Achieving bit-compatibility in a sparse direct symmetric solver

Jennifer Scott
STFC Rutherford Appleton Laboratory
in collaboration with Jonathan Hogg

PMAA 2012 London, 28th June 2012

Outline of talk

- Introduction and motivation
- Multifrontal method
  - Brief overview
  - Implementation within HSL_MA97, with emphasis on robustness, efficiency and bit compatibility
- Numerical results and comparisons
- Concluding remarks

Sparse linear system

Solve

$$Ax = b$$

with A large, sparse, symmetric and possibly indefinite (may be singular).

For example, saddle-point systems arise in a number of important applications

$$\begin{pmatrix} H & B^T \\ B & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} b \\ c \end{pmatrix}$$

Direct method

- Compute explicit factorization

$$SAS = (PL)D(PL)^T$$

- where L (unit) is lower triangular,
- D block diagonal with 1×1 and 2×2 blocks.
- S is a diagonal scaling matrix chosen to improve performance.
- P is a permutation matrix chosen to limit fill in L and for numerical stability.

Why develop a new direct solver?

- HSL specialises in sparse matrix computations.
- Largest collection of sparse direct solvers anywhere.
- Older multifrontal codes `MA27` (Duff and Reid '83) and `MA57` (Duff) are well-known and remain widely used ... account for more than half of HSL downloads, frequently for use within optimization packages (saddle-point systems).
- Also out-of-core multifrontal code `HSL_MA77` designed for very large problems.

But not parallel (except through use of multithreaded BLAS).

Design aims

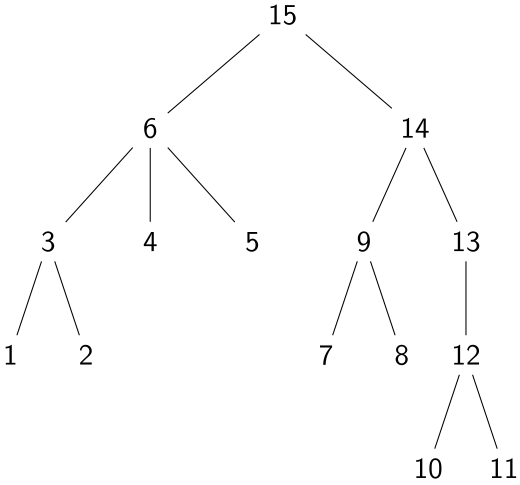
- Multifrontal code for multicore architectures.
- Efficient, robust, flexible, user-friendly code that is fully tested, supported and maintained.
- Provide basis for future research (replace `MA57`).

Package: `HSL_MA97`.

Language: Fortran 95 and OpenMP.

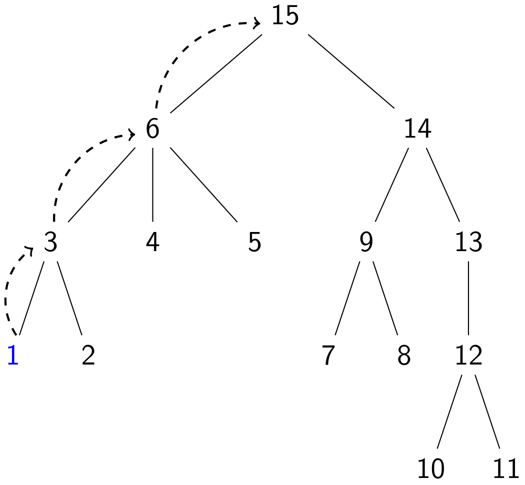
Multifrontal approach

Represent the sparse problem as a tree of dense subproblems



Multifrontal approach

Represent the sparse problem as a tree of dense subproblems



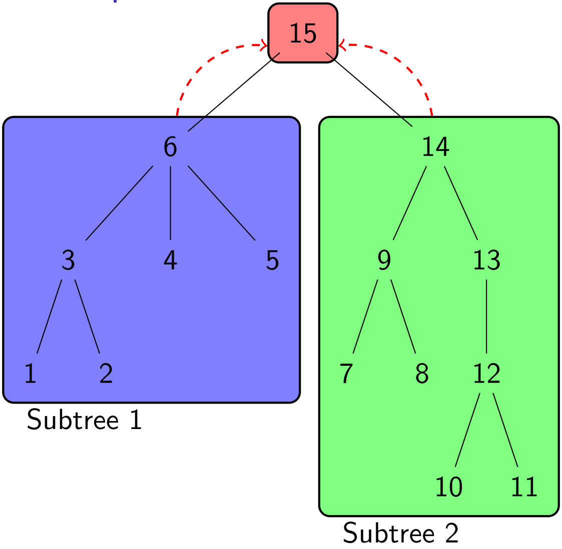
Notes on multifrontal

- Tree depends on ordering (eg nested dissection).
- Each (non-root) node has single parent and each non-leaf node has ≥ 1 child.
- Leaf nodes: small, lots of them, each involves little work.
- Near root: large nodes, account for most of the computational work.

Why choose multifrontal?

- Naturally adapts to parallel implementation by processing multiple independent subtrees simultaneously.
  Tree-level parallelism.

Tree-level parallelism



Why choose multifrontal?

- Parallelism can be exploited within each dense subproblem. Node-level parallelism.

Parallel multifrontal codes for symmetric systems include:

MUMPS (MPI code, Amestoy, L'Excellent et al),

WSMP (IBM, Gupta),

TAUCS (Toledo et al).

Subtree factorization

Basic work unit for parallel multifrontal is factorization of subtree.

Serial case: single subtree factorization.

Parallel case: a number of subtrees are factorized simultaneously.

Subtree factorization computes

- the entries of L and D associated with the nodes within the subtree, and
- the contribution associated with root of subtree to next (higher) level in tree.

Work at a node

At each node, dense $m \times m$ frontal matrix

$$\begin{pmatrix} F_1 & F_2^T \\ F_2 & E \end{pmatrix}.$$

Rows/columns of F_1 are fully summed (do not appear higher up tree and so are elimination candidates).

1. Factorization: $F_1 = L_1 D L_1^T$
2. Solve: $L_2 = F_2 L_1^{-1}$. (L_1, L_2) are computed columns of L.
3. Update: $E \leftarrow E - L_2 (L_2 D)^T$ (BLAS 3).

   E is generated element that is passed up tree.

Achieving good performance

Key to good performance is efficiency of dense factorization.

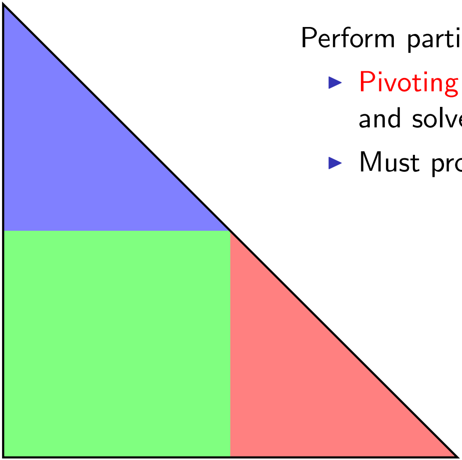
Can't use LAPACK since doesn't perform partial factorization.

Can't use LAPACK for $F_1 = L_1 D L_1^T$ because, for stability, entries in F_2 must be considered.

Instead, developed recursive factorization procedure that

- incorporates threshold partial pivoting,
- exploits symmetry, and
- is cache agnostic.

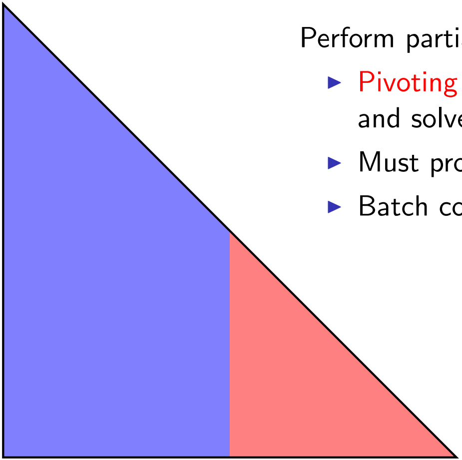
Recursive factorization of frontal matrix



Perform partial factorization at node.

- Pivoting — can't factorize blue part and solve with green part.
- Must proceed column-by-column.

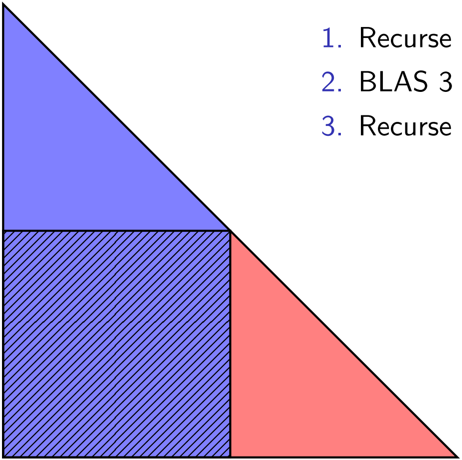
Recursive factorization of frontal matrix



Perform partial factorization at node.

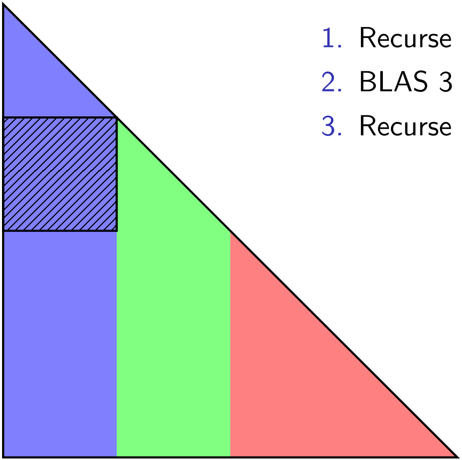
- Pivoting — can't factorize blue part and solve with green part.
- Must proceed column-by-column.
- Batch columns together

Recursive factorization of frontal matrix



1. Recurse on blue area
2. BLAS 3 outer product of shaded area
3. Recurse again as required

Recursive factorization of frontal matrix



1. Recurse on blue area
2. BLAS 3 outer product of shaded area
3. Recurse again as required

Recursive factorization

Recursion continues until number of columns is small or recursion depth exceeds some maximum.

At lowest level, use dense factorization kernel. Exploit BLAS 3.

Note: numerical stability very important so use threshold partial pivoting (want to compute inertia and minimise number of steps of refinement and hence number of solves).

If columns fail pivot test — get delayed (if delayed to higher up tree, increases flop count and fill in L).

Tree-level parallelism

Tree-level parallelism is exposed recursively. At each node i:

- If amount of work associated with subtree rooted at i is small ($< 10^5$ flops), subtree factorization code used.
- Otherwise, task created for each child node (and subtree rooted at child).

  If only small amount of work in consecutive children, merge into single task (necessary to avoid slow down).

  Once all child node tasks have run (in parallel), subtree factorization code performs assembly and factorization operations at i.

Bit compatibility

What? The computed factorization and solutions are bit-for-bit identical regardless of the number of threads used.

That is, results are reproducible with successive runs using identical input data yielding identical output data (assuming they are performed using identical hardware and operating system environments).

Is non-bit compatibility a problem?

- Non-reproducibility well understood within scientific computing community, so may not be seen as a problem by this group.
- But code's end users may have no idea how code been executed so may be unable to judge whether (eg) rounding errors have been propogated unfavourably.
- Of course, non-reproducible results could be seen as a positive feature since requires user to consider whether the program's results are what is expected.
- But can be very worrying for user to see different runs returning different results.

Why do we want bit compatibility?

Many potential reasons:

- Aids users in debugging their program.
- Later part of user's program may be unstable so sensitive to output from solver.
- Increases confidence if results are repeatable.
- Can be requirement in some application areas (eg financial computations).
- Requested by some HSL users (may be inexperienced or limited background in numerical mathematics and scientific computing).

Note: nice article on this by Kai Diethelm in *Computing in Science and Engineering*, January 2012

Achieving bit compatibility

Two issues:

- How do we achieve bit compatibility when executing our solver in parallel?
- What does it cost us in terms of performance?

Achieving bit compatibility

Must consider both levels of parallelism.

Tree-level parallelism:
Requires assembly order of child nodes to be fixed.

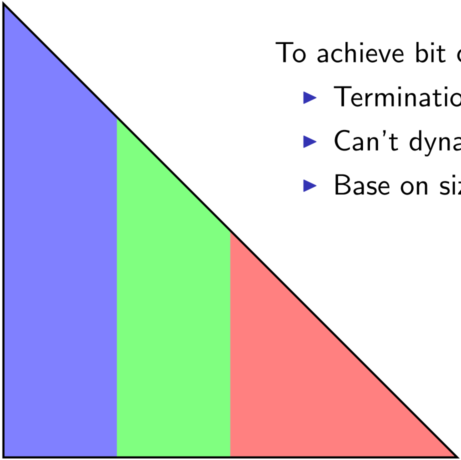
That is, at each node i, once all the child node tasks have been run, the contributions from the child nodes must be added into the frontal matrix in the same order, independently of number of threads.

Achieving bit compatibility

Node-level parallelism:

- Blocking must be independent of the number of threads (cannot optimize block size).
- Data-parallel approach is used so that each individual sum is effectively calculated in serial.

Node-level parallelism



To achieve bit compatibility

- Termination of recursion must be fixed.
- Can't dynamically adjust.
- Base on size of node and depth in tree.

Cost of bit compatibility

Imposing bit compatibility does mean a loss of efficiency.

Our experiments suggest resulting overhead within our solver is around 20-30 per cent.

We feel this is acceptable.

Note: HSL has a wide user base and our aim is always to achieve reliability and robustness and this sometimes is at the cost of some loss of efficiency.

Comment on bit compatibility

`HSL_MA97` uses BLAS routines and our tests found bit-compatibility dependent on BLAS library used.

- Bit-compatibility not achieved with the GotoBLAS (which is normally our prefered choice for BLAS).
- For the Intel MKL, bit-compatibility achieved provided BLAS 3 used.
- No problems were encountered using ACML or ATLAS BLAS libraries.

Numerical problems

Problems from University of Florida Sparse Matrix Collection.

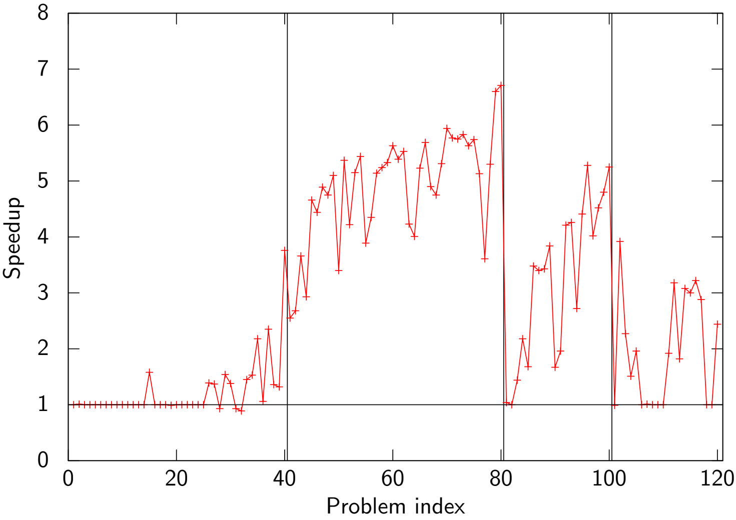
- Test Set 1: 40 small indefinite matrices (including some KKT systems).
- Test Set 2: 40 positive-definite matrices.
- Test Set 3: 20 general indefinite matrices (non-KKT systems).
- Test Set 4: 20 (mostly larger) KKT indefinite matrices.

For full details and complete results, see STFC Technical Report RAL-TR-2011-24.

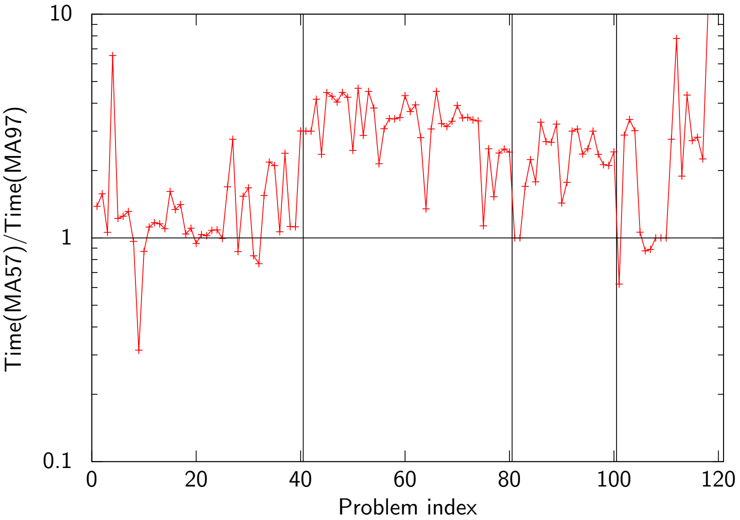
Test environment

Processor	2 × Intel Xeon E5620
Physical Cores	8
Memory	24 GB
Compiler	Intel Fortran 12.0.0
	ifort -g -fast -openmp
BLAS	MKL 10.3.0

Speedup on 8 cores



MA57 vs HSL_MA97 factorize performance (8 cores)



Other tests

Comparisons to test competitiveness of HSL_MA97
(not fully comprehensive study).

PARDISO (Intel MKL 10.3) (Schenk).

- Lacks some features included in Version 4.0.0 (and later) available for fee from Uni. Basel (eg bit-compatibility).
- By default, uses iterative refinement.

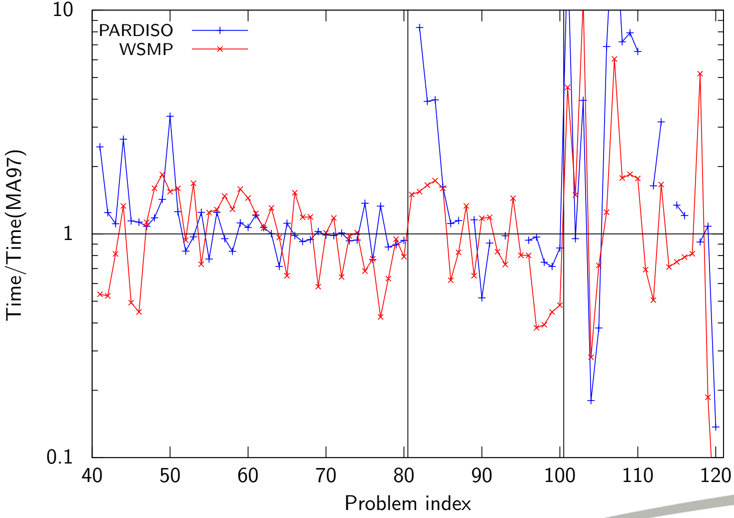
WSMP v11.5.20 (Gupta, IBM).

- Bit-compatible only if same number of threads used.

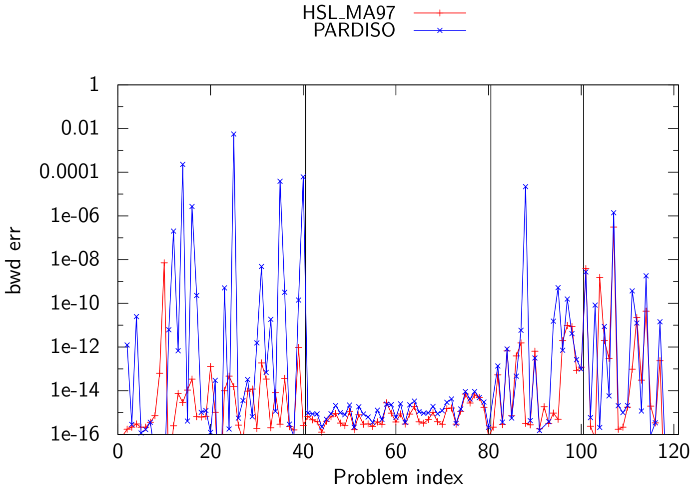
We allow up to 5 steps of iterative refinement.

Comparison with WSMP and PARDISO

Note: points above the line indicate better performance by `HSL_MA97`.



Comparison of scaled backward errors



These are without external iterative refinement but include iterative refinement within PARSISO.

Comparison with WSMP and PARDISO

After iterative refinement, the number of problems that failed to achieve an accurate solution were:

Solver	Failed
HSL_MA97	0
PARDISO	20
WSMP	2

Concluding remarks

- Efficient solution of sparse symmetric linear systems is long-standing challenge.
- Multicore machines: new challenges, need to redesign solvers.
- `HSL_MA97`: new general-purpose parallel multifrontal sparse direct solver for symmetric (indefinite) systems.
- Important features include:
  - bit-compatibility
  - use of sophisticated dense factorization kernels
- Resulting code is robust and efficient when applied to tough indefinite systems.

HSL_MA97 is available as part of the HSL mathematical software library (free to academics).

Please go to www.hsl.rl.ac.uk

Thank you!