Science & Technology
Facilities Council

Technical Report

A note on the solve phase of a multicore solver

J. D. Hogg and J. A. Scott

June 8, 2010

RAL-TR-2010-007

Library and Information Services
SFTC Rutherford Appleton Laboratory
Harwell Science and Innovation Campus
Didcot
OX11 0QX
UK
Tel: +44 (0)1235 445384
Fax: +44(0)1235 446403
Email: library@rl.ac.uk

ISSN 1358-6254

RAL-TR-2010-007

A note on the solve phase of a multicore solver

J. D. Hogg and J. A. Scott[1]

ABSTRACT

When using a direct solver to solve large sparse linear systems of equations, the solve phase that follows the numerical factorization performs a relatively small proportion of the total number of numerical operations. However, it is responsible for a much higher proportion of the memory traffic. This makes it a potential bottleneck on newer multicore architectures that have a higher ratio of computational power to memory bandwidth than exhibited by traditional shared-memory parallel machines.

In this note, we illustrate the problem through experiments and test a number of different approaches that aim to improve the performance of the solve phase on multicore machines. We extend the DAG-based approach that has been successfully used for the factorize phase on multicore machines to the solve phase and explore techniques for reducing memory throughput. Numerical experiments that illustrate the difficulties and the effectiveness of our approaches are performed on large-scale problems from practical applications using the sparse multicore solver `HSL_MA87`.

Keywords: sparse symmetric linear systems, direct solver, parallel, multicore, solve phase, DAG-based.

AMS(MOS) subject classifications: 65F05, 65F50, 65Y05, 65Y10

---

[1] Computational Science and Engineering Department, Rutherford Appleton Laboratory, Chilton, Oxfordshire, OX11 0QX, UK.
Email: jonathan.hogg@stfc.ac.uk and jennifer.scott@stfc.ac.uk
Work supported by EPSRC grant EP/E053351/1.
Current reports available from `http://www.numerical.rl.ac.uk/reports/reports.html`.

June 8, 2010

1 Introduction

A classical problem in scientific computing is the solution of linear systems

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

where the $n \times n$ matrix A is large and sparse. A direct method performs a matrix factorization. In the general case, $A = LU$ where L is a lower triangular matrix and U is an upper triangular matrix. If A is symmetric and indefinite, the factorization takes the form $A = LDL^T$, where L is unit lower triangular and D is block diagonal with 1×1 and 2×2 blocks, while if A is symmetric and positive definite, a Cholesky factorization $A = LL^T$ is computed. Once a factorization has been found, the system may be solved for one or more right-hand sides $\boldsymbol{b}$ through a forward substitution

$$L\boldsymbol{y} = \boldsymbol{b},$$

to solve for the partial solution $\boldsymbol{y}$, then a backward substitution

$$U\boldsymbol{x} = \boldsymbol{y},$$

to obtain the solution $\boldsymbol{x}$. A direct solver normally has four phases:

Ordering: Determine an a priori pivot order that attempts to minimize fill in the factors.

Analyse: Perform a symbolic factorization to set up the data structures and prepare for the numerical factorization.

Factorize: Perform the numerical factorization.

Solve: Perform the forward and backward substitutions using the computed factors.

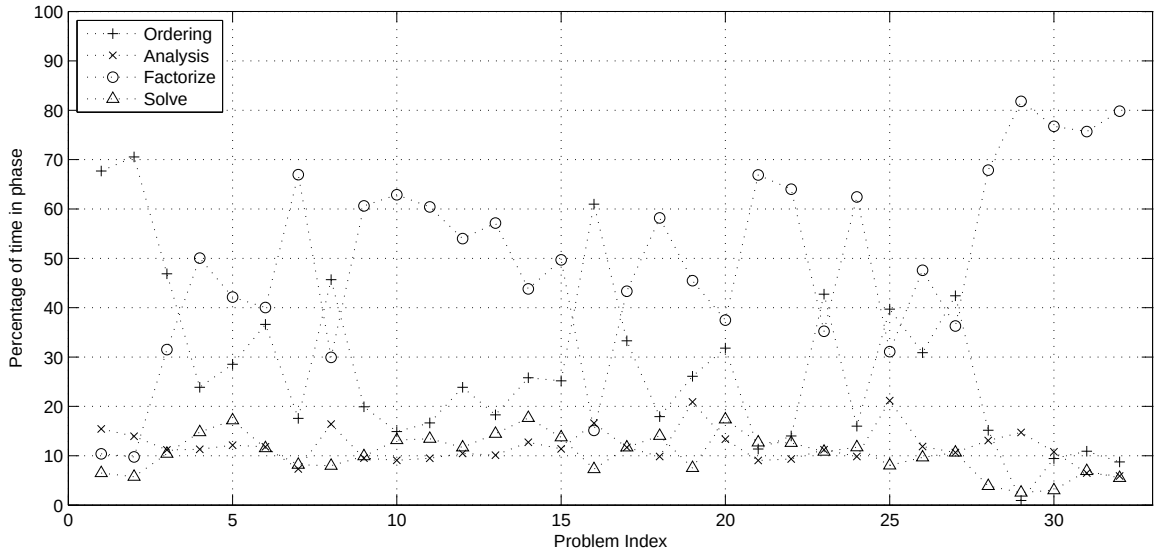
Traditionally, the factorize phase has generally required the greatest portion of the total execution time. It typically involves the majority of the floating-point operations, and is computation bound; the other phases are memory bound. Over the past decade or so, the increase in computational capacity (floating-point operations per second) has vastly outstripped the increase in memory bandwidth (bytes per second). This has led to the use of small, fast memory caches near the execution cores, enabling near peak floating-point performance to be achieved on workloads that have a high ratio of computation to memory accesses, such as matrix-matrix multiplication. Unfortunately, for workloads with a low ratio, such as matrix-vector multiplication, the memory bandwidth is the limiting factor. The result of this for sparse direct solvers has been an increase in the proportion of the computation time taken by the solve phase. This is particularly the case on new multicore chips that currently have a single memory channel for 2 or more cores.

To illustrate this, we ran our sparse direct solver, `HSL_MA87` [7], from the HSL mathematical software library [9] on a two-way quad-core Harpertown system (see Table 1.1). `HSL_MA87` uses a task-based DAG (directed acyclic graph) and has been developed specifically for solving large-scale sparse linear systems on multicore architectures. On a test set of 32 positive-definite problems of order in the range 2×10^4 to 1.6×10^6 arising from a variety of application areas (details are given in [7]), typically more than 98% of the floating-point operations were performed by the factorize phase, but 20–40% of the Level-2 cache misses were in the solve phase. The proportion of the total runtime spent in each phase is shown in Figure 1.1. Here only the factorize phase is run in parallel and the solve phase is for a single right-hand side. There is scope to parallelize the ordering and analyse phases, and this is an ongoing area of research, however in this paper we will limit our attention to the solve phase. We remark that in her evaluation of the solver SuperLU on multicore architectures [13], Li also highlighted the problems associated with the solve phase. We also note that, in a recent paper, Amestoy, Duff, Guermouche and Slavova [1] have examined the solve phase of a multifrontal algorithm in a distributed memory environment.

Table 1.1: Specifications of our two-way quad systems.

	Harpertown	Nehalem
Architecture	Intel(R) Xeon(R) CPU E5420	Intel(R) Xeon(R) CPU E5540
Clock	2.50 GHz	2.53Ghz
Cores	2×4	2×4
Level-1 cache	32 K on each core	128 K on each core
Level-2 cache	6 M for each pair of cores	128 K on each core
Level-3 cache	None	8192 K shared by 4 cores
Memory	32 GB for all cores	24 GB for all cores
BLAS	Intel MKL 10.1	Intel MKL 11.0
Compiler	Intel 11.0 with option -fast	Intel 11.0 with option -fast

Figure 1.1: The proportion of the total run time for solving $A\boldsymbol{x} = \boldsymbol{b}$ that is spent in each phase of HSL_MA87 on our 8-core Harpertown system.



In many instances where a direct solver is used, a number of calls to the solve phase may follow a single call to the factorize phase. The following four examples highlight how common and important the technique is.

Iterative refinement performs multiple solves to improve the quality of a solution. It is often required if static pivoting is used during the factorize phase (see, for example, [14]). In recent years, the use of static pivoting has gained in popularity and variants are used in a number of well-known sparse solvers (including SuperLU [12] and PARDISO [19]) because of the software design and implementation simplifications it offers and, in many cases, the performance improvements it allows.

Mixed precision solvers use a low precision factorization as a preconditioner for a higher precision iterative method. For example, in the sparse mixed precision solver described by Hogg and Scott [8], the factorize phase is performed in single precision (to take advantage of the savings in both the storage and computation time offered by single precision) and then iterative refinement or FGMRES [18] in double precision is used to recover accuracy.

Preconditioning more generally often uses a direct solver. A preconditioner may be computed using a variant of a direct algorithm (such as an incomplete matrix factorization); each application of the preconditioner will then involve calls to a solve phase.

Interior-point methods often require multiple solves, each using iterative refinement, before their augmented system matrices need refactorizing.

This paper is organised as follows. In Section 2, we discuss combining the factorize and solve phases and illustrate the savings this can achieve on a multicore machine. We then consider in Section 3 a number of approaches to parallelize the solve phase and present results to show their effectiveness. The best results are achieved using a DAG-based block approach. Section 4 considers various techniques for reducing the memory traffic, again with the view to improving the performance of the solve phase on a multicore machine. Finally, we present our conclusions in Section 5.

For simplicity, we restrict our discussion to the symmetric positive-definite case, however our ideas can be adapted to more general systems. All the presented results are computed using `HSL_MA87` (modified as necessary) and (with the exception of the results presented in Figure 3.3) are run on our two-way quad-core Harpertown system. All times are wall-clock times in seconds on a lightly loaded machine. The test problems are all symmetric and positive definite and are taken from the University of Florida Sparse Matrix Collection [2]. Each problem was ordered using the Metis graph partitioning package [10, 11]. Reported times are elapsed times in seconds, measured using the system clock.

2 Combined factorize and solve

For very large problems, direct solvers have been developed that optionally hold the matrix factors in files on disk; examples include the serial HSL packages `MA42` [5], `HSL_MA77` [17], and `HSL_MA78` [16]. The solve phase of such solvers is significantly more expensive than for a conventional direct solver because the matrix factor L must be read from disk once for the forward substitution and again for the backward substitution (in the general case, U is read for the backward substitution). A standard technique to reduce this overhead is to offer an option that combines the factorize and solve phases into a single phase, which we call the *factor_solve* phase. In this case, the forward substitution is performed at the same time as the numerical factorization, performing the required elimination operations using the portion of L that has just been computed, before it is written to file. Consequently, L needs to be read only once, for the backward substitution.

Release 2 of `HSL_MA87` offers a factor_solve phase. As in the out-of-core case, the forward elimination operations for one or more right-hand sides are performed as soon as each block of L is computed, while it is still in cache. This allows the forward substitution to be performed in parallel. Our implementation

of factor_solve requires additional memory since it uses a right-hand side array per thread. An alternative approach would be to employ a global right-hand side with locks to prevent write conflicts, but this is potentially less efficient, so we prefer to use extra storage. Numerical results comparing the time for the factorize phase followed by the solve phase (denoted by $F + S$) with that of factor_solve (denoted by F_S) for a single right-hand side are given in Table 2.2. We also report the separate solve time (S). The problems used here are the largest in our test set; the results for the smaller problems were consistent with these. We see that the saving resulting from combining the factorize and solve phases is approximately equal to half the cost of the separate solve, hence we conclude that the forward substitution is almost without cost.

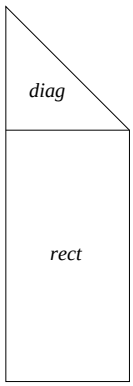
Table 2.2: Times (in seconds) on 8 cores for a call to factorize followed by a call to solve ($F + S$), a call to factor_solve (F_S) and a separate call to solve (S).

Problem	$F + S$	F_S	S
JGD_Trefethen/Trefethen_20000	19.2	18.9	0.63
ND/nd24k	51.0	50.0	2.09
Oberwolfach/bone010	76.8	73.5	7.11
GHS_psdef/audikw_1	114.7	110.9	8.31

3 Parallel solve

At the end of the factorize phase of `HSL_MA87`, the L factor is stored in the form of a set of trapezoidal block columns, one of which is similar to that shown in Figure 3.2. The square diagonal block L_{diag} is stored using full storage, with contiguous rows. The $k \times m$ rectangular block L_{rect} is also stored as a dense matrix by rows. The row storage allows arbitrary blocking by rows while maintaining data contiguity. The k columns of the trapezoidal block correspond to variables that were eliminated at the same step of the factorization.

Figure 3.2: Trapezoidal block column, consisting of a square diagonal block L_{diag} and a rectangular off-diagonal block L_{rect}.



In the serial solve phase, at each stage of the forward substitution, all the active components of the partial solution vector $\boldsymbol{y}$ can be gathered into a temporary dense vector $(\boldsymbol{u}, \boldsymbol{v})^T$, with $\boldsymbol{u}$ of size k and $\boldsymbol{v}$ of size m. The diagonal solve

$$\boldsymbol{u} \leftarrow L_{diag}^{-1}\boldsymbol{u}, \tag{3.1}$$

followed by the forward update operation

$$\boldsymbol{v} \leftarrow \boldsymbol{v} - L_{rect}\boldsymbol{u}, \tag{3.2}$$

can be performed before $(\boldsymbol{u}, \boldsymbol{v})^T$ is scattered into $\boldsymbol{y}$. Similarly, during the backward substitution, all the active components of the partial solution $\boldsymbol{y}$ can be gathered into a temporary vector $\boldsymbol{w}$ of size k and the active variables of the solution vector $\boldsymbol{x}$ can be gathered into a temporary vector $\boldsymbol{z}$ of size m. The backward update operation

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - L_{rect}^T \boldsymbol{z}, \tag{3.3}$$

and then the diagonal solve

$$\boldsymbol{w} \leftarrow L_{diag}^{-T} \boldsymbol{w}, \tag{3.4}$$

can be carried out before $\boldsymbol{w}$ is scattered into $\boldsymbol{x}$. These operations may be performed using the Level 2 BLAS kernels _TRSV and _GEMV. In the case of multiple right-hand sides, the Level 3 kernels _TRSM and _GEMM can be used. We remark that if k and m are small it may not pay to gather the components into a dense vector to allow the BLAS to be used; it may be more efficient to use indirect addressing (see, for example, [3]).

3.1 Straightforward parallelization

There are two straightforward ways to try and parallelize the solve phase:

Threaded BLAS: perform the linear algebra operations in parallel by using a version of the BLAS that has been specifically designed for use on a multicore machine. This option has the disadvantage that the factorize phase of our solver is designed to use non-threaded BLAS (its performance is adversely effected if multi-threaded BLAS are used), and there is no common portable threading interface to the BLAS.

OpenMP DO: effectively provide our own threading of the BLAS by splitting the matrix-vector multiplication into chunks using the OpenMP parallel do loop construct. The backward substitution requires some care to do the reduction on $\boldsymbol{w}$ efficiently.

A disadvantage of both these approaches is that, in effect, they reduce the size of the matrix involved in the BLAS calls. We found the backward substitution for the OpenMP DO approach hard to implement in an efficient manner. We tried both splitting the destination vector across threads and several reduction approaches. A reduction approach implemented using a single thread to gather contributions gave the best results but these were disappointing (see Tables 3.3 and 3.4 in Section 3.4).

3.2 Node approach

An alternative approach to parallelizing the solve phase is to exploit the potential for parallelism within the assembly tree that is set up by the analyse phase and used by the factorize phase. A trapezoidal block column of L is associated with each node of the tree. The forward substitution starts at the leaf nodes and moves up the tree towards the root. The diagonal solve (3.1) and forward update operations (3.2) at a node may be performed as soon as all the operations at each of its descendants in the tree have been completed. Thus for each node it is necessary to keep a count of the number of descendants that have yet to complete. For the backward substitution, the computation proceeds from the root down to the leaf nodes. The solve (3.4) and backward update operations (3.3) associated with a child can only be done after those at its parent have finished. We will refer to this strategy as the *node approach*.

The forward substitution must allow multiple processes to generate updates to the same part of the partial solution vector. To avoid write-write conflicts we are aware of three possible techniques:

Global vector plus locks: A single global vector is used and locks are maintained to prevent write conflicts. This minimises memory requirements and flop counts but this saving is offset by the need to wait for locks to be freed.

Local vector plus reduction: Each thread accumulates its updates to the partial solution vector. At each node the updates are summed into the partial solution vector. This is more straightforward than using a global vector but involves extra addition operations as well as more memory.

Multifrontal: Rather than gathering all updates in a single vector, each node stores its contribution from the update operations separately in a small vector. These are gathered by the parent before a diagonal solve is performed. Again, this involves extra addition operations and more memory than using a global vector. This approach is used, for example, by the solve phase of the sparse direct solvers MUMPS [1] and WSMP [6].

For a small number of threads, using local vectors requires fewer floating-point operations and less additional memory than the multifrontal strategy. On our 8-core shared-memory machine, our experience was that there was little to choose between the performance of the local vector and multifrontal approaches, with the use of the former being typically faster by less than 3 per cent; local vectors are used in the results reported in Section 3.4. In the future we anticipate the balance between the two approaches may change as larger numbers of cores become commonplace.

3.3 DAG Block approach

A problem with the node-based approach described in the previous subsection is that the amount of available parallelism is limited. In particular, the root node can be large and account for a significant proportion of the total work. It is therefore advantageous to allow more than one block column at a node and, as in the numerical factorize phase (see [7] for details), to subdivide each of the block columns by rows, potentially allowing finer-grained parallelism. Thus L_{rect} in Figure 3.2 is subdivided into blocks L_{rect1}, L_{rect2}, ... and the computation then proceeds by blocks. In particular, in the forward substitution, the forward update operation (3.2) is replaced by independent block forward update tasks

$$\begin{aligned} \boldsymbol{v}_1 &\leftarrow \boldsymbol{v}_1 - L_{rect1}\boldsymbol{u} \\ \boldsymbol{v}_2 &\leftarrow \boldsymbol{v}_2 - L_{rect2}\boldsymbol{u} \\ &\ldots. \end{aligned}$$

The tasks are dynamically scheduled for execution based on the dependencies among them, that are represented implicitly using a DAG. We block the solution in accordance with the column blocking and for each block we keep a dependency count of the number of forward update tasks still to be performed on it. Once a count reaches zero, the corresponding diagonal solve may be carried out. This in turn allows further forward update tasks to be performed. For the backwards solve, updates must be tracked on a block-by-block basis, which can be done by efficiently exploiting the depth-first postordering of the assembly tree. This allows us to loop over only the descendants of each node as it completes. For each node, a pointer to the last row corresponding to the next ancestor to be completed is maintained to allow updates to be quickly and cheaply determined. The ancestors must complete in order from the root to a node and cannot overlap because of the dependency structure — each is (indirectly) dependent on the next.

In this DAG-based block approach, both the forward and backward substitutions must allow multiple processes to generate updates to the same part of the solution vector simultaneously. The techniques described in the previous section may be adapted to this case. Based on our numerical experimentation, Release 2 of `HSL_MA87` employs the local vector approach and it is used in Section 3.4.

3.4 Numerical results

In Tables 3.3 to 3.6, we present numerical results for the approaches to parallelizing the solve phase discussed in Sections 3.1 to 3.3. Runs were made for all 32 problems in our test set; detailed results are given for every 8th problem for a single right-hand side and for ten right-hand sides. We report the time for

the serial solve phase as implemented within Release 1 of `HSL_MA87` together with the times and speedups on 4 and 8 cores for the four strategies: threaded BLAS, OpenMP DO, the node approach and the block approach. When run on 4 cores, all threads are assigned to the same quad-core processor. Note that, the extra book-keeping required by the node and block approaches incurs an overhead so that the serial times for these versions are greater than those for Release 1 (typically by up to 10 per cent); speedups are reported relative to the Release 1 times (and thus may be less than 1).

We see that on 4 cores (a single quad-core processor), the maximum speedup for a single right-hand side is limited to less than 1.30. Once we move to 8 cores (two quad-core processors) the node and block approaches achieve speedups in excess of 2 for many of our test problems. For ten right-hand sides, slightly higher speedups are achieved (the best being 3.19 for the block approach applied to our largest problem GHS_psdef/audikw_1). These modest speedups suggest we are limited by memory bandwidth; this is confirmed using profiling. The small gain over serial performance on a single quad-core processor is due to a mixture of concealing the latency and ensuring the memory channel is saturated. When running on 8 cores, we are using a second quad-core processor with an additional memory channel, allowing better performance.

There is little to choose between the threaded BLAS and the OpenMP DO. Both achieve disappointing speedups, generally of less than 1.5, even on 8 cores. Moreover, their performance is consistently poorer than both the node and block approaches. The former is best on 4 cores while on 8 cores the latter is generally the fastest approach. The block scheme produces finer-grained parallelism at the cost of additional overheads; this is beneficial when running on two quad-core processors.

Figure 3.3 shows the speedup achieved in the factorize and solve phases on both the older (2008) Harpertown and more recent (2009) Nehalem architectures, using the block approach for the solve phase. On both machines the speedup achieved in the solve phase is considerably worse than that in the factorize phase. Using the Nehalem architecture generally improves the speedup, though it is still less than 3 for most problems. Our understanding is that the serial run uses less of the available memory bandwidth on the Nehalem architecture, giving greater scope for parallel speedup.

4 Reducing memory traffic

As we have seen, the total memory traffic is the limiting factor for the solve phase. In this section, we consider the viability of using in-memory compression techniques to reduce this traffic on multicore architectures.

In-memory compression works by taking a block of data in memory and producing a smaller representation of it. This smaller representation can then later be decompressed to recover the original data. For the solve phase this is potentially advantageous because we can perform the compression and decompression of factor data in the processor's cache, and move only the compressed factor through the channel between cache and main memory. This effectively increases the amount of memory bandwidth available at the cost of compression/decompression computations.

The success of this technique will depend largely on two factors:

Compression ratio: the amount by which the factor data can be reduced. A compression ratio of 2 would mean the data required half as much storage after compression compared with before.

Decompression rate: how fast the data can be decompressed. We will measure this in Mbytes/s.

If we had infinite processing capacity and were only restricted by the memory bandwidth, we would expect a speedup in the solve time equal to the compression ratio. If processing capacity is finite then this is an upper limit, and instead we may be limited by the decompression rate. We remark that, although compression may give a speedup in the solve phase, there is a penalty in the factorize phase when the data is compressed. As this can be amortised over several solves, and we want to explore the potential of data compression, we choose not to concern ourselves with this at present.

Table 3.3: Serial times and times on 4 cores (in seconds) together with speedups (sp) for a single right-hand side. The fastest times are in bold.

Problem	serial	threaded BLAS		OpenMP DO		node		block	
		4	sp	4	sp	4	sp	4	sp
Rothberg/cfd2	0.30	0.29	1.04	0.29	1.04	**0.24**	1.24	0.25	1.24
AMD/G3_circuit	1.50	1.47	1.02	1.56	0.96	**1.18**	1.27	1.26	1.19
Chen/pkustk14	0.77	0.73	1.05	0.71	1.08	**0.62**	1.24	0.63	1.23
GHS_psdef/audikw_1	8.31	7.72	1.08	7.39	1.12	**6.86**	1.21	6.90	1.20

Table 3.4: Serial times and times on 8 cores (in seconds) together with speedups (sp) for a single right-hand side. The fastest times are in bold.

Problem	serial	threaded BLAS		OpenMP DO		node		block	
		8	sp	8	sp	8	sp	8	sp
Rothberg/cfd2	0.30	0.24	1.24	0.28	1.07	0.15	2.02	**0.14**	2.14
AMD/G3_circuit	1.50	1.36	1.11	1.65	0.91	**0.79**	1.91	0.84	1.79
Chen/pkustk14	0.77	0.55	1.41	0.65	1.18	**0.34**	2.25	**0.34**	2.27
GHS_psdef/audikw_1	8.31	5.23	1.59	5.51	1.51	3.80	2.18	**3.61**	2.30

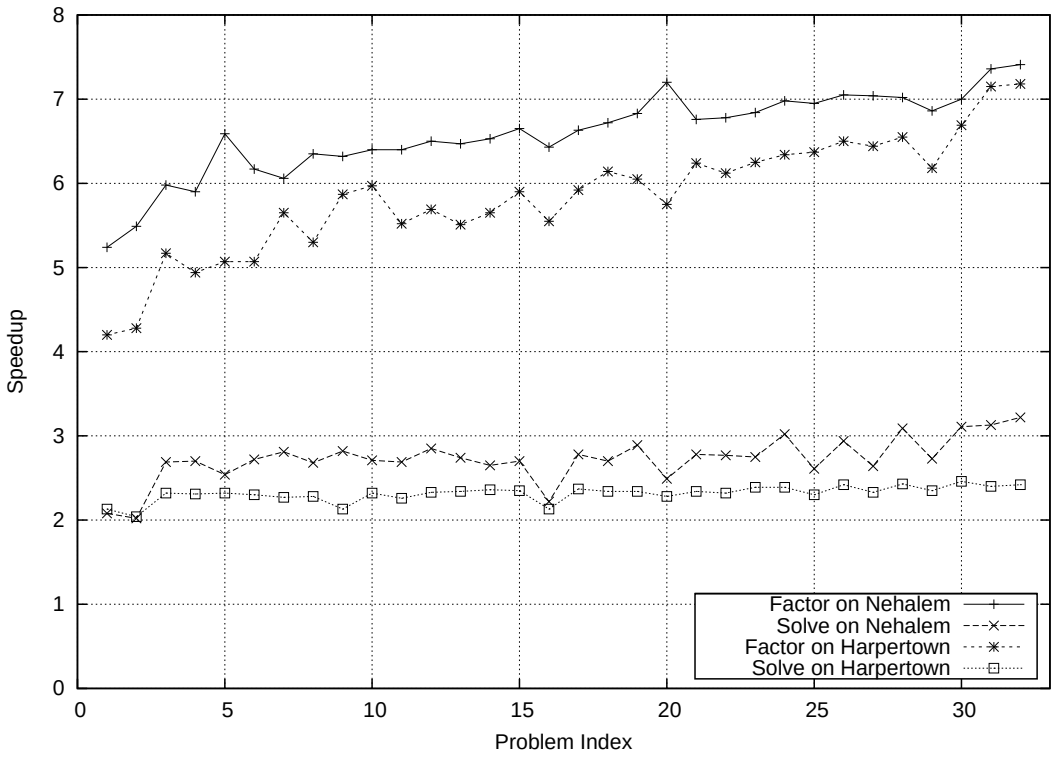
Table 3.5: Serial times and times on 4 cores (in seconds) together with speedups (sp) for ten simultaneous right-hand sides. The fastest times are in bold.

Problem	serial	threaded BLAS		OpenMP DO		node		block	
		4	sp	4	sp	4	sp	4	sp
Rothberg/cfd2	0.63	0.53	1.20	0.58	1.09	0.40	1.59	**0.38**	1.68
AMD/G3_circuit	3.47	3.27	1.06	3.69	0.94	2.53	1.37	**2.46**	1.41
Chen/pkustk14	1.57	1.20	1.31	1.33	1.18	0.92	1.70	**0.84**	1.86
GHS_psdef/audikw_1	17.02	11.93	1.43	11.57	1.47	10.27	1.66	**8.61**	1.98

Table 3.6: Serial times and times on 8 cores (in seconds) together with speedups (sp) for ten simultaneous right-hand sides. The fastest times are in bold.

Problem	serial	threaded BLAS		OpenMP DO		node		block	
		8	sp	8	sp	8	sp	8	sp
Rothberg/cfd2	0.63	0.52	1.22	0.59	1.07	0.31	2.03	**0.29**	2.15
AMD/G3_circuit	3.47	3.42	1.01	4.08	0.85	2.47	1.40	**2.46**	1.41
Chen/pkustk14	1.57	1.08	1.45	1.28	1.23	0.62	2.54	**0.56**	2.79
GHS_psdef/audikw_1	17.02	9.72	1.75	9.87	1.72	6.58	2.59	**5.34**	3.19

Figure 3.3: Speedups on 8 cores of the factorize and solve phases on two different multicore architectures.



We consider two options for data compression within our solver:

Generic algorithm: use a generic in-memory compression algorithm to store the compressed factor data in main memory, expanding out blocks of L into cache when required.

Factor sparsity: avoid storing explicit zeros in the computed factor. Explicit zeros may occur naturally, however some zero entries are introduced in the analyse phase through the use of node amalgamation (see Section 4 of [4]). These zeros can be removed before the solve phase and the trapezoidal block columns (Figure 3.2) are then held as sparse, rather than dense, matrices, thus reducing the number of entries in the factors and flops required during the solve. The penalty is that the BLAS cannot be used during the solve phase without first copying the blocks back into dense storage.

Table 4.7 shows the compression ratios achieved for a number of our test problems using the LZO compression library [15] (specifically LZO_1_1X). We note that LZO gives a compression ratio comparable

Table 4.7: The size of L (in Mbytes) and compression ratios (CR).

		LZO				LZO	
	`nemin` = 32	`nemin` = 32		`nemin` = 1		`nemin` = 1	
Problem	size	size	CR	size	CR	size	CR
CEMW/tmt_sym	579	333	1.74	283	2.04	269	2.14
Schmid/thermal2	981	567	1.73	489	2.01	463	2.12
GHS_psdef/crankseg_1	315	283	1.11	292	1.08	269	1.17
DNVS/shipsec1	407	328	1.24	342	1.19	303	1.34
GHS_psdef/audikw_1	10829	10310	1.05	10304	1.05	10009	1.08

to that of the bzip2 data compressor (see `http://www.bzip.org/`), but is considerably faster during decompression (on our two-way quad Harpertown system we measured the decompression rates for LZO and bzip2 to be approximately 2450 and 12.0 Mbytes/s, respectively). The size of L (that is, the storage in Mbytes needed to hold the dense trapezoidal block columns and the corresponding row index lists) and the compression ratios are given for two values of the node amalgamation parameter, `nemin`. The value `nemin` = 32 is the default setting within `HSL_MA87` and was chosen to achieve good factorization performance (see [7]), while `nemin` = 1 (that is, node amalgamation is disabled) represents the sparsest representation of the factor we are likely to achieve by removing zeros (a sparser representation may occur if the original matrix data file contained explicit zeros that were treated as non-zeros during the analyse phase). The compression ratio (CR) is given relative to the uncompressed data with `nemin` = 32 (column 2). The test problems reported on here were chosen to illustrate the three kinds of behaviour that we saw: compression ratios close to 1.0 (GHS_psdef/audikw_1), savings of around 50 per cent in the size of L for `nemin` = 1 (CEMW/tmt_sym and Schmid/thermal2), and small savings in the size of L (GHS_psdef/crankseg_1 and DNVS/shipsec1).

In Table 4.8, we present solve times with and without LZO compression; the ratios between the times are also given. We see for the problems with high compression ratios (CEMW/tmt_sym and Schmid/thermal2) modest gains in performance are achieved, but disappointingly nothing approaching the compression ratio limit.

Table 4.9 gives the solve time using the block approach for smaller values of `nemin`. While our experience in serial [7] was that smaller values can sometimes significantly reduce the solve time, this is not true to the same extent for our parallel approach. Table 4.10 demonstrates the reason for this — a small `nemin` generally results in many more tasks (using smaller blocks), and this substantially increases the overheads of running in parallel.

Table 4.8: Solve times (in seconds) and ratios of the times for the block approach with and without LZO compression (`nemin` = 32).

	4 cores			8 cores		
Problem	without	with	ratio	without	with	ratio
CEMW/tmt_sym	0.44	0.38	1.16	0.30	0.27	1.11
Schmid/thermal2	0.76	0.67	1.13	0.53	0.48	1.10
GHS_psdef/crankseg_1	0.19	0.19	1.00	0.11	0.11	1.00
DNVS/shipsec1	0.25	0.24	1.04	0.15	0.14	1.07
GHS_psdef/audikw_1	6.90	6.92	1.00	3.61	3.59	1.01

Table 4.9: Solve times (in seconds) using the block approach with a range of `nemin` values on 8 cores. The fastest times are in bold.

Problem	`nemin` = 1	`nemin` = 8	`nemin` = 32
CEMW/tmt_sym	1.59	0.42	**0.30**
Schmid/thermal2	2.62	0.70	**0.53**
GHS_psdef/crankseg_1	0.11	**0.10**	0.11
DNVS/shipsec1	0.17	**0.15**	**0.15**
GHS_psdef/audikw_1	4.06	**3.28**	3.61

Table 4.10: Total number of tasks during the solve phase using the block approach with a range of `nemin` values.

Problem	`nemin` = 1	`nemin` = 8	`nemin` = 32
CEMW/tmt_sym	974,094	157,960	38,372
Schmid/thermal2	1,572,784	262,434	66,512
GHS_psdef/crankseg_1	7,108	4,842	2,358
DNVS/shipsec1	29,402	17,450	6,640
GHS_psdef/audikw_1	262,960	131,856	69,722

5 Conclusions

In this paper, we have highlighted a problem facing direct solvers on multicore chips and evaluated possible solutions, showing a limited potential to improve on serial performance through the use of a parallel node or block solve approach and compression of the factor data.

An important implication of this bottleneck is that it currently severely limits the potential speedup on multicore architectures of algorithms that rely on repeated calls to the solve phase. As noted in Section 1, such algorithms include static pivoting techniques for speeding up and simplifying the factorization of systems that are not positive definite and mixed precision approaches that perform the factorization in single precision but require the solution in double precision, as well as the use of direct solvers in the computation of preconditioners and interior-point methods. These algorithms are only effective if the solve phase is many times cheaper than the factorize phase. While this was true on more traditional computer architectures, our study has shown that on multicore machines this assumption is not necessarily valid. This suggests it may be necessary to develop alternatives to iterative refinement and other iterative methods that use direct solvers as a preconditioning step. These alternatives may include computing a more accurate factorization to reduce the number of iterations required (albeit at the cost of a more expensive factorize phase). Alternatively, they may seek to only read each part of the factors into cache once, performing multiple operations with that part while it is there.

In our multicore solver `HSL_MA87` we currently use the DAG block-based scheduling method and recommend that, if repeated calls to the solve phase will follow the factorization, the user should experiment with using a value of the node amalgamation parameter, `nemin`, that is less than the default setting. Additionally, where appropriate, the combined factor-solve option should be employed.

6 Acknowledgements

We are grateful to our colleague John Reid for commenting on a draft of this paper.

References

[1] P. Amestoy, I. S. Duff, A. Guermouche, and Tz. Slavova. Analysis of the solution phase of a parallel multifrontal approach. *Parallel Computing*, 36:3–15, 2010.

[2] T. A. Davis. The University of Florida sparse matrix collection. Technical Report, University of Florida, 2007. `http://www.cise.ufl.edu/~davis/techreports/matrices.pdf`.

[3] I. S. Duff. MA57 - A new code for the solution of sparse symmetric definite and indefinite sysetms. Technical Report RAL-TR-2002-024, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England, 2002.

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

[5] I. S. Duff and J. A. Scott. The design of a new frontal code for solving sparse unsymmetric systems. *ACM Trans. Math. Softw.*, 22(1):30–45, 1996.

[6] A Gupta and M. Joshi. WSMP: A high-performance shared- and distributed-memory parallel sparse linear equation solver. Technical Report RC 22038, IBM T. J. Watson Research Center, Yorktown Heights, NY, 2001.

[7] J. D. Hogg, J. K. Reid, and J. A. Scott. Design of a multicore sparse Cholesky factorization using DAGs. Technical Report RAL-TR-2009-027, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England, 2009.

[8] J.D. Hogg and J.A. Scott. A fast and robust mixed precision solver for the solution of sparse symmetric linear systems. *ACM Trans. Math. Softw.*, 37(2), 2010. Article 17, 24 pages.

[9] HSL. A collection of Fortran codes for large-scale scientific computation, 2007. See `http://www.hsl.rl.ac.uk/`.

[10] G. Karypis and V. Kumar. METIS - family of multilevel partitioning algorithms, 1998. See `http://glaros.dtc.umn.edu/gkhome/views/metis`.

[11] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Scientific Computing*, 20:359–392, 1999.

[12] X. S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31:302–325, 2005.

[13] X. S. Li. Evaluation of SuperLU on multicore architectures. *J. Phys.: Conf. Ser.*, 125, 2008. 6 pages.

[14] X.S. Li and J. W. Demmel. A scalable sparse direct solver using static pivoting. In *Proceedings of the 9th SIAM conference on Parallel Processing for Scientic Computing, San Antonio, Texas*, 1999.

[15] M. F. X. J. Oberhumer. LZO — a real-time data compression library. See `http://www.oberhumer.com/opensource/lzo/`.

[16] J. K. Reid and J. A. Scott. An efficient out-of-core multifrontal solver for large-scale unsymmetric element problems. *Intl. J. Numer. Methods Engrng.*, 77(7):901–921, 2009.

[17] J. K. Reid and J. A. Scott. An out-of-core sparse Cholesky solver. *ACM Trans. Math. Softw.*, 36(2), 2009. Article 9, 33 pages.

[18] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Scientific and Statistical Computing*, 14:461–469, 1994.

[19] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Journal of Future Generation Computer Systems*, 20:475–487, 2004.