	●
ncvxqp5	62500	424966				●	●
ncvxqp7	87500	574962				●	●
ncvxqp9	16554	54040			●	●	●
olesnik0	88263	744216			●	●	●
qpband	20000	45000			●	●	●
sit100	10262	61046			●	●	●
sparsine	50000	1548988				●	●
spmsrtls	29995	229947			●	●	●
stokes64	12546	140034			●	●	●
stokes64s	12546	140034			●	●	●
stokes128	49666	558594			●	●	●
tuma1	22967	87760			●	●	●
tuma2	12992	49365			●	●	●
turon_m	189924	1690876				●	●
GHS_psdef							
apache1	80800	542184	Yes		●	●	●
apache2	715176	4817870	Yes				●
audikw_1	943695	74651847	Yes				●
bmw7st_1	141347	7318399	Yes				●
bmwcra_1	148770	10641602	Yes				●
copter1	17222	211064					●
copter2	55476	759952					●
crankseg_1	52804	10614210	Yes		●		●
crankseg_2	63838	14148858	Yes		●		●
cvxbqp1	50000	349968	Yes		●	●	●
finance256	37376	298496					●
ford1	18728	101576					●
ford2	100196	544688					●
gridgena	48962	512084	Yes		●	●	●
hood	220542	9895422	Yes				●
inline1	503712	36816170	Yes				●
jnlbrng1	40000	199200	Yes		●	●	●
minsurfo	40806	203622	Yes		●	●	●
ldoor	952203	42493817	Yes				●
obstclae	40000	197608	Yes		●	●	●
oilpan	73752	2148558	Yes		●	●	●

Name	*m*	*nnz(A)*	*Pos Def*	*Singular*	*Test Set 1*	*Test Set 2*	*Test Set 3*
GHS_psdef *(cont.)*							
opt1	15449	1930655					●
pds10	16558	149658					●
pwt	36519	326107					●
ramage02	16830	2866352					●
s3dkq4m2	90449	5427725	Yes		●	●	●
s3dkt3m2	90449	3686223	Yes		●	●	●
srb1	54924	2962152					●
torsion1	40000	197608	Yes		●	●	●
vanbody	47072	2239056	Yes		●	●	●
wathen100	30401	471601	Yes		●	●	●
wathen120	36441	565761	Yes		●	●	●
Grund							
meg4	5860	25258			●	●	
Gset							
G6	800	38352			●		
G7	800	38352			●		
G8	800	38352			●		
G9	800	38352			●		
G10	800	38352			●		
G11	800	3200			●		
G12	800	3200			●		
G13	800	3200			●		
G18	800	9388			●		
G19	800	9322			●		
G20	800	9344			●		
G21	800	9334			●		
G27	2000	39980			●	●	
G28	2000	38890			●	●	
G29	2000	38890			●	●	
G30	2000	38890			●	●	
G31	2000	38890			●	●	
G32	2000	8000			●	●	
G33	2000	8000			●	●	
G34	2000	8000			●	●	
G39	2000	23556			●	●	
G40	2000	23532			●	●	
G41	2000	23570			●	●	
G42	2000	23558			●	●	
G56	5000	24996		Yes	●	●	
G57	5000	20000			●	●	
G59	5000	59140			●	●	
G61	7000	34296		Yes	●	●	
G62	7000	28000			●	●	
G64	7000	82918			●	●	
G65	8000	32000			●	●	
G66	9000	36000			●	●	
G67	10000	40000			●	●	●

Name	*m*	*nnz(A)*	*Pos Def*	*Singular*	*Test Set 1*	*Test Set 2*	*Test Set 3*
		Gupta					
gupta1	31802	2164210					●
gupta2	62064	4248286					●
gupta3	16783	9323427					●
		HB					
494_bus	494	1666	Yes		●		
662_bus	662	2474	Yes		●		
685_bus	685	3249	Yes		●		
1138_bus	1138	4054	Yes		●	●	
bcsstk01	48	400	Yes		●		
bcsstk02	66	4356	Yes		●		
bcsstk03	112	640	Yes		●		
bcsstk04	132	3648	Yes		●		
bcsstk05	153	2423	Yes		●		
bcsstk06	420	7860	Yes		●		
bcsstk07	420	7860	Yes		●		
bcsstk08	1074	12960	Yes		●	●	
bcsstk09	1083	18437	Yes		●	●	
bcsstk10	1086	22070	Yes		●	●	
bcsstk11	1473	34241	Yes		●	●	
bcsstk12	1473	34241	Yes		●	●	
bcsstk13	2003	83883	Yes		●	●	
bcsstk14	1806	63454	Yes		●	●	
bcsstk15	3948	117816	Yes		●	●	
bcsstk16	4884	290378	Yes		●	●	
bcsstk17	10974	428650	Yes		●	●	●
bcsstk18	11948	149090	Yes		●	●	●
bcsstk19	817	6853	Yes		●		
bcsstk20	485	3135	Yes		●		
bcsstk21	3600	26600	Yes		●	●	
bcsstk22	138	696	Yes		●		
bcsstk23	3134	45178	Yes		●	●	
bcsstk24	3562	159910	Yes		●	●	
bcsstk25	15439	252241	Yes		●	●	●
bcsstk26	1922	30336	Yes		●	●	
bcsstk27	1224	56126	Yes		●	●	
bcsstk28	4410	219024	Yes		●	●	
bcsstk29	13992	619488					●
bcsstk30	28924	2043492					●
bcsstk31	35588	1181416					●
bcsstk32	44609	2014701					●
bcsstm01	48	24		Yes	●		
bcsstm02	66	66	Yes		●		
bcsstm04	132	132		Yes	●		
bcsstm05	153	153	Yes		●		
bcsstm06	420	420	Yes		●		
bcsstm07	420	7252	Yes		●		
bcsstm08	1074	1074	Yes		●	●	
bcsstm09	1083	1083	Yes		●	●	
bcsstm10	1086	22092			●	●	
bcsstm11	1473	1473	Yes		●	●	

Name	*m*	*nnz(A)*	*Pos Def*	*Singular*	*Test Set 1*	*Test Set 2*	*Test Set 3*
HB *(cont.)*							
bcsstm12	1473	19659	Yes		●	●	
bcsstm13	2003	21181		Yes	●	●	
bcsstm19	817	817	Yes		●		
bcsstm20	485	485	Yes		●		
bcsstm21	3600	3600	Yes		●	●	
bcsstm22	138	138	Yes		●		
bcsstm23	3134	3134	Yes		●	●	
bcsstm24	3562	3562	Yes		●	●	
bcsstm25	15439	15439	Yes		●	●	●
bcsstm26	1922	1922	Yes		●	●	
bcsstm27	1224	56126			●	●	
gr_30_30	900	7744	Yes		●		
lund_a	147	2449	Yes		●		
lund_b	147	2441	Yes		●		
nos1	237	1017	Yes		●		
nos2	957	4137	Yes		●		
nos3	960	15844	Yes		●		
nos4	100	594	Yes		●		
nos5	468	5172	Yes		●		
nos6	675	3255	Yes		●		
nos7	729	4617	Yes		●		
plat362	362	5786	Yes		●		
plat1919	1919	32399	Yes		●	●	
saylr3	1000	3750			●	●	
saylr4	3564	22316			●	●	
sherman1	1000	3750			●	●	
zenios	2873	1314		Yes	●	●	
INPRO							
msdoor	415863	19173163	Yes				●
Koutsovasilis							
F1	343791	19173163	Yes				●
F2	71505	5294285				●	●
Li							
pli	22695	1350309					●
Lin							
Lin	256000	1766400				●	●
Lourakis							
bundle1	10581	770811	Yes		●	●	●
Mathworks							
Kuu	7102	340200	Yes		●	●	
Muu	7102	170134	Yes		●	●	
McRae							
ecology1	1000000	4996000					●
ecology2	999999	4995991	Yes				●

Name	*m*	*nnz(A)*	*Pos Def*	*Singular*	*Test Set 1*	*Test Set 2*	*Test Set 3*
Mulvey							
finan512	74752	596992	Yes		●	●	●
pfinan512	74752	596992					●
Nasa							
nasa1824	1824	39208	Yes		●	●	
nasa2146	2146	72250	Yes		●	●	
nasa2910	2910	174296	Yes		●	●	
nasa4704	4704	104756	Yes		●	●	
nasarb	54870	2677324	Yes			●	●
pwt	36519	326107					●
shuttle_eddy	10429	103599					●
skirt	125598	196520					●
ND							
nd3k	9000	3279690	Yes		●	●	
nd6k	18000	6897316	Yes		●	●	●
nd12k	36000	14220946	Yes				●
nd24k	72000	28715634	Yes				●
Nemeth							
nemeth01	9506	725054			●	●	
nemeth02	9506	394808			●	●	
nemeth03	9506	394808			●	●	
nemeth04	9506	394808			●	●	
nemeth05	9506	394808			●	●	
nemeth06	9506	394808			●	●	
nemeth07	9506	394812			●	●	
nemeth08	9506	394816			●	●	
nemeth09	9506	395506			●	●	
nemeth10	9506	401448			●	●	
nemeth11	9506	408264			●	●	
nemeth12	9506	446818			●	●	
nemeth13	9506	474472			●	●	
nemeth14	9506	496144			●	●	
nemeth15	9506	539802			●	●	
nemeth16	9506	587012			●	●	
nemeth17	9506	629620			●	●	
nemeth18	9506	695234			●	●	
nemeth19	9506	818302			●	●	
nemeth20	9506	971870			●	●	
nemeth21	9506	1173746			●	●	
nemeth22	9506	1358832			●	●	
nemeth23	9506	1506810			●	●	
nemeth24	9506	1506550			●	●	
nemeth25	9506	1511758			●	●	
nemeth26	9506	1511760			●	●	
Norris							
fv1	9604	85264	Yes		●	●	
fv2	9801	87025	Yes		●	●	
fv3	9801	87025	Yes		●	●	

Name	*m*	*nnz(A)*	*Pos Def*	*Singular*	*Test Set 1*	*Test Set 2*	*Test Set 3*
Oberwolfach							
bone010	986703	47851783	Yes				●
boneS01	127224	5516602	Yes			●	●
boneS10	914818	40878706	Yes				●
filter2D	1668	10750			●	●	
filter3D	106437	2707179				●	●
flowmeter0	9669	67391			●	●	
gas_sensor	66917	1703365			●	●	●
gyro_k	17361	1021159	Yes		●	●	●
gyro_m	17361	340341	Yes		●	●	●
gyro	17361	1021159	Yes		●	●	●
LF10	18	82	Yes		●		
LF10000	19998	99982	Yes			●	●
LFAT5	14	46	Yes		●		
LFAT5000	19994	79966	Yes		●	●	●
rail_1357	1357	8985			●	●	
rail_5177	5177	35185			●	●	
rail_20209	20209	139233			●	●	●
rail_79841	79841	553921			●	●	●
spiral	1434	18228			●	●	
t2dah	11445	176117			●	●	●
t2dah_a	11445	176117			●	●	●
t2dah_e	11445	176117	Yes		●	●	●
t2dal	4257	37465			●	●	
t2dal_a	4257	37465			●	●	
t2dal_bci	4257	37465			●	●	
t2dal_e	4257	4257	Yes		●	●	
t3dh	79171	4352105			●	●	●
t3dh_a	79171	4352105			●	●	●
t3dh_e	79171	4352105			●	●	●
t3dl	20360	509866			●	●	●
t3dl_a	20360	509866			●	●	●
t3dl_e	20360	20360	Yes		●	●	●
Okunbor							
aft01	8205	125567	Yes		●	●	
Pajek							
dictionary28	52652	178076		Yes			●
GD97_b	47	264		Yes	●		
GD99_b	64	252		Yes	●		
geom	7343	23796		Yes	●	●	
Journals	124	12068	Yes		●		
Reuters911	13332	296076		Yes	●	●	●
Sandi_authors	86	248		Yes	●		
Stranke94	10	90			●		
USAir97	332	4252		Yes	●		
PARSEC							
benzene	8219	242669			●	●	
CO	221119	7666057					●
Ga10As10H30	113081	6115633					●

Name	*m*	*nnz(A)*	*Pos Def*	*Singular*	*Test Set 1*	*Test Set 2*	*Test Set 3*
PARSEC*(cont.)*							
Ga19As19H42	133123	8884839					●
Ga3As3H12	61349	5970947				●	●
Ga41As41H72	268096	18488476					●
GaAsH6	61439	3381809				●	●
Ge87H76	112985	7892195				●	●
Ge99H100	112985	8451395				●	●
H2O	67024	2216736				●	●
Na5	5832	305630			●	●	
Si10H16	17077	875923			●	●	●
Si2	769	17801			●		
Si34H36	97569	5156379				●	●
Si41Ge41H72	185639	15011265					●
Si5H12	19896	738598				●	●
Si87H76	240369	10661631					●
SiH4	5041	171903			●	●	
SiNa	5743	198787			●	●	
SiO	33401	1317655				●	●
SiO2	155331	11283505					●
Pothen							
barth5	15606	107362					●
bodyy4	17546	121550	Yes		●	●	●
bodyy5	18589	128853	Yes		●	●	●
bodyy6	19366	134208	Yes		●	●	●
mesh1e1	48	306	Yes		●		
mesh1em1	48	306	Yes		●		
mesh1em6	48	306	Yes		●		
mesh2e1	306	2018	Yes		●		
mesh2em5	306	2018	Yes		●		
mesh3e1	289	1377	Yes		●		
mesh3em5	289	1377	Yes		●		
onera_dual	85567	419201					●
pwt	36519	326107					●
shuttle_eddy	10429	103599					●
skirt	12598	196520					●
tandem_dual	94069	460493					●
tandem_vtx	18454	253350					●
Rajat							
rajat06	10922	46983					●
rajat07	14842	63913					●
rajat08	19362	83443					●
rajat09	24482	105573					●
rajat10	30202	130303					●
Rothberg							
3dtube	45330	3213618					●
cfd1	70656	1825580	Yes		●	●	●
cfd2	123440	3085406	Yes			●	●
gearbox	153746	9080404					●
struct3	53570	1173694					●

Name	*m*	*nnz(A)*	*Pos Def*	*Singular*	*Test Set 1*	*Test Set 2*	*Test Set 3*
Schenk_AFE							
af_0_k101	503625	17550675	Yes				●
af_1_k101	503625	17550675	Yes				●
af_2_k101	503625	17550675	Yes				●
af_3_k101	503625	17550675	Yes				●
af_4_k101	503625	17550675	Yes				●
af_5_k101	503625	17550675	Yes				●
af_shell1	504855	17562051					●
af_shell2	504855	17562051					●
af_shell3	504855	17562051	Yes				●
af_shell4	504855	17562051	Yes				●
af_shell5	504855	17579155					●
af_shell6	504855	17579155					●
af_shell7	504855	17579155	Yes				●
af_shell8	504855	17579155	Yes				●
af_shell9	504855	17588846					●
af_shell10	1508065	52259885					●
Schenk_IBMNA							
c-18	2169	15145			●	●	
c-19	2327	21817			●	●	
c-20	2921	20445			●	●	
c-21	3509	32145			●	●	
c-22	3792	28870			●	●	
c-23	3969	31079			●	●	
c-24	4119	35669			●	●	
c-25	3797	49635			●	●	
c-26	4307	34537			●	●	
c-27	4563	30927			●	●	
c-28	4598	30590			●	●	
c-29	5033	43731			●	●	
c-30	5321	65693			●	●	
c-31	5339	78571			●	●	
c-32	5975	54471			●	●	
c-33	6317	56123			●	●	
c-34	6611	64333			●	●	
c-35	6537	62891			●	●	
c-36	7479	65941			●	●	
c-37	8204	74676			●	●	
c-38	8127	77689			●	●	
c-39	9271	116587			●	●	
c-40	9941	81501			●	●	
c-41	9769	101635			●	●	
c-42	10471	110285			●	●	●
c-43	11125	123659			●	●	●
c-44	10728	85000			●	●	●
c-45	13206	174452			●	●	●
c-46	14913	130397			●	●	●
c-47	15343	211401			●	●	●
c-48	18354	166080			●	●	●
c-49	21132	157040			●	●	●
c-50	22401	180245			●	●	●

Name	*m*	*nnz(A)*	*Pos Def*	*Singular*	*Test Set 1*	*Test Set 2*	*Test Set 3*
Schenk_IBMNA *(cont.)*							
c-51	23196	203048			●	●	●
c-52	23948	202708			●	●	●
c-53	30235	355139			●	●	●
c-54	31793	385987			●	●	●
c-56	35910	380240			●	●	●
c-57	37833	403373			●	●	●
c-60	43640	298570			●	●	●
c-61	43618	310016			●	●	●
c-62	41731	559341			●	●	●
c-64	51035	707985			●	●	●
c-64b	51035	707601			●	●	●
c-65	48066	360428			●	●	●
c-66	49989	444853			●	●	●
c-66b	49989	444851			●	●	●
c-67	57975	530229			●	●	●
c-67b	57975	530583			●	●	●
c-73	169422	1279274				●	●
c-73b	169422	1279274					●
c-big	345241	2340859					●
Schmid							
thermal1	82654	574458	Yes		●	●	●
thermal2	1228045	8580313	Yes				●
Simon							
olafu	16146	1015156	Yes		●	●	●
raefsky4	19779	1316789	Yes		●	●	●
UTEP							
Dubcova1	16129	253009	Yes		●	●	●
Dubcova2	65025	1030225	Yes		●	●	●
Dubcova3	146689	3636643	Yes				●
Wissgott							
parabolic_fem	525825	3674625	Yes				●
Zaoui							
kkt_power	2063494	12771361					●

Optimization test sets

Test Set 4: Linear programming problems			
Problem	n	m	description
cont11	80396	160792	Linearized PDE
cont1	40398	160792	Linearized PDE
cont4	40398	160792	Linearized PDE
neos1	1892	131581	Unknown; submitted to NEOS
neos2	1560	132568	Unknown; submitted to NEOS
neos3	6624	512209	Unknown; submitted to NEOS
ns1687037	43749	50622	Unknown; submitted to NEOS
All problems were available from `http://plato.asu.edu/ftp/lptestset/`.			

Test Set 5: Non-linear programming problems				
Problem	n	m	origin	description
bearing_400	160000	0	1	Pressure distribution model
cont5_1	90600	90300	3	Parabolic boundary control problem
cont5_2_1	90600	90300	3	Parabolic boundary control problem
cont5_2_2	90600	90300	3	Parabolic boundary control problem
cont5_2_3	90600	90300	3	Parabolic boundary control problem
cont5_2_4	90600	90300	3	Parabolic boundary control problem
dirichlet_120	53881	241	1	Transition state problem
ex1_160	204798	103037	2	Elliptic control problem
ex1_320	203522	101761	2	Elliptic control problem
ex4_2_320	204798	103037	2	Elliptic control problem
NARX_CFy	43973	46744	4	System identification

origins:

1	[Dolan et al. 2004]	http://www-unix.mcs.anl.gov/ more/cops/
2	[Maurer and Mittelmann 2000]	http://plato.asu.edu/ftp/ampl_files/ellip_ampl/
3	[Mittelmann 2001; Mittelmann and Troeltzsch 2001]	http://plato.asu.edu/ftp/ampl_files/parabol_ampl/
4	[Mittelmann et al. 2007]	http://plato.asu.edu/ftp/ampl_files/sysid_ampl/

Appendix B

Full Ipopt results

Here we present full results on `fox` for IPOPT using a number of different sparse symmetric indefinite solvers. Full details of the experimental setup are described in Section 8.3.

B.1 Netlib lp problems

The first table presents interior point iteration counts, with factorization counts in brackets, a star indicates a failure to converge.

Table of Iteration Counts on Netlib lp test set

Problem	Iteration (Factorization) count				
	`MA57`	`HSL_MA77`	`HSL_MA79`	DSIS	Pardiso
25fv47	203 (211)	205 (220)	257 (308)	206 (218)	203 (204)
80bau3b	232 (323)	213 (241)	219 (270)	214 (251)	216 (217)
adlittle	65 (66)	65 (66)	68 (73)	65 (66)	65 (66)
afiro	25 (28)	25 (28)	26 (31)	25 (27)	25 (26)
agg	175 (184)	171 (174)	188 (201)	171 (172)	171 (172)
agg2	166 (176)	164 (171)	167 (180)	164 (169)	164 (165)
agg3	165 (170)	165 (171)	166 (177)	168 (175)	165 (166)
bandm	78 (79)	78 (80)	78 (79)	78 (79)	78 (79)
beaconfd	37 (40)	35 (37)	35 (36)	35 (36)	35 (36)
blend	19 (20)	19 (20)	20 (29)	19 (20)	19 (20)
bnl1	195 (203)	195 (202)	201 (215)	195 (203)	199 (200)
bnl2	287 (344)	277 (304)	289 (362)	279 (307)	306 (330)
boeing1	85 (89)	84 (86)	84 (87)	84 (85)	84 (85)
boeing2	62 (73)	63 (70)	57 (61)	60 (62)	57 (58)
bore3d	194 (202)	195 (202)	194 (239)	206 (211)	137 (140)
brandy	85 (94)	85 (95)	83 (99)	85 (96)	87 (88)
capri	152 (153)	152 (153)	152 (155)	152 (153)	166 (168)
cycle	171 (344)	195 (400)	171 (346)	187 (375)	110 (125)
czprob	420 (427)	424 (426)	419 (422)	425 (429)	417 (418)
d2q06c	283 (292)	283 (291)	302 (369)	283 (290)	283 (284)
d6cube	53 (59)	53 (60)	54 (74)	53 (57)	42* (45)
degen2	78 (141)	62 (107)	102 (217)	96 (158)	64 (65)
degen3	276 (283)	62 (118)	84 (169)	63 (120)	353 (374)
dfl001	171 (178)	170 (177)	166 (179)	176 (362)	3000* (3351)
e226	71 (84)	69 (74)	69 (71)	69 (77)	69 (70)
etamacro	113 (117)	113 (120)	113 (117)	114 (120)	80 (81)
fffff800	392 (411)	393 (408)	395 (457)	392 (404)	389 (390)
finnis	237 (496)	153 (173)	149 (177)	154 (174)	147 (148)
fit1d	30 (31)	30 (31)	30 (31)	30 (31)	30 (31)
fit1p	122 (123)	122 (123)	122 (123)	122 (123)	122 (123)

Table of Iteration Counts on Netlib lp test set*(cont.)*

Problem	Iteration (Factorization) count				
	MA57	HSL_MA77	HSL_MA79	DSIS	Pardiso
fit2d	38 (39)	38 (39)	38 (39)	38 (39)	38 (39)
fit2p	67 (68)	67 (68)	67 (68)	67 (68)	67 (68)
ganges	404 (426)	401 (411)	374 (399)	400 (417)	399 (400)
gfrd-pnc	144 (145)	144 (146)	144 (145)	144 (145)	144 (145)
greenbea	3000* (3004)	3000* (3007)	3000* (3004)	3000* (3004)	3000* (3001)
greenbeb	2718 (3745)	2319 (2578)	2295 (2512)	2729 (3808)	2081 (2136)
grow15	149 (155)	148 (155)	148 (155)	148 (154)	153 (154)
grow22	157 (163)	158 (164)	157 (164)	157 (163)	160 (161)
grow7	108 (114)	110 (117)	108 (114)	110 (116)	111 (112)
israel	135 (139)	135 (140)	139 (148)	132 (136)	135 (136)
kb2	45 (46)	45 (46)	48 (50)	45 (46)	45 (46)
lotfi	58 (71)	55 (57)	55 (62)	61 (69)	55 (57)
maros	783 (1245)	830 (1204)	813 (1351)	819 (1176)	508 (509)
nesm	531 (532)	533 (535)	538 (543)	510 (511)	539 (540)
perold	1317 (1323)	1152 (1156)	1278 (1285)	1346 (1350)	1344 (1345)
pilot	286 (287)	288 (291)	293 (299)	286 (287)	286 (287)
pilot_ja	465 (513)	467 (516)	466 (523)	478 (530)	435 (436)
pilot_we	229 (230)	229 (231)	271 (306)	229 (230)	229 (230)
pilot4	3000* (8758)	3000* (7847)	3000* (5507)	3000* (8291)	252* (255)
pilot87	363 (519)	337 (367)	317 (372)	360 (419)	306 (307)
pilotnov	340 (431)	340 (424)	340 (430)	340 (421)	290 (291)
recipe	50 (92)	45 (52)	48 (57)	45 (78)	33 (34)
sc105	19 (20)	19 (20)	19 (20)	19 (20)	19 (20)
sc205	26 (27)	26 (27)	26 (27)	26 (27)	26 (27)
sc50a	17 (18)	17 (18)	17 (18)	17 (18)	17 (18)
sc50b	15 (16)	15 (16)	15 (18)	15 (16)	15 (16)
scagr25	406 (407)	411 (413)	405 (407)	401 (402)	405 (406)
scagr7	138 (139)	138 (140)	139 (142)	138 (139)	138 (139)
scfxm1	144 (178)	136 (141)	135 (143)	135 (138)	135 (136)
scfxm2	356 (444)	329 (337)	328 (335)	329 (336)	328 (329)
scfxm3	423 (509)	399 (412)	397 (405)	399 (410)	397 (398)
scorpion	35 (43)	35 (43)	35 (42)	45 (90)	37 (38)
scrs8	85 (120)	85 (100)	116 (241)	83 (101)	83 (84)
scsd1	15 (16)	15 (16)	15 (18)	15 (16)	15 (16)
scsd6	17 (18)	17 (18)	17 (18)	17 (18)	17 (18)
scsd8	18 (19)	18 (19)	18 (19)	18 (19)	18 (19)
sctap1	41 (49)	41 (43)	41 (47)	41 (42)	41 (42)
sctap2	32 (37)	32 (34)	32 (37)	32 (33)	32 (33)
sctap3	31 (38)	31 (34)	31 (34)	31 (32)	31 (32)
seba	214 (215)	214 (216)	214 (215)	214 (215)	214 (215)
share1b	356 (357)	357 (359)	325 (337)	355 (356)	359 (360)
share2b	29 (30)	29 (31)	29 (30)	29 (30)	29 (30)
shell	560 (611)	560 (620)	611 (727)	560 (618)	560 (562)
ship04l	71 (75)	71 (78)	71 (82)	71 (75)	66 (71)
ship04s	59 (63)	59 (66)	59 (66)	59 (63)	48 (49)
ship08l	85 (89)	85 (92)	83 (88)	85 (89)	94 (95)
ship08s	158 (163)	156 (163)	162 (173)	157 (161)	71 (72)
ship12l	99 (103)	66 (74)	66 (80)	67 (72)	69 (70)
ship12s	100 (104)	100 (107)	105 (118)	100 (104)	70 (71)
sierra	60 (67)	58 (65)	58 (64)	58 (62)	58 (59)
stair	70 (71)	70 (73)	70 (71)	70 (71)	70 (71)
standata	52 (65)	52 (56)	52 (55)	52 (58)	52 (53)
standgub	53 (78)	53 (60)	53 (80)	53 (60)	52 (53)
standmps	69 (75)	69 (71)	69 (71)	69 (70)	69 (70)

Table of Iteration Counts on Netlib lp test set*(cont.)*					
Problem	Iteration (Factorization) count				
	`MA57`	`HSL_MA77`	`HSL_MA79`	DSIS	Pardiso
stocfor1	44 (45)	44 (46)	44 (45)	44 (45)	44 (45)
stocfor2	174 (178)	174 (176)	174 (178)	174 (175)	174 (175)
stocfor3	159 (161)	140 (147)	??? (163)	159 (160)	159 (160)
truss	64 (67)	64 (67)	64 (66)	65 (66)	64 (65)
tuff	143 (147)	144 (151)	141 (145)	144 (148)	133 (134)
vtp_base	341 (342)	341 (342)	341 (343)	341 (342)	341 (342)
wood1p	72 (118)	73 (123)	72 (117)	72 (113)	81 (82)
woodw	70 (71)	70 (72)	70 (71)	70 (71)	70 (71)

The next table shows timing results for these codes. Again a star indicates failure to converge to the optimum.

Table of run times on the Netlib lp test set					
Problem	Overall run time				
	`MA57`	`HSL_MA77`	`HSL_MA79`	DSIS	Pardiso
25fv47	1.11	3.27	3.62	2.87	7.58
80bau3b	4.99	14.52	7.41	14.91	10.20
adlittle	0.04	0.06	0.06	0.21	0.89
afiro	0.02	0.03	0.02	0.21	0.98
agg	0.28	0.65	0.42	0.83	1.58
agg2	0.48	0.96	0.89	1.04	1.79
agg3	0.47	1.00	0.74	1.04	1.98
bandm	0.12	0.24	0.18	0.42	0.58
beaconfd	0.06	0.09	0.08	0.22	0.98
blend	0.02	0.03	0.03	0.21	0.98
bnl1	0.61	1.73	1.26	1.65	4.21
bnl2	5.08	10.36	7.48	12.00	12.87
boeing1	0.17	0.33	0.24	0.42	1.06
boeing2	0.07	0.13	0.09	0.22	0.78
bore3d	0.22	0.43	0.28	0.62	1.39
brandy	0.11	0.20	0.18	0.22	0.78
capri	0.22	0.42	0.32	0.62	1.38
cycle	10.63	29.28	11.52	15.64	7.65
czprob	1.63	5.51	2.03	5.71	2.53
d2q06c	5.08	24.01	13.82	14.67	12.88
d6cube	1.18	3.79	2.53	5.38	5.32*
degen2	0.70	0.71	1.06	1.24	1.74
degen3	13.38	6.80	7.62	5.93	40.56*
dfl001	256.39	268.42	234.11	413.2	101.00
e226	0.12	0.23	0.13	0.22	0.83
etamacro	0.26	0.59	0.31	0.63	1.59
fffff800	1.15	3.16	2.67	2.86	2.39
finnis	0.87	0.85	0.56	0.83	1.16
fit1d	0.11	0.28	0.21	0.44	0.78
fit1p	0.38	1.11	0.90	1.04	1.18
fit2d	1.66	3.92	3.24	4.39	2.40
fit2p	1.50	4.20	3.86	3.99	1.58
ganges	1.66	4.51	2.52	4.28	2.59
gfrd-pnc	0.24	0.65	0.33	0.63	1.36
greenbea	47.00*	244.50*	105.37*	139.55*	265.19*
greenbeb	65.32	213.46	96.34	189.75	196.71
grow15	0.40	0.69	0.46	0.84	1.44
grow22	0.61	1.05	0.68	1.24	1.18
grow7	0.15	0.24	0.18	0.42	1.37

Table of run times on the Netlib lp test set *(cont.)*

Problem	Overall run time				
	`MA57`	`HSL_MA77`	`HSL_MA79`	DSIS	Pardiso
israel	0.15	0.27	0.22	0.42	0.98
kb2	0.03	0.04	0.04	0.21	0.78
lotfi	0.07	0.11	0.11	0.22	0.98
maros	5.87	26.47	11.93	15.41	13.71
nesm	3.01	10.00	4.64	11.18	5.71
perold	5.71	20.61	7.05	14.59	12.44
pilot	9.83	27.94	12.82	21.95	25.89
pilot_ja	4.62	19.73	6.00	9.75	14.66
pilot_we	1.05	3.16	2.58	3.09	6.12
pilot4	46.27*	104.95*	69.64*	59.87*	3.48*
pilot87	74.95	102.77	66.45	85.89	66.61
pilotnov	60.92	52.68	24.08	29.55	11.45
recipe	0.06	0.07	0.06	0.22	0.73
sc105	0.02	0.03	0.02	0.22	0.98
sc205	0.03	0.05	0.04	0.22	0.98
sc50a	0.02	0.02	0.02	0.21	0.98
sc50b	0.02	0.02	0.02	0.22	0.98
scagr25	0.51	1.26	0.77	1.43	1.38
scagr7	0.08	0.16	0.11	0.22	0.58
scfxm1	0.25	0.44	0.35	0.63	1.18
scfxm2	1.07	2.12	1.44	2.05	1.99
scfxm3	1.77	4.06	1.82	3.88	2.99
scorpion	0.06	0.12	0.08	0.22	0.76
scrs8	0.22	0.51	1.12	0.63	1.18
scsd1	0.03	0.07	0.04	0.22	0.78
scsd6	0.05	0.12	0.06	0.23	0.98
scsd8	0.10	0.22	0.11	0.45	0.98
sctap1	0.08	0.15	0.12	0.22	0.98
sctap2	0.17	0.40	0.30	0.44	1.18
sctap3	0.23	0.58	0.41	0.65	0.97
seba	0.40	1.07	0.60	1.24	1.18
share1b	0.28	0.50	0.42	0.62	0.98
share2b	0.03	0.05	0.03	0.22	0.58
shell	1.26	3.98	2.86	3.87	2.60
ship04l	0.22	0.59	0.31	0.64	1.58
ship04s	0.14	0.37	0.20	0.43	1.38
ship08l	0.50	1.41	0.71	1.68	3.40
ship08s	0.50	1.50	0.72	1.86	2.37
ship12l	0.72	1.46	1.01	1.70	3.60
ship12s	0.39	1.17	0.81	1.67	2.77
sierra	0.33	0.71	0.39	0.85	2.38
stair	0.16	0.29	0.18	0.43	0.97
standata	0.12	0.29	0.17	0.43	0.78
standgub	0.14	0.32	0.23	0.43	1.38
standmps	0.16	0.37	0.21	0.43	0.78
stocfor1	0.04	0.06	0.05	0.22	0.78
stocfor2	0.95	3.13	1.35	2.88	2.19
stocfor3	7.66	37.16	10.66	25.23	11.28
truss	1.01	3.23	1.26	4.79	2.28
tuff	0.34	0.69	0.40	0.63	2.17
vtp_base	0.29	0.58	0.43	0.82	1.37
wood1p	2.47	4.71	3.37	3.93	4.01
woodw	1.11	3.29	1.43	4.38	2.57

B.2 Mittelmann benchmark problems

The next table shows iteration and factorization counts on the larger problems of Test Sets 4 and 5. `Ipopt` reported optimal solutions for all problems except *cont11*, *cont1*, and *cont4* where it reported convergence to a point of local infeasibility. Deviations from this behaviour are indicated as follows:
f —failed to reach the optimal solution.
m—virtual memory exceeded 8GiB.
t —run time exceeded 1 hour.
d —out-of-core solver exceeded 100GiB of disk space.

Problem	Iteration (Factorization) count				
	`MA57`	`HSL_MA77`	`HSL_MA79`	DSIS	Pardiso
Test Set 4					
cont11	64 (66)	62 (66)	62 (65)	55 (58)	62 (64)
cont1	50 (52)	48 (52)	48 (50)	48 (50)	48 (50)
cont4	84 (86)	82 (86)	81 (83)	83 (85)	81 (83)
neos1	269 (271)	269 (271)	269 (271)	269 (271)	269 (270)
neos2	309 (313)	307 (309)	309 (313)	309 (313)	307 (308)
neos3	22 (29)	20 (25)	18 (20)	18 (21)	19 (20)
ns1687037	84 (85)	71 (73)	d	84 (85)	84 (85)
Test Set 5					
bearing_400	16 (16)	16 (16)	16 (16)	16 (16)	16 (16)
cont5_1	16 (26)	16 (23)	17 (23)	16 (18)	16 (17)
cont5_2_1	69 (178)	50 (74)	95 (209)	51 (90)	41 (42)
cont5_2_2	62 (121)	49 (75)	50 (73)	50 (75)	48 (49)
cont5_2_3	76 (144)	61 (95)	59 (91)	61 (99)	55 (56)
cont5_2_4	14 (25)	14 (25)	12 (18)	14 (22)	14 (15)
dirichlet_120	56 (119)	56 (119)	56 (119)	56 (119)	56 (119)
ex1_160	10 (11)	10 (12)	10 (11)	10 (11)	10 (11)
ex1_320	9 (10)	9 (11)	9 (10)	9 (10)	9 (10)
ex4_2_320	10 (11)	10 (12)	10 (11)	10 (11)	10 (11)
NARX_CFy	352 (878)	160 (402)	241 (595)	278 (677)	t (2945)

The next table reports serial execution times for Test Sets 4 and 5.

Problem	Overall run time				
	`MA57`	`HSL_MA77`	`HSL_MA79`	DSIS	Pardiso
Test Set 4					
cont11	246.9	234.7	235.2	233.4	76.0
cont1	57.5	182.6	80.2	115.5	52.6
cont4	80.6	270.2	116.3	166.3	134.5
neos1	378.6	722.6	193.5	802.5	1268.5
neos2	378.5	590.4	327.1	898.6	1079.0
neos3	1189.9	575.3	325.7	550.6	3570.0
ns1687037	708.3	494.2	d	404.9	495.3
Test Set 5					
bearing_400	66.6	85.5	90.5	70.6	67.3
cont5_1	96.8	112.0	149.1	65.7	27.7
cont5_2_1	669.8	354.3	1560.7	307.4	61.9
cont5_2_2	431.8	345.5	480.4	260.4	73.7
cont5_2_3	505.9	454.1	596.8	366.0	82.8
cont5_2_4	221.4	183.5	181.3	130.2	54.9
dirichlet_120	569.0	532.3	1824.9	441.7	578.8
ex1_160	43.6	62.1	53.1	52.5	27.6
ex1_320	41.9	56.4	53.2	44.3	25.6
ex4_2_320	43.8	64.1	52.8	52.5	27.6
NARX_CFy	1593.8	1312.4	292.4	3008.3	7300.0t

The next table reports the average time per factorization on Test Sets 4 and 5. These times may be misleading as different solvers are factorizing different systems, possibly with significantly more difficult numerics. Further, other `Ipopt` operations such as function evaluation and Hessian calculations may cause significant variation, particularly for the non-linear problems of Test Set 5.

Problem	Time per factorization				
	`MA57`	`HSL_MA77`	`HSL_MA79`	DSIS	Pardiso
Test Set 4					
cont11	3.74	3.56	3.62	4.02	1.19
cont1	1.11	3.51	1.60	2.31	1.05
cont4	0.94	3.14	1.40	1.96	1.62
neos1	1.40	2.67	0.71	2.96	4.70
neos2	0.93	1.91	1.05	2.87	2.84
neos3	41.03	22.93	16.28	26.22	178.50
ns1687037	8.33	6.77	-	4.76	5.83
Test Set 5					
bearing_400	4.16	5.34	5.65	4.41	4.21
cont5_1	3.72	4.87	6.48	3.65	1.63
cont5_2_1	3.76	4.79	7.47	3.42	1.47
cont5_2_2	3.57	4.61	6.58	3.47	1.50
cont5_2_3	3.51	4.78	6.56	3.70	1.47
cont5_2_4	8.85	7.34	10.07	5.92	3.66
dirichlet_120	4.78	4.47	15.34	3.71	4.86
ex1_160	3.96	5.17	4.83	4.77	2.51
ex1_320	4.19	5.13	5.31	4.42	2.56
ex4_2_320	3.98	5.34	4.80	4.77	2.51
NARX_CFy	1.82	3.26	0.49	4.44	2.48

The following table shows parallel performance and speedup of DSIS, with Pardiso (on 8 threads only) included for comparison. Only the subset of problems for which results on 8 threads were obtainable are shown.

Problem	DSIS				PARDISO
threads=	1	2	4	8	8
Test Set 4					
cont11	233.4	fail (–)	105.3 (1.62)	92.3 (1.85)	45.9 (1.66)
cont1	115.5	88.9 (1.30)	73.1 (1.58)	62.6 (1.85)	31.2 (1.69)
cont4	166.3	138.7 (1.20)	110.8 (1.50)	96.5 (1.72)	95.0 (1.42)
neos1	802.5	728.9 (1.10)	508.9 (1.58)	401.6 (2.00)	422.6 (3.00)
neos2	898.6	713.0 (1.26)	554.1 (1.62)	430.3 (2.09)	395.4 (2.73)
neos3	550.6	416.9 (1.32)	291.0 (1.89)	244.0 (2.26)	1411.3 (2.53)
ns1687037	404.9	345.3 (1.17)	303.9 (1.33)	241.8 (1.67)	193.3 (2.56)
Test Set 5					
bearing_400	70.6	65.6 (1.08)	62.2 (1.14)	61.4 (1.15)	57.9 (1.16)
cont5_1	65.7	48.3 (1.36)	40.2 (1.63)	33.6 (1.96)	9.6 (2.88)
cont5_2_1	307.4	237.2 (1.30)	143.5 (2.14)	114.2 (2.69)	19.0 (3.25)
cont5_2_2	260.4	222.6 (1.17)	181.6 (1.43)	fail (–)	22.3 (3.30)
cont5_2_3	366.0	258.3 (1.42)	179.6 (2.04)	fail (–)	24.2 (3.42)
cont5_2_4	130.2	87.2 (1.49)	76.1 (1.71)	72.6 (1.79)	33.25 (1.65)
dirichlet_120	441.7	315.0 (1.40)	234.7 (1.88)	195.3 (2.26)	205.2 (2.82)
ex1_160	52.5	41.8 (1.26)	34.5 (1.52)	32.3 (1.63)	15.3 (1.80)
ex1_320	44.3	35.0 (1.27)	29.6 (1.50)	26.7 (1.66)	14.4 (1.78)
ex4_2_320	52.5	41.8 (1.26)	34.4 (1.53)	31.3 (1.68)	15.3 (1.80)
NARX_CFy	3008.3	2703.2 (1.11)	fail (–)	fail (–)	fail (–)

Index