Technical Report

A fast and robust mixed precision solver for the solution of sparse symmetric linear systems

J. D. Hogg and J. A. Scott

September 3, 2008

RAL-TR-2008-023

ISSN 1358-6254

RAL-TR-2008-023

A fast and robust mixed precision solver for the solution of sparse symmetric linear systems

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

ABSTRACT

The main bottleneck for emerging computing architectures is memory bandwidth. The amount of data moved around within a sparse direct solver can be approximately halved by using single precision arithmetic. However, the cost of this is a potential loss of accuracy in the solution of the linear systems. Double precision iterative methods preconditioned by a single precision factorization can enable the recovery of high precision solutions more quickly than a sparse direct solver run using double precision arithmetic. The gains from the reduced memory bandwidth are expected to be particularly prominent on multicore machines where the ratio of computational power to memory bandwidth is higher.

In this paper, we develop a practical algorithm to apply such a mixed precision approach and suggest parameters and techniques to minimize the number of solves required by the iterative recovery process. These experiments provide the basis for our new code `HSL_MA79` — a fast, robust, mixed precision sparse symmetric solver that will be included in the mathematical software library HSL.

Numerical results for a wide range of problems from practical applications are presented.

Keywords: Gaussian elimination, sparse symmetric linear systems, iterative refinement, FGMRES, multifrontal method, mixed precision, Fortran 95.

AMS(MOS) subject classifications: 65F05, 65F10, 65F50.

---

[1] Computational Science and Engineering Department, Rutherford Appleton Laboratory, Chilton, Oxfordshire, OX11 0QX, England, UK.
Email: jonathan.hogg@stfc.ac.uk & jennifer.scott@stfc.ac.uk
Work supported by EPSRC grants EP/F006535/1 and EP/E053551/1.
Current reports available from `http://www.numerical.rl.ac.uk/reports/reports.html`.

September 3, 2008

1 Introduction

A common task in scientific software packages is solving linear systems

$$A\boldsymbol{x} = \boldsymbol{b} \tag{1.1}$$

where A is a large sparse symmetric matrix and $\boldsymbol{b}$ is the known right hand side. There are two common approaches to the solution of such systems:

Direct Methods: these are generally variants of Gaussian elimination, involving a factorization $PAP^T \rightarrow LDL^T$ of the system matrix A, where L is unit lower triangular, D is block diagonal (with 1×1 and 2×2 blocks), and P is permutation matrix. The solution process is completed by performing forward and then backward substitutions (that is, by first solving a lower triangular system and then an upper triangular system). Direct methods are popular because, when properly implemented, they are generally robust and achieve a high level of accuracy, making them suitable for use as general-purpose black-box solvers for a wide range of problems. The main limitation of direct methods is that the memory required normally increases rapidly with problem size.

Iterative Methods: these involve some iterative scheme and are often based on using Krylov subspaces of A. In general, their performance is dependent upon the availability of an appropriate preconditioner. For large-scale problems, a carefully chosen and tuned preconditioned iterative method will often run significantly faster than a direct solver and will require far less memory; indeed for very large problems, an iterative method is often the only available choice. Unfortunately, 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.

In this paper, we are concerned with using a direct method to obtain an approximate solution to (1.1) and then applying an iterative method to refine the solution. In other words, we use the direct factorization as a preconditioner for an iterative solver. All our results are obtained using multifrontal solvers, but much of our work will apply equally to left- and right-looking supernodal solvers. Modern direct solvers for sparse matrices make extensive use of Level 3 BLAS kernels [12] in their aim to achieve close to dense performance on modern cache-based architectures (see, for example, [20] for details of currently available sparse symmetric direct solvers). As such, they are limited by the rate of transfer of data for their floating point operands through and between main memory and the cache hierarchy. With the recent emergence of multicore processors and with future chips likely to have ever larger numbers of cores, this data transfer rate is expected to become an ever tighter constraint. Approximately half the storage required by a double precision factorization is needed when using single precision arithmetic (by which we mean working with 32-bit floating-point numbers), potentially allowing larger problems to be solved, and the movement of data is much reduced. Moreover, on a number of architectures, such as Cell processors and general-purpose computing on graphics cards (GPGPU), single precision arithmetic is currently more highly optimised (and hence faster) than double precision computation. Buttari, Dongarra, and Kurzak [6] 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. Single precision is potentially particularly advantageous for an out-of-core direct solver (that is, a solver that stores the factors and possibly some of its work arrays in files) because the amount of disk access is also approximately halved.

In general, direct solvers use double precision arithmetic, although some packages (including those from the HSL mathematical software library [22]) also offer a single precision version. The accuracy required when solving the system (1.1) is application dependent. If the required accuracy is less than $\mathcal{O}(10^{-5})$ (which may be all that is appropriate if, for example, the problem data is not known to high accuracy), single precision may generally be used. However, users frequently request greater accuracy or, if the problem is ill-conditioned, higher precision may be necessary to ensure a solution with the sought-after accuracy. In such cases, the double precision version of the solver has traditionally been used. Motivated by the advantages of using single precision on modern architectures, recent studies [4, 7, 8] have shown that it may be possible to use a matrix factorization computed using the single precision version of a direct solver as a preconditioner for a simple iterative method that is used to regain higher precision.

Our aim is to design and develop a mixed precision sparse solver for the solution of symmetric (possibly indefinite) linear systems and to demonstrate its performance in terms of efficiency and robustness. Section 2 describes the algorithms used in our mixed precision solver and establishes default values for the parameters involved, extending both the work of Buttari et al [7] and the preliminary MATLAB experimental results of Arioli and Duff [4]. Section 3 describes our Fortran 95 implementation of the mixed precision solver as the new package `HSL_MA79`. `HSL_MA79` is built upon the HSL multifrontal solvers `MA57` [13] and `HSL_MA77` [25] and is intended for inclusion within HSL. Numerical results for `HSL_MA79` are given in Section 4 and our conclusions are presented in Section 5.

All reported experiments are performed on a Dell Precision T5400 with two Intel E5420 quad core processors running at 2.5GHz backed by 8GB of RAM. In all our tests, we use the Goto BLAS [19] and the gfortran-4.3 compiler with -O1 optimization. All timings are elapsed times in seconds. We work with two test sets; all but three of the problems are drawn from the University of Florida Sparse Matrix Collection [10] and all are symmetric with either real or integer valued entries.

Test Set 1. 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 2. 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 efficient to perform a lot of tests on the smaller test problems. Furthermore, `MA57` is not able to solve the largest problems in Test Set 2 (because of insufficient memory), while `HSL_MA77` is an out-of-core solver that is specifically designed for solving large problems. 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) [21, 26][1]. The FGMRES iterative solver employed in our experiments uses the HSL implementation `MI15`.

2 Algorithm

As we have already observed, use of single precision arithmetic reduces the movement of data between memory hierarchies and cache and the processing units, speeding up core operations. This is illustrated in Figure 2.1. Here we show the performance of the Level 3 BLAS kernel for matrix-matrix multiplication in single precision (`SGEMM`) and in double precision (`DGEMM`) for square matrices of order up to 1000. Since `GEMM` is used extensively within direct solvers, this demonstrates the potential advantage of performing the factorization using single precision. Figure 2.2 shows how this translates into a performance gain for the factorization phase of the sparse symmetric solver `MA57` (problems from Test Set 1 that take at least 0.01 seconds to factorize on our test machine). While we do not see gains of quite a factor of two from the matrix-matrix multiply, 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 pursuing a mixed precision approach.

In our early runs of `MA57`, floating point underflows caused single precision to underperform double, because of the way modern Intel CPUs handle such events. This problem was reduced by setting a processor flag to flush denormals to zero, avoiding a cascade of slow operations, though there is still a potential speed drop if a significant number occur (indeed a similar situation can occur in double precision

[1]The direct solvers were unable to solve problem `GHS_indef/boyd1` with this choice of scaling so, in this case, we scale using `MC64` [13, 14].

Figure 2.1: The performance of single (SGEMM) and double (DGEMM) precision matrix-matrix multiplication for a range of sizes of square matrices. These experiments used the Goto BLAS [19].

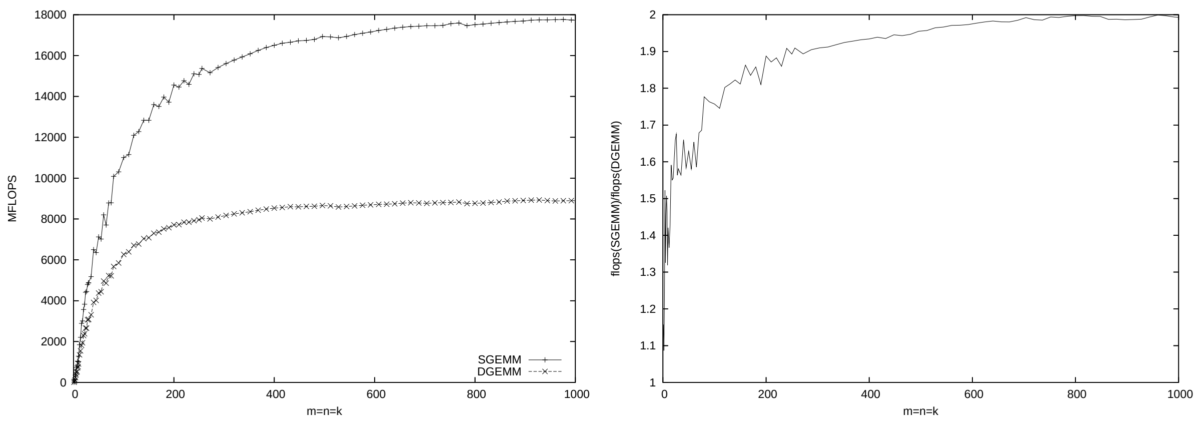
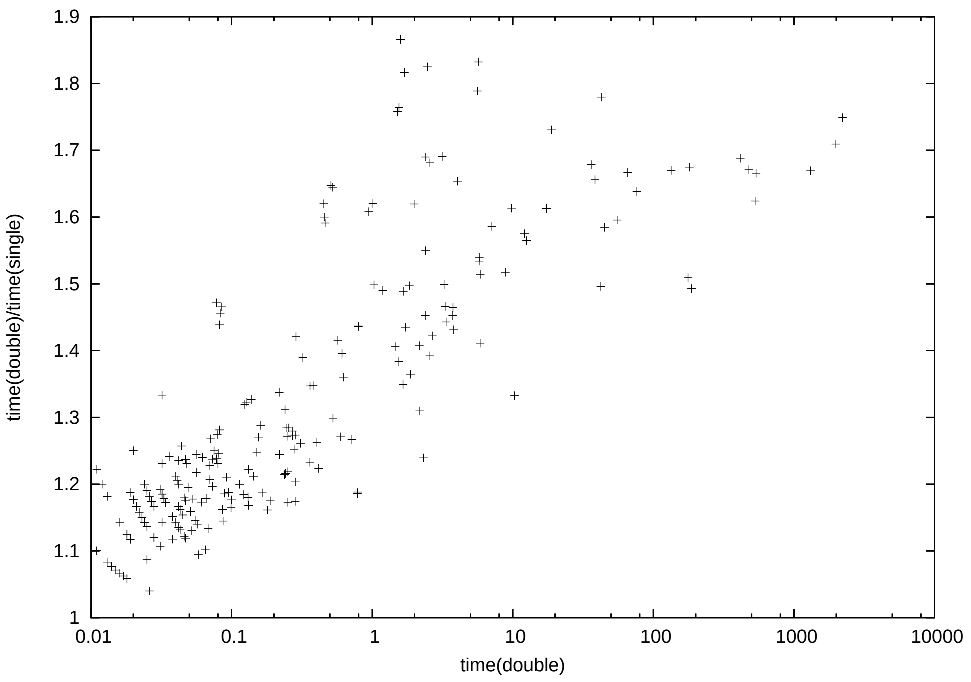


Figure 2.2: A comparison of the times required by the factorization phase of MA57 when run in single and double precision (Test Set 1).



but is less common due to a much larger exponent range). There is also one problem (not shown) that has a ratio of well over two. This was an indefinite problem and, in double precision, the pivot sequence chosen by the analyse phase of `MA57` had to be modified more than in the single precision case, resulting in a higher flop count and denser factors.

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, do we want to resort to applying more computationally expensive alternatives. In the worst case, we fall back to performing a double precision factorization.

Setting $\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}$, we define the norm of the scaled residual (the backward error) to be

$$\beta = \frac{\|\boldsymbol{r}\|_\infty}{\|A\|_\infty \|\boldsymbol{x}\|_\infty + \|\boldsymbol{b}\|_\infty}. \tag{2.1}$$

The computed solution is assumed to be of the required accuracy when $\beta \leq \gamma$, where γ is a parameter chosen by the user.

Algorithm 1 outlines our basic mixed precision approach. The factorization is performed in single precision then, if $\beta > \gamma$, iterative refinement in double precision is performed. If the required 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 restarted. In the next two sections, we discuss the iterative refinement and FGMRES steps.

Algorithm 1 Mixed precision solver

```
Input: Desired 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 $\beta \leq \gamma$ then exit
  Perform iterative refinement (Algorithm 2)
  if $\beta \leq \gamma$ then exit
  Perform FGMRES (Algorithm 3)
  if $\beta \leq \gamma$ then exit
  if $prec = single$ then
    Set $prec = double$ and cycle
  else
    Set error flag and exit
  end if
end loop
```

2.1 Iterative Refinement

Iterative refinement is a simple first order method used to improve a computed solution of (1.1); it is outlined in Algorithm 2. Here β_k is the norm of the scaled residual (2.1) on the kth iteration. 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 [29] 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 paper) 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. [7, 8].

Algorithm 2 Mixed precision iterative refinement

```
Input: Single precision factors of $A$, desired accuracy $\gamma$, minimum reduction $\delta$ and
    maximum number of iterations 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 $k = 1$.
for $k = 1$,i_maxitr do
  if $\beta_k \leq \gamma$ exit
  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 $\boldsymbol{r}_{k+1} \geq 2\boldsymbol{r}_k$ then Set error flag and exit (stagnation)
  Set $k = k + 1$
end for
$\boldsymbol{x} = \boldsymbol{x}_k$
```

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 [4]), a large number of iterations are needed before any substantial reduction in the residual is achieved. To detect 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 desired accuracy γ. The number of additional iterations can be reduced or eliminated by choosing a small δ. For different values of δ, Table 2.1 reports the number of problems belonging to Test Set 1 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 $\delta = 0.5$.

Table 2.1: The number of problems in Test Set 1 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`$= 10$).

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

Our implementation of iterative refinement also offers an option to terminate once a chosen maximum 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 residual is $\beta = 10^{-7}$ and the default improvement parameter $\delta = 0.3$ is used. If stagnation has not occurred, after 13 iterations the 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 required accuracy was achieved or stagnation was recognised before this limit was reached).

2.2 Preconditioned FGMRES

For our examination of FGMRES, we use the 62 problems from Test Set 1 that failed to achieve the requested accuracy using iterative refinement with any δ. We call this the Reduced Test Set 1.

Algorithm 3 Mixed Precision FGMRES right preconditioned by a direct solver with adaptive restarting

```
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 [28])
    $\boldsymbol{y}_k = \arg\min_{\boldsymbol{y}} \| \|\boldsymbol{r}\|\boldsymbol{e}_1 - H_k\boldsymbol{y}_k\|$ (Minimize the 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
```

In Algorithm 1, FGMRES [27] refers to a right preconditioned variant of FGMRES. Arioli, Duff, Gratton, and Pralet [5] have shown that, in cases where iterative refinement fails, FGMRES will often succeed and is more robust than either iterative refinement or GMRES. Arioli and Duff [4] prove a convergence result for the use of a mixed precision FGMRES algorithm to recover double precision accuracy following a single precision factorization. This motivates our use of FGMRES.

Our variant of FGMRES, shown as Algorithm 3, is essentially that given in [4] but additionally uses an adaptive restart parameter. Here $\boldsymbol{e}_1$ denotes the first column of the identity matrix. Algorithm 3 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 residual 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 residual (it may increase), we restore the solution from the previous outer iteration before restarting. In our experiments, we compared adaptive restarting with using 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 desired 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.

Following Arioli and Duff, we consider converting the matrix factor that has been computed in single precision into double precision, allowing us to perform the forward and backward substitutions in either single or double precision. Performing the solves in single precision has obvious speed advantages per iteration but 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. Table 2.2 attempts to capture to what extent 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,

$$\frac{1}{|\mathcal{P}|}\sum_{i\in\mathcal{P}}\left(\mathrm{Solves}_i(\mathit{single}) - \mathrm{Solves}_i(\mathit{double})\right), \tag{2.2}$$

  where $\mathrm{Solves}_i(\mathit{double})$ ($\mathrm{Solves}_i(\mathit{single})$) 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 required accuracy.
- The geometric mean of the ratio of the number of solves in single precision to the number in double precision, that is,

$$\left(\prod_{i\in\mathcal{P}}\frac{\mathrm{Solves}_i(\mathit{single})}{\mathrm{Solves}_i(\mathit{double})}\right)^{\frac{1}{|\mathcal{P}|}}. \tag{2.3}$$

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

Table 2.2: 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 1 ($\delta = 0.3$, `f_maxitr`=128).

`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 (2.2))	14.9	15.6	16.2	12.9	12.9
Average ratio of number of solves (see (2.3))	2.8	2.5	3.1	2.1	2.1

The reduction in the number of solves when using double precision has to be set against the slow down experienced because of increased data movement through the memory hierarchy. 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 then double precision (as is the case in the last line of Table 2.2) is faster, in addition to allowing us to solve more problems. Because of this, we shall use double precision solves for FGMRES in the remainder of this paper. Note that iterative refinement will still use single precision solves as similar experiments showed there to be limited 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 1.

In Table 2.3, 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 1 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` $= 4$.

Table 2.3: The number of solves performed within FGMRES for Reduced Test Set 1 using a range of values for `restart` following unsuccessful iterative refinement. Results are shown for $\delta = 0.3$ and `f_maxitr`$= 64$. The 23 problems that failed for all values of `restart` are not shown.

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

3 Implementing the mixed precision strategy

In this section, we discuss the design and development of our new mixed precision solver `HSL_MA79`. The code is written in Fortran 95 and, at its heart, uses the HSL direct solvers `MA57` [13] and `HSL_MA77` [25]. We start by giving a brief overview of these solvers, highlighting some of their key features that are important for `HSL_MA79`.

3.1 `MA57`

`MA57` is designed to solve sparse symmetric linear systems (1.1); the system matrix may be either positive definite or indefinite. The multifrontal method is used [17]. A detailed discussion of the design of `MA57` and its performance is given by Duff [13]. Relevant work on the pivoting and scaling strategies available within `MA57` is given by Duff and Pralet [15, 16].

In common with other HSL solvers, `MA57` is available in both double and single precision versions. It offers a range of options, including solving for multiple right-hand sides, computing partial solutions, error analysis, a matrix modification facility, and a stop and restart facility. Although the default settings for the control parameters should work well in general, there are several parameters available to enable the user to tune the code for his or her problem class or computer architecture.

Like many modern symmetric direct solvers, `MA57` has three distinct phases: an analyse phase that works only with the sparsity pattern to set up data structures for the factorization, the numerical factorization phase that uses these data structures to compute the matrix factor, and a solve phase that may be called any number of times after the factorization is complete to solve repeatedly for different right-hand sides.

Built-in scaling is available through a symmetrized version of the well-known package `MC64` [14]. The system matrix is explicitly scaled internally to the package, as are the right-hand sides and the solution, so that the user need not be concerned with this.

The efficiency of a direct method, in terms of both the storage needed and the work performed, is dependent on the order in which the elimination operations are performed, that is, the order in which the pivots are selected. For symmetric matrices that are positive definite, the pivotal sequence chosen using the sparsity pattern of A alone can be used during the factorization without modification. For symmetric indefinite problems, a tentative pivot sequence is chosen based upon the sparsity pattern (treating zeros on the diagonal as entries) and this is modified if necessary (possibly to include 2×2 pivots) during the factorization to maintain numerical stability. `MA57` offers the user a number of ordering options, including variants of the minimum degree algorithm [1, 2] and multilevel nested dissection through an interface to the well-known MeTiS package [23]. Based on the study by Duff and Scott [18], by default, `MA57` chooses between MeTiS and approximate minimum degree (avoiding problems with dense rows [3, 11]).

3.2 `HSL_MA77`

`HSL_MA77` [25] is also a multifrontal solver that is designed to solve positive definite and indefinite sparse symmetric systems. It too has separate analyse, factorize and solve phases. The fundamental difference between `MA57` and `HSL_MA77` is that the latter is an out-of-core solver, that is, it is designed to allow the matrix data, the computed factors and some of the intermediate work arrays to be held in files. The advantage of this is that it enables much larger problems to be solved.

Storing data in files potentially adds a significant overhead to the time required to factor and then solve the linear system. To minimise this overhead, Reid and Scott [24] have written a set of Fortran subroutines that manage a virtual memory system so that actual input/output occurs only when really necessary. These routines are available within HSL as the package `HSL_OF01` and are used by `HSL_MA77` to efficiently handle all input and output. Results reported by Reid and Scott [25] illustrate the effectiveness of `HSL_OF01` in minimizing the out-of-core overhead; they also present results for problems that were too large for `MA57` to solve on their test machine. On problems that `MA57` is able to solve, the performance of HSL_MA77 is favourable: on some problems it is faster than MA57, while on others the converse is true. The main exception is the solve time. The factor data has to be read from file twice: once for the forward substitution and once for the back substitution. This is independent of the number of right-hand sides. Thus, for a single right-hand side (or small number of right-hand sides), the solve phase of HSL_MA77 can be significantly more expensive than the corresponding phase of MA57, although for a large number of right-hand sides there is a smaller relative difference.

As with MA57, HSL_MA77 offers a range of options. These include allowing the files to be replaced by arrays (so that, if there is sufficient space, the data is all stored in main memory). The user can specify the initial sizes of these arrays and an overall limit on their total size. If an array is found to be too small, the code will continue using a combination of files and arrays. Another important option allows the user to specify whether he or she is running the code on a 32-bit or 64-bit architecture. On a 64-bit machine, it is possible to run problems with much larger frontal matrices (on a 32-bit machine, the maximum front size is limited to approximately 16,000).

We remark that HSL_MA77 requires the user to input the pivot order. This can be computed by calling MeTiS or the HSL package MC47 that implements an approximate minimum degree algorithm; alternatively, the user may choose to call the analyse phase of MA57 and allow it to select the pivot sequence. Each of these alternatives computes a pivot order of 1×1 pivots; 2×2 pivots may be selected during the factorization. In some cases, it may be advantageous to specify a tentative pivot sequence that includes 2×2 pivots; HSL_MA77 allows the user to do this. A pivot order that contains 2×2 pivots may be found using the package HSL_MC68, and this remains an area of active research.

3.3 Design of HSL_MA79

HSL_MA79 is designed to provide a robust and efficient implementation of Algorithm 1 for both positive definite and indefinite systems, using existing HSL packages as its main building blocks. In particular, it uses the direct solvers MA57 and HSL_MA77 and the implementation of FMGRES offered by MI15, together with the scaling packages MC30, MC64, and MC77. HSL_MA79 includes a range of options but it is not our intention to incorporate all possibilities available within the direct solvers MA57 and HSL_MA77. Instead, we have designed a general purpose package that is straightforward to use and, by limiting the number of parameters that have to be set, we do not require the user to have a detailed knowledge and understanding of all the different components of the algorithms used.

The following procedures are available to the user:

- MA79_factor_solve accepts the matrix A, the right-hand sides $\boldsymbol{b}$, and the required accuracy. Based on the matrix, it selects whether to use MA57 or HSL_MA77; by default, the single precision is selected as the initial precision. The code then implements Algorithm 1. The matrix factorization is retained for further solves.

- MA79_refactor_solve uses the information returned from a previous 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 the computed factors generated by MA79_factor_solve (or 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). A warning is issued if the computed factors are unable to achieve the requested accuracy.

- 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.

Derived types are used to pass data between the different routines. In particular, `MA79_control` has components that control the action within the package and `MA79_info` has components that return information from subroutine calls. The control components are given default values when a variable of type `MA79_control` is declared and may be altered thereafter. Full details are provided in the user documentation.

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

3.3.1 `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. The user is also required to state whether the matrix is known to be positive definite (if it is, it is generally advantageous to take advantage of this by switching off numerical pivoting). `HSL_MA79` first checks the user-supplied data for errors (any out-of-range matrix entries are removed and duplicates are summed). The user may supply a pivot order; otherwise, the analyse phase of `MA57` is called to compute the pivot order. Statistics on the forthcoming factorization (such as the maximum frontsize, the number of flops, and the number of entries in the factor) are 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. `MA57` is selected unless one or more of the following holds:

(i) It is not possible to allocate the arrays required by `MA57`. (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`)

(ii) The matrix is positive definite with a maximum frontsize greater than 1500.

(iii) The matrix is not positive definite and the user-supplied pivot sequence includes 2×2 pivots.

(iv) The user has chosen `HSL_MA77`.

The reason for (ii) is that tests have shown that `HSL_MA77` generally outperforms `MA57` under these conditions (see [25]). If the user has flagged the problem as positive definite, any supplied 2×2 pivots are replaced by two consecutive 1×1 pivots. 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 (which may be significant if the problem is not very large). Furthermore, if `MA57` is chosen, the analyse phase has already been performed, whereas for `HSL_MA77`, `ma77_analyse` still has to be called. The process of refactorizing in double precision is also more expensive for `HSL_MA77` because it is necessary to reload the matrix data and repeat its analysis phase (this can be avoided for `MA57`). We remark that, if (iii) or (iv) applies, the analyse phase of `MA57` is not called; instead, the code proceeds directly to calling `MA77_analyse`.

`HSL_MA79` offers the user the option of choosing the direct solver. In particular, the results of Reid and Scott [25] show that `HSL_MA77` can be significantly faster than `MA57` on some problems and the user may have previous experience that indicates for his or her application `HSL_MA77` outperforms `MA57`. Should the user request `MA57` but the problem is found to be too large, a warning flag is issued and the computation proceeds using `HSL_MA77`. At the start of the factorization with `MA57`, `HSL_MA79` allocates the required arrays based on the analyse statistics (allowing 10 per cent additional space in the indefinite case to accommodate pivots that may be delayed by numerical pivoting). If during the factorization, `MA57` requires larger work arrays because of delayed pivots, `HSL_MA79` uses the stop and restart facility offered by `MA57` to allocate larger arrays (the array size is doubled) and to restart the factorization from the failure point. If there is insufficient memory to allocate these arrays, `HSL_MA79` switches to `HSL_MA77`. 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 1 and its development through Section 2. 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 [9]), `MC64` (symmetrized scaling based on maximum matching by Duff and Koster [14, 15]), and `MC77` using the 1 or ∞ norms (iterative process of simultaneous norm equilibration [26]). The default is the ∞ norm equilibration scaling from `MC77` because recent tests [21] on a large number of problems from practical applications have shown that, in general, this provides a good fast scaling.

`HSL_MA79` offers complete control of the parameters in Algorithms 2 and 3, in addition to the ability to disable any particular method of recovering precision in Algorithm 1 (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.

An important feature of `MA79_factor_solve` is that it returns (using the derived type `MA79_info`) 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, to allow the user to examine the fine detail of the factorization, the full information type or array from the factorization code (`MA57B` or `MA77_factor`) is returned to the user as a component of the derived type `MA79_info`.

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.

3.3.2 `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` can take advantage of the experience gained on the initial factorization. In particular, information on the size of the computed factor and work arrays used by `MA57` can be used in allocating arrays for subsequent factorizations so that, unless the new numerical values lead to significantly more delayed pivots, `MA57` will not need to stop and restart because of the arrays not being of adequate size. If the initial factorization found it was necessary (because of memory limitations) to switch from using `MA57` to the out-of-core code `HSL_MA77`, on a second or subsequent factorization it will be more efficient to go straight for using `HSL_MA77`. Similarly, if the requested accuracy was only achieved by using a double precision factorization, if the user still wants the same accuracy, it will generally not be worthwhile to attempt the computation in mixed precision. Thus, the aim of `MA79_refactor_solve` is to use the experience from the preceding call to `MA79_factor_solve` to more efficiently factorize and solve subsequent systems with an unchanged sparsity pattern. Although by default `MA79_refactor_solve` selects the precision for the factorization, an optional argument allows the user to specify the precision.

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. Note that, if necessary, existing sparse matrix manipulation packages within HSL can be employed by the user to assist in putting the data into the required format. The only other parameters the user must set are the required accuracy and the right-hand side vectors.

`MA79_refactor_solve` avoids recalling the analyse phase of `MA57` (or `HSL_MA77`) and uses the pivot sequence that was computed by (or input to) MA79_factor_solve. If the user wishes to experiment with a different pivot sequence, MA79_factor_solve must be called. The user may, however, reset the block size used by the dense kernels and the pivoting threshold u, as well as the parameters used by iterative refinement and FGMRES.

3.3.3 MA79_solve

After a call to MA79_factor_solve (or to MA79_refactor_solve), MA79_solve may be called to solve for additional right-hand sides. If MA57 has been used to compute the factorization (or HSL_MA77 was run in-core), the cost of each additional solve is generally much less than the factorization time but, if HSL_MA77 was run with the factors held on disk, the solve time can be significant (see [25]). If the solve is performed at the same time as the factorization, the factors can be used to perform the forward substitution as they are generated, cutting the input and output (and hence the time) for the solve approximately in half.

There is, of course, no guarantee that the requested accuracy will be achieved for one or more of the user's right-hand sides. In this case, MA79_solve will return a warning together with the computed solution and information on size of the corresponding scaled residual. In particular, if the factorization has been performed in single precision, the required accuracy may not be achieved without resorting to a double precision factorization, which will not occur automatically with MA79_solve. If the user wants to switch to double precision after a warning from MA79_solve, he or she should call MA79_refactor_solve, explicitly specifying the factorization is to be performed in double precision.

Between calls to MA79_solve, the user may change any of the parameters relating to the iterative methods, such as the minimum improvement parameter (δ) and the parameters f_maxitr and i_maxitr that control the maximum number of iterations.

4 Numerical Results

In this section, we present results obtained using a pre-release version 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 and test examples described in Section 1. 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 Section 2 are used to control the solution recovery).

Figures 4.3 and 4.4 compare the performance of mixed precision and double precision for Test Sets 1 and 2, respectively, with MA57 selected as the direct solver within HSL_MA79 for Test Set 1 and HSL_MA77 for Test Set 2. We note that, if the mixed precision approach is used, 31 problems in Test Set 1 and 27 problems in Test Set 2 fail to achieve the required accuracy (that is, mixed precision fails for about 10 per cent of our test problems). From Figure 4.3, 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 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 1, mixed precision is more than 1.5 times faster than double precision. For the problems in Test Set 2 with the out-of-core solver HSL_MA77, 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 tolerance 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 4.4, we report the number and percentage of problems in each test set for which the required accuracy was achieved after iterative refinement and after iterative refinement followed by FGMRES. We also report the number of problems that had to switch to a double precision factorization to achieve the required accuracy and the number that failed to achieve this even in double precision. There were just two

Figure 4.3: Ratio of times to solve (1.1) with HSL_MA79 in mixed precision and double precision modes on Test Set 1 using MA57 as the solver, with accuracy $\gamma = 5 \times 10^{-15}$.

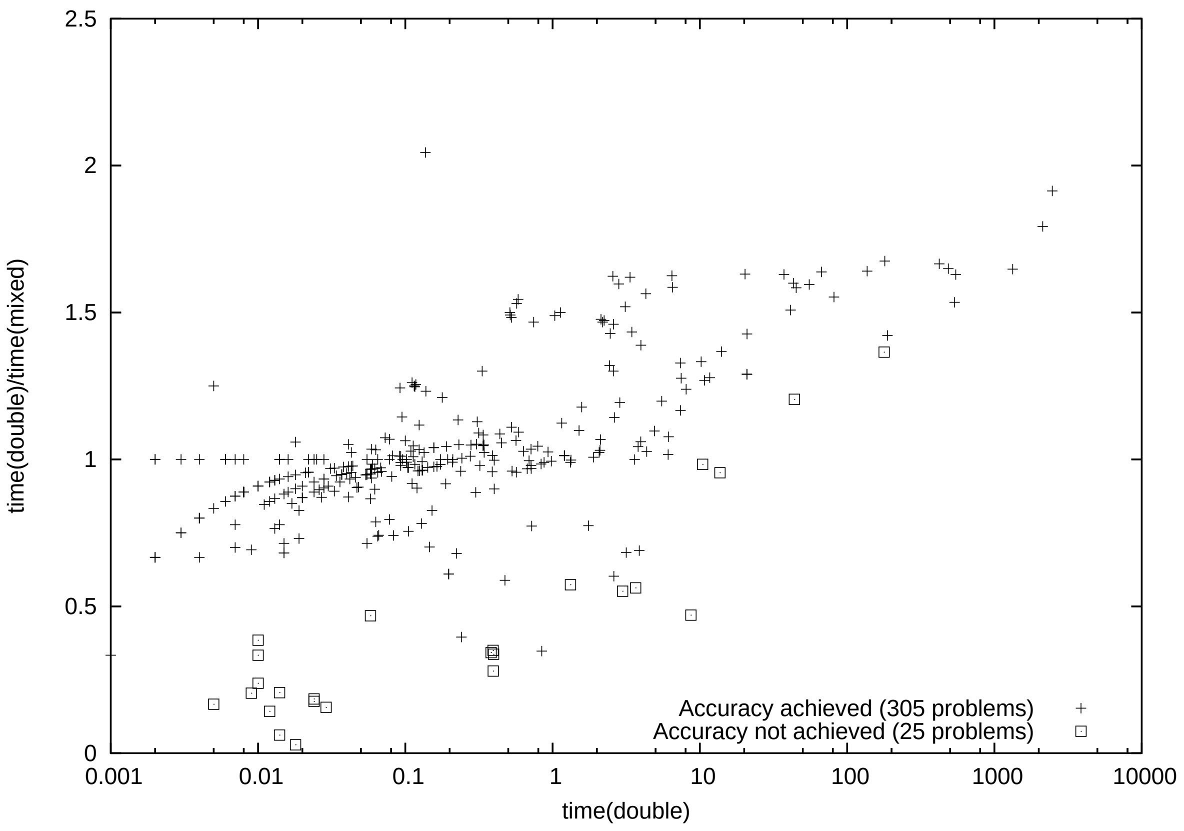
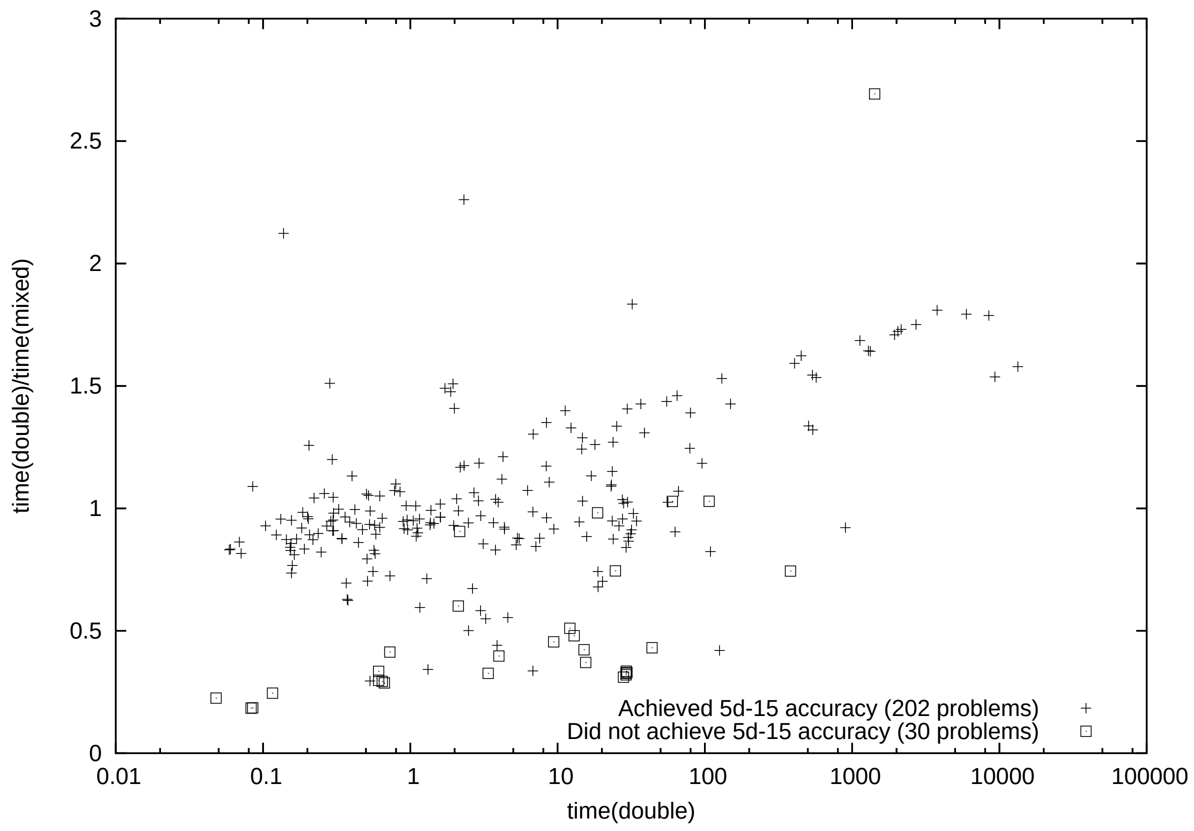


Figure 4.4: Ratio of times to solve (1.1) with `HSL_MA79` in mixed precision and double precision modes on test set 2 using `HSL_MA77` as the solver, with accuracy $\gamma = 5 \times 10^{-15}$.



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).

Table 4.4: Number of problems that exited at each stage of Algorithm 1 implemented as `HSL_MA79`.

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

As expected, when mixed precision fails to reach the desired 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 (1.1), but also the times for each phase of the solution process in mixed precision and in double precision. Table 4.5 reports timings for the various phases for a subset of problems of different sizes from Test Set 2. The problems are ordered by the total time required to solve (1.1) using double precision. The time for the analyse phase (which here includes the time to scale the matrix) is independent of the precision. The "Single Solve" time is the time needed to perform forward and back substitution (for a single right hand side) in single precision (for the mixed precision case) or double precision (for the double precision case). It is worth noting that, once a single solve has been performed, future solves are often faster because of caching by the operating system. The advantage of mixed precision on larger problems is clearly because such problems have a large ratio for the factorize to solve time.

Table 4.5: Times to solve (1.1) for a subset of problems from Test Set 2 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.

	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

5 Conclusions and future directions

In this paper, 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 [4] and Buttari et al [7], we have designed and developed a practical and robust sparse mixed precision solver; the new package `HSL_MA79` will be made available within the HSL Library. Numerical experiments on a large number of problems has 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 switch to double precision and is tuned to minimize the work performed before the switch is made. 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).

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 also plan 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 paper, 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 desired accuracy γ, the theory [4] 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 will be the subject of a separate study.

6 Code availability

All the codes discussed in this paper have been developed for inclusion in the mathematical software library HSL. All use of HSL requires a licence. Individual HSL packages (together with their dependencies and accompanying documentation) are available without charge to individual academic users for their personal (non-commercial) research and for teaching; licences for other uses involve a fee. Details of the packages and how to obtain a licence plus conditions of use are available at `www.cse.clrc.ac.uk/nag/hsl/`.

7 Acknowledgements

We are grateful to our colleagues John Reid, Iain Duff and Mario Arioli for useful discussions relating to this work.

References

[1] P. Amestoy, T. Davis, and I. Duff, *An approximate minimum degree ordering algorithm*, SIAM J. Matrix Analysis and Applications, 17 (1996), pp. 886–905.

[2] ——, *Algorithm 837: AMD, an approximate minimum degree ordering algorithm*, ACM Transactions On Mathematical Software, 30 (2004), pp. 381–388.

[3] P. Amestoy, H. Dollar, J. Reid, and J. Scott, *An approximate minimum degree algorithm for matrices with dense rows*, Technical Report RAL-TR-2007-020, Rutherford Appleton Laboratory, 2007.

[4] M. Arioli and I. Duff, *FGMRES to obtain backward stability in mixed precision*, Technical Report RAL-TR-2008-006, Rutherford Appleton Laboratory, 2008.

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

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

[7] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov, *Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy*, ACM Transactions On Mathematical Software, (2008). to appear.

[8] A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek, and J. Kurzak, *Mixed precision iterative refinement techniques for the solution of dense linear systems*, International J. High Performance Computing Applications, 21 (2007), pp. 457–466.

[9] A. Curtis and J. Reid, *On the automatic scaling of matrices for Gaussian elimination*, IMA J. Applied Mathematics, 10 (1972), pp. 118–124.

[10] T. Davis, *The University of Florida sparse matrix collection*, Technical Report, University of Florida, 2007. http://www.cise.ufl.edu/∼davis/techreports/matrices.pdf.

[11] H. Dollar and J. Scott, *A note on fast approximate minimum degree orderings for matrices with some dense rows*, Technical Report RAL-TR-2007-022, Rutherford Appleton Laboratory, 2007.

[12] J. Dongarra, J. D. Croz, S. Hammarling, and I. Duff, *A set of level 3 basic linear algebra subprograms*, ACM Transactions On Mathematical Software, 16 (1990), pp. 1–17.

[13] I. Duff, *MA57– a new code for the solution of sparse symmetric definite and indefinite systems*, ACM Transactions On Mathematical Software, 30 (2004), pp. 118–154.

[14] I. Duff and J. Koster, *On algorithms for permuting large entries to the diagonal of a sparse matrix*, SIAM J. Matrix Analysis and Applications, 22 (2001), pp. 973–996.

[15] I. Duff and S. Pralet, *Strategies for scaling and pivoting for sparse symmetric indefinite problems*, SIAM J. Matrix Analysis and Applications, 27 (2005), pp. 313–340.

[16] ———, *Towards a stable static pivoting strategy for the sequential and parallel solution of sparse symmetric indefinite systems*, SIAM J. Matrix Analysis and Applications, 29 (2007), pp. 1007–1024.

[17] I. Duff and J. Reid, *The multifrontal solution of indefinite sparse symmetric linear systems*, ACM Transactions On Mathematical Software, 9 (1983), pp. 302–325.

[18] I. Duff and J. Scott, *Towards an automatic ordering for a symmetric sparse direct solver*, Technical Report RAL-TR-2006-001, Rutherford Appleton Laboratory, 2005.

[19] K. Goto and R. van de Geijn, *High performance implementation of the level-3 BLAS*, ACM Transactions On Mathematical Software, 34 (2008). To appear.

[20] N. Gould, J. Scott, and Y. Hu, *A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations*, ACM Transactions On Mathematical Software, 33 (2007).

[21] J. D. Hogg and J. A. Scott, *The effects of scalings on the performance of a sparse symmetric indefinite solver*, Technical Report RAL-TR-2008-007, Rutherford Appleton Laboratory, 2008.

[22] HSL, *A collection of Fortran codes for large-scale scientific computation*, 2007. See http://www.cse.clrc.ac.uk/nag/hsl/.

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

[24] J. Reid and J. Scott, *HSL_OF01, a virtual memory system in Fortran*, Tech. Rep. RAL-TR-2006-026, Rutherford Appleton Laboratory, 2006. Revised Nov. 2007. To appear in *ACM Transactions on Mathematical Software*.

[25] ———, *An out-of-core sparse Cholesky solver*, Technical Report RAL-TR-2006-013, Rutherford Appleton Laboratory, 2006. Revised Nov. 2007. Submitted to ACM Transactions on Mathematical Software.

[26] D. Ruiz, *A scaling algorithm to equilibrate both rows and columns norms in matrices*, Tech. Rep. RAL-TR-2001-034, Rutherford Appleton Laboratory, 2001.

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

[28] ———, *Iterative Methods for Sparse Linear Systems*, SIAM, 2 ed., 2003, pp. 273–275.

[29] R. Skeel, *Iterative refinement implies numerical stability for Gaussian elimination*, Mathematics of Computation, 35 (1980), pp. 817–832.