Science & Technology Facilities Council
Rutherford Appleton Laboratory

The out-of-core challenge for large-scale problems

Jennifer A. Scott (j.a.scott@rl.ac.uk)

Joint work with John Reid

Sparse systems

Problem: we wish to solve

$$\mathbf{Ax} = \mathbf{b}$$

where $\mathbf{A}$ is

LARGE

s p a r s e

- Problem sizes constantly grow larger
- 40 years ago large might have meant order 10^2
- Today order $> 10^7$ not unusual
- For direct methods storage requirements generally grow more rapidly than problem size

Options for large problems

Possibilities:

- Iterative method ... but preconditioner?
- Combine iterative and direct methods?
- Buy a bigger machine ... but **expensive** and **inflexible**
- Parallel direct solver?
- Use an out-of-core solver

Options for large problems

Possibilities:

- Iterative method ... but preconditioner?
- Combine iterative and direct methods?
- Buy a bigger machine ... but **expensive** and **inflexible**
- Parallel direct solver?
- Use an out-of-core solver

An **out-of-core solver** holds the matrix factors in **files** and may also hold the matrix data and some work arrays in files.

Note: out-of-core working has become even more important because of more limited local memories on distributed memory machines

Out-of-core solvers

- Idea of out-of-core solvers **not** new: band and frontal solvers developed in 1970s and 1980s held matrix data and factors out-of-core.
- For example, MA32 in HSL (superseded in 1990s by MA42).
- 30 years ago John Reid at Harwell developed a Cholesky out-of-core multifrontal code TREESOLV for element applications.

Out-of-core solvers

- Idea of out-of-core solvers **not** new: band and frontal solvers developed in 1970s and 1980s held matrix data and factors out-of-core.
- For example, `MA32` in HSL (superseded in 1990s by `MA42`).
- 30 years ago John Reid at Harwell developed a Cholesky out-of-core multifrontal code `TREESOLV` for element applications.

More recent codes include:

- `BCSEXT-LIB` (Boeing)
- `Oblio` (Dobrian and Pothen)
- `TAUCS` (Toledo and students)
- `MUMPS` currently developing out-of-core version
- Also work by Rothberg and Schreiber

HSL_MA77

Our new out-of-core solver is HSL_MA77

- HSL_MA77 is designed to solve **LARGE** sparse symmetric systems
- First release for **positive definite** problems (Cholesky $\mathbf{A} = \mathbf{L}\mathbf{L}^{\mathbf{T}}$); coming VERY soon is version for **symmetric indefinite problems** ($\mathbf{A} = \mathbf{L}\mathbf{D}\mathbf{L}^{\mathbf{T}}$)
- Matrix $\mathbf{A}$ may be either in assembled form or a sum of element matrices

$$\mathbf{A} = \sum_{\mathbf{k=1}}^{\mathbf{m}} \mathbf{A}^{(\mathbf{k})}$$

  where $\mathbf{A}^{(k)}$ has nonzeros in a small number of rows and columns and corresponds to the matrix from element k.

- Matrix data, matrix factor, and the main work space held in **files**

Aim today: to provide brief introduction to HSL_MA77 and to present some numerical results .... hope you will go away wanting to try the code

Basic multifrontal algorithm

HSL_MA77 implements a **multifrontal algorithm**

Assume that $\mathbf{A}$ is a sum of element matrices.

Basic **multifrontal** algorithm may be described as follows:

Given a pivot sequence:

do for each pivot

assemble all elements that contain the pivot into a dense matrix;

eliminate the pivot and any other variables that are found only here;

treat the reduced matrix as a new **generated** element

end do

Language

HSL is a **Fortran** library

`HSL_MA77` written in **Fortran 95**, **PLUS** we use allocatable structure components and dummy arguments (part of Fortran 2003, implemented by current compilers).

Advantages of using allocatables:

- more efficient than using pointers
  - pointers must allow for the array being associated with an array section (eg `a(i,:)`) that is not a contiguous part of its parent
  - optimization of a loop involving a pointer may be inhibited by the possibility that its target is also accessed in another way in the loop
- avoids the memory-leakage dangers of pointers

Language (continued)

Other features of F95 that are important in design of HSL_MA77:

- **Automatic and allocatable arrays** significantly reduce complexity of code and user interface, (especially in indefinite case)
- We selectively use **long** (64-bit) integers (`selected_int_kind(18)`)
- Multifrontal algorithm can be naturally formulated using **recursive procedures**

```
 ....
      call factor (root)
 ....
      recursive subroutine factor (node)
! Loop over children over node
      do i = 1,number_children
         call factor (child(i))
      end do
! Assemble frontal matrix and partially factorize
 ....
      end subroutine factor
```

Input/Output in HSL_MA77

For HSL_MA77 to perform well, the I/O **must** be efficient. I/O involves:

- writing the original real and integer data
- analyse phase (integer data only)
  - reading data for input matrix
  - writing data at each node of the assembly tree
  - reading data at each node
  - writing reordered data ready for factorization
- factorization phase
  - reading integer data at each node of the tree
  - reading real data for each leaf node
  - writing columns of L as they are computed
  - writing Schur complements to stack
  - reading matrix from stack
- solve phase
  - reading integer/ real factor data once for forward sub. and once for back sub.

Input/Output in Fortran

In Fortran 77/90/95 - direct access I/O is entirely **record based**

- Fine if every read/write is of the same amount of data
- **But** we need to read/write different numbers of reals and integers at each stage of the computation
- Note: we do not want to be restricted to only accessing the data in the same order as it was written so sequential access (which is less efficient) not an option

Input/Output in Fortran

In Fortran 77/90/95 - direct access I/O is entirely **record based**

- Fine if every read/write is of the same amount of data
- **But** we need to read/write different numbers of reals and integers at each stage of the computation
- Note: we do not want to be restricted to only accessing the data in the same order as it was written so sequential access (which is less efficient) not an option

We have got around these limitations while adhering to the strict Fortran standard by writing our own **virtual memory management system**

Virtual memory management

We have a separate HSL package `HSL_OF01` that handles all i/o

- `HSL_OF01` also written in Fortran 95
- It provides read/write facilities for one or more direct access files through a single **in-core buffer** (work array)
- The buffer is divided into fixed length pages
- The page length is the same as the record length in the file(s)
- Our handling of the buffer aims to avoid actual input-output operations whenever possible

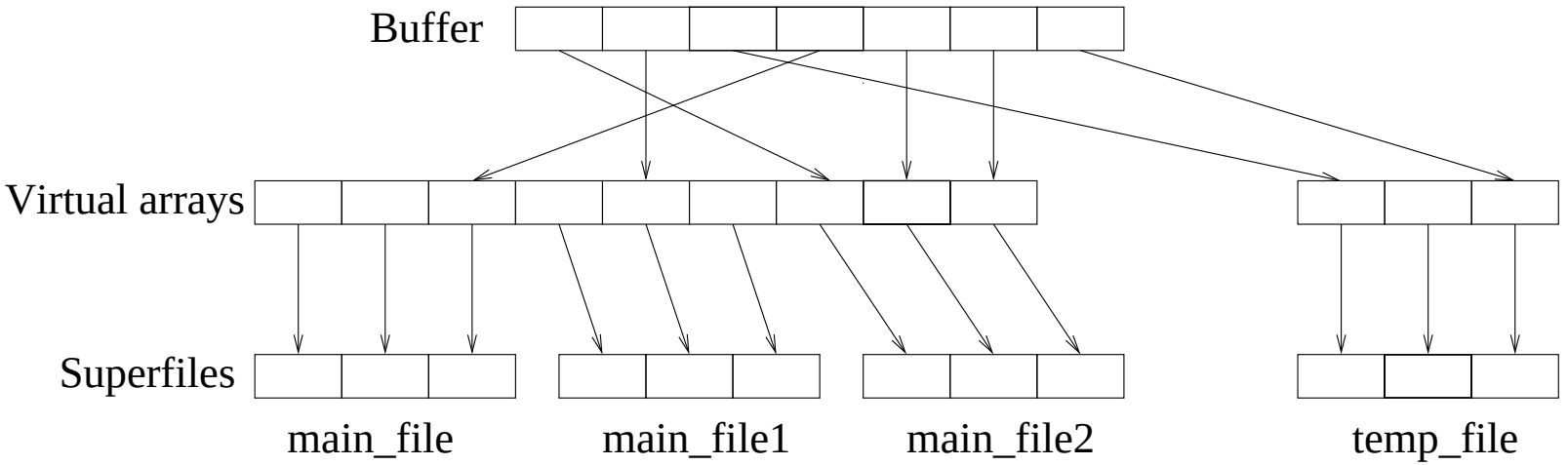
Virtual memory management

Each set of data (such as the reals in the matrix and its factor) is accessed as a **virtual array** i.e. as if it were a very long array

- Long integers are used for addresses in the virtual array
- Most active pages of the virtual array are held in the buffer
- Any contiguous section of the virtual array (of any length) may be read or written
- Each virtual array is associated with a **primary file**
- For very large problems, the virtual array may be too large for a single file. In this case, one or more **secondary files** are used

The primary and secondary files are **direct access files**.

Virtual memory management



- In this example, two superfiles associated with the in-core buffer
- First superfile has two secondaries, the second has none

Use of the buffer

- Buffer divided into fixed length **pages** (user chooses number/length)
- Most recently accessed pages of the virtual array are held in the buffer
- For each page in the buffer, we store:
  - **unit number** of its primary file
  - **page number** within corresponding virtual array
- Required page(s) found using simple hash function

Use of the buffer to minimise i/o

Aim to **minimise** number of i/o operations by:

- Using wanted pages that are already in buffer first
- If buffer full, free the least recently accessed page
- Only write page to file if it has changed since entry into buffer
- Optional flag to indicate if transferred data unlikely to be needed again before other data in the buffer (eg writing out factor data)
- When reading, optional flag to indicate data will not be read again (eg reading stack data)

Advantages

Advantages of this approach for developing sparse solvers:

- All i/o is **isolated**... assists with code design, development, debugging, and maintenance
- User is shielded from i/o but can control where files are written and can save data for future solves
- Possible for the primary and secondary files to reside on different devices
- Actual i/o is **not** needed if user has supplied long buffer
- `HSL_OF01` can be used in development of other solvers

Use of HSL_OF01 within HSL_MA77

- HSL_MA77 has one **integer** buffer and one **real** buffer
- The integer buffer is associated with a file that holds the integer data for the matrix $\mathbf{A}$ and the matrix factor
- The real buffer is associated with two files:
  - one holds the real data for the matrix $\mathbf{A}$ and the matrix factor
  - the other is used for the multifrontal stack (work space)
- The indefinite case uses a further real file to hold delayed pivots
- The user supplies pathnames together with names for the primary files
- HSL_OF01 options used to minimise i/o (eg. when reading from multifrontal stack, flag set to indicate not required again and when writing factor data, flag set to indicate data not required soon)

Use of HSL_OF01 within HSL_MA77

- HSL_MA77 has one **integer** buffer and one **real** buffer
- The integer buffer is associated with a file that holds the integer data for the matrix $\mathbf{A}$ and the matrix factor
- The real buffer is associated with two files:
  - one holds the real data for the matrix $\mathbf{A}$ and the matrix factor
  - the other is used for the multifrontal stack (work space)
- The indefinite case uses a further real file to hold delayed pivots
- The user supplies pathnames together with names for the primary files
- HSL_OF01 options used to minimise i/o (eg. when reading from multifrontal stack, flag set to indicate not required again and when writing factor data, flag set to indicate data not required soon)

NOTE: HSL_MA77 includes option for the files to be replaced by **in-core arrays** (faster for problems for which user has enough memory) . A combination of files and arrays may be used.

Stream I/O

Fortran 2003 includes **stream i/o** (sometimes called binary i/o)

- Allows a stream of bytes to be read/written
- File is opened with `ACCESS = "STREAM"` specified
- Addresses are specified by bytes, rather than by records
- Modelled on binary stream file in C

Stream I/O

Advantages of stream i/o for `HSL_OF01`

- Code is significantly simplified
- Buffers no longer needed
- Reduces input parameters (page length/number of pages no needed)

Stream I/O

Advantages of stream i/o for `HSL_OF01`

- Code is significantly simplified
- Buffers no longer needed
- Reduces input parameters (page length/number of pages no needed)

Disadvantages

- Not part of Fortran 95
- Not yet offered by all compilers (existing extensions that offered stream i/o are not necessarily portable)
- To include in HSL we will need both Fortran 95 and 2003 versions

Stream I/O

Advantages of stream i/o for `HSL_OF01`

- Code is significantly simplified
- Buffers no longer needed
- Reduces input parameters (page length/number of pages no needed)

Disadvantages

- Not part of Fortran 95
- Not yet offered by all compilers (existing extensions that offered stream i/o are not necessarily portable)
- To include in HSL we will need both Fortran 95 and 2003 versions

What about performance?

Numerical experiments

- We have performed a limited number of experiments
- All times are wall clock times in **seconds**
- First compare `HSL_OF01` with using stream I/O
- Times are for complete `HSL_MA77` solution

Numerical experiments

	HSL_OF01	Stream I/O
m_t1	19.4	23.3
shipsec1	24.1	28.3
troll	35.8	48.0
inline_1	121.5	189.1

Numerical experiments

	HSL_OF01	Stream I/O
m_t1	19.4	23.3
shipsec1	24.1	28.3
troll	35.8	48.0
inline_1	121.5	189.1

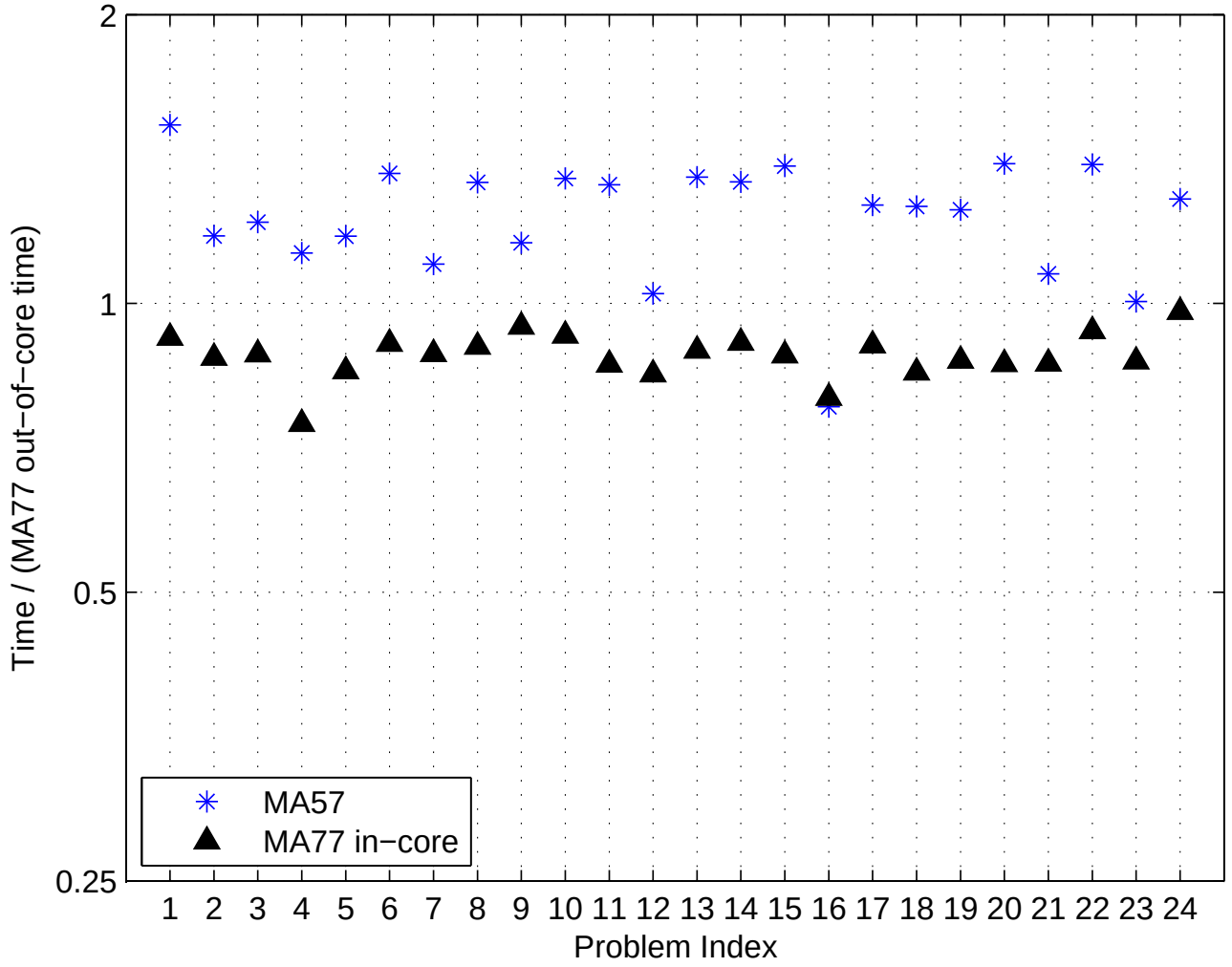
- Timings are effected by what else is happening on machine
- These timings were for lightly loaded machine
- If two problems run together, stream i/o seems more sensitive (sometimes more than double)

Conclude: at present, not planning to use stream i/o

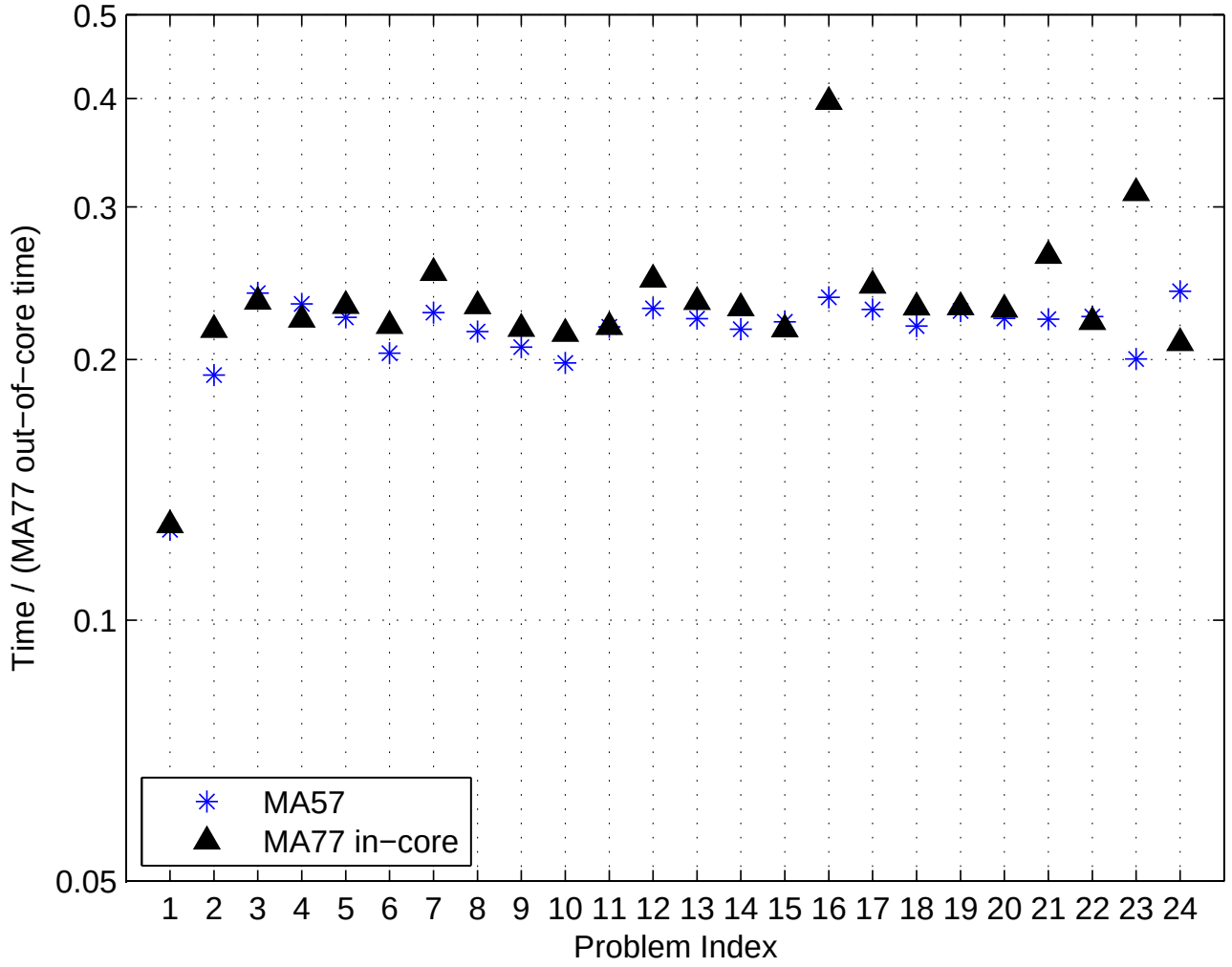
Comparisons with MA57

- Test set of 24 problems of order up to $1.5 * 10^6$ from a range of applications
- All available in University of Florida Sparse Matrix Collection
- Tests used double precision (64-bit) reals on a Dell Precision 670 with 4 Gbytes of RAM
- f95 compiler with the -O3 option and ATLAS BLAS and LAPACK
- Comparisons with flagship HSL solver MA57 (Duff)
  - Multifrontal solver (replaced earlier package MA27)
  - Primarily designed for indefinite problems (option to switch off numerical pivoting)

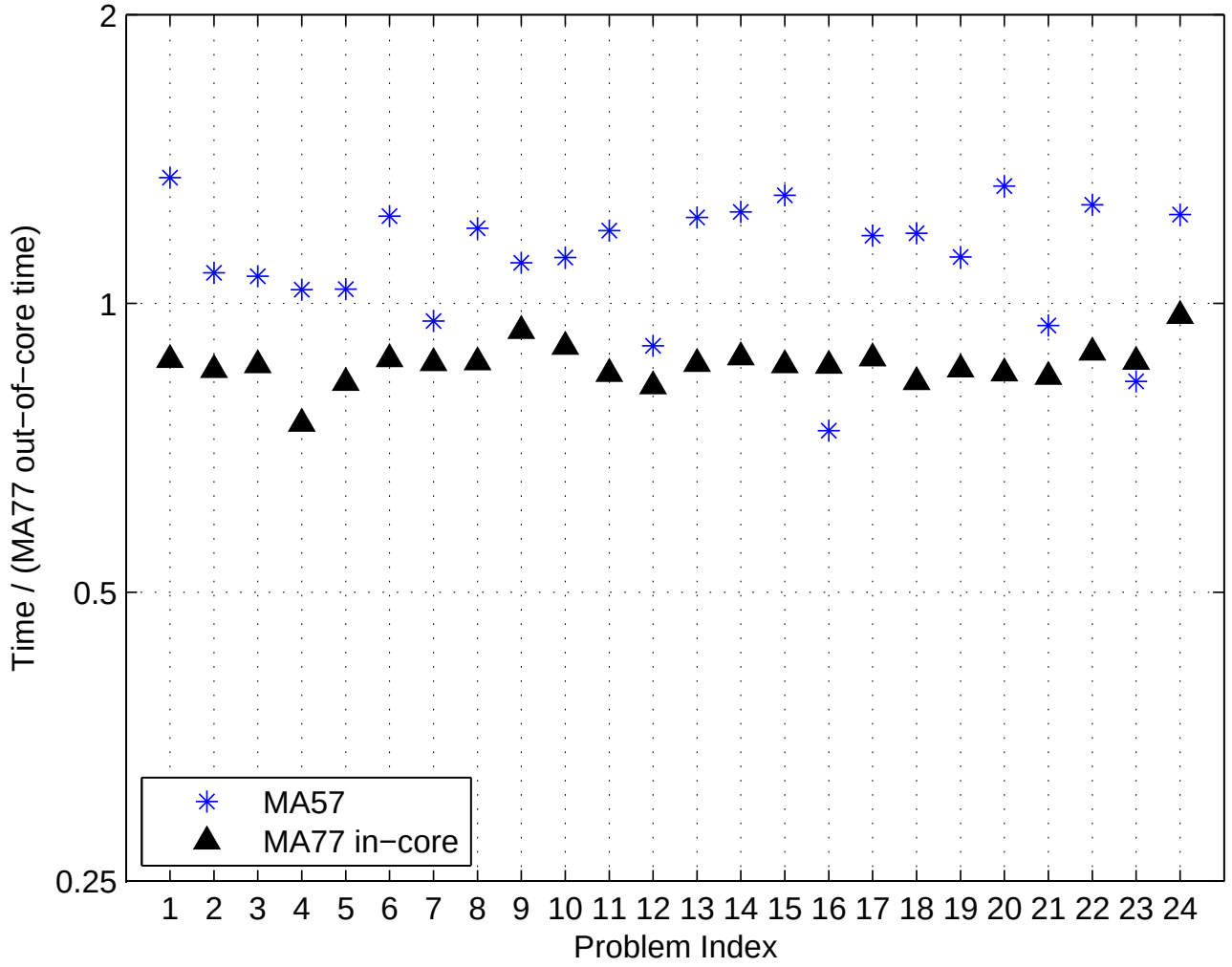
Factorization time compared with MA57



Solve time compared with MA57



Total time compared with MA57



Times (in seconds) for larger problems

Phase	inline_1 ($n = 503,712$)	bones10 ($n = 914,898$)	nd24k ($n = 72,000$)	bone010 ($n = 986,703$)
Input	4.87	6.25	2.86	8.00
Ordering	14.2	22.8	16.4	34.7
MA77_analyse	4.20	6.70	22.1	26.7
MA77_factor(0)	90.6	174.6	1284	1491
MA77_factor(1)	93.0	190.2	1243	1861
MA77_solve(1)	5.30	12.0	13.4	294
MA77_solve(8)	10.6	20.3	16.5	309
MA77_solve(64)	60.5	121	90.2	497

MA57 not able to solve these on our test computer (insufficient memory).

Mflop rates for larger problems

Phase	`inline_1` ($n = 503,712$)	`bones10` ($n = 914,898$)	`nd24k` ($n = 72,000$)	`bone010` ($n = 986,703$)
`MA77_factor(0)`	1600	1615	1917	2632
`MA77_factor(1)`	1523	1478	1948	2355
`MA77_solve(1)`	130	120	123	18
`MA77_solve(8)`	650	574	778	140
`MA77_solve(64)`	948	760	1488	701

Note: these are using wall clock times

Unsymmetric element problems

- Recently developed out-of-core multifrontal code for unsymmetric element problems. Code is called HSL_MA78
- Based on the design of HSL_MA77
- Again uses HSL_OF01 to handle out-of-core
- Separate package HSL_MA74 written to compute the partial factorization of the dense unsymmetric frontal matrices
  - Implements a block factorization ... employs level 3 BLAS
  - Incorporates threshold pivoting (options for partial, diagonal or rook pivoting)
  - Also option for static pivoting (prevents delayed pivots but may produce inaccurate factorization)
- HSL_MA78 solves $AX = B$ or $A^T X = B$

Comparison with frontal solver

HSL_MA42_ELEMENT is an unsymmetric out-of-core (uni-)frontal code

	n	Time (secs)		Factors ($*10^6$)	
		MA42_ELEMENT	MA78	MA42_ELEMENT	MA78
crplat2	18010	1.85	1.84	4.35	2.94
ship_001	34920	10.5	13.4	15.5	15.6
m_t1	97578	552	101	135.5	56.2
shipsec8	114919	950	101	196.0	55.6
troll	213453	3042	74	672.0	63.7

These results illustrate the benefits of the multifrontal algorithm.

Appeal: We need large test problems in element form from real applications.

Concluding remarks

- Writing the solver has been (and still is) a major project
- Positive definite and unsymmetric elements codes performing well
- Out-of-core working adds an overhead but not prohibitive (exception is solve phase)
- Indefinite kernel almost done (separate HSL package)
- Version for complex arithmetic will be developed

Concluding remarks

- Writing the solver has been (and still is) a major project
- Positive definite and unsymmetric elements codes performing well
- Out-of-core working adds an overhead but not prohibitive (exception is solve phase)
- Indefinite kernel almost done (separate HSL package)
- Version for complex arithmetic will be developed

THANK YOU!