Preprint
RAL-P-2012-005

Achieving bit compatibility in sparse direct solvers

JD Hogg, JA Scott

October 2012

Submitted for publication in Parallel Computing

ISSN 1361- 4762

Achieving bit compatibility in sparse direct solvers

Jonathan D. Hogg and Jennifer A. Scott[1]

ABSTRACT

In some applications that reply on the numerical solution of linear systems it can be important that the computed results are reproducible. When designing a parallel sparse direct solver the goal of efficiency potentially conflicts with that of achieving bit-by-bit identical results. This paper focuses on two approaches to achieving bit compatibility independently of the number of processors the solver is run on.

The first, use of a fixed summation order, is demonstrated as a practical solution for multifrontal solvers. This is due to the low number of summands involved in the multifrontal assembly operations.

The second is based on using extended precision. An analysis is presented that demonstrates that the use of extended precision alone is insufficient to ensure bit compatibility because of rounding discontinuities. An algorithm is presented to detect possibly problematic summations, allowing alternate approaches to be used only when needed. The large number of summands encountered in supernodal algorithms makes them suitable for extended precision summation.

A multifrontal solver and a supernodal solver from the HSL mathematical software library are used to explore the performance and feasibility of the two approaches on linear systems arising from practical applications.

Keywords: sparse linear systems, direct solver, multifrontal, supernodal, parallel, bit compatible.

AMS(MOS) subject classifications: 65F05, 65F50, 65Y05

---

[1] Scientific Computing Department, STFC Rutherford Appleton Laboratory,
Harwell Oxford, Oxfordshire, OX11 0QX, UK.
Emails: jonathan.hogg@stfc.ac.uk and jennifer.scott@stfc.ac.uk
Work supported by EPSRC grant EP/I013067/1.

October 11, 2012

1 Background and motivation

In recent years, there has been significant interest in the development of parallel sparse direct linear solvers. In some applications, users want reproducibility in the sense that two runs of the solver with identical input data should produce identical output, that is, users want the matrix factorization and solution to be bit-by-bit identical. This bit compatibility is sometimes called deterministic behaviour. For sequential solvers, achieving bit compatibility is not a problem. But the request for reproducibility is not always met by a parallel solver and this can be an issue. Expert users will understand the methods used, why a solver may produce results that are not bit compatible and how to deal with this. Indeed, non-reproducible results could be seen as a positive feature since they require the user to consider whether the results are as expected. Nevertheless, bit compatibility is a useful to aid in debugging by making problems reproducible. Moreover, the solver will frequently be employed within a larger package, possibly composed of subroutines from many different areas and written by many different software developers. In such cases, the end user will neither understand nor control the solver and consequently may not know how to handle the results and cope with a lack of bit compatibility. Further, some industries (e.g. nuclear, finance) require reproducible results to satisfy regulatory requirements.

One example where lack of bit compatibility of the solver can have a marked effect is in the solution of large-scale non-linear optimization problems. The Ipopt [9] package is designed to solve large-scale non-linear optimization problems of the form

$$\begin{aligned} \min_{x\in\mathbb{R}^k} \quad & f(x) \\ \text{s.t.} \quad & g_L \le g(x) \le g_U, \\ & x_L \le x \le x_U, \end{aligned}$$

where $f(x) : \mathbb{R}^k \to \mathbb{R}$ is the objective function and $g(x) : \mathbb{R}^k \to \mathbb{R}^l$ are the constraint functions. Ipopt implements a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. At each stage, this involves solving an indefinite sparse symmetric linear system of the form

$$\begin{pmatrix} W + \Sigma + \delta_w I & B \\ B^T & -\delta_c I \end{pmatrix} \begin{pmatrix} d^x \\ d^\lambda \end{pmatrix} = \begin{pmatrix} b^x \\ b^\lambda \end{pmatrix}, \tag{1.1}$$

where δ_w and δ_c are chosen such that the matrix has inertia $(k, l, 0)$. W and B are the Hessian of $f(x)$ and the Jacobian of $g(x)$, respectively, evaluated at the current trial point and Σ is a diagonal matrix. In tests with Ipopt, we have observed that if a solver that does not guarantee bit compatibility is used, different paths may be followed on two runs with identical input data, possibly resulting in a different solution being returned or, in the extreme case, a solution being successfully computed on only one of the runs. This will potentially cause the application developer significant difficulties, not least of which will be debugging the software that depends on Ipopt and explaining to customers why different runs on identical data produce different outcomes.

A number of parallel sparse direct solvers, including PARDISO [8] and WSMP [2], yield bit-compatible results on runs with the same number of processors, but not when run on different numbers of processors. Our interest is in developing sparse direct solvers that produce bit-compatible results, independently of the number of processors used.

In this paper, we explore the feasibility of two possible approaches to achieving bit compatibility. The first considers the feasibility of using a fixed ordering for summation under different update techniques. In the second approach, we consider the use of extended precision, demonstrating that high accuracy alone is insufficient to guarantee bit compatibility because of the presence of rounding discontinuities. We propose a procedure to identify and circumvent this issue when it occurs.

The layout of the paper is as follows. The basic summation problem is described in Section 2. Section 3 examines its embedding within sparse direct solvers and the applicability of ordered summation as a solution. Section 4 considers the use of extended precision as an alternative, presenting an analysis of when it breaks down and how to detect such an event. An exploration of practical performance is given in Section 5 and conclusions are presented in Section 6.

2 The bit-compatible summation problem

The critical issue for bit compatibility within a linear solver is the way in which N numbers (or, more generally, matrices) are assembled

$$sum = \sum_{j=1}^{N} s_j,$$

where the s_j are computed using one or more processors. The assembly is commutative but, because of the potential rounding of the intermediate results, is not associative so that the result *sum* depends on the order in which the s_j are assembled. Typically, subtasks that return the s_j values will complete in different orders on different runs due to interference from other processes and other environmental conditions. If the assembly operation takes the s_j in the order in which they become available, the value of *sum* will vary on each run. To prevent this, it is sufficient to wait to start the assembly until all the s_j are available and then to assemble them in their natural order (or in any other deterministic way). This ensures all the summations are performed in the same order, regardless of how much time each processor takes to perform the subtask(s) it has been assigned. The overhead for achieving reproducibility is the additional waiting times that are required to artificially synchronize the subprocesses. The overall increase in the total runtime can be significant if many assembly operations are needed.

For a chosen number of processors P, it is possible to produce a fixed schedule for each processor such that summations are always performed in the same order. Such a schedule can be executed on a differing number of processors p by running on exactly P threads. If $p < P$, jitter is likely to render the schedule inefficient, while if $p > P$, $p - P$ processors are idle so again the schedule is inefficient. A limited version of this technique is offered by PARDISO (Version 4.1.2). Because of the inflexibility of this approach, particularly when moving between machines of different size, we seek an alternative methodology. This must either achieve bit compatibility by transforming summations so that they can be subdivided independently of the number of processors or by performing summations in such a way that the result is bit compatible regardless of the ordering.

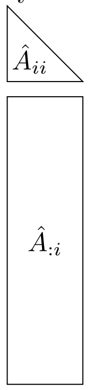
3 Ordered summation in a sparse direct solver

3.1 Sparse direct algorithm

The following framework describes the factorization phase of a generic supernode-orientated sparse Cholesky solver; it is sufficiently general that we can describe both supernodal and multifrontal approaches within the same framework. We assume here and throughout the remainder of the paper that the system matrix is denoted by A and that it is of order n. The required input to the factorization phases is an assembly tree that describes the data flow dependencies arising from the structure of the matrix factors. The assembly tree, which is computed by the analyse phase of the solver, is a directed graph in which vertices represent nodes (a collection of consecutive matrix columns with the same sparsity pattern) and paths represent data dependence. Each node has at most one outgoing edge, hence the graph is a forest (and a tree if the matrix is irreducible). We refer to the destination of an outgoing edge from node i as the parent of i. The assembly tree is enumerated in a postorder (that is, each node is numbered lower than its parent).

The factorization is normally performed in place. We denote the partially factorized matrix by $\hat{A}$ and initialise it as $\hat{A} = A$. The final factors that overwrite $\hat{A}$ are denoted by L. As $\hat{A}$ is symmetric and L lower triangular, it is sufficient to store only the lower triangular parts. If node i has columns $s_i, s_i + 1, \dots s_{i+1} - 1$, only the non-zeroes in those columns are stored. These columns are conceptually

Figure 3.1: Conceptual layout of data for a node of $\hat{A}$.



split into two parts: a diagonal block constituting rows $s_i, s_i + 1, \ldots s_{i+1} - 1$ (henceforth $\hat{A}_{ii}$), and a rectangular block constituting all remaining non-zero rows (henceforth $\hat{A}_{:i}$). Observe that if i is a root of the assembly tree, the rectangular block is empty. Figure 3.1 illustrates these matrices diagrammatically.

During the factorization, a temporary update matrix U_i is formed. Each entry of this matrix is added in to a destination matrix that is determined by its row and column. Adjacent entries in U_i need not be adjacent in their destination matrix. We refer to this sparse scatter/reduction as a sparse expand-add operation. In a multifrontal approach, part of U_i is relabelled F_i to denote a partial frontal matrix.

With this notation, the outline sparse Cholesky factorization algorithm is as follows:

```
for each node $i$ in postorder do
  Perform the dense Cholesky factorization of the diagonal block: $L_{ii} \leftarrow \text{Cholesky}(\hat{A}_{ii})$.
  Perform the dense triangular solve of the rectangular block using the diagonal block: $L_{:i} \leftarrow \hat{A}_{:i} L_{ii}^{-T}$.
  Form the outer product of the rectangular block with itself to form a temporary matrix: $U_i \leftarrow L_{:i} L_{:i}^T$.
  Either Multifrontal Update:
      For each child $j$ of $i$, assemble the contribution $F_j$ to $U_i$ using sparse expand-add.
      Assemble the columns of $U_i$ that belong to its parent $k$ into $(\hat{A}_{kk}, \hat{A}_{:k})$ using sparse expand-add.
      Relabel the remaining columns of $U_i$ as $F_i$.
  Or Supernodal Update:
      Assemble each column of $U_i$ into the node to which it belongs using sparse expand-add.
end for
```

Note that in some implementations, the matrix U_i is not formed explicitly and the matrix-matrix multiplication $L_{:i} L_{:i}^T$ is performed as part of the sparse expand-add operation.

Within this algorithm, parallelism is typically exploited on two levels:

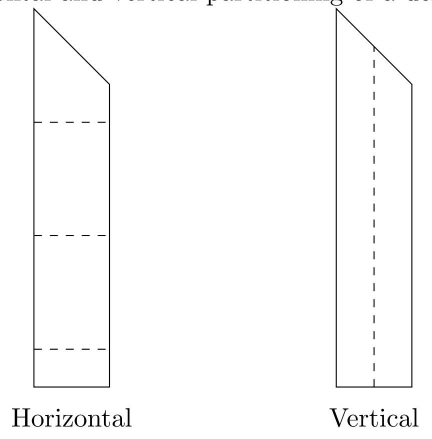
Node level: At each node i, the dense linear algebra operations are parallelized.

Tree level: Multiple nodes of the assembly tree are processed simultaneously so long as all the children of each node have completed. Synchronization of the (multifrontal or supernodal) update operations is required.

3.2 Bit-compatible summations

Before we proceed to a detailed analysis, we remark that it is possible to achieve a bit-compatible solution without having bit-compatible factors L. As the solution contains only n numbers derived from the entirety of L and b, it is quite possible for any differences to be cancelled or lost in rounding during the triangular solves that complete the solution process (these solves must themselves be bit compatible). However, if L is not bit compatible, it is always possible to find some right-hand side b to expose this. We therefore must

Figure 3.2: Horizontal and vertical partitioning of a dense nodal matrix



ensure that the entries of L are bit compatible. The techniques used to ensure bit compatibility during the factorization phase are trivially applied to achieve the same for the solves, hence we omit the details.

We now consider the operations that require care to ensure overall bit compatibility and discuss how to achieve this. For efficiency, the dense linear algebra operations within a node are blocked and operations on an individual block are executed using BLAS routines. Some of these operations can be performed simultaneously on different blocks. Typically the partitions into blocks will be either horizontal or vertical (or a combination of both), as shown in Figure 3.2. To ensure bit compatibility, the following must be considered:

1. **The BLAS calls** must be bit compatible. This can be achieved by choosing a BLAS library that guarantees this (often the serial variant will do so)[1]. However, the block size must not be altered between runs or as the number of processors varies (and so it cannot be tuned for load balancing).

2. **Horizontal partitioning** only affects bit compatibility in so far as it determines the block size used by the BLAS.

3. **Vertical partitioning** is equivalent to splitting the node into two (or more) nodes with the same sparsity pattern, so the techniques discussed below for the inter-nodal sparse expand-add operations apply.

At the inter-nodal (tree-) level, the following must be considered to ensure bit compatibility:

4. **Sparse expand-add operations** have the potential to break bit compatibility. For a given node, i, contributions come either from its children $j_1, j_2, \ldots, j_{nc}$ (multifrontal) or its descendants $j_1, j_2, \ldots, j_{nd}$ (supernodal). These contributions may become available in any postorder of the assembly tree. Hence, to achieve bit compatibility, the sparse expand-add operations must be performed in a bit-compatible fashion.

Table 3.1 shows the values of nc for a variety of matrices. Table 3.2 shows the number of updates to each block of the matrix using a DAG-based supernodal approach [6], and is more representative of inter-nodal dependencies than nd, which bounds it. The problems listed are drawn from the University of Florida Sparse Matrix Collection [1], and use a nested dissection ordering to reduce fill. Problems are listed in increasing order of the number of floating point operations to factorize A, with A varying in size from 75,000 to 1,500,00 with the number of entries in A in the range 300,000–40,000,000.

In the multifrontal approach, at each node i a number of matrices equal to the nc, the number of its children, must be summed. Given that nc is typically small, and that large matrices can be partitioned

[1]Our experiments show that the Intel MKL BLAS are bit compatible, but some other common BLAS libraries are not.

Table 3.1: Distribution of number of children per node for various problems

Problem	0	1	2	3	4	≥5	max
Mulvey/finan512	22611	14658	19115	208	548	193	11
MaxPlanck/shallow_water1	32538	7968	18464	6922	71	4	5
UTEP/Dubcova3	16384	0	16213	85	0	0	3
Nasa/nasasrb	2479	3079	2410	31	2	0	4
CEMW/tmt_sym	183469	126066	171401	5891	95	0	4
Schmid/thermal2	280401	228623	274915	2247	11	0	4
Rothberg/gearbox	5627	7950	5556	28	3	0	4
INPRO/msdoor	20706	6750	13951	2837	360	0	4
DNVS/m_t1	1964	1521	1915	24	0	0	3
McRae/ecology2	442927	39728	159529	92580	32743	2	5
Boeing/pwtk	6681	7416	6463	89	10	2	6
Chen/pkustk13	4184	4862	4108	36	1	0	4
BenElechi/BenElechi1	7361	6811	7247	52	1	0	4
Rothberg/cfd2	10634	15807	10609	12	0	0	3
DNVS/thread	595	663	581	5	1	0	4
DNVS/shipsec8	4037	4470	2967	414	54	13	16
GHS_psdef/crankseg_2	1627	266	1605	9	1	0	4
DNVS/fcondp2	6545	6240	6246	47	68	0	4
Schenk_AFE/af_shell3	22773	12114	22586	93	0	0	3
DNVS/troll	6873	8658	6805	32	1	0	4
GHS_psdef/bmwcra_1	4387	7836	4349	17	1	0	4
GHS_psdef/crankseg_1	1352	428	1340	4	1	0	4
Um/2cubes_sphere	11798	16129	11687	55	0	0	3
GHS_psdef/ldoor	47571	14903	33163	6674	353	0	4
DNVS/ship_003	3616	5037	2885	238	39	12	55
DNVS/fullb	6187	8333	4748	479	101	30	29
Um/offshore	31410	39623	31098	154	1	0	4
GHS_psdef/inline_1	18315	28552	18093	109	1	0	4
Chen/pkustk14	4155	4905	3776	146	7	8	16
GHS_psdef/apache2	309992	25988	59025	29630	27974	24773	7
Koutsovasilis/F1	12062	18430	11847	104	2	0	4
Oberwolfach/boneS10	41028	50972	39967	521	6	0	4
AMD/G3_circuit	636810	113742	317266	127064	21660	107	7
JGD_Trefethen/Trefethen_20000	1649	3738	987	241	46	10	6
ND/nd24k	346	7597	345	0	0	0	2

Table 3.2: Distribution of number of block dependencies in `HSL_MA87` for various problems

Problem	0	1	2	3	4	≥ 5	max
Mulvey/finan512	22611	13449	6517	1325	5006	8425	1539
MaxPlanck/shallow_water1	32538	6399	6386	5065	3541	12044	2092
UTEP/Dubcova3	16384	1	5212	2717	1082	7299	746
Nasa/nasasrb	2481	1102	1023	630	514	2325	401
CEMW/tmt_sym	183469	72449	49364	39367	30771	111627	1675
Schmid/thermal2	280403	122737	80632	68273	53331	181016	1681
Rothberg/gearbox	5627	3605	2151	1480	1235	5319	535
INPRO/msdoor	20706	4714	3464	2578	2131	11223	385
DNVS/m_t1	1965	888	619	339	247	1676	215
McRae/ecology2	442927	33471	15562	36855	45253	193644	2122
Boeing/pwtk	6681	4367	1740	1398	1062	5768	388
Chen/pkustk13	4184	1820	1166	1022	871	4398	524
BenElechi/BenElechi1	7367	4669	1802	1111	965	5881	289
Rothberg/cfd2	10634	7820	4235	2673	2246	9859	872
DNVS/thread	595	240	245	162	97	827	250
DNVS/shipsec8	4037	2756	1176	1108	721	2557	488
GHS_psdef/crankseg_2	1631	189	531	246	125	1399	324
DNVS/fcondp2	6545	3395	1859	1871	923	4997	555
Schenk_AFE/af_shell3	22785	5008	6983	4033	4593	14827	372
DNVS/troll	6873	4181	2034	2048	1421	6405	622
GHS_psdef/bmwcra_1	4402	3342	1938	1543	1049	5059	486
GHS_psdef/crankseg_1	1353	224	486	195	114	1185	394
Um/2cubes_sphere	11798	5487	4058	3332	2706	12861	2283
GHS_psdef/ldoor	47571	10315	8067	6117	5160	26337	467
DNVS/ship_003	3626	2289	1218	1067	689	3646	411
DNVS/fullb	6187	4047	2060	2186	1402	4857	569
Um/offshore	31410	14456	10166	8276	6594	32308	1994
GHS_psdef/inline_1	18318	10321	8049	6141	4564	19250	568
Chen/pkustk14	4184	2025	1443	919	853	4849	969
GHS_psdef/apache2	309992	21166	9690	6625	9187	122154	2143
Koutsovasilis/F1	12065	6706	4673	3626	2790	14558	754
Oberwolfach/boneS10	41029	17661	16770	10640	8581	40363	468
AMD/G3_circuit	636810	78068	90758	85720	62685	263231	3530
JGD_Trefethen/Trefethen_20000	1649	409	247	160	112	4094	6670
ND/nd24k	346	259	268	238	229	11498	1974

such that summations can be safely performed in parallel, the overhead of enforcing a defined order of the summation is relatively small. However, for each data block in the supernodal approach, a number of matrices equal to the block dependencies are summed. Given the relatively large numbers (several thousand) for many nodes, an enforced order is likely to prove detrimental to efficiency.

4 Local use of extended precision

One suggested alternative to enforcing a predefined ordering on a summation is to use extended precision [3]. However, this alone is not sufficient to guarantee even most significant digit reproducibility. The key difference between accuracy and reproducibility is rounding.

Let us denote numbers in the desired precision using a subscript d and those in the extended precision using a subscript e. Conversions will be denoted using subscripted brackets.

To illustrate the problems, take the following example with d having a 3 decimal digit mantissa, and e having a 6 decimal digit mantissa, using round to nearest, ties away from zero:

$$(1.499 \times 10^0)_e + \big((4.990 \times 10^{-4})_e + (5.000 \times 10^{-7})_e\big) = 1.499500_e = 1.500_d \tag{4.2}$$

$$\big((1.499 \times 10^0)_e + (4.990 \times 10^{-4})_e\big) + (5.000 \times 10^{-7})_e = 1.499499_e = 1.499_d \tag{4.3}$$

Both summations differ only in associativity and results are accurate to the desired precision, however they have only a single digit in common.

This problem can be safely ignored if we are certain the extended precision result does not lie close to the rounding discontinuity of the $e \to d$ conversion. If the maximum error in the summation sum is known to be at most δ, we require $(sum_e + \delta_e)_d = (sum_e - \delta_e)_d$. If this equality does not hold, the summation in extended precision may not yield a bit-compatible result in the desired precision.

The difficulty clearly lies in determining a good estimate for δ. Interval arithmetic could be used, but such analysis is very heavyweight for the purpose at hand. We therefore use a related but simpler technique: a single additional extended precision number is used to estimate the error when adding each summand. The error can be estimated using similar techniques to those used in compensated summation (see the summation chapter of [5]). If $|a| > |b|$ and

$$\begin{aligned} sum &= (a+b)_e\,, \\ corr &= b - (sum - a), \end{aligned}$$

then $sum + corr$ is exactly $a + b$, with $corr$ containing the least significant bits of the result. For a given summation, δ is at most the sum of $|corr|$ for each individual addition performed.

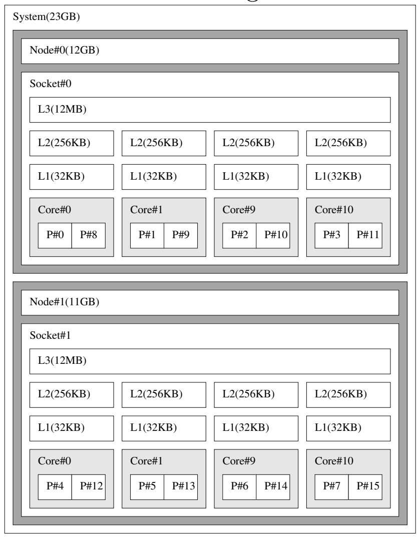
Accuracy can be further improved by using the full algorithm for compensated summation, and the correction term $corr$ can still be used to determine a suitable $\delta = \delta^*$. This requires no additional storage, but some additional floating-point operations.

If the result of a summation is equally likely to be any extended precision number, the probability that it lies within a small δ of a rounding discontinuity is small. For example, if d is IEEE double precision and e is IEEE quad precision, there are approximately 2^{60} quad precision numbers for each double precision number. Given, for example, $\delta = 2^{-110}$ (i.e. sum is accurate to the last few binary digits of the quad precision) the probability is $2^{-50} \approx 9 \times 10^{-16}$.

However, empirical experiments reveal that the assumption of uniform probability distribution on sum is invalid. Using the supernodal sparse direct solver `HSL_MA87` [6], we observed that for most problems, 5-10% of all the summations involved yield a result within δ^* of a rounding discontinuity. To illustrate why this is so, consider the summation $(1.000 \times 10^0 + 1.005 \times 10^{-1})_d$. As the exponents of the summands differ only by one, the final digit of 1.005×10^{-1} causes an exact coincidence of the sum with the rounding discontinuity.

To overcome this, an arbitrary small number τ may be added to the mantissa of sum in extended precision before conversion to the desired precision. If the number is less than ϵ_d (where $\epsilon_d =$

Figure 5.3: Description of the machine `mitchell`.



	`mitchell`
Processor	2 × Intel Xeon E5620
Physical Cores	8
Memory	24GB
BLAS	MKL 10.3.0

Compilers:

```
ifort -O3 -xSSE4.1 -no-prec-div -ip
gfortran-4.6 -O3 -msse3 -march=native
```

$\inf\{\epsilon : (1.0)_d \neq (1.0+\epsilon)_d\}$), the result will be accurate to double precision. We repeated our experiments with $\tau = 0.2345 \times 2^{exponent(\epsilon_d)}$ (where $exponent(\epsilon_d)$ means the floating-point exponent of d in IEEE double precision), and found no *sum* values lay within δ^* of a rounding discontinuity. As τ is added to the mantissa of *sum*, caution may be required to maintain bit compatibility when the mantissa of *sum* lies close to 0 or the radix.

5 Results

To assess the impact of the algorithms discussed in this paper on the performance of a direct solver, we conduct tests on the 8-core machine detailed in Figure 5.3. Results are given for both the gfortran and ifort compilers, as some implementations perform significantly better under one compiler rather than another.

We are not able to compare the methods of Sections 3 and 4 within the same solver (to do so would require us to develop of a completely new sparse direct solver and this is an objective that is far beyond the scope of this paper). Instead, we compare a multifrontal solver and a supernodal solver written by the same team; both are written in Fortran 95 and use OpenMP and both use the same code to perform the analysis phase.

`HSL_MA87` **[6]** is a dedicated sparse Cholesky solver with a *supernodal* update. It utilises a DAG-based runtime approach to fully exploit parallelism within the factorization. It is optimised for large systems. Version 2.1.0 is modified to conduct the experiments.

`HSL_MA97` **[7]** is primarily written as a sparse symmetric indefinite solver, but also offers a Cholesky factorization with a *multifrontal* update. It was carefully designed from the ground up to be a bit-compatible solver using the techniques described in Section 3. It is optimised for small and medium systems. Version 2.0.0 is used in our experiments.

As the codes are designed and tuned for different problems, they are not directly comparable. However, such a comparison does provide an upper bound on the overheads of the approaches of Sections 3 and 4.

To accommodate the summation in local precision, minimal modifications are made to `HSL_MA87` (a more considered implementation could potentially be more efficient, for example, in its memory management). Two different extended precision implementations are tested. The first is the double-double algorithm of the `libqd` package by Hida, Li and Bailey [4] that approximates quad precision using two doubles. The second is the fully accurate IEEE implementation of quad precision offered by the Fortran compilers using

Table 5.3: Comparison of performance overhead of ensuring bit compatibility. Ratios are given of time to complete divided by time for the original bit-incompatible `HSL_MA87` to complete. Brackets () are used in the overhead columns to denote lack of bit compatibility. d-d and quad denote double-double and quad precision respectively. Asterisks * indicate that at some stage during the calculation, a value of *sum* was encountered that lay within within δ^* of a rounding discontinuity.

	gfortran				ifort			
	`HSL_MA97`	`HSL_MA87`	`HSL_MA87`	`HSL_MA87`	`HSL_MA97`	`HSL_MA87`	`HSL_MA87`	`HSL_MA87`
Problem		overhead	d-d	quad		overhead	d-d	quad
Mulvey/finan512	0.56	(1.10)	1.49	2.76	0.42	(1.21)	1.81*	2.23
MaxPlanck/shallow_water1	1.27	(1.09)	1.13	1.22	0.52	(1.30)	1.73*	1.87
UTEP/Dubcova3	0.86	(1.20)	2.02*	4.86	0.82	(1.28)	2.50*	2.90
Nasa/nasasrb	1.21	(1.21)	2.22*	6.23	0.90	(1.03)	2.44*	3.50
CEMW/tmt_sym	0.48	(1.09)	1.41	2.53	0.44	(1.23)	1.86*	1.98
Schmid/thermal2	0.45	(1.10)	1.50	2.70	0.42	(1.28)	1.94*	2.04
Rothberg/gearbox	1.52	(1.15)	1.98	5.75	1.32	(1.12)	2.65*	3.42
INPRO/msdoor	1.17	(1.20)	2.21	5.93	0.95	(1.18)	2.68*	3.40
DNVS/m_t1	1.47	(1.11)	1.88	5.29	1.40	(1.10)	2.51*	3.16
McRae/ecology2	0.49	(1.23)	1.53	2.53	0.41	(1.26)	1.74*	1.80
Boeing/pwtk	1.20	(1.10)	2.07	5.63	1.23	(1.11)	2.60*	3.34
Chen/pkustk13	1.52	(1.06)	1.78	4.94	1.46	(1.08)	2.28*	3.08
BenElechi/BenElechi1	1.32	(1.09)	1.94	5.23	1.36	(1.15)	2.65*	3.38
Rothberg/cfd2	1.32	(1.12)	1.89	4.57	1.33	(1.12)	2.43*	2.97
DNVS/thread	1.90	(1.04)	1.57*	3.86*	1.77	(1.02)	1.93*	2.64*
DNVS/shipsec8	1.69	(1.00)	1.61*	4.09*	1.52	(1.10)	2.14*	2.77*
GHS_psdef/crankseg_2	1.38	(1.07)	1.73	4.42	1.34	(1.05)	2.09*	2.67
DNVS/fcondp2	1.48	(1.10)	1.76	5.00	1.24	(1.00)	1.99*	2.67
Schenk_AFE/af_shell3	1.35	(1.14)	1.98*	5.19	1.10	(1.17)	2.51*	2.99
DNVS/troll	1.14	(1.19)	1.41	3.79	1.21	(1.09)	2.19*	2.90
GHS_psdef/bmwcra_1	1.47	(1.12)	1.86*	4.82*	1.30	(1.09)	2.28*	2.92*
GHS_psdef/crankseg_1	1.57	(1.06)	1.74	4.64	1.48	(1.03)	2.25*	2.81
Um/2cubes_sphere	1.61	(1.07)	1.60	3.72	1.30	(1.08)	1.88*	2.46
GHS_psdef/ldoor	1.30	(1.14)	1.95	5.06	1.06	(1.14)	2.38*	3.07
DNVS/ship_003	1.57	(1.06)	1.68*	4.24*	1.51	(1.07)	2.03*	2.65*
DNVS/fullb	1.88	(1.05)	1.65	4.46	1.62	(1.06)	1.99*	2.58
Um/offshore	1.24	(1.07)	1.67	3.89	1.15	(1.08)	2.02*	2.57
GHS_psdef/inline_1	1.52	(1.10)	1.86	4.86	1.16	(1.08)	2.30*	3.06
Chen/pkustk14	1.55	(1.06)	1.71	4.47	1.32	(1.05)	2.03*	2.70
GHS_psdef/apache2	1.26	(1.07)	1.63	3.50	1.01	(1.12)	1.85*	2.08
Koutsovasilis/F1	1.53	(1.07)	1.69	4.11	1.11	(1.01)	1.93*	2.50
Oberwolfach/boneS10	1.59	(1.08)	1.75*	4.26*	1.17	(1.04)	2.01*	2.62*
AMD/G3_circuit	0.63	(1.07)	1.50	2.92	0.50	(1.06)	1.79*	1.93
JGD_Trefethen/Trefethen_20000	3.12	(1.00)	2.19	10.75	1.40	(1.00)	2.29*	4.16
ND/nd24k	2.10	(1.00)	1.67	4.24*	1.44	(1.00)	1.87*	2.73

`real(selected_real_kind(32))`. As the latter aims for full quad precision accuracy, it requires more operations than the double-double approach, but uses a similar amount of storage.

Table 5.3 shows the relevant efficiency of the ordered summation multifrontal method and two extended precision supernodal implementations. For each problem, the ratios of the factorization phase runtimes against that of the unmodified supernodal code are given. The calculation of δ^* and detection of proximity to a rounding discontinuity are disabled in our experiments (although the addition of τ as discussed at the end of Section 4 is performed). An asterisk indicates a rounding discontinuity was detected; for these bit compatibility is not guaranteed. However, in each case, for a series of three runs with the right-hand side corresponding to an exact solution $x = 1$, the computed solution was observed bit compatible. The entries in the columns 3 and 7 labelled "overhead" are the ratios of the time for the modified `HSL_MA87` with $d = e =$ double to the original `HSL_MA87` time. These estimate the overhead of the inefficient memory management employed, and have been rounded to 1.00 where the difference is negligible.

It is clear that the double-double implementation is significantly faster than the quad precision implementation of extended precision (particularly in the case of the gfortran compiler). However, in our tests there were fewer problems for which quad precision requires additional calculations to ensure bit-compatibility. After allowing for the possible inefficient implementation of local precision within `HSL_MA87`, our experiments have not demonstrated a clear winner between the Section 3 and Section 4 approaches. For most of our test problems, `HSL_MA97` dominates but as the problem size increases, `HSL_MA87` becomes more competitive. While this is likely an inherent property of the codes, it is encouraging that imposing bit compatibility on the multifrontal approach has apparently not seriously effected its efficiency.

6 Conclusions

We have presented two approaches of achieving bit compatibility within a sparse direct solver. The traditional approach of ordered summation performs well, but the use of extended precision offers a viable alternative. The existence of rounding discontinuities requires special care is taken in extended precision. We have developed a procedure to detect when this extra care might be required, and have demonstrated that addition of a small number to the extended precision sum significantly reduces the incidence of this problem.

Our exploratory results suggest that, on current machines, the cost of using extended precision is significant but not prohibitive. Furthermore, our fixed order summation multifrontal solver provides good performance, particularly on our smallest and medium-sized problems. It is likely that as the balance between the cost of floating-point arithmetic and other computational resources changes, the use of extended precision will become justified to allow more parallelism to be exposed without sacrificing bit compatibility.

Further work on an efficient implementation of our detection techniques would prove very useful to the development of future sparse direct solvers as well as to other areas of linear algebra.

References

[1] T. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Transactions on Mathematical Software, 38 (2011).

[2] A. Gupta, M. Joshi, and V. Kumar, *WSMP: A high-performance serial and parallel sparse linear solver*, Technical Report RC 22038 (98932), IBM T.J. Watson Research Center, 2001. `http://www.cs.umn.edu/∼agupta/doc/wssmp-paper.ps`.

[3] Y. He and C. Ding, *Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications*, The Journal of Supercomputing, 18 (2001), pp. 259–277.

[4] Y. Hida, X. Li, and D. Bailey, *Algorithms for quad-double precision floating point arithmetic*, in Computer Arithmetic, 2001. Proceedings. 15th IEEE Symposium on, 2001, pp. 155–162.

[5] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM, 1996.

[6] J. D. Hogg, J. K. Reid, and J. A. Scott, *Design of a multicore sparse Cholesky factorization using DAGs*, SIAM Journal on Scientific Computing, 32 (2010), pp. 3627–3649.

[7] J. D. Hogg and J. A. Scott, `HSL_MA97`*: a bit-compatible multifrontal code for sparse symmetric systems*, Technical Report RAL-TR-2011-024, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England, 2011.

[8] O. Schenk and K. Gärtner, *Solving unsymmetric sparse systems of linear equations with PARDISO*, Journal of Future Generation Computer Systems, 20 (2004), pp. 475–487.

[9] A. Wachter and L. T. Biegler, *On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming*, Mathematical Programming, 106(1) (2006). 25–57.