Preprint
STFC-P-2023-002

Algebraic preconditioning in low precision works

J Scott, M Tuma

May 2023

Submitted for publication in ACM Transactions on Mathematical Software

ISSN 2753-5819

STFC Author Identifiers (ORCIDs)

Author ORCIDs are provided where available.

Jennifer Scott	0000-0003-2130-1091

Algebraic preconditioning in low precision works

Jennifer Scott[*] Miroslav Tůma[†]

May 1, 2023

Abstract

The emergence of low precision floating-point arithmetic in computer hardware has led to a resurgence of interest in the use of mixed precision numerical linear algebra. For linear systems of equations, there has been renewed enthusiasm for mixed precision variants of iterative refinement, with the emphasis so far being mainly on dense systems. We consider the iterative solution of large sparse systems using algebraic preconditioners. The focus is on the robust computation of incomplete factorization preconditioners in half precision arithmetic and employing them to solve symmetric positive definite systems to higher precision accuracy; however, the proposed ideas can be applied more generally. Even for well-scaled problems, incomplete factorizations can break down because of small entries on the diagonal. When using half precision arithmetic, overflows are an additional potential source of breakdown. We examine how breakdowns can be avoided and we implement our strategies within new half precision Fortran sparse incomplete Cholesky factorization software. Results are reported for a range of problems from practical applications. These demonstrate that, even for highly ill-conditioned problems, half precision preconditioners can replace double precision preconditioners, although unsurprisingly this can be at the cost of additional iterations of a Krylov solver.

Keywords: sparse matrices, sparse linear systems, preconditioning, incomplete factorizations, half precision arithmetic, mixed precision arithmetic, iterative refinement.

[*]STFC Rutherford Appleton Laboratory, Harwell Campus, Didcot, Oxfordshire, OX11 0QX, UK and School of Mathematical, Physical and Computational Sciences, University of Reading, Reading RG6 6AQ, UK. Correspondence to: `jennifer.scott@stfc.ac.uk`.

[†]Department of Numerical Mathematics, Faculty of Mathematics and Physics, Charles University, Czech Republic. `mirektuma@karlin.mff.cuni.cz`.

1 Introduction

We are interested in solving sparse linear systems $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ is nonsingular and $x, b \in \mathbb{R}^n$. The majority of algorithms for solving such systems fall into two main categories: direct methods and iterative methods. Direct methods transform A using a finite sequence of elementary transformations into a product of simpler sparse matrices in such a way that solving linear systems of equations with the factor matrices is comparatively easy and inexpensive. For example, for a general nonsymmetric matrix, $A = PLUQ$, where L is a lower triangular matrix, U is an upper triangular matrix, and P and Q are permutation matrices chosen to preserve sparsity in the factors and ensure the factorization is stable. Direct methods, when properly implemented, are robust and can be confidently used as block-box solvers for computing solutions with predictable accuracy, which is typically double precision. However, they require significant expertise to implement efficiently (particularly in parallel). They also need large amounts of memory (which increases non linearly with the size and density of A) and the matrix factors normally contain many more non zero entries than A; these extra entries are termed the fill-in and much effort goes into trying to minimise the amount of fill-in.

By contrast, iterative methods compute a sequence of approximations $x^{(0)}, x^{(1)}, x^{(2)}, \ldots$ that (hopefully) converge to the solution in an acceptable number of iterations. The number of iterations (and whether or not convergence occurs at all) depends on $x^{(0)}$, A and b as well as the required accuracy in x. Basic implementations of iterative solvers are relatively straightforward as they only use the sparse matrix A indirectly, through matrix-vector products and, most importantly, their memory demands are limited to a (small) number of vectors of length n, making them attractive for very large problems and problems where A is not available explicitly. However, preconditioning is usually essential to enhance convergence of the iterative method. Preconditioning seeks to transform the system into one that is more tractable and from which the required solution of the original system can easily be recovered. Determining and computing effective preconditioners is highly problem dependent and generally very challenging. Algebraic preconditioners that are built using an incomplete factorization of A in which entries that would be part of a complete factorization are dropped are frequently used, especially when the underlying physics of the problem is difficult to exploit. Such preconditioners can be employed within more sophisticated methods; for example, to precondition subdomain solves in domain decomposition schemes or as smoothers in multigrid methods.

The performance differences for computing and communicating in different precision formats has led to a long history of efforts to enhance numerical algorithms by combining precision formats. The goals of mixed-precision algorithms are to accelerate the computational time by using lower-precision formats while maintaining the high accuracy of the output, and by reducing the memory requirements, extend the size of problems that can be solved. Numerical linear algebra software, and linear system solvers in particular, typically use double precision (64-bit) arithmetic, although some packages (including the BLAS and LAPACK routines and some sparse solvers, such as those in the HSL mathematical software library [24]) have always offered single precision (32-bit) versions. In the late 2000s, single precision arithmetic was more highly optimised (and hence faster) than double precision computation on what were then state-of-the-art architectures, such as Intel chips with SSE instructions and Sony/Toshiba/IBM (STI) Cell processors (see, for example, [9, 27]). This speed advantage, combined with the potential memory savings and reduction in data movement resulting from working in single precision, led to a number of studies into the feasibility of factorizing a matrix in single precision and then using the factors as a preconditioner for a simple iterative method in high precision to regain higher precision accuracy [5, 10, 11]. Hogg and Scott [23] extended this work by developing a Fortran mixed precision sparse solver for symmetric (possibly indefinite) linear systems; this code is available as `HSL_MA79` within the HSL library. It uses a single precision multifrontal method to compute the sparse factors and then either mixed precision iterative refinement or FGMRES [5, 35] is employed to achieve double precision accuracy.

In the past few years, the emergence of lower precision arithmetic in hardware has led to further interest in mixed precision algorithms. The key difference compared to earlier work is the use of half precision

(16-bit) arithmetic, motivated by NVIDIA, Google, and AMD manufacturing hardware that is capable of performing half precision arithmetic, driven primarily by gaming but becoming increasingly important for machine learning. Half precision arithmetic is at least four times faster than double precision arithmetic, and possibly much more than that on some hardware, notably on NVIDIA tensor cores. A comprehensive state-of-the-art survey of work on mixed precision numerical linear algebra routines is given in [20] (see also [1]). In particular, there have been important ideas and theory on mixed precision iterative refinement methods that employ the matrix factors computed in low precision as a preconditioner to recover higher precision accuracy [2, 12, 13]. Numerical experiments for these hybrid methods have largely focussed on demonstrating the potential to accelerate the solution of dense systems; to date, much less work has centred on the sparse case. For sparse systems, the benefits of employing single precision arithmetic in solving double precision sparse linear systems using multiple cores are evaluated by Zounon et al [38]. Amestoy et al [3] investigate the potential of mixed precision iterative refinement to enhance methods for sparse systems based on a particular class of approximate sparse factorizations. They employ the well-known parallel sparse direct solver MUMPS [4], which is able to exploit block low-rank factorizations and static pivoting to compute approximate factors. In common with all other currently available sparse direct solvers, MUMPS does not support the use of half precision arithmetic and developing an efficient half precision sparse solver would be a major undertaking, requiring 16-bit versions of the dense linear algebra routines that provide the building blocks behind sparse direct solvers. Consequently, as in [38], the reported results in [3] are restricted to combining single and double precision arithmetic. Higham and Pranesh [22] focus on symmetric positive definite linear systems. They compute a Cholesky factorization using low precision arithmetic and employ the factors as preconditioners in GMRES-based and CG-based iterative refinement. While they are interested in the sparse case, their MATLAB experiments (which simulate low precision using their `chop` function [21]) store the sparse test examples as dense matrices and their Cholesky factorizations are computed using dense routines. The reported theoretical and numerical results demonstrate the potential for low precision (complete) factors to be used to obtain high precision accuracy. Most recently, Carsen and Khan [14] have considered using sparse approximate inverse preconditioners (SPAI) that are based on Frobenius norm minimization [19]. They are interested in using low precision to compute the preconditioners and then employ GMRES-based iterative refinement.

Our emphasis is on low precision incomplete factorization preconditioners. Although our ideas can be used for general sparse linear systems, we focus on the sparse symmetric positive definite case. We use half precision arithmetic to construct incomplete Cholesky factorizations that are then employed as preconditioners to recover double precision accuracy. Our primary objective is to show that for a range of problems (some of which are highly ill-conditioned) it is possible to successfully obtain and use low precision incomplete factors. We consider the potential sources of overflow during the incomplete factorization and look at how to prevent it. Our numerical experiments use sparse matrix software that we have developed in Fortran. Half precision and double precision versions are tested on systems coming from practical applications.

This paper offers the following novel contributions:

(a) it considers the practicalities of computing incomplete factorizations using low precision arithmetic;

(b) it looks at using local and global modifications combined with scaling to prevent breakdowns during the factorization, particularly those that are a consequence of computing the factors in low precision;

(c) it develops level-based incomplete Cholesky factorization software in half precision;

(d) it demonstrates that the use of half precision incomplete factorization preconditioners is effective in practice.

The rest of the paper is organised as follows. In Section 2, we briefly recall incomplete factorizations of sparse matrices and consider the challenges that incomplete Cholesky factorizations can face when low precision arithmetic is employed. In Section 3, we summarise basic mixed precision iterative refinement algorithms. Breakdowns in incomplete factorizations in low precision and ways to avoid them are discussed in Section 4. In particular, the local and global modifications to avoid breakdowns are described. Numerical results for a range of problems coming from practical applications are presented in Section 5 and concluding remarks as well as future directions are given in Section 6.

Terminology. We use the term high precision to refer to precision formats that provide high accuracy at the cost of a larger memory volume (in terms of bits) and low precision to refer to precision formats that compose of fewer bits (smaller memory volume) and provide low(er) accuracy. Unless stated otherwise, we mean IEEE double precision (64-bit) when using the term high precision (denoted by fp64) and the 1985 IEEE standard 754 half precision (16-bit) when using the term low precision (denoted by fp16, with unit roundoff u_ℓ). bfloat16 is another form of half-precision arithmetic that was introduced by Google in its tensor processing units and formalized by Intel; we do not use it in this paper. Table 1.1 summarises the parameters for different precision arithmetic.

Table 1.1: Parameters for bfloat16, fp16, fp32, and fp64 arithmetic: the number of bits in the significand and exponent, unit roundoff u, smallest positive (subnormal) number x^s_{min} , smallest normalized positive number x_{min}, and largest finite number x_{max}, all given to three significant figures. † In Intel's bfloat16 specification, subnormal numbers are not supported.

	Signif.	Exp.	u	x^s_{min}	x_{min}	x_{max}
bfloat16	8	8	3.91×10^{-3}	†	1.18×10^{-38}	3.39×10^{38}
fp16	11	5	4.88×10^{-4}	5.96×10^{-8}	6.10×10^{-5}	6.55×10^{4}
fp32	24	8	5.96×10^{-8}	1.40×10^{-45}	1.18×10^{-38}	3.40×10^{38}
fp64	53	11	1.11×10^{-16}	4.94×10^{-324}	2.22×10^{-308}	1.80×10^{308}

2 Incomplete factorizations

In this section, we briefly recall incomplete factorizations of sparse matrices and then discuss how breakdown can occur, particularly when using low precision.

2.1 A brief introduction to incomplete factorizations

The incomplete factorizations that we are interested in are of the form $A \approx LU$, where L and U are sparse lower and upper triangular matrices, respectively (for simplicity of notation, the permutations P and Q are omitted). If A is a symmetric positive definite (SPD) matrix then $U = L^T$. There are three main classes of incomplete factorization preconditioners. Firstly, threshold-based $ILU(\tau)$ methods in which the locations of permissible fill-in in the factors are determined in conjunction with the numerical factorization of A; entries of the computed factors that are smaller than a prescribed threshold are dropped. Secondly, memory-based $ILU(m)$ methods in which the amount of memory available for the incomplete factorization is prescribed and only the largest entries are retained at each stage of the factorization. Thirdly, structure-based $ILU(\ell)$ methods in which an initial symbolic phase determines the location of permissible entries using only the sparsity pattern of A. The memory requirements for the incomplete factors are then determined before the numerical factorization is performed. The simplest such approach (for which the symbolic phase is trivial) is an $ILU(0)$ factorization (or $IC(0)$ in the SPD case) that limits entries in the incomplete factors to positions corresponding to entries in A (no fill-in is permitted). $ILU(0)$ preconditioners are frequently used for comparison purposes when assessing the performance of other approaches.

The different approaches have been developed, modified and refined over many years. Variants have been proposed that combine the ideas and/or employ them in conjunction with discarding entries in A (sparsification) before the factorization commences. For more details on possible variants we refer, for example, to [15, 36].

Algorithm 2.1 outlines a basic generic (right-looking) incomplete Cholesky (IC) factorization of a SPD matrix. The output is the so-called *square-root* form (rather than the square-root free LDLT form). Later, we will see that in low precision we need to use this form. The algorithm assumes a target sparsity pattern $\mathcal{S}\{L\}$ for L is provided, where

$$\mathcal{S}\{L\} = \{(i,j) \mid l_{ij} \neq 0,\ 1 \leq j \leq i \leq n\}.$$

Modifications can be made to incorporate threshold dropping strategies and to determine $\mathcal{S}\{L\}$ as the method proceeds. At each major step, a right-looking factorization algorithm applies the outer product updates to the part of the matrix that has not yet been factored as they are generated (Steps 7–11). In left-looking variants, the updates are not applied immediately; instead, all updates from previous columns are applied together to the current column before it is factorized.

Algorithm 2.1. Basic right-looking IC factorization
Input: *SPD matrix A and a target sparsity pattern $\mathcal{S}\{L\}$*
Output: *Incomplete Cholesky factorization $A \approx LL^T$.*

```
1: Initialize $l_{ij} = a_{ij}$ for all $(i,j) \in \mathcal{S}\{L\}$
2: for $k = 1 : n$ do                                          ▷ Start of k-th major step
3:     $l_{kk} \leftarrow (l_{kk})^{1/2}$                                  ▷ Diagonal entry is the pivot
4:     for $i = k+1 : n$ such that $(i,j) \in \mathcal{S}\{L\}$ do
5:         $l_{ik} \leftarrow l_{ik}/l_{kk}$       ▷ Scale pivot column k of the incomplete factor by the pivot
6:     end for
7:     for $j = k+1 : n$ such that $(j,k) \in \mathcal{S}\{L\}$ do
8:         for $i = j : n$ such that $(i,j) \in \mathcal{S}\{L\}$ do
9:             $l_{ij} \leftarrow l_{ij} - l_{ik}l_{jk}$                              ▷ Update operation
10:        end for
11:    end for
12: end for                                              ▷ column k of L has been computed
```

2.2 Challenges for incomplete factorizations in low precision

For arbitrary choices of the sparsity pattern $\mathcal{S}\{L\}$, the incomplete Cholesky factorization exists if A is an M-matrix or a H-matrix with positive diagonal entries [32, 31]. But for a general SPD matrix, there is no such guarantee and an incomplete factorization algorithm can (and frequently does) break down.

When using fp16 arithmetic, there are three places where breakdown can potentially occur. We refer to these as problems B1, B2, and B3 as follows.

- B1: The computed diagonal entry l_{kk} (which is termed the pivot at step k) may be unacceptably small or negative.
- B2: The scaling $l_{ik} \leftarrow l_{ik}/l_{kk}$ may overflow.
- B3: The update $l_{ij} \leftarrow l_{ij} - l_{ik}l_{jk}$ may overflow.

B1 breakdown is not necessarily related to the use of low precision arithmetic but can occur when using higher precision arithmetic. Prescaling A and factorizing $S_1^{-1}AS_2^{-1}$, where S_1 and S_2 are diagonal scaling matrices, is typically built into single and double precision sparse direct solvers and is often applied by default. In the symmetric case, symmetry is preserved by choosing $S_1 = S_2 = S$. Because no single choice of scaling always results in the best performance of a sparse factorization algorithm (in terms of the time, factor sizes, memory requirements and data movement), a number of different possibilities (with different associated costs) are generally offered so that a user can experiment and select the best for their application. For incomplete factorizations, scaling is also important and it can reduce the incidence of B1 breakdowns. This is illustrated for SPD matrices in [37], where it was reported that the cheap scaling in which the entries in column j of A are normalised by the 2-norm of column j is generally a good choice. For nonsymmetric matrices, it may be beneficial to permute large entries on to the diagonal before the factorization begins. However, scaling alone cannot guarantee to prevent B1 breakdowns. If breakdown does happen then modifications need to be made to the scaled matrix that is being factorized, either before or during the factorization; this is discussed in Section 4.

Observe that the occurrence of underflows when using fp16 arithmetic does not prevent the computation of the incomplete factors, although underflows could potentially lead to a loss of information that affects the quality of the preconditioner. However, provided the problem has been well scaled, the dropping strategy used within the incomplete factorization has more influence on the computed factors than underflows do. A subnormal floating-point number is a nonzero number with magnitude less than the absolute value of the smallest normalized number. Floating-point operations on subnormals can be very slow, because they often require extra clock cycles, which introduces a high overhead. If an off-diagonal factor entry is subnormal, it can again be replaced by zero without significantly affecting the preconditioner quality.

2.3 Using the low precision factors

Each application of an incomplete LU factorization preconditioner is equivalent to solving a system $LUv = w$. This involves a solve with the lower triangular L factor followed by a solve with the upper triangular U factor; these are referred to as forward and back substitutions, respectively. Algorithm 2.2 outlines a simple lower triangular solve; it can be modified for sparse w.

Algorithm 2.2. Forward substitution: lower triangular solve $Ly = w$
Input: *Lower triangular matrix L with nonzero diagonal entries and right-hand side w.*
Output: *The dense solution vector y.*

```
1: Initialise $y_j = w_j$, $1 \le j \le n$
2: for $j = 1 : n$ do
3:     $y_j \leftarrow y_j / l_{jj}$
4:     for $i = j + 1 : n$ do
5:         if $l_{ij} \neq 0$ then
6:             $y_i \leftarrow y_i - l_{ij} y_j$
7:         end if
8:     end for
9: end for
```

If the factors are computed and stored in half precision arithmetic u_ℓ and the forward and back substitutions applied in precision $u = u_\ell$ then overflows can occur at Steps 3 and 6. We can try and avoid this using simple scaling of the right hand side so that we solve $LUv = w/\|w\|_\infty$ and then set $y = v \times \|w\|_\infty$ (see [13]). Nevertheless, as in problems B2 and B3 above, overflows can still happen. The safe computations that we introduce in Section 4 can reduce (but not eliminate) the incidence of such overflows; alternatively, higher precision can be used for the triangular solves.

3 LU and Cholesky factorization based iterative refinement

Iterative refinement seeks to improve the accuracy of a computed solution $\tilde{x}$ by iteratively repeating the following steps until either the required accuracy is achieved or a prescribed limit on the number of iterations is reached.

1. Compute the residual $r = b - A\tilde{x}$.

2. Solve the correction equation $Ad = r$.

3. Update the computed solution $\tilde{x} \leftarrow \tilde{x} + d$.

A number of variants exist. The most common is LU-IR, which computes the LU factors of A in low precision u_ℓ, and then solves the correction equation by forward and back substitution using the computed LU factors in precision u_ℓ (traditionally, this was single precision but the interest now is also in half precision). The computation of the residual is performed in precision u_r and the update is performed in the working precision u ($u_r \le u \le u_\ell$). This is outlined in Algorithm 3.1.

Algorithm 3.1. LU-based iterative refinement in three precisions (LU-IR)
Input: *Non singular matrix A and vector b, three precisions satisfying $u_r \le u \le u_\ell$*
Output: *Computed solution x of the system $Ax = b$*

```
1: Compute the factorization $A = LU$ in precision $u_\ell$
2: Initialize $x_1$ (e.g., by solving $LUx_1 = b$ using substitution in precision $u_\ell$)
3: for  $i = 1$ : itmax or until converged do               ▷ itmax is the maximum iteration count
4:     Compute $r_i = b - Ax_i$ in precision $u_r$
5:     Solve $LUd_i = r_i$ by $d_i = U^{-1}L^{-1}r_i$ by substitution in precision $u_\ell$
6:     Compute $x_{i+1} \leftarrow x_i + d_i$ in precision $u$
7: end for
```

Algorithm 3.2. Krylov-based iterative refinement in five precisions (Krylov-IR)
Input: *Non singular matrix A and vector b, a Krylov subspace method, and five precisions u_r, u_g, u_p, u and u_ℓ*
Output: *Computed solution x of the system $Ax = b$*

```
1: Compute the factorization $A = LU$ in precision $u_\ell$
2: Solve $LUx_1 = b$ by substitution in precision $u_\ell$
3: for  $i = 1$ : itmax or until converged do               ▷ itmax is the maximum iteration count
4:     Compute $r_i = b - Ax_i$ in precision $u_r$
5:     Solve $U^{-1}L^{-1}Ad_i = U^{-1}L^{-1}r_i$ by the Krylov method in precision $u_g$, with $U^{-1}L^{-1}A$ performed in
   precision $u_p$
6:     Compute $x_{i+1} \leftarrow x_i + d_i$ in precision $u$
7: end for
```

Using fp16 arithmetic to accelerate the LU factorization restricts the ability of LU-IR to solve moderately ill-conditioned problems. To extend the range of problems that can be tackled, Carson and Higham [12] propose a variant that uses GMRES preconditioned by the LU factors to solve the correction equation. This is outlined in Algorithm 3.2 with the Krylov subspace method set to GMRES. Carson and Higham use two precisions $u = u_\ell$ and $u_r = u_g = u_p = u^2$; this was later extended to allow up to five precisions [2, 13]. If the LU factorization is performed in fp16 arithmetic, then LU-IR is only guaranteed to reduce the solution error if the condition number $\kappa(A)$ satisfies $\kappa(A) \ll 2 \times 10^3$, whereas if u_g and u_p correspond to fp64 then it is reduced by GMRES-IR provided $\kappa(A) \ll 3 \times 10^7$. Note that Algorithm 3.2 requires two convergence tests and stopping criteria; firstly, for the Krylov method on Step 5 and secondly, for testing the updated solution.

If A is SPD then a Cholesky factorization replaces the LU factorization. A potential problem is that definiteness can be lost when the matrix is rounded to fp16 precision and the Cholesky factorization can suffer break down. To avoid this, Higham and Pranesh [22] prescale and shift the matrix and, for fp16 arithmetic, they additionally seek to minimize the chance of underflow and of subnormal numbers by multiplying the matrix entries by a scalar to bring them close to the overflow level. The absolute values of the entries of the scaled and shifted matrix $A^{(\ell)}$ are bounded by θx_{max}, where $0 < \theta \le 1$ is a chosen parameter. If the factorization breaks down, the shift is doubled, $A^{(\ell)}$ is recomputed, and the factorization restarted. This doubling of the shift is a standard remedy for the breakdown of incomplete Cholesky factorizations (in double precision arithmetic) [28, 37]. Note that because the growth factor for Cholesky factorization is 1, overflows do not occur during the low precision factorization of $A^{(\ell)}$. This is in contrast to the overflow that can happen in an incomplete factorization (problems B2 and B3).

In the SPD case, a natural choice is to choose the Krylov method in Algorithm 3.2 to be the conjugate gradient (CG) method. The supporting rounding error analysis applies only to GMRES, because it relies on the backward stability of GMRES and preconditioned CG is not guaranteed to be backward stable [18]. This is also the case for MINRES. However, the numerical results reported in [22] suggest that in practice CG-IR generally works as well as GMRES-IR.

An obvious way to generalise GMRES-IR is to replace $U^{-1}L^{-1}$ with a preconditioner M^{-1}. Some work on this has recently been reported by Lindquist et al. [29, 30], but only for dense problems, combining single and double precision arithmetic. In addition, Amestoy et al. [3] use the option within the MUMPS solver to compute sparse factors in single precision using block low-rank factorizations and static pivoting and then employ them within GMRES-IR to recover double precision accuracy.

4 Breakdown-free incomplete factorizations in low precision

In this section, we first look at how we can safely predict the possible occurrence of the breakdown problems B1, B2 and B3 before overflow occurs. We then consider two possible approaches for overcoming breakdown which, following [6], we refer to as local and global modifications. We say that an operation is *safe* in the precision being used if it cannot overflow; otherwise it is *unsafe*. We look at IC factorizations but the general ideas are also applicable to ILU factorizations.

4.1 Safe detection of overflow problems

To detect problem B1, we simply need to check at the start of major step k of the factorization algorithm that the pivot satisfies $l_{kk} \geq \tau$, where the chosen threshold parameter $\tau > 0$ depends on the precision used. Problem B2 can occur at Step 5 of Algorithm 2.1 in which all entries in the pivot column are scaled by the square root of the pivot. Algorithm 4.1 can be used to safely check whether a safe scaling is possible. By first finding the entry of largest absolute value in the column and setting it to a, the safe test only needs to be applied once with d set to the current l_{kk}. This limits the overhead of the safe check.

Algorithm 4.1. Safe test for a safe scaling of a scalar in low precision arithmetic
Input: *Scalar a with $|a| \leq x_{max}$ to be scaled, scaling factor $0 < d \leq x_{max}$.*
Output: *Scaled $v = a/d$ or a negative $flag$ indicating unsafe to scale.*

```
1:  $flag = 0$
2:  if $d \geq 1$ then                                ▷ Safe to scale
3:      $v = a/d$
4:  else
5:      if $d \geq |a|/x_{max}$ then                  ▷ Safe to scale
6:          $v = a/d$
7:      else                                          ▷ Unsafe to scale
8:          $flag = -1$
9:      end if
10: end if
```

Problem B3 can occur during the update operations at Step 9 of Algorithm 2.1. Algorithm 4.2 can be used to safely check whether safe updates are possible. Again, the test does not have to be applied to individual scalar entries. The maximum product corresponds to the square of the maximum scaled entry computed in Step 5 of Algorithm 2.1. The work for safe checking is bounded by the number of numerical operations.

Algorithm 4.2. Safe test for a safe update operation in low precision arithmetic
Input: *Input scalars a, b, c such that $|a|, |b|, |c| \le x_{max}$.*
Output: *$v = a - bc$ or negative $flag$ indicating unsafe to perform update.*

```
1:  $flag = 0$
2:  if $|b| < 1$ or $|c| < 1$ then                         ▷ Safe to compute bc
3:      $w = bc$
4:  else
5:      if $|b| \le x_{max}/|c|$ then                      ▷ Safe to compute division
6:          $w = bc$                                       ▷ Safe to compute bc
7:      else
8:          $flag = -1$; exit                              ▷ Unsafe to compute bc; terminate
9:      end if
10: end if
11: if $a = 0$ or $w = 0$ then
12:     $v = a - w$
13: else if $a > 0$ and $w > 0$ or $a < 0$ and $w < 0$ then
14:     $v = a - w$
15: else if $a > 0$ and $x_{max} - a \ge -w$ then
16:     $v = a - w$
17: else if $a < 0$ and $x_{max} + a \ge w$ then
18:     $v = a - w$
19: else
20:     $flag = -1$                                        ▷ Unsafe to compute subtraction
21: end if
```

4.2 Local modifications

As already observed, the occurrence of small or negative pivots (problem B1) can occur within incomplete factorizations, whatever precision is employed. Local diagonal modifications were first described in the 1970s by Kershaw [26]. The idea is to simply modify an individual diagonal entry of A during the factorization if it is found to be too small (or negative) to use as a pivot, that is, at the Step 3 of Algorithm 2.1 the computed pivot is perturbed by some positive quantity if it is not sufficiently large. A simple rule that employs a given threshold $\tau > 0$ is:

$$\text{if } l_{kk} < \tau \ \text{ then } \ l_{kk} \leftarrow \sqrt{x_{min}}. \tag{4.1}$$

This strategy is motivated by the hope that if only a few of the pivots are unstable (very small or nonpositive), the resulting incomplete factors may still yield a satisfactory preconditioner but, unfortunately, this is frequently not the case. Consequently, many more sophisticated rules have been proposed, usually for problems coming from specific applications: there is no generally accepted way to perform effective local modifications. Further discussion is given in the classical monograph [17] (see also [16]). For other classes of algebraic preconditioners, local modifications have also been found to be generally ineffective (for example, for factorized approximate inverse preconditioners [7]). Observe that local diagonal modifications are sometimes used by sparse direct solvers to prevent breakdown; this is usually referred to as static pivoting. It is cheap to incorporate within a direct solver but while it can be successful, it is also possible that by the time an unstable pivot is found, it is too late to save the stability of the factorization and locally perturbing the pivot effectively just amplifies numerical noise.

Next, assume that problem B2 has been detected, that is, overflow when scaling the off-diagonal entries in column k at Step 5 of Algorithm 2.1 has been detected using Algorithm 4.1. We want to show how B2 overflow can be avoided using the following result. Here and in the subsequent lemmas the scalars are all reals.

Lemma 4.1. *Assume the entries of a vector $w = \{w_k\} \in I\!R^m$ and the scalar $d > 0$ satisfy $|w_k|, d < x_{max}$ $(1 \le k \le m)$. Define $c \le x_{max}$ as follows.*

(i) If $d \ge 1$ then set $c = d$.

(ii) Otherwise, if $d < 1$ and $\sqrt{d} \times x_{max} \ge \max_{1\le k\le m} |w_k|$ then set $c = d$.

(iii) Otherwise, set $c = d \times \beta$ where $\beta > 1$ is such that $c \le x_{max}$ and $\sqrt{c} \times x_{max} \ge \max_{1\le k\le m} |w_k|$.

Then the entries of the scaled vector $w/\sqrt{c}$ satisfy $|w_k|/\sqrt{c} \le x_{max}$ $(1 \le k \le m)$ (that is, there is no overflow in the entries of the scaled vector).

The proof is straightforward because, in *(iii)*, $d < 1$ and an appropriate $\beta > 1$ clearly exists.

Setting $d = l_{kk}$ and $w = (l_{k+1,k}, \dots, l_{nk})^T$, Lemma 4.1 shows how to modify the diagonal entry so that the column scaling operation in Algorithm 2.1 can be performed safely. For $d < 1$, β can be chosen so that $c = 1$ but for the scaling of the pivot column we need to choose β to be small because the size of β corresponds to the size of the modification to l_{kk}. If β is found by successive increments, then in precision u, the operations needed for selecting $c = d \times \beta$ are safe if the test in *(ii)* is used within this sequence of guesses.

Finally, consider the most interesting problem B3, that is, avoiding overflow in the update operation at Step 9 of Algorithm 2.1. The update operation has two parts. First, the product of two entries l_{ik} and l_{jk} in column k that have been scaled in Step 5 by the square root of the corresponding diagonal entry must be computed. This product is then subtracted from entry l_{ij} of the current Schur complement (Steps 7-11). The next result relates to the product of the scaled entries.

Lemma 4.2. *If scalars w_1, w_2 satisfy $|w_1|, |w_2| \le x_{max}$ then there exists a scalar c such that $0 < c \le x_{max}$ and $|(w_1/\sqrt{c})(w_2/\sqrt{c})| \le x_{max}$.*

The proof is trivial because the result clearly holds if $c = x_{max}$.

Lemma 4.2 demonstrates that if the product of two scalars is unsafe, then c can be found such that, when each is scaled by $\sqrt{c}$, the product of the scaled scalars is a safe operation. If we employ Lemma 4.2 in Algorithm 2.1 then choosing c corresponds to increasing the diagonal entry l_{kk}. Again, we want to limit the size of the modification so c should be chosen to be as small as possible, for example, using a simple strategy of successively increasing the initial value of the diagonal entry. The safe test checks whether $\sqrt{c \times x_{max}} \ge \max\{|w_1|, |w_2|\}$. If not, then c is increased.

Note that it may be necessary to select $c \gg 1$ for the product of the off-diagonal entries within the update operation to be safe. This is potentially much larger than the smallest value needed for avoiding problem B2. Clearly, setting w_1 and w_2 in Lemma 4.2 to be any two components of $w = (l_{k+1,k}, \dots, l_{nk})^T$, B2 is automatically avoided.

To prevent problem B3, the subtraction $l_{ij} \leftarrow l_{ij} - l_{ik}l_{jk}$ must also be made safe. This can be achieved using the following result.

Lemma 4.3. *Assume the scalars e, f satisfy $|e| \le x_{max} - 1$, $|f| \le x_{max}$, Define g as follows.*

(i) If $sign(e) = sign(f)$ then set $g = f$.

(ii) Otherwise, find $0 < \beta \le x_{max}$ such that $|f|/\beta \le x_{max} - |e|$ and set $g = f \times \beta$.

Then $|e - g| \le x_{max}$.

Proof. If $sign(e) = sign(f)$ then the result is immediate. Otherwise, it is sufficient to choose $\beta = x_{max}$. □

Setting $f = l_{ik}l_{kj}$ and $e = l_{ij}$ with $|l_{ij}| \le x_{max} - 1$, an additional scaling that corresponds to increasing the diagonal entry l_{kk} to at most x_{max}^2 at the start of the k-th major step may be needed. Although such a large modification is not useful in practice, it can be implemented without overflow because only the square root of the modified diagonal entry is used. Consequently, the update operation can be made safe and overflow of the computed entries of the Schur complement prevented.

Lemma 4.3 indicates that the square-root IC factorization is the preferred implementation if there is a danger of diagonal entries overflowing. We observed this when performing preliminary numerical experiments on practical problems using fp16 arithmetic.

4.3 Global modifications in incomplete Cholesky factorizations

While local modifications are inexpensive to implement within a right-looking factorization algorithm, it is frequently the case that even a handful of pivot modifications will result in a poor preconditioner. Furthermore, local modifications cannot be efficiently incorporated into a left-looking approach because such modifications would require previously computed columns to be modified, not just the current column. Global strategies are generally more successful in terms of the quality of the resulting preconditioner (see, for example, [6]) and they can be used within a right- or left-looking algorithm. They have been widely used since the late 1970s; theoretical and numerical results may be found in [28, 31, 37]. When breakdown happens the straightforward strategy is to select a shift $\alpha > 0$, replace the scaled matrix $S^{-1}AS^{-1}$ by $S^{-1}AS^{-1} + \alpha I$ (I is the identity matrix) and restart the factorization. The factors of the shifted matrix are used to precondition the original scaled matrix. If A_D and A_E are, respectively, the diagonal and off-diagonal parts of $S^{-1}AS^{-1}$, then there is always some α for which $(1 + \alpha)A_D + A_E$ is diagonally dominant. Provided the target sparsity pattern of the incomplete factors contains the positions of the diagonal entries, then it can be shown that the incomplete factorization of this shifted matrix does not break down [31]. Diagonal dominance is sufficient for avoiding breakdown but it is not a necessary condition and an incomplete factorization may be breakdown free for much smaller values of α (particularly if A has been well scaled). An appropriate α is not usually known a priori: too large a value may harm the quality of the incomplete factors when used as a preconditioner for the original system and too small a value will not prevent breakdown, necessitating more than one restart, with a successively larger α. Typically, the shift is doubled after a breakdown, although more sophisticated strategies are sometimes used. For instance, in their limited-memory incomplete Cholesky factorization package `HSL_MI28`, Scott and Tůma [37] seek to reduce the number of restarts (and hence the preconditioner computation time) by increasing α more rapidly if successive breakdowns occur at essentially the same major step of the factorization. They also allow the user to supply a nonzero value for the initial shift α_0 (potentially useful when factorizing a sequence of related problems since reusing an earlier nonzero shift might be beneficial). If using α_0 is breakdown free then `HSL_MI28` offers an option to save the computed factors, reduce the shift and if, that is also breakdown free, to replace the computed factors with the new factors. This all happens without any intervention from the user.

When using fp16 arithmetic, detecting problem B1 uses a simple threshold test and Algorithms 4.1 and 4.2 can be used to safely detect problems B2 and B3. If the threshold test fails or one of these algorithms returns a negative flag, then α is increased and the factorization restarted. Our experiments on SPD matrices confirm that, provided we prescale A, the number of times we must increase α and restart is generally small (see the statistics *nmod* and $nofl$ is the tables of results in Section 5).

5 Numerical experiments

In this section, we investigate the effectiveness and reliability of half precision IC preconditioners. Our test examples are SPD matrices taken from the SuiteSparse Collection; they are listed in Table 5.1. In the top part of the table are those we classify as being well-conditioned (those for which our estimate *cond*2 of the 2-norm condition number is less than 10^7) and in the lower part are ill-conditioned examples. We have selected problems coming from a range of application areas and of different sizes and densities. Many of the problems are initially poorly scaled and some (including the first three problems in Table 5.1) contain entries that overflow in fp16 and thus prescaling of A is essential. The scaling used in all our experiments is such that the l_2 norm of each column of the scaled matrix is equal to 1. It is computed and applied to A in double precision. We performed tests using equilibration scaling (implemented using the HSL routine `MC77` [33, 34]) and found that the resulting preconditioner is of a similar quality; this is consistent with the findings reported in [37]. The right-hand side vector b is constructed by setting the solution x to be the vector of 1's.

Table 5.1: Statistics for our test examples. Those in the top half are considered to be well conditioned and those in the lower half to be ill conditioned. $nnz(A)$ denotes the number of entries in the lower triangular part of A. $normA$ and $normb$ are the infinity norms of A and b. $cond2$ is a computed estimate of the condition number of A.

Identifier	n	$nnz(A)$	$normA$	$normb$	$cond2$
HB/bcsstk27	1224	2.87×10^{4}	2.96×10^{7}	9.74×10^{5}	2.41×10^{4}
Nasa/nasa2146	2146	3.72×10^{4}	2.79×10^{8}	9.05×10^{6}	1.72×10^{3}
Cylshell/s1rmq4m1	5489	1.43×10^{5}	8.14×10^{6}	1.73×10^{5}	1.81×10^{6}
MathWorks/Kuu	7102	1.74×10^{5}	4.73×10^{2}	5.01	1.58×10^{4}
Pothen/bodyy6	19366	7.71×10^{4}	1.09×10^{5}	9.81×10^{4}	9.91×10^{4}
GHS_psdef/wathen120	36441	3.01×10^{5}	1.52×10^{3}	2.66×10^{2}	9.58×10^{2}
GHS_psdef/jnlbrng1	40000	1.20×10^{5}	3.29×10^{1}	2.00×10^{-1}	1.83×10^{2}
Williams/cant	62451	2.03×10^{6}	2.92×10^{5}	5.05×10^{3}	8.06×10^{3}
UTEP/Dubcova2	65025	5.48×10^{5}	6.67×10^{1}	1.18	3.33
Cunningham/qa8fm	66127	8.63×10^{5}	4.28×10^{-3}	9.51×10^{-4}	8.00
Mulvey/finan512	74752	3.36×10^{5}	3.91×10^{2}	3.78×10^{1}	2.51×10^{1}
GHS_psdef/apache1	80800	3.11×10^{5}	8.10×10^{5}	6.76×10^{-1}	4.18×10^{2}
Williams/consph	83334	3.05×10^{6}	6.61×10^{5}	7.20×10^{3}	1.25×10^{5}
AMD/G2_circuit	150102	4.38×10^{5}	2.27×10^{4}	2.17×10^{4}	2.02×10^{4}
Boeing/msc01050	1050	1.51×10^{4}	2.58×10^{7}	1.90×10^{6}	4.58×10^{15}
HB/bcsstk11	1473	1.79×10^{4}	1.21×10^{10}	7.05×10^{8}	2.21×10^{8}
HB/bcsstk26	1922	1.61×10^{4}	1.68×10^{11}	8.99×10^{10}	1.66×10^{8}
HB/bcsstk24	3562	8.17×10^{4}	5.28×10^{14}	4.21×10^{13}	1.95×10^{11}
HB/bcsstk16	4884	1.48×10^{5}	4.12×10^{10}	9.22×10^{8}	4.94×10^{9}
Cylshell/s2rmt3m1	5489	1.13×10^{5}	9.84×10^{5}	1.73×10^{4}	2.50×10^{8}
Cylshell/s3rmt3m1	5489	1.13×10^{5}	1.01×10^{5}	1.73×10^{3}	2.48×10^{10}
Boeing/bcsstk38	8032	1.82×10^{5}	4.50×10^{11}	4.04×10^{11}	5.52×10^{16}
Boeing/msc10848	10848	6.20×10^{5}	4.58×10^{13}	6.19×10^{11}	9.97×10^{9}
Oberwolfach/t2dah_e	11445	9.38×10^{4}	2.20×10^{-5}	1.40×10^{-5}	7.23×10^{8}
Boeing/ct20stif	52329	1.38×10^{6}	8.99×10^{11}	8.87×10^{11}	1.18×10^{12}
DNVS/shipsec8	114919	3.38×10^{6}	7.31×10^{12}	4.15×10^{11}	2.40×10^{13}
Um/2cubes_sphere	101492	8.74×10^{5}	3.43×10^{10}	3.59×10^{10}	2.59×10^{8}
GHS_psdef/hood	220542	5.49×10^{6}	2.23×10^{9}	1.51×10^{8}	5.35×10^{7}
Um/offshore	259789	2.25×10^{6}	1.44×10^{15}	1.16×10^{15}	4.26×10^{9}

We use Algorithm 3.2 (with our incomplete Cholesky factors replacing the complete LU factors). Our interest is in exploring whether we can recover (close to) double precision accuracy using preconditioners computed in fp16 arithmetic, although we are aware that in practice much less accuracy in the computed solution may be sufficient (indeed, in many practical situations, inaccuracies in the supplied data may mean low precision accuracy in the solution is all that can be justified). We thus use two precisions: u_ℓ for the factorization and $u_r = u_g = u_p = u$, where u is the fp64 unit roundoff. Iterative refinement is terminated when the normwise backward error for the computed solution satisfies

$$res = \frac{\|b - Ax\|_\infty}{\|A\|_\infty \|x\|_\infty + \|b\|_\infty} \le \delta. \qquad (5.1)$$

In our experiments, we set $\delta = 10^3 \times u$. The implementations of CG and GMRES used are `MI21` and `MI24`, respectively, from the HSL software library [24]. Except for the results in Table 5.3, the CG and the GMRES convergence tolerance is $\delta_{krylov} = \sqrt{u}$ and the limit on the number of iterations for each application of CG and GMRES is 1000.

The reported experiments are performed on a Windows 11-Pro-based machine with an Intel(R) Core(TM) i5-10505 CPU processor (3.20GHz). Our results are for a right-looking implementation of the level-based incomplete Cholesky factorization $IC(\ell)$, using a range of values of $\ell \geq 0$. The sparsity pattern of L is computed using the approach of Hysom and Pothen [25]. This is able to compute the patterns of the rows of L independently and thus in parallel. Our software is written in Fortran and compiled using the NAG compiler (Version 7.1, Build 7118). This is currently the only Fortran compiler that supports the use of fp16. The Nag-compiled code evaluates scalar fp16 operations in single precision, and rounds to half precision when assigned to a variable or passed as an actual argument to a non-intrinsic or non-mathematical procedure. Because of all the conversions needed, half precision is slower than single precision and so no timings are reported in our experiments. We refer to the $IC(\ell)$ factorizations computed using half precision and double precision arithmetics as fp16-$IC(\ell)$ and fp64-$IC(\ell)$, respectively.

The key difference between the fp16 and fp64 versions of our $IC(\ell)$ software is that for the former, during the incomplete factorization, we incorporate the safe scaling and update operations (as discussed in the previous section). In addition, the fp16 version allows the preconditioner to be applied in either half or double precision arithmetic; the former is for LU-IR and the latter for Krylov-IR (recall Algorithms 3.1 and 3.2). In the Krylov-IR case, the solves with L and L^T require the L factor to be in double precision. There are two possible ways to handle this. The first is to make an explicit copy of L by casting the data into double precision but this negates the important benefit that half precision offers of reducing memory requirements. The alternative is to cast the entries on the fly. This is straightforward to incorporate into a serial triangular solve routine, and only requires a temporary double precision array of length n.

In all the reported experiments we use global modifications. This is because local modifications proved unreliable. Although the theory presented in Section 4.2 shows it is possible to make local modifications to avoid the breakdown problems B1, B2 and B3, in practice we unfortunately have no robust method of making the modifications that leads to reliably high-quality preconditioners. Consider, for example, the extreme case that at the start of the major step k of Algorithm 2.1 the diagonal entry l_{kk} is equal to zero and the absolute values of all the remaining entries of column k are x_{max}. A local modification that replaces l_{kk} by some chosen value less than 1 prevents B1, but the corresponding column then does not scale (entries overflow) so that B1 is transferred into a B2 problem.

In the results tables, $nnz(L)$ is the number of entries in the incomplete factor L; $iouter$ denotes the number of iterative refinement steps (that is, the number of times the loop starting at Step 3 in Algorithm 3.2 is executed) and $totits \leq itmax$ is the total number of CG (or GMRES) iterations performed; $resint$ is the initial scaled residual (that is, (5.1) with $x = x_1 = L^{-T}L^{-1}b$) and $resfinal$ is the scaled residual for the computed solution; $nmod$ and $nofl$ are the numbers of times problems B1 and B3 occur during the incomplete factorization, with the latter for fp16 only. This is because our right-looking approach allows us to monitor the diagonal entries and increase the global shift as necessary and restart, which then avoids problem B2. Problem B2 could then occur in column k if, for example, after the shift, the diagonal entry l_{kk} is close to the shift α and $|l_{ik}| \geq \alpha \times x_{max}$ for some $i > k$. In all the experiments on well-conditioned problems, we found $nofl = 0$ and so this statistic is omitted from the corresponding tables of results. As expected, $nmod > 0$ can occur for fp16 and fp64, and for both well-conditioned and ill-conditioned examples.

5.1 Results for LU-IR

LU-IR is attractive because the application of the preconditioner is performed in half precision arithmetic. Table 5.2 reports results for LU-IR for the well-conditioned test problems. The iteration count is limited to 1000. We see that for three problems we failed to achieve the requested accuracy within this limit. Additionally, for problem Williams/consph, the refinement procedure diverges and the process is stopped when the norm of the residual approaches x_{max} in double precision. Results are given for the only two ill-conditioned problems that were successfully solved using LU-IR; for the other test examples, we failed to achieve convergence.

Table 5.2: LU-IR results for well-conditioned problems. The preconditioner is fp16-$IC(3)$. *resint* and *resfinal* are the initial and final residuals; $nnz(L)$ is the number of entries in the $IC(3)$ factor; *iters* is the number of refinement steps; and *nmod* denotes the number of times problem B1 occurs during the factorization. > 1000 indicates the requested accuracy was not obtained within the iteration limit. † indicates the refinement procedure breaks down.

Identifier	*resinit*	*resfinal*	$nnz(L)$	*iters*	*nmod*
HB/bcsstk27	8.23×10^{-5}	6.13×10^{-14}	4.88×10^{4}	13	0
Nasa/nasa2146	9.43×10^{-5}	1.89×10^{-13}	7.89×10^{4}	15	0
Cylshell/s1rmq4m1	5.63×10^{-5}	3.80×10^{-8}	3.15×10^{5}	> 1000	0
MathWorks/Kuu	1.69×10^{-4}	2.14×10^{-13}	7.65×10^{5}	194	0
Pothen/bodyy6	7.50×10^{-3}	2.20×10^{-13}	1.76×10^{5}	817	2
GHS_psdef/wathen120	4.44×10^{-4}	1.70×10^{-14}	8.30×10^{5}	5	0
GHS_psdef/jnlbrng1	1.31×10^{-3}	6.40×10^{-14}	2.77×10^{5}	16	0
Williams/cant	3.26×10^{-4}	4.61×10^{-8}	9.95×10^{6}	> 1000	0
UTEP/Dubcova2	1.54×10^{-3}	2.19×10^{-13}	6.22×10^{6}	736	0
Cunningham/qa8fm	4.08×10^{-4}	2.27×10^{-15}	5.14×10^{6}	5	0
Mulvey/finan512	3.22×10^{-4}	3.61×10^{-15}	4.08×10^{6}	5	0
GHS_psdef/apache1	8.52×10^{-5}	1.03×10^{-7}	1.54×10^{6}	> 1000	0
Williams/consph	1.05×10^{-4}	†	2.02×10^{7}	†	0
AMD/G2_circuit	8.16×10^{-4}	7.46×10^{-8}	1.04×10^{6}	> 1000	0
Oberwolfach/t2dah_e	6.95×10^{-4}	9.12×10^{-15}	3.29×10^{5}	5	0
Um/2cubes_sphere	1.03×10^{-3}	1.33×10^{-15}	8.70×10^{6}	5	0

5.2 Dependence of the iteration counts on the CG tolerance

The results in Table 5.3 illustrate the dependence of the number of iterative refinement steps and the total iteration count on the convergence tolerance δ_{krylov} used by CG within CG-IR. The preconditioner is $IC(3)$ computed using fp16 arithmetic. For three examples, we test δ_{krylov} ranging from 10^{-9} to 10^{-1}. The first problem is well-conditioned and the other two are ill-conditioned. As expected, the number of outer iterations increases with δ_{krylov}, but the variations in the total iteration count are relatively modest. This confirms the choice of $\sqrt{u}$, which is used for all remaining experiments.

Table 5.3: The effects of changing the CG convergence tolerance δ_{krylov} used in CG-IR. The preconditioner is fp16-$IC(3)$. *iouter* and *totits* denote the number of outer iterations and the total number of CG iterations, respectively.

	UTEP/Dubcova2								
δ_{krylov}	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}
Outer iterations	2	2	2	2	2	2	4	6	10
Total iterations	73	64	58	49	68	55	57	58	70
	HB/bcsstk26								
δ_{krylov}	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}
Outer iterations	2	2	2	2	3	3	4	6	11
Total iterations	121	107	93	78	103	81	84	97	96
	Cylshell/s2rmt3m1								
δ_{krylov}	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}
Outer iterations	2	2	2	2	3	3	4	5	9
Total iterations	130	125	120	116	94	83	117	85	96

5.3 Results for $IC(0)$

Table 5.4: Results for CG-IR using an $IC(0)$ preconditioner: well-conditioned problems. *resint* and *resfinal* are the initial and final residuals. $nnz(L)$ is the number of entries in the $IC(0)$ factor. *iouter* and *totits* denote the number of outer iterations and the total number of CG iterations, respectively > 1000 indicates CG tolerance not reached on outer iteration *iouter*. *nmod* denotes the number of times problem B1 occurs during the factorization. A count in bold indicates the fp16 result is within 10 per cent of (or is better than) the corresponding fp64 result.

		Preconditioner fp16-$IC(0)$				
Identifier	*resinit*	*resfinal*	$nnz(L)$	*iouter*	*totits*	*nmod*
HB/bcsstk27	4.26×10^{-4}	2.66×10^{-17}	2.87×10^{4}	2	**43**	0
Nasa/nasa2146	3.27×10^{-4}	2.91×10^{-17}	3.72×10^{4}	2	**28**	0
Cylshell/s1rmq4m1	1.49×10^{-4}	2.45×10^{-17}	1.15×10^{5}	2	224	0
MathWorks/Kuu	3.25×10^{-3}	4.09×10^{-17}	1.43×10^{5}	2	312	4
Pothen/bodyy6	2.12×10^{-4}	1.75×10^{-16}	7.03×10^{4}	2	161	2
GHS_psdef/wathen120	1.09×10^{-2}	8.76×10^{-17}	3.01×10^{5}	2	**21**	0
GHS_psdef/jnlbrng1	9.97×10^{-3}	1.21×10^{-16}	1.20×10^{5}	2	**50**	0
Williams/cant	4.62×10^{-3}	8.34×10^{-7}	1.46×10^{6}	1	> 1000	9
UTEP/Dubcova2	5.55×10^{-3}	3.93×10^{-17}	4.19×10^{5}	2	**268**	0
Cunningham/qa8fm	3.58×10^{-3}	1.04×10^{-16}	8.63×10^{5}	2	**14**	0
Mulvey/finan512	2.88×10^{-3}	4.97×10^{-17}	3.36×10^{5}	2	**17**	0
GHS_psdef/apache1	3.63×10^{-4}	4.15×10^{-14}	3.11×10^{5}	1	**248**	0
Williams/consph	8.58×10^{-5}	4.63×10^{-17}	3.05×10^{6}	2	**593**	7
AMD/G2_circuit	7.78×10^{-4}	9.83×10^{-16}	4.38×10^{5}	2	**779**	0
		Preconditioner fp64-$IC(0)$				
Identifier	*resinit*	*resfinal*	$nnz(L)$	*iouter*	*totits*	*nmod*
HB/bcsstk27	4.31×10^{-4}	2.66×10^{-17}	2.87×10^{4}	2	43	0
Nasa/nasa2146	3.06×10^{-4}	2.91×10^{-17}	3.72×10^{4}	2	28	0
Cylshell/s1rmq4m1	1.61×10^{-4}	2.27×10^{-17}	1.43×10^{5}	2	190	0
MathWorks/Kuu	9.47×10^{-4}	4.09×10^{-17}	1.74×10^{5}	2	148	0
Pothen/bodyy6	8.68×10^{-5}	1.05×10^{-16}	7.71×10^{4}	2	124	0
GHS_psdef/wathen120	1.09×10^{-2}	1.04×10^{-16}	3.01×10^{5}	2	21	0
GHS_psdef/jnlbrng1	9.96×10^{-3}	1.34×10^{-16}	1.20×10^{5}	2	50	0
Williams/cant	2.63×10^{-3}	5.83×10^{-8}	2.03×10^{6}	1	> 1000	8
UTEP/Dubcova2	5.36×10^{-3}	3.93×10^{-17}	5.48×10^{5}	2	270	0
Cunningham/qa8fm	3.57×10^{-3}	1.09×10^{-16}	8.63×10^{5}	2	14	0
Mulvey/finan512	2.92×10^{-3}	4.15×10^{-17}	3.36×10^{5}	2	17	0
GHS_psdef/apache1	3.63×10^{-4}	4.23×10^{-14}	3.11×10^{5}	1	236	0
Williams/consph	8.26×10^{-5}	4.90×10^{-17}	3.05×10^{6}	2	594	7
AMD/G2_circuit	5.44×10^{-4}	6.55×10^{-16}	4.38×10^{5}	2	772	0

As already noted, $IC(0)$ is a very simple preconditioner but one that is frequently reported on in publications. Results for CG-IR using an $IC(0)$ preconditioner computed in half and double precision arithmetics are given in Tables 5.4 and 5.5 for well-conditioned and ill-conditioned problems, respectively. If the total iteration count (*totits*) for fp16 is within 10 per cent of the count for fp64 (or is less than the fp64 count) then it is highlighted in bold. We see that, for well-conditioned problems, using fp16 arithmetic to compute the $IC(0)$ factorization is often as good as using fp64 arithmetic. For problem Williams/cant, the CG method on the first outer iteration fails to converge within the 1000 iteration count for the fp16 and the fp64 preconditioners. *nofl* is omitted from Tables 5.4 and 5.5 because it was equal to 0 for all our test examples. However, for many problems (particularly the ill-conditioned ones), $nmod > 0$ for both half precision and double precision and this can lead to a poor quality preconditioner, indicated by high iteration counts, with the limit of 1000 iterations being exceeded on the second outer iteration for a number of test examples (such as Boeing/bcsstk38 and Boeing/msc01050). Although the requested accuracy is not achieved, there are still significant reductions in the initial residual so the preconditioners may be acceptable if less accuracy is required. Nevertheless, the fp16 performance is often competitive with that of fp64, and the fp16 factor can be sparser than for fp64 because entries of the scaled A can underflow

Table 5.5: Results for CG-IR and GMRES-IR using an $IC(0)$ preconditioner: ill-conditioned problems. $resint$ is the initial residual; $resfinal$ is the final CG-IR scaled residual. $nnz(L)$ is the number of entries in the $IC(0)$ factor. $iouter$ and $totits$ denote the number of outer iterations and the total number of CG iterations with the GMRES statistics in parentheses. > 1000 indicates CG (or GMRES) tolerance not reached on outer iteration $iouter$. $nmod$ denotes the number of times problem B1 occurs during the factorization. A count in bold indicates the fp16 result is within 10 per cent of (or is better than) the corresponding fp64 result. * denotes early termination of CG.

		Preconditioner fp16-$IC(0)$						
Identifier	$resinit$	$resfinal$	$nnz(L)$	$iouter$		$totits$		$nmod$
Boeing/msc01050	1.45×10^{-5}	1.71×10^{-14}	9.49×10^{3}	2	(2)	> 1000	(**1161**)	7
HB/bcsstk11	5.38×10^{-4}	4.65×10^{-17}	1.52×10^{4}	2	(2)	**1099**	(753)	5
HB/bcsstk26	6.71×10^{-4}	1.18×10^{-16}	1.33×10^{4}	2	(2)	**455**	(**393**)	2
HB/bcsstk24	4.50×10^{-5}	7.71×10^{-12}	7.97×10^{4}	2	(2)	> 1000	(1115)	3
HB/bcsstk16	3.36×10^{-3}	4.82×10^{-17}	1.27×10^{5}	2	(2)	102	(90)	4
Cylshell/s2rmt3m1	1.41×10^{-4}	3.27×10^{-17}	1.02×10^{5}	2	(2)	**490**	(**482**)	0
Cylshell/s3rmt3m1	1.13×10^{-5}	2.66×10^{-17}	1.02×10^{5}	2	(2)	1741	(1483)	3
Boeing/bcsstk38	3.25×10^{-2}	3.16×10^{-10}	1.63×10^{5}	2	(2)	> 1000	(> 1000)	8
Boeing/msc10848	5.79×10^{-6}	7.90×10^{-18}	6.18×10^{5}	2	(2)	1325	(**601**)	2
Oberwolfach/t2dah_e	4.88×10^{-2}	$4.81\times10^{-9}*$	9.38×10^{4}	1	(2)	**15**	(**33**)	0
Boeing/ct20stif	3.83×10^{-3}	1.29×10^{-9}	1.30×10^{6}	2	(2)	> 1000	(> 1000)	7
DNVS/shipsec8	2.31×10^{-3}	1.98×10^{-9}	1.53×10^{6}	2	(2)	> 1000	(> 1000)	8
Um/2cubes_sphere	1.88×10^{-2}	1.63×10^{-16}	8.74×10^{5}	2	(2)	**16**	(**14**)	0
GHS_psdef/hood	1.91×10^{-3}	4.38×10^{-17}	5.06×10^{6}	2	(2)	**583**	(**487**)	2
Um/offshore	1.51×10^{-2}	3.84×10^{-16}	2.25×10^{6}	2	(2)	**523**	(**284**)	0
		Preconditioner fp64-$IC(0)$						
Identifier	$resinit$	$resfinal$	$nnz(L)$	$iouter$		$totits$		$nmod$
Boeing/msc01050	6.03×10^{-3}	4.37×10^{-16}	1.51×10^{4}	2	(2)	860	(1198)	8
HB/bcsstk11	7.49×10^{-4}	7.44×10^{-17}	1.79×10^{4}	2	(2)	1036	(593)	4
HB/bcsstk26	1.59×10^{-3}	2.37×10^{-16}	1.61×10^{4}	2	(2)	552	(456)	4
HB/bcsstk24	4.59×10^{-5}	6.19×10^{-12}	8.17×10^{4}	2	(2)	> 1000	(906)	3
HB/bcsstk16	2.76×10^{-3}	3.97×10^{-17}	1.48×10^{5}	2	(2)	80	(79)	0
Cylshell/s2rmt3m1	1.45×10^{-4}	2.91×10^{-17}	1.13×10^{5}	2	(2)	472	(467)	0
Cylshell/s3rmt3m1	1.14×10^{-5}	2.22×10^{-17}	1.13×10^{5}	2	(2)	896	(1330)	0
Boeing/bcsstk38	1.21×10^{-2}	5.94×10^{-10}	1.82×10^{5}	2	(2)	> 1000	(> 1000)	7
Boeing/msc10848	1.07×10^{-5}	2.63×10^{-18}	6.20×10^{5}	2	(2)	1188	(640)	3
Oberwolfach/t2dah_e	4.86×10^{-2}	$4.84\times10^{-9}*$	9.38×10^{4}	1	(2)	15	(33)	0
Boeing/ct20stif	3.65×10^{-3}	7.52×10^{-10}	1.38×10^{6}	2	(2)	> 1000	(> 1000)	7
DNVS/shipsec8	2.75×10^{-4}	6.14×10^{-10}	3.38×10^{6}	2	(2)	> 1000	(> 1000)	4
Um/2cubes_sphere	1.89×10^{-2}	2.17×10^{-16}	8.74×10^{5}	2	(2)	16	(14)	0
GHS_psdef/hood	9.82×10^{-4}	5.01×10^{-17}	5.49×10^{6}	2	(2)	752	(519)	2
Um/offshore	1.51×10^{-2}	3.84×10^{-16}	2.25×10^{6}	2	(2)	519	(284)	0

in half precision arithmetic. For problem Oberwolfach/t2dah_e, the CG algorithm terminates before the requested accuracy has been achieved; this is because the curvature encountered within the CG algorithm is found to be too small, triggering an error return.

We have also run GMRES-IR with the $IC(0)$ preconditioners on all our test problems. For the well-conditioned problems, the iterations counts using GMRES are broadly similar to those for CG. For the ill-conditioned problems, the GMRES-IR iteration counts are given in parentheses in the *iouter* and *totits* columns of Table 5.5. Our findings are consistent with those reported in [22], namely that GMRES-IR is robust and it may require fewer iterations than CG-IR. However, it is important to remember that each GMRES iteration is more expensive than a CG iteration so simply comparing the iteration count may be misleading.

5.4 Results for $IC(\ell)$

Figure 5.1 illustrates the influence of the number of levels ℓ in the $IC(\ell)$ preconditioner computed in half precision and double precision. Typically, ℓ is chosen to small (large values lead to slow computation times and loss of sparsity in L) although there are cases where larger ℓ may be employed [25]. In general, it is hoped that as ℓ increases, the additional fill-in in L will result in a better preconditioner (but there is no guarantee of this). We observe that for the well-conditioned problem UTEP/Dubcova2, the half precision and double precision preconditioners behave similarly. For this example, as ℓ increases from 1 to 9 the number of entries in the incomplete factor increases from 1.77×10^6 to 3.65×10^7. For the ill-conditioned problem Cylshell/s2rmt3m1, the corresponding increase is from 1.43×10^5 to 5.58×10^5. In this case, the iteration count for fp64+$IC(\ell)$ steadily decreases as ℓ increases but for fp16+$IC(\ell)$ the decrease is much less and it stagnates for $\ell > 4$. In the rest of this section, we perform experiments with $\ell = 3$.

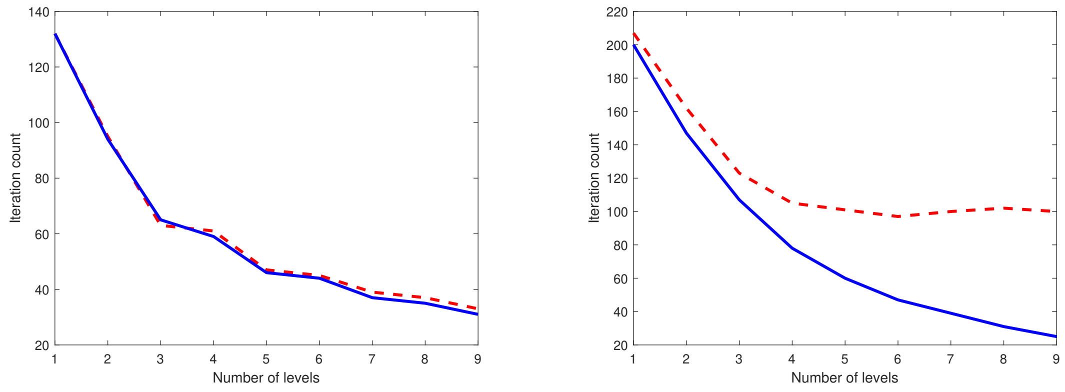

Figure 5.1: CG-IR total iteration counts for the fp16+$IC(\ell)$ preconditiioner (solid line) and fp64+$IC(\ell)$ preconditioner (dashed line) with k ranging from 1 to 9 for problems UTEP/Dubcova2 (left) and Cylshell/s2rmt3m1 (right).

Tables 5.6 and 5.7 present results for CG-IR with the fp16-$IC(3)$ and fp64-$IC(3)$ preconditioners for the well-conditioned and ill-conditioned test sets, respectively. The latter also reports total iteration counts for GMRES-IR (the number of outer iterations *iouter* for GMRES-IR and CG-IR are the same for all the test examples). In double precision arithmetic, B1 breakdowns do not happen for any of our test cases and so the *nmod* statistic is omitted. B3 breakdowns occur for a small number of the ill-conditioned problems. We see that fp16-$IC(3)$ performs as well as fp64-$IC(3)$ on most of the well-conditioned problems. The same is not true for the ill-conditioned problems. While the fp16-$IC(3)$ preconditioner combined with CG-IR and GMRES-IR is able to return a computed solution with a small residual, for many examples the iteration counts are significantly greater than for the fp64-$IC(3)$ preconditioner. However, for a small number of problems the computation of the fp64-$IC(3)$ preconditioner fails in that the computed factor has very large entries, which do not overflow in double precision but make it useless as a preconditioner.

Table 5.6: Results for CG-IR using an $IC(3)$ preconditioner: well-conditioned problems. $resint$ and $resfinal$ are the initial and final residuals. $nnz(L)$ is the number of entries in the $IC(3)$ factor. $iouter$ and $totits$ denote the number of outer iterations and the total number of CG iterations, respectively. > 1000 indicates CG tolerance not reached on outer iteration $iouter$. $nmod$ denotes the number of times problem B1 occurs during the factorization (for fp64-$IC(3)$ it is equal to 0 for all our test cases and so omitted). A count in bold indicates the fp16 result is within 10 per cent of (or is better than) the corresponding fp64 result.

Preconditioner fp16-$IC(3)$						
Identifier	$resinit$	$resfinal$	$nnz(L)$	$iouter$	$totits$	$nmod$
HB/bcsstk27	8.23×10^{-5}	3.42×10^{-17}	4.88×10^{4}	2	13	0
Nasa/nasa2146	9.43×10^{-5}	2.91×10^{-17}	7.89×10^{4}	2	**13**	0
Cylshell/s1rmq4m1	5.63×10^{-5}	2.10×10^{-17}	3.15×10^{5}	2	**54**	0
MathWorks/Kuu	1.69×10^{-4}	5.20×10^{-17}	7.65×10^{5}	2	**36**	0
Pothen/bodyy6	7.50×10^{-3}	2.10×10^{-16}	1.76×10^{5}	2	96	2
GHS_psdef/wathen120	4.44×10^{-4}	6.37×10^{-17}	8.30×10^{5}	2	**7**	0
GHS_psdef/jnlbrng1	1.31×10^{-3}	1.34×10^{-16}	2.77×10^{5}	2	**16**	0
Williams/cant	3.26×10^{-4}	1.54×10^{-16}	9.95×10^{6}	2	**1193**	0
UTEP/Dubcova2	1.54×10^{-3}	3.93×10^{-17}	6.22×10^{6}	2	**63**	0
Cunningham/qa8fm	4.08×10^{-4}	1.04×10^{-16}	5.14×10^{6}	2	**6**	0
Mulvey/finan512	3.22×10^{-4}	4.15×10^{-17}	4.08×10^{6}	2	**6**	0
GHS_psdef/apache1	8.52×10^{-5}	6.82×10^{-14}	1.54×10^{6}	1	**114**	0
Williams/consph	1.05×10^{-4}	4.35×10^{-17}	2.02×10^{7}	2	**198**	0
AMD/G2_circuit	8.16×10^{-4}	4.10×10^{-16}	1.04×10^{6}	2	**246**	0
Preconditioner fp64-$IC(3)$						
Identifier	$resinit$	$resfinal$	$nnz(L)$	$iouter$	$totits$	
HB/bcsstk27	3.93×10^{-5}	2.66×10^{-17}	4.88×10^{4}	2	10	
Nasa/nasa2146	4.32×10^{-5}	2.58×10^{-17}	7.89×10^{4}	2	13	
Cylshell/s1rmq4m1	3.55×10^{-5}	2.10×10^{-17}	3.15×10^{5}	2	49	
MathWorks/Kuu	2.50×10^{-4}	4.09×10^{-17}	7.65×10^{5}	2	32	
Pothen/bodyy6	9.31×10^{-3}	3.86×10^{-16}	1.76×10^{5}	2	51	
GHS_psdef/wathen120	1.77×10^{-4}	6.37×10^{-17}	8.30×10^{5}	2	7	
GHS_psdef/jnlbrng1	9.40×10^{-4}	1.07×10^{-16}	2.77×10^{5}	2	14	
Williams/cant	3.19×10^{-4}	1.50×10^{-16}	9.95×10^{6}	2	1132	
UTEP/Dubcova2	1.65×10^{-3}	4.58×10^{-17}	6.22×10^{6}	2	65	
Cunningham/qa8fm	3.29×10^{-5}	1.04×10^{-16}	5.14×10^{6}	2	5	
Mulvey/finan512	3.72×10^{-5}	4.97×10^{-17}	4.08×10^{6}	2	6	
GHS_psdef/apache1	8.28×10^{-5}	7.58×10^{-14}	1.54×10^{6}	1	113	
Williams/consph	3.39×10^{-4}	6.07×10^{-3}	2.02×10^{7}	1	> 1000	
AMD/G2_circuit	8.42×10^{-5}	4.10×10^{-16}	1.04×10^{6}	2	237	

Table 5.7: Results for CG-IR and GMRES-IR using an $IC(3)$ preconditioner: ill-conditioned problems. $resint$ is the initial residual; $resfinal$ is the final CG-IR scaled residual. $nnz(L)$ is the number of entries in the $IC(3)$ factor. $iouter$ and $totits$ denote the number of outer iterations and the total number of CG iterations with the GMRES statistics in parentheses. > 1000 indicates CG (or GMRES) tolerance not reached on outer iteration $iouter$. $nmod$ and $nofl$ denote the numbers of times problems B1 and $B3$ occur during the factorization (for fp64-$IC(3)$ they are equal to 0 for all our test cases and so omitted). A count in bold indicates the fp16 result is within 10 per cent (or better) of the corresponding fp64 result. ‡ indicates failure to compute the factorization because of enormous growth in its entries.

		Preconditioner fp16-$IC(3)$						
Identifier	$resinit$	$resfinal$	$nnz(L)$	$iouter$	$totits$		$nmod$	$nofl$
Boeing/msc01050	1.60×10^{-3}	3.70×10^{-16}	3.74×10^{4}	2	277	(1029)	2	0
HB/bcsstk11	5.84×10^{-4}	1.86×10^{-17}	4.18×10^{4}	2	274	(202)	0	2
HB/bcsstk26	6.55×10^{-5}	9.25×10^{-17}	3.43×10^{4}	2	104	(92)	2	0
HB/bcsstk24	5.91×10^{-7}	3.77×10^{-17}	2.27×10^{5}	2	534	(294)	1	1
HB/bcsstk16	6.25×10^{-4}	4.82×10^{-17}	4.89×10^{5}	2	21	(21)	0	0
Cylshell/s2rmt3m1	8.43×10^{-6}	3.27×10^{-17}	2.60×10^{5}	2	123	(120)	0	0
Cylshell/s3rmt3m1	1.56×10^{-6}	2.66×10^{-17}	2.60×10^{5}	2	**654**	(1166)	1	1
Boeing/bcsstk38	4.79×10^{-5}	9.29×10^{-16}	5.64×10^{5}	2	821	(350)	2	0
Boeing/msc10848	7.72×10^{-7}	2.63×10^{-18}	2.51×10^{6}	2	159	(113)	0	0
Oberwolfach/t2dah_e	6.95×10^{-4}	1.41×10^{-16}	3.29×10^{5}	2	**7**	(**7**)	0	0
Boeing/ct20stif	3.01×10^{-5}	1.28×10^{-9}	6.70×10^{6}	2	$>$ 1000	(1041)	2	0
DNVS/shipsec8	5.64×10^{-6}	2.70×10^{-13}	1.22×10^{7}	2	1107	(1069)	2	0
Um/2cubes_sphere	1.03×10^{-3}	1.63×10^{-16}	8.70×10^{6}	2	**6**	(**6**)	0	0
GHS_psdef/hood	7.05×10^{-4}	5.01×10^{-17}	2.78×10^{7}	2	**414**	(**366**)	1	3
Um/offshore	6.27×10^{-4}	1.92×10^{-16}	2.08×10^{7}	2	**165**	(**114**)	0	4

		Preconditioner fp64-$IC(3)$				
Identifier	$resinit$	$resfinal$	$nnz(L)$	$iouter$	$totits$	
Boeing/msc01050	1.41×10^{-4}	3.35×10^{-16}	3.74×10^{4}	2	41	(1010)
HB/bcsstk11	1.48×10^{-5}	2.79×10^{-17}	4.18×10^{4}	2	38	(35)
HB/bcsstk26	9.24×10^{-5}	1.18×10^{-16}	3.43×10^{4}	2	82	(75)
HB/bcsstk24	4.01×10^{-7}	2.05×10^{-17}	2.27×10^{5}	2	85	(75)
HB/bcsstk16	7.01×10^{-4}	3.40×10^{-17}	4.89×10^{5}	2	18	(17)
Cylshell/s2rmt3m1	8.61×10^{-6}	3.63×10^{-17}	2.60×10^{5}	2	107	(105)
Cylshell/s3rmt3m1	2.00×10^{-6}	6.12×10^{-5}	2.60×10^{5}	1	$>$ 1000	($>$ 1000)
Boeing/bcsstk38	4.67×10^{-9}	8.57×10^{-16}	5.64×10^{5}	2	167	(124)
Boeing/msc10848	8.73×10^{-10}	5.27×10^{-18}	2.51×10^{6}	2	49	(41)
Oberwolfach/t2dah_e	5.49×10^{-6}	1.41×10^{-16}	3.29×10^{5}	2	6	(6)
Boeing/ct20stif	5.82×10^{-8}	1.47×10^{-9}	6.70×10^{6}	2	$>$ 1000	(1006)
DNVS/shipsec8	7.50×10^{-7}	1.26×10^{-16}	1.22×10^{7}	2	266	(212)
Um/2cubes_sphere	2.25×10^{-5}	1.36×10^{-16}	8.70×10^{6}	2	5	(5)
GHS_psdef/hood	‡	‡	‡	‡	‡	‡
Um/offshore	‡	‡	‡	‡	‡	‡

This indicates that it is not just in half precision arithmetic that it is necessary to monitor the possibility of growth occurring in the factor entries, something that is not currently considered when computing incomplete Cholesky factorizations in double (or single) precision arithmetic.

Note that, when using fp16 arithmetic, the initial residual *resint* is typically larger than for fp64 arithmetic. However, if the user does not require double precision accuracy in the computed solution, it may be unnecessary to perform any refinement steps (or a small number of steps may be sufficient), even when using half precision incomplete factors.

Finally, we observe that, in addition to symmetrically prescaling A, Higham and Pranesh [22] shift the scaled matrix and multiply all the entries by a scalar μ to limit underflow. Our $IC(\ell)$ implementation incorporates scaling and shifting by default but not μ (thus our reported results correspond to $\mu = 1.0$). We have performed experiments with $\mu = 0.1x_{max}$ [22]. We found that there was no consistent advantage in using $\mu \neq 1$ (for some examples, the iteration count varied but the changes were modest and the count was not always reduced). This is perhaps because the underflows are not such a concern when constructing incomplete factorizations as they are for complete factorizations.

6 Concluding remarks and future directions

As far as we are aware, our work in the first to construct and employ incomplete factorizations using half precision arithmetic to solve large-scale sparse linear systems. Our results demonstrate that, when carefully implemented, the use of fp16 level-based incomplete factorization preconditioners may not impact on the overall accuracy of the computed solution, even when a small tolerance is imposed on the requested scaled residual. Unsurprisingly, the number of iterations of the Krylov subspace method that is used in the refinement process can be greater for fp16 factors compared to fp64 factors but generally this increase is only significant for highly ill conditioned systems. The numerical experiments support the view that, for many real-world problems, it is sufficient to employ half precision arithmetic. Its use may be particularly advantageous if the linear system does not need to be solved to high accuracy. Our study also encourages us to conjecture that by building safe operations into sparse direct solvers it should be possible to build efficient and robust half precision variants and, because this would lead to substantial memory savings, it should allow direct solvers to be used (in combination with an appropriate refinement process) to solve much larger problems than is currently possible.

It is of interest to explore other classes of algebraic preconditioners, to consider how they can be safely computed using fp16 arithmetic and how effective they are compared to higher precision versions. For sparse approximate inverse (SPAI) preconditioners, we anticipate that it may be possible to combine the systematic dropping of subnormal quantities with avoiding overflows in local linear solves as we have discussed without significantly affecting the preconditioner quality because such changes may only influence a small number of the computed columns of the SPAI preconditioner (see also [14] for a recent analysis of SPAI preconditioners in mixed precision). For other approximate inverses, such as AINV [8] and AIB [36], the sizes of the diagonal entries follow from maintaining a generalized orthogonalization property and avoiding overflows using local or global modifications may be possible but challenging. We also plan to consider the construction of the `HSL_MI28` [37] IC preconditioner using low precision arithmetic. `HSL_MI28` uses an extended memory approach for the robust construction of the factors. Once the factors have been computed, the extended memory can be freed, limiting the size of the factors and the work needed in the substitution steps, generally without a significant reduction in the preconditioner quality. A challenge here is safely allowing intermediate quantities of absolute value at least x_{min} to be retained in the factors or in the extended memory.

References

[1] A. Abdelfattah, H. Anzt, E. G. Boman, E. Carson, T. Cojean, J. Dongarra, A. Fox, M. Gates, N. J. Higham, X. S. Li, J. Loe, P. Luszczek, S. Pranesh, S. Rajamanickam, T. Ribizel, B. F. Smith, K. Swirydowicz,

S. Thomas, S. Tomov, Y. M. Tzai, and U. Meier Yang. A survey of numerical linear algebra methods utilizing mixed-precision arithmetic. *International J. High Performance Computing Applications*, 35(4):344–369, 2021.

[2] P. Amestoy, A. Buttari, N. J. Higham, J.-Y. L'Excellent, T. Mary, and B. Vieuble. Five-precision GMRES-based iterative refinement. Technical Report MIMS EPrint: 2021.5, Manchester Institute for Mathematical Sciences, University of Manchester, 2021.

[3] P. Amestoy, A. Buttari, N. J. Higham, J.-Y. L'Excellent, T. Mary, and B. Vieuble. Combining sparse approximate factorizations with mixed precision iterative refinement. *ACM Transactions on Mathematical Software*, 49(1):4:1–4:28, 2023.

[4] P. R. Amestoy, I. S. Duff, and J-Y L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computer methods in applied mechanics and engineering*, 184(2-4):501–520, 2000.

[5] M. Arioli and I. S Duff. Using FGMRES to obtain backward stability in mixed precision. *Electronic Transactions on Numerical Analysis*, 33:31–44, 2009.

[6] M. Benzi. Preconditioning techniques for large linear systems: a survey. *J. of Computational Physics*, 182(2):418–477, 2002.

[7] M. Benzi, J. K. Cullum, and M. Tůma. Robust approximate inverse preconditioning for the conjugate gradient method. *SIAM J. on Scientific Computing*, 22(4):1318–1332, 2000.

[8] M. Benzi, C. D. Meyer, and M. Tůma. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM J. on Scientific Computing*, 17(5):1135–1149, 1996.

[9] A. Buttari, J. Dongarra, and J. Kurzak. Limitations of the Playstation 3 for high performance cluster computing. Technical Report MIMS EPrint: 2007.93, Manchester Institute for Mathematical Sciences, University of Manchester, 2007.

[10] 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*, 34:17:1–17:22, 2008.

[11] 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(4):457–466, 2007.

[12] E. Carson and N. J. Higham. A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems. *SIAM J. on Scientific Computing*, 39(6):A2834–A2856, 2017.

[13] E. Carson and N. J. Higham. Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM J. on Scientific Computing*, 40(2):A817–A847, 2018.

[14] E. Carson and N. Khan. Mixed precision iterative refinement with sparse approximate inverse preconditioning. *SIAM J. on Scientific Computing*, 2023. To appear.

[15] T. F. Chan and H. A. van der Vorst. Approximate and incomplete factorizations. In *Parallel Numerical Algorithms, ICASE/LaRC Interdisciplinary Series in Science and Engineering IV. Centenary Conference, D.E. Keyes, A. Sameh and V. Venkatakrishnan, eds.*, pages 167–202, Dordrecht, 1997. Kluver Academic Publishers.

[16] H. Fang and D. P. O'Leary. Modified Cholesky algorithms: a catalog with new approaches. *Math. Program.*, 115(2, Ser. A):319–349, 2008.

[17] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization.* Academic Press, London and New York, 1981.

[18] A. Greenbaum. Estimating the attainable accuracy of recursively computed residual methods. *SIAM J. on Matrix Analysis and Applications*, 18:535–551, 1997.

[19] M. J. Grote and T. Huckle. Parallel preconditioning with sparse approximate inverses. *SIAM J. on Scientific Computing*, 18(3):838–853, 1997.

[20] N. J. Higham and T. Mary. Mixed precision algorithms in numerical linear algebra. *Acta Numerica*, 31:347–414, 2022.

[21] N. J. Higham and S. Pranesh. Simulating low precision floating-point arithmetic. *SIAM J. on Scientific Computing*, 41(5):C585–C602, 2019.

[22] N. J. Higham and S. Pranesh. Exploiting lower precision arithmetic in solving symmetric positive definite linear systems and least squares problems. *SIAM J. on Scientific Computing*, 43(1):A258–A277, 2021.

[23] J. D. Hogg and J. A. Scott. A fast and robust mixed precision solver for sparse symmetric systems. *ACM Transactions on Mathematical Software*, 37(2):1–24, 2010. Article 4, 19 pages.

[24] HSL. A collection of Fortran codes for large-scale scientific computation, 2023. `http://www.hsl.rl.ac.uk`.

[25] D. Hysom and A. Pothen. Efficient parallel computation of ILU(k) preconditioners. In *SC '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing, Portland, OR*, pages 1–29. ACM/IEEE, 1999.

[26] D. S. Kershaw. The incomplete Cholesky-conjugate gradient method for the iterative solution of systems of linear equations. *J. of Computational Physics*, 26:43–65, 1978.

[27] J. Kurzak and J. Dongarra. Implementation of mixed precision in solving systems of linear equations on the cell processor. *Concurrency and Computation: Practice and Experience*, 19(10):1371–1385, 2007.

[28] C.-J. Lin and J. J. Moré. Incomplete Cholesky factorizations with limited memory. *SIAM J. on Scientific Computing*, 21(1):24–45, 1999.

[29] N. Lindquist, P. Luszczek, and J. Dongarra. Improving the performance of the GMRES method using mixed-precision techniques. In *Driving Scientific and Engineering Discoveries Through the Convergence of HPC, Big Data and AI: 17th Smoky Mountains Computational Sciences and Engineering Conference, SMC 2020, Oak Ridge, TN, USA, August 26–28, 2020, Revised Selected Papers 17*, pages 51–66. Springer, 2020.

[30] N. Lindquist, P. Luszczek, and J. Dongarra. Accelerating restarted GMRES with mixed precision arithmetic. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):1027–1037, 2021.

[31] T. A. Manteuffel. An incomplete factorization technique for positive definite linear systems. *Mathematics of Computation*, 34:473–497, 1980.

[32] J. A. Meijerink and H. A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Mathematics of Computation*, 31(137):148–162, 1977.

[33] D. Ruiz. A scaling algorithm to equilibrate both rows and columns norms in matrices. Technical Report RAL-TR-2001-034, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England, 2001.

[34] D. Ruiz and B. Uçar. A symmetry preserving algorithm for matrix scaling. Technical Report INRIA RR-7552, INRIA, Grenoble, France, 2011.

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

[36] Y. Saad. *Iterative Methods for Sparse Linear Systems.* SIAM, Philadelphia, PA, second edition, 2003.

[37] J. A. Scott and M. Tůma. `HSL_MI28`: an efficient and robust limited-memory incomplete Cholesky factorization code. *ACM Transactions on Mathematical Software*, 40(4):24:1–19, 2014.

[38] M. Zounon, N. J. Higham, C. Lucas, and F. Tisseur. Performance impact of precision reduction in sparse linear systems solvers. *PeerJ Computer Science*, 8:e778:1–22, 2022.