# The design of an out-of-core multifrontal solver for the 21st century

John K. Reid and Jennifer A. Scott

Computational Science and Engineering Department,
Rutherford Appleton Laboratory,
Chilton, Oxfordshire, OX11 0QX, England, UK
{j.k.reid, j.a.scott}@rl.ac.uk

## Extended Abstract

The popularity of direct methods for solving large sparse linear systems of equations $Ax = b$ stems from their generality and robustness. Indeed, if numerical pivoting is properly incorporated, direct solvers rarely fail for numerical reasons; the main reason for failure is a lack of memory. Although increasing amounts of main memory have enabled direct solvers to solve many problems that were previously intractable, their memory requirements generally increase much more rapidly than problem size so that they can quickly run out of memory. Buying a machine with more memory is an expensive and inflexible solution since there will always be problems that are too large for the chosen quantity of memory. Using an iterative method may be a possible alternative but for many of the "tough" systems that arise from practical applications, the difficulties involved in finding and computing a good preconditioner can make iterative methods infeasible. Another possibility is to use a direct solver that is able to hold its data structures on disk, that is, an out-of-core solver.

The advantage of using disk storage is that it is many times cheaper than main memory per megabyte, making it practical and cost-effective to add tens or hundreds of gigabytes of disk space to a machine. By holding the main data structures on disk, properly implemented out-of-core direct solvers are very reliable since they are much less likely than in-core solvers to run out of memory.

The idea of out-of-core linear solvers is not new (see, for example, [3],[6] and, recently, [2], [7]). Our aim is to design and develop a sparse symmetric out-of-core solver for inclusion within the mathematical software library HSL [5]. Our new solver, which is called `hsl_ma77`, implements an out-of-core multifrontal algorithm and is designed for the efficient solution of both positive-definite and indefinite sparse linear systems with one or more right-hand sides. Input of the system matrix $A$ may be by rows or by square symmetric elements.

The multifrontal method is a variant of sparse Gaussian elimination and involves the matrix factorization

$$A = (PL)D(PL)^T,$$

where $P$ is a permutation matrix and $L$ is a unit lower triangular matrix. In the positive-definite case, $D$ is diagonal; in the indefinite case, $D$ is block diagonal

with blocks of size $1 \times 1$ and $2 \times 2$. Solving $Ax = b$ is completed by performing forward elimination followed by back substitution. The basic multifrontal algorithm for element problems is summarised below. The assemblies can be held

---

Given a pivot sequence:
**do** for each pivot
Assemble all elements that involve the pivot into a full matrix (the *frontal* matrix)
Perform static condensation, that is, eliminate the pivot variable and any others that do not appear in any elements not yet assembled;
Treat the reduced matrix as a new element (a *generated* element)
**end do**

---

as a tree, called an *assembly* tree. Each *node* of the assembly tree represents an element. When a pivot is eliminated and a generated element created, a node is added to the tree whose children are the elements that involve the pivot (these may be original elements and/or generated elements).

The multifrontal method can be extended to non-element problems by regarding row $i$ of $A$ as a packed representation of a $1 \times 1$ element (the diagonal $a_{ii}$) and, for each $a_{ij} \neq 0$, a $2 \times 2$ element of the form

$$A^{(ij)} = \begin{pmatrix} 0 & a_{ij} \\ a_{ij}^T & 0 \end{pmatrix}.$$

When $i$ is chosen as pivot, the $1 \times 1$ element plus the subset of $2 \times 2$ elements $A^{(ij)}$ for which $j$ has not yet been selected as a pivot must be assembled. Rows that have an identical pattern may be treated together by grouping the corresponding variables into so-called supervariables.

The efficiency of the multifrontal method depends on the implementation details, including:

- The choice of pivot order.
- Merging nodes to increase the pivot block size.
- The ordering of the children at each node.
- Use of dense linear algebra computations to perform static condensations.

We briefly discuss each of these in the context of `hsl_ma77`.

### Pivot order

Although finding a good pivot order has been a subject of much research during the last 20 years, there is no one algorithm that produces the best ordering for all problems. Thus an early design decision for `hsl_ma77` was that the pivot order must be supplied by the user. Several stand-alone packages already exist that can be used, including a number of HSL routines. Statistics on the number of

entries in the factors and the number of floating-point operations (based on the sparsity pattern) are returned by the analyse phase and these allow the user to compare the effectiveness of different ordering algorithms.

**Merging of nodes**

Merging a parent and child reduces data movement and increases the pivot block size, allowing better use of cache. However, merging nodes may result in more fill in the factors and more floating-point operations. In `hsl_ma77`, if the list of uneliminated variables at a generated element is contained within the list of variables at its parent, the two nodes are merged. A parent and one of its children are also merged if both involve fewer than a prescribed number of eliminations. The user is able to choose the value of this node amalgamation parameter.

**Ordering the children**

At each node of the assembly tree, all the children must be processed and their generated elements computed before the frontal matrix at the parent can be factorized. However, the children can be processed in any order and this can significantly effect the total storage required. The simplest strategy is to wait until the last child of a node has been processed and then allocate the frontal matrix and assemble all the generated elements from its children into it. This requires the generated element from each child to be stacked and there may be many child elements that are nearly as big as the parent. To reduce storage requirements, Guermouche and L'Excellent [4] propose computing, for each node, the optimal point at which to allocate the frontal matrix and start the assembly. Given a parent node $i$ with $n_i$ children, let $f_i$ be the memory needed for the frontal matrix at node $i$, let the size of the generated element at the $j$th child be $g_j$ and let the storage required to generate child $j$ be $s_j$. Then, if the frontal matrix is allocated and the assembly started after $p$ children have been processed, Guermouche and L'Excellent show that the storage needed to process $i$ is

$$s_i = \max\left(\max_{j=1,p}\left(\sum_{k=1}^{j} g_k + s_j - g_j\right), f_i + \sum_{k=1}^{p} g_k, f_i + \max_{j>p} s_j\right).$$

Their algorithm for finding the *split point*, that is, the $p$ that gives the smallest $s_i$, then proceeds as follows: for each $p$ ($1 \leq p \leq n_i$), order the children in decreasing order of $s_j$, then reorder the first $p$ children in decreasing order of $s_j - g_j$. Finally, compute the resulting $s_i$ and take the split point to be the $p$ that gives the smallest $s_i$. They prove this gives the optimal $s_i$. This algorithm is implemented within the analyse phase of `hsl_ma77`.

**Dense linear algebra**

For fast performance, it is essential that efficient implementations of the dense linear algebra computations that lie at the heart of the factorization and solve phases are employed. Recently, Andersen et. al. [1] proposed a block hybrid storage format that aims to minimise storage requirements and to reduce caching overhead. They showed how the Cholesky factorization and solution of dense systems can be efficiently implemented using this format. Their kernels cannot

be used directly within hsl_ma77 since, at each stage, a partial factorization must be performed. Reid has developed a separate HSL package hsl_ma54 that modifies the kernels of [1] to perform a partial Cholesky factorization. hsl_ma54 is employed within hsl_ma77. The blocksize is a parameter that the user of hsl_ma77 can tune for optimum performance. So far, kernels for the positive definite case have been written; further work on the kernels for the indefinite case is still to be completed.

**Out-of-core working**

As already noted, the most important feature of hsl_ma77 is that it is an out-of-core solver. It allows the matrix factor and the multifrontal stack, as well as the original matrix data, to be held in direct-access files. All input and output to disk is performed through a set of Fortran subroutines that manage a virtual memory system so that actual i/o occurs only when really necessary. This set of subroutines is available within HSL as the package hsl_of01. Handling i/o through a separate package was part of the original element out-of-core solver of Reid [6] and our approach is essentially a modification of that used by the earlier code. We note that, because i/o can add a significant overhead, the user can request that arrays be used instead of direct-access files. In this case, the code works in-core but if the workspace supplied by the user is found to be too small, a switch to using files is automatically made.

**Summary**

We are currently performance testing hsl_ma77. Results so far are encouraging; results for a range of practical problems will be included in our talk. The packages hsl_of01, hsl_ma54, and hsl_ma77 will be included within the next release of HSL.

## References

1. B.S. Andersen and J.A. Gunnels and F.G. Gustavson and J.K. Reid and J. Wasniewski A fully portable high performance minimal storage hybrid format Cholesky algorithm. *ACM Transactions on Mathematical Software*, **31**, 201–208, 2005.
2. F. Dobrian and A. Pothen. A comparison between three external memory algorithms for factorising sparse matrices. *in* 'Proceedings of the SIAM Conference on Applied Linear Algebra', 2003.
3. I.S. Duff. Design features of a frontal code for solving sparse unsymmetric linear systems out-of-core. *SIAM J. Scientific and Statistical Computing*, **5**, 270–280, 1984.
4. A. Guermouche and J.-Y. L'Excellent. Optimal memory minimization algorithms for the multifrontal method. Technical Report RR5179, INRIA, 2004.
5. HSL. A collection of Fortran codes for large-scale scientific computation, 2004. See http://hsl.rl.ac.uk/.
6. J.K. Reid. TREESOLV, a Fortran package for solving large sets of linear finite-element equations. Report CSS 155, AERE Harwell, 1984.
7. V. Rotkin and S. Toledo. The design and implementation of a new out-of-core sparse Cholesky factorization method. *ACM Transactions on Mathematical Software*, **30**(1), 19–46, 2004.