Science & Technology Facilities Council

Technical Report
RAL-TR-2016-017

A new sparse LDLT solver using a Posteriori Threshold Pivoting

J Hogg

November 2016

ISSN 1358-6254

A new sparse LDL^T solver using A Posteriori Threshold Pivoting

Jonathan Hogg

ABSTRACT

We describe version 2.0 of the SPRAL Sparse Symmetric Indefinite Direct Solver (SSIDS). Version 1 performed the factorization only on a single GPU. The new version adds support for CPU and hybrid execution, capable of utilising multiple GPUs.

Notably, the CPU algorithm uses OpenMP 4.0 tasks to implement a multifrontal algorithm with dense factorizations using a posteriori threshold pivoting based on the same threshold technique as traditional threshold partial pivoting.

Keywords: sparse symmetric factorization, indefinite systems, a posteriori pivoting, OpenMP

AMS(MOS) subject classifications: 65F30, 65F50

Scientific Computing Department, STFC Rutherford Appleton Laboratory, Harwell Campus, Oxfordshire, OX11 0QX, UK.

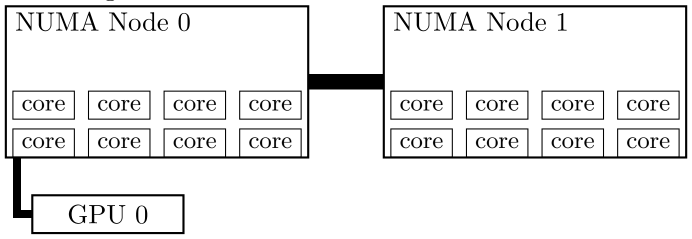
This work was supported in part by the NLAFET Project funded by the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement 671633.

It has also been published as NLAFET Working Note 6.

November 28, 2016

Contents

Figure 1: Architecture for GPU test machine



1 Introduction

In 2014 we released version 1.0 of SSIDS that provided a sparse direct solver to solve $Ax = b$ through a symmetric indefinite factorization $A = LDL^T$. It is described in [4]. This work was limited to a single GPU, and could not make use of the CPU. In this paper we describe version 2.0 of SSIDS. This adds a new CPU implementation that is faster than existing implementations without using the GPU, but is also able to use the GPU code from version 1.0 to perform heterogeneous computation on the CPU and one or more GPUs.

The CPU implementation uses the same sort of a posteriori threshold pivoting (APTP) algorithm as our GPU implementation, but builds it on top of the OpenMP 4.0 tasking system. It combines this with a fail-in-place technique to ensure all permutations during the dense factorization are local. We demonstrate that the combination provides considerably better performance than existing techniques.

Whilst we could have used the more capable runtime systems provided by, for example, StarPU [1], this would add additional dependencies to the SPRAL mathematical software library and may cause problems for portability, for example if the user wishes to use a conflicting runtime system at a higher level. We therefore restrict ourselves to OpenMP 4.0 and CUDA as the mechanisms for expressing parallelism. The code is written in a mixture of Fortran, C++ and CUDA.

The paper is organised as follows. Section 2 describes high level parallelization that splits the factorization into one or more subtrees that are assigned to available resources (CPUs or GPUs) using a traditional method. These parts are then combined using only the CPU. Section 3 describes the design of the CPU implementation, and its memory management techniques that prove crucial to its performance. This calls the dense factorization kernels described in Section 4. Theoretical results supporting the use of full pivoting at the finest granularity are given in Section 5. Finally, numerical results are given in Section 6 and conclusions in Section 7.

2 High-level parallel strategy

We consider a machine to consist of one or more NUMA nodes, each of which may have one or more GPUs attached. This is illustrated by our main GPU test machine, pictured in Figure 1. We use the term resource to mean either all the cores of a NUMA node, or a single GPU. Often the communications between NUMA nodes are sufficiently fast that it is beneficial to consider the cores across all NUMA nodes as a single resource (this is our default).

OpenMP task parallelism cannot easily express task dependencies between different resources, for the following reasons:

- Different NUMA regions have to use different OpenMP parallel regions with a `proc_bind` clause to ensure task execution on the desired region. However, OpenMP tasks bind to the nearest parallel region or task group, and task dependencies are not considered with other task binding regions, so we cannot use tasks to synchronise between different teams on different regions. We could use non-task synchronisations such as OpenMP locks or POSIX mutexes. However, this could lead to tasks sitting idle longer than necessary.

- We could (in theory) include tasks that just wait on GPU events, but again this may lead to tasks idling longer than necessary. Combined with the level-set rather than task-based design of the existing GPU code, mixing the two modes was considered a bad idea.

Even if such parallelism was easy to express, inter-resource communication is more expensive than intra-resource communication, so should be minimized. This is achieved through tree parallelism. The assembly tree is split into one or more leaf subtrees per resource. The remaining root subtree(s) are then executed solely on the combined CPU resources, because (at present) the GPU code does not support external contributions. In the future it may make sense to offload large dense matrix operations to the GPU.

Assignment of leaf subtrees to resources is done in a round robin fashion. First, the leaf subtrees are ordered in descending order of floating-point operation count. The largest subtree is assigned to the fastest resource, the next largest subtree to the next fastest resource, and so on until all subtrees are assigned. If there are more subtrees than resources, we loop back around again.

We define the load balance as:

$$\mathit{balance} = \max_i \frac{n_{res} x_i/\alpha_i}{\sum_j (x_j/\alpha_j)}$$

where n_{res} is the number of resources, x_i is the total number of floating point operations assigned to resource i and α_i is proportional to the speed of the resource (eg flop/s). In a perfectly balanced situation, $\mathit{balance} = 1.0$. In our implementation, we treat all NUMA regions as having $\alpha = 1.0$, and all GPUs as having the same value relative to that (e.g. if the GPUs are twice as fast as the total cores of a NUMA region we set $\alpha = 2.0$ for the GPUs). Furthermore, we prevent scheduling of subtrees on the GPU with size less than some minimum, gpu_{min}, because a minimum amount of work is needed for sufficient parallelism to exist and to compensate for communication and start-up latencies.

We define b_{min} to be the minimum value of *balance* we are willing to accept and use Algorithm 1 to find a subtree partition that meets it. Note the existence of a maximum iteration count, for we do not wish the root subtree to grow too large.

Algorithm 1 `find_subtree_partition()`

```
Initialise partition as set of independent subtrees in assembly forest.
Assign subtrees to resources and calculate balance.
while  $balance > b_{\min}$ and # iterations < max_itr  do
  Find largest subtree in partition.
  Move root node of subtree to root subtree, creating a new subtree from each child.
  If there are now insufficient subtrees of size at least $\text{gpu}_{\min}$, exit
  Assign subtrees to resources and calculate balance.
end while
Return partition with least balance value encountered.
```

With the subtree partition and assignment determined, data structures are set up on each resource. The CPU and GPU implementations are both fully self contained, allowing different underlying data structures (and factorizations if desired). They communicate only through contribution block objects that provide a means to access the contribution block from the root of a subtree to its parent, and a way to free the underlying memory of that block when it is no longer required.

We use the OpenMP `proc_bind()` directive to bind threads to the desired NUMA nodes, and rely on a first-touch policy to ensure memory is allocated in the appropriate locations. Synchronisations are done by locating the leaf subtree and root subtree calculations in different OpenMP parallel regions: one must complete before the other can start. Each GPU gets its own CPU thread (which will be largely idle) that waits on a CUDA event before exiting the leaf subtree parallel region. We thus have a construct that looks similar to that shown in Algorithm 2.

Algorithm 2 High-level OpenMP structure

```
! Leaf subtrees
!$OMP parallel proc_bind(spread) num_threads(num_regions * max_gpu_per_region)
    if( thread_num<num_regions ) then
        ! Represents a NUMA region
        !$OMP parallel proc_bind(close) num_threads(region_num_threads)
            Process leaf subtree(s) in parallel
        !$OMP end parallel
    else
        ! Represents a GPU
        Launch subtrees on GPU
        Wait for all subtrees to complete
    end if
!$OMP end parallel
```

3 Subtree factorization on CPU

The CPU factorization is implemented in C++ and follows a traditional multifrontal design. An OpenMP 4.0 task is created for each node, with a dependency *inout* on itself and *in* on its parent node. The reason for an *in* and not an *out* dependency on the parent node is due to the semantics of OpenMP. The OpenMP 4.0 standard [6] states dictates the following meanings for dependency types:

in "The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an out or inout dependence-type list."; and

out/inout "The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an in, out, or inout dependence-type list."

As we want to be able to factorize all children of a node in parallel (each is generating a separate contribution block), the semantics we want are *in*, not *out* (which would serialise them). Rather than the above workaround with children depending in a reverse fashion on their parents, in an ideal world the parent node would have an *in* dependence on each child. However OpenMP does not allow an arbitrary number of dependencies per task (a node may have an arbitrary number of children).

Near the leaf nodes of the tree, handling each (tiny) node as a separate task generates significant overhead. In these cases, we group nodes together into small leaf subtrees that are factorized as a single task using a separate kernel. This kernel can be further optimized to exploit the small nature of the nodes it contains. However, in our implementation, the structure is still largely as discussed below.

Each node is notionally a $m \times m$ (symmetric) frontal matrix F of which n columns are fully summed and can be eliminated (subject to numerical stability), and $m-n$ are non-fully summed and form the contribution block to be passed to the parent. In our implementation, the fully summed columns are stored directly into the matrix factors, whilst the $(m-n)\times(m-n)$ contribution block is stored separately in temporary memory.

At each node we perform the following operations:

assemble_pre() Allocates memory for both the fully summed columns (which must be zeroed) and the contribution block (which need not be). Assembles contribution from children into the fully summed columns only.

factor() Factorizes the fully summed columns and calculates the contribution block.

assemble_post() Assembles contribution from children into the contribution block.

The contribution block does not need to be zeroed (an expensive operation), as the `_gemm()` operations can be told to treat it as zero during calculation.

For a large node, each of these operations is parallelized (node parallelism). Taskgroups are used to ensure all parallel tasks within each operation are completed before the next operation begins. This restriction could be removed with some effort, but the additional complication of the code was deemed not to be worthwhile. The assemble tasks use a 1D parallelization across block columns of each child node, whilst the factor task uses the task parallel scheme described in Section 4.

Of particular note is the memory management. For security reasons, the OS must ensure all memory is zero upon allocation, and thus maintains a cache of pre-cleared pages that are zeroed at times the machine is not busy. For certain memory-hungry applications, such as direct solvers, this cache can quickly become exhausted, and memory allocation slows down. To minimize this overhead, we implement our own memory management:

- Entries of the factors are managed using a stack allocator as they are never released (until the user releases the factorization). To avoid the need for explicit zeroing in the assemble_pre() operation, this uses `calloc()` as the underlying memory allocation mechanism, which is able to exploit the fact that OS pages are zero when allocated.
- Work spaces used by more than one thread and storage for contribution blocks is allocated using a buddy system allocator. This allows memory to be recovered without loss, at the expense of requiring more space than some other methods.
- Work spaces used by a single thread use a static workspace maintained by each thread sufficient to store a single block of the matrix.

4 A posteriori threshold pivoting kernel

The factor() operation performs the dense factorization

$$\begin{pmatrix} F_{11} & F_{21}^T \\ F_{21} & F_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & \\ L_{21} & I \end{pmatrix} \begin{pmatrix} D_{11} & \\ & F_{21} - L_{21}D_{11}L_{21}^T \end{pmatrix} \begin{pmatrix} L_{11} & L_{21}^T \\ & I \end{pmatrix},$$

where L_{11} and L_{21} are lower triangular and D_{11} is block diagonal with 1×1 and 2×2 blocks. Pivots may only be selected from F_{11}. In practice, only the lower triangle of F is stored because of symmetry.

Most existing solvers use one of two underlying approaches:

Threshold Partial Pivoting (TPP) ensures that all entries of $l_{ij} < u^{-1}$ for some fixed threshold u before accepting a pivot. If D_{11} can be stably inverted (an easy condition to meet) then growth in the factors is bounded and the factorization is backwards stable. This requires global communication on each column factorized to perform the threshold test. This approach is used, for example, in MA57 [3] and HSL_MA97 [5].

Supernode Bunch-Kaufmann (SBK) applies the stable dense LDL^T approach of Bunch and Kaufmann[2] to F_{11} and then performs a solve with F_{21} to obtain L_{21}. As entries of F_{21} are not considered during pivoting the overall factorization may be unstable. This approach is used, for example, in PARDISO [7].

Clearly, SBK involves less communication and is hence faster (especially as core count increases, assuming n is kept reasonably small by splitting supernodes if appropriate). Its lack of stability can often be countered by appropriate numerical pre-treatments (e.g. use of a matching-based ordering) and then a few steps of an iterative method (e.g. iterative refinement or FGMRES). However, even with the most sophisticated (and expensive) techniques, the numerical instability cannot be overcome for some matrices, and a stable factorization (i.e. using TPP) must be performed instead.

In this section, we describe an implementation of an a posteriori threshold pivoting (APTP) approach that aims to combine the speed of SBK on stable matrices with the stability of TPP on numerically challenging ones. This is achieved by testing the condition $l_{ij} < u^{-1}$ on a block column basis rather than on a column-wise one.

The new algorithm is shown as Algorithm 3. Tasks are presented as subroutine calls that update (i.e. have an *inout* dependency on) arguments before the semicolon, and have an input dependency on all blocks after it. The variable $nelim_j$ is special as we do not express OpenMP task dependencies involving it, but instead use atomic updates. The tasks are as follows:

Factor($A_{jj}, nelim_j$**)** first stores a backup of block A_{jj} and then performs a pivoted factorization $A_{jj} = P_j L_{jj} D_{jj} L_{jj}^T P_j$, where P_j is a permutation. Initialise $nelim_j$ to block size for atomic reduction for number of successful pivots in block j.

ApplyN($A_{ij}, nelim_j$**;** A_{jj}**) and ApplyT(**$A_{ji}, nelim_j$**;** A_{jj}**)** first store a backup of block A_{ij} (respectively A_{ji}) and then performs the operation $L_{ij} = P_j A_{ij} (L_{jj} D_{jj})^{-T}$ (respectively $L_{ji} = (L_{jj} D_{jj})^{-1} A_{ji} P_j^T$) on uneliminated entries (i.e. the full block for ApplyN and only those in columns corresponding to failed pivots for ApplyT), before finding the first column $nelim_{ij}$ in L_{ij} ($nelim_{ji}$ inL_{ji}) that contains an entry $l_{pq} > u^{-1}$. This is then atomically reduced to find the global $nelim_j = \min(nelim_j, nelim_{ij})$.

Eliminated entries only exist in the ApplyT case (i.e. those in columns corresponding to successful pivots), in which case the rows are permuted according to P_j.

Adjust($nelim_j$**; All blocks** $A_{:j}$ **and** $A_{j:}$**)** adjusts $nelim_j$ down by one if we would otherwise accept only the first column of a 2×2 pivot. The dependence on all blocks in the column ensures atomic reductions are complete.

UpdateNN(A_{ik}**;** A_{ij}**,** A_{kj}**,** $nelim_j$**), UpdateNT(**A_{ik}**;** A_{ij}**,** A_{jk}**,** $nelim_j$**) and UpdateTT(**A_{ik}**;** A_{ji}**,** A_{jk}**,** $nelim_j$**)** perform the update operations $A_{ik} = A_{ik} - L_{ij} D_{jj} L_{kj}^T$ (UpdateNN), $A_{ik} = A_{ik} - L_{ij} D_{jj} L_{jk}$ (UpdateNT), and $A_{ik} = A_{ik} - L_{ji}^T D_{jj} L_{jk}$ (UpdateTT) respectively on uneliminated entries. If A_{ik} is in row or column j, then failed rows or columns (those with an index greater than $nelim_j$) are first restored from the backup before they are updated by the eliminated columns. The backup memory may then be reused for another block. The 'N' and 'T' refer to the orientation of the input blocks compared to a standard implementation of LDL^T that only features UpdateNN.

Note that we are using a fail-in-place approach, where columns and rows corresponding to failed pivots are not permuted outside their block as in most implementations. This leads to the multiple variants of Apply and Update operations, as such entries must be kept up-to-date and participate in pivoting. Figure 2 visualises the data structure part-way through the factorization in which some pivots have failed. However, the overhead of such is not as high as might first appear. If a block has been completely eliminated (the most common case), the UpdateNT and UpdateTT operations become no-ops, and ApplyT becomes a row permutation that would be required anyway.

The extra cost imposed by APTP over TPP is the requirement to take a backup of the pivotal block column and the restoration and update of any failed pivots. The benefit is a more coarse-grained synchronisation (i.e. fewer communications).

After completion of this algorithm, failed entries are permuted to the back of the matrix. At this stage the entries can be refactorized (they have likely been updated since they failed) using either APTP or TPP, or passed directly to the parent. We originally considered not permuting the failed entries, but instead performing swaps and running again on the same block partitioning. However this turned out not to be needed in practice as the number of failed pivots (that could be subsequently eliminated) was so small.

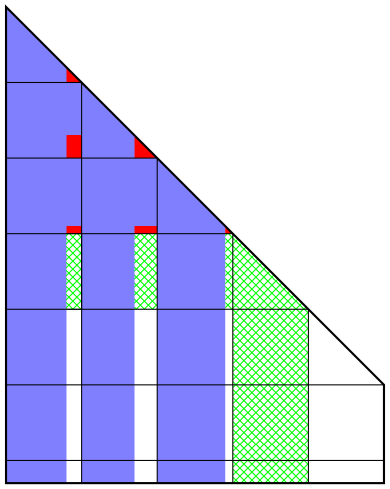
We note that there are a number of options for implementation of the Factor() task:

- Recursively perform an APTP factorization with a smaller block size.
- Perform complete pivoting (see Section 5).
- Use an implementation of TPP.

Algorithm 3 A posteriori threshold pivoting LDL^T factorization.

```
for $j = 1, \ldots, \texttt{nblk}$ do
  Factor( $A_{jj}$, $nelim_j$ )
  for $i = 1, \ldots, j-1$ do
    ApplyT( $A_{ji}$, $nelim_j$; $A_{jj}$ )
  end for
  for $i = j+1, \ldots, \texttt{mblk}$ do
    ApplyN( $A_{ij}$, $nelim_j$; $A_{jj}$ )
  end for
  Adjust( $nelim_j$; All blocks $A_{:j}$ and $A_{j:}$ )
  for $k = 1, \ldots, j-1$ do
    for $i = k, \ldots, j-1$ do
      UpdateTT( $A_{ik}$; $A_{ji}$, $A_{jk}$, $nelim_j$ )
    end for
    for $i = j, \ldots, \texttt{mblk}$ do
      UpdateNT( $A_{ik}$; $A_{ij}$, $A_{jk}$, $nelim_j$ )
    end for
  end for
  for $k = j, \ldots, \texttt{nblk}$ do
    for $i = k, \ldots, \texttt{mblk}$ do
      UpdateNN( $A_{ik}$; $A_{ij}$, $A_{kj}$, $nelim_j$ )
    end for
  end for
end for
```

Figure 2: Part-way through execution of Algorithm 3. Iterations $j = 1, 2, 3$ have completed, and iteration $j = 4$ is about to begin. Blue entries have been eliminated. Red entries have failed to be eliminated twice (by column or row). Green entries indicate uneliminated entries considered for elimination on iteration $j = 4$.



In practice, we use a two-level approach involving an outer block size *nb* (an option accessible to the user) and an inner block size *nbi* (fixed at compile time). APTP is performed in parallel using the outer block size, and recurses to perform APTP in serial with the smaller inner block size. This inner block APTP then uses complete pivoting for full blocks and a simple implementation of TPP for partial blocks.

In addition to Algorithm 3, we also offer a more aggressive variant that removes the Adjust() task, replacing it with a global abort in the event that any $l_{ij} < u^{-1}$. This provides a stable algorithm with the same degree of parallelism as Cholesky. If a more advanced tasking system were used instead of OpenMP, we could instead use speculative execution to achieve the same acceleration.

5 Relation between full pivoting and threshold pivoting

As described in the previous section, at the most fine-grained level we use complete pivoting on a small dense block. The reason for doing so is that such a block fits in cache, so there is little penalty to searching the entire matrix for a good pivot. Furthermore, it should cause any failed pivots of the APTP algorithm to occur as late in each block as possible (if the cause is a poor diagonal pivot rather than an unusually large off-diagonal entry).

Algorithm 4 Complete pivoting algorithm

```
1: Find maximum uneliminated entry in position $(t, m)$ with value $a_{tm}$.
2: if $|a_{tm}| < \delta$ then
3:     All remaining pivots are 0.
4: else if $m == t$ then
5:     Use as $1 \times 1$ pivot
6: else
7:     $\Delta = a_{mm}a_{tt} - a_{mt}a_{mt}$
8:     if $|\Delta| \geq \frac{1}{2}|a_{mt}|^2$ then
9:         $2 \times 2$ pivot
10:    else
11:        $1 \times 1$ pivot on $\max(|a_{mm}|, |a_{tt}|)$
12:    end if
13: end if
```

The complete pivoting algorithm proceeds column-wise in a traditional fashion, but the choice of the next pivot is determined by the location of the maximum entry as shown in Algorithm 4. A brief summary of the algorithm is that we first test if all remaining entries in the block are effectively zero. If so, we record all uneliminated columns as zero pivots. If not, we try and use the largest entry as the pivot, either as a 1×1 if it lies on the diagonal, or as the off-diagonal entry of a 2×2 pivot if it is not. The test $|\Delta| \geq \frac{1}{2}|a_{mt}|^2$ on line 8 is a very conservative condition on the stability of the 2×2 pivot. As we show below, if the condition does not hold, then $\max(|a_{mm}|, |a_{tt}|)$ must be large in its own right and is hence safe to use as a 1×1 pivot instead.

5.1 Stability

Algorithm 4 is as stable as traditional threshold partial pivoting with a pivot threshold of $u = 0.25$. Recall that this is implied by all entries of L being bounded by $u^{-1} = 4$ with all 2×2 pivots inverted in a stable fashion. We now show this is the case for each of the three nonsingular execution paths. For ease of reference, we relabel $a_{11} = a_{mm}, a_{21} = a_{tm}, a_{22} = a_{tt}$.

Immediate 1×1 pivot case
Trivially

$$|l_{i1}| = \frac{|a_{i1}|}{|a_{11}|} \le 1 < 4.$$

Successful 2×2 pivot case
Without loss of generality we have (up to relabelling)

$$|a_{21}| \ge |a_{11}| \ge |a_{22}|, \tag{1}$$

$$\Delta = a_{11}a_{22} - a_{21}a_{21}.$$

The stability test used in recent HSL solvers (e.g. `HSL_MA97`) for 2×2 pivots is the following set of conditions:

(a) $|\Delta| \ge \frac{1}{2}\delta|a_{21}|$,

(b) $|\Delta| \ge \frac{1}{2}|a_{11}||a_{22}|$,

(c) $|\Delta| \ge \frac{1}{2}|a_{21}||a_{21}|$.

The first condition ensures that the pivot is nonsingular, while the latter two ensure any cancellation in the calculation is insignificant. Given condition (1), observe that (c)$\Rightarrow$(b). Furthermore, since $|a_{21}| \ge \delta$, it follows that (c)$\Rightarrow$(a). However, it is possible for condition (c) to be violated, consider for example

$$\begin{pmatrix} \alpha - \epsilon & \alpha \\ \alpha & \alpha - \epsilon \end{pmatrix} \Rightarrow \Delta = \epsilon(\epsilon - 2) = O(\epsilon).$$

So, assuming condition (c) holds, we now consider the size of entries of L:

$$\begin{pmatrix} l_{i1} \\ l_{i2} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{21} \\ a_{21} & a_{22} \end{pmatrix}^{-1} \begin{pmatrix} a_{i1} \\ a_{i2} \end{pmatrix} = \frac{1}{\Delta} \begin{pmatrix} a_{22}a_{i1} - a_{21}a_{i2} \\ -a_{21}a_{i1} + a_{11}a_{i2} \end{pmatrix}.$$

Using $|a_{21}| \ge |a_{i1}|, |a_{i2}|$ and the triangular inequality we get

$$\begin{pmatrix} |l_{i1}| \\ |l_{i2}| \end{pmatrix} \le \frac{1}{\Delta} \begin{pmatrix} |a_{22}||a_{21}| + |a_{21}||a_{21}| \\ |a_{21}||a_{21}| + |a_{11}||a_{21}| \end{pmatrix}.$$

By condition (c) we have $\frac{1}{\Delta} \le 2\frac{1}{|a_{21}||a_{21}|}$, combined with (1) to give

$$|l_{i1}|, |l_{i2}| \le 2\frac{1}{|a_{21}||a_{21}|}\left(|a_{21}||a_{21}| + |a_{21}||a_{21}|\right) = 4 = (0.25)^{-1}.$$

This corresponds to a traditional pivoting threshold of $u = 0.25$. This bound is achieved for the following matrix in the limit $\epsilon \to 0$,

$$\begin{pmatrix} \alpha - \epsilon & \alpha & \alpha - \epsilon \\ \alpha & \frac{1}{2}\frac{\alpha^2}{\alpha - \epsilon} & -(\alpha - \epsilon) \\ \alpha - \epsilon & -(\alpha - \epsilon) & 0 \end{pmatrix}.$$

Failed 2×2 pivot case
If condition (c) does not hold, we fall back on a 1×1 pivot corresponding to the maximum (absolute) diagonal value. The failure of (c) means that

$$|\Delta| = |a_{11}a_{22} - a_{21}a_{21}| < \frac{1}{2}|a_{21}||a_{21}|.$$

For the above to hold, $a_{11}a_{22}$ and $a_{21}a_{21}$ must have the same sign. Furthermore, from (1) we have $|a_{21}a_{21}| \geq |a_{11}a_{22}|$. Combining these allows us to write

$$|a_{21}a_{21}| - |a_{11}a_{22}| < \frac{1}{2}|a_{21}||a_{21}|.$$

As wlog $a_{11} \geq a_{22}$, rearranging we have

$$\frac{1}{2}|a_{21}||a_{21}| < |a_{11}||a_{22}| < |a_{11}||a_{11}|,$$

and hence

$$\frac{1}{|a_{11}|} < \frac{\sqrt{2}}{|a_{21}|}.$$

Which allows us to bound entries of L:

$$|l_{i1}| = \frac{|a_{i1}|}{|a_{11}|} \leq \frac{|a_{21}|}{|a_{11}|} \leq \sqrt{2} < 4.$$

6 Results

We present results on the following machines:

Haswell desktop A desktop machine with a i7-4790 processor, providing 4 cores.

Haswell compute node A HPC node with 2×E5-2695 v3 processors, for a total of 28 cores.

Ivy Bridge/K40 compute node A HPC node with 2×E5-2650 v2 processors, for a total of 16 CPU cores, and a K40 GPU.

We use the following codes:

SPRAL SSIDS The code described in this paper. All results are against the version with git tag `PAPER_20160922`.

HSL_MA86 A task-based supernodal symmetric indefinite solver from HSL. It uses a block column orientated factorization, so is more limited in the parallelism it can exploit than the other codes here. We use version 1.5.0.

HSL_MA87 A task-based supernodal Cholesky solver from HSL. We use version 2.4.0.

HSL_MA97 A multifrontal symmetric indefinite solver that uses a recursive parallel factorization at the node level. We use version 2.4.0.

PARDISO A supernodal solver that uses Supernode Bunch-Kaufmann and a static pivoting scheme. We use the version provided with the MKL as stated for each machine.

We use the following Test Sets, listed in Tables 1–3 respectively.

Test Set 1 Symmetric indefinite problems that require few delayed pivots. No scaling is performed. Results are presented with SSIDS, HSL_MA86, HSL_MA97 and PARDISO.

Test Set 2 Symmetric indefinite problems that require significant scaling and pivoting. Matrices are scaled and ordered using a matching-based ordering and scaling. Results are presented with SSIDS, HSL_MA97 and PARDISO only, as HSL_MA86 does not offer the capability to use a matching-based ordering without significant additional work.

Test Set 3 Positive-definite problems. Results are presented with SSIDS, HSL_MA87, HSL_MA97 and PARDISO.

Table 1: Test Set 1: Easy Indefinite. Statistics as reported by the analyse phase of SSIDS with default settings, assuming no delays.

Problem	n $\times 10^3$	$nz(A)$ $\times 10^6$	$nz(L)$ $\times 10^6$	$flops$ $\times 10^9$
Oberwolfach/t2dal	4.26	0.02	0.28	0.02
GHS_indef/dixmaanl	60.00	0.18	1.58	0.05
Oberwolfach/rail_79841	79.84	0.32	4.43	0.33
GHS_indef/dawson5	51.54	0.53	5.69	0.90
Boeing/bcsstk39	46.77	1.07	9.61	2.66
Boeing/pct20stif	52.33	1.38	12.60	5.63
GHS_indef/copter2	55.48	0.41	12.70	6.10
GHS_indef/helm2d03	392.26	1.57	33.00	6.16
Boeing/crystk03	24.70	0.89	10.90	6.26
Oberwolfach/filter3D	106.44	1.41	23.80	8.71
Koutsovasilis/F2	71.50	2.68	23.70	11.30
McRae/ecology1	1000.00	3.00	72.30	18.20
Cunningham/qa8fk	66.13	0.86	26.70	22.10
Oberwolfach/gas_sensor	66.92	0.89	27.00	22.10
Oberwolfach/t3dh	79.17	2.22	50.60	70.10
Lin/Lin	256.00	1.01	126.00	285.00
GHS_indef/sparsine	50.00	0.80	207.00	1390.00
PARSEC/Ge99H100	112.98	4.28	669.00	7070.00
PARSEC/Ga10As10H30	113.08	3.11	690.00	7280.00
PARSEC/Ga19As19H42	133.12	4.51	823.00	9100.00

6.1 Haswell desktop

Results are on a desktop machine with an i7-4790 processor, compiled with:

- gcc 6.2.0 with flags "-g -O2 -march=native"
- Intel MKL BLAS 11.0.3
- metis 4.0.3

Figures 4–5 show the comparative performance of SSIDS on a range of problems on this machine.

Clearly the code is outperformed by the dedicated Cholesky solver HSL_MA87 for positive definite problems. However on the indefinite problems it generally outperforms both HSL_MA86 and HSL_MA97 and is often comparable with PARDISO. On the larger problems it significantly outperforms PARDISO, but is outperformed by HSL_MA86. The very large problems failed to run with any solver on this machine as they ran out of memory.

6.2 Haswell compute node

Results on a HPC node with 2×E5-2695 v3 processors, compiled with:

- gcc 6.1.0 with flags "-g -O2 -march=native"
- Intel MKL BLAS 11.3.1
- metis 4.0.3

Table 2: Test Set 2: Hard Indefinite. Statistics as reported by the analyse phase of SSIDS with default settings, using matching-based ordering, assuming no delays.

Problem	n $\times 10^3$	$nz(A)$ $\times 10^6$	$nz(L)$ $\times 10^6$	$flops$ $\times 10^9$
TSOPF/TSOPF_FS_b39_c7	28.22	0.37	2.61	0.26
TSOPF/TSOPF_FS_b162_c1	10.80	0.31	1.89	0.36
QY/case39	40.22	0.53	3.87	0.40
TSOPF/TSOPF_FS_b39_c19	76.22	1.00	7.28	0.75
TSOPF/TSOPF_FS_b39_c30	120.22	1.58	11.10	1.10
GHS_indef/cont-201	80.59	0.24	7.12	1.11
GHS_indef/stokes128	49.67	0.30	6.35	1.16
TSOPF/TSOPF_FS_b162_c3	30.80	0.90	6.37	1.41
TSOPF/TSOPF_FS_b162_c4	40.80	1.20	7.32	1.43
GHS_indef/ncvxqp1	12.11	0.04	3.56	2.52
GHS_indef/darcy003	389.87	1.17	23.20	3.01
GHS_indef/cont-300	180.90	0.54	17.20	3.58
GHS_indef/bratu3d	27.79	0.09	7.49	4.72
GHS_indef/cvxqp3	17.50	0.07	6.33	5.27
TSOPF/TSOPF_FS_b300	29.21	2.20	13.40	6.92
TSOPF/TSOPF_FS_b300_c1	29.21	2.20	13.50	7.01
GHS_indef/d_pretok	182.73	0.89	24.80	7.42
GHS_indef/turon_m	189.92	0.91	24.70	7.60
TSOPF/TSOPF_FS_b300_c2	56.81	4.39	27.00	14.10
TSOPF/TSOPF_FS_b300_c3	84.41	6.58	40.50	21.40
GHS_indef/ncvxqp5	62.50	0.24	22.90	24.30
GHS_indef/ncvxqp3	75.00	0.27	39.30	63.70
GHS_indef/ncvxqp7	87.50	0.31	51.00	101.00

Figure 3: Timing results on Haswell desktop for Test Set 1 (easy indefinite).

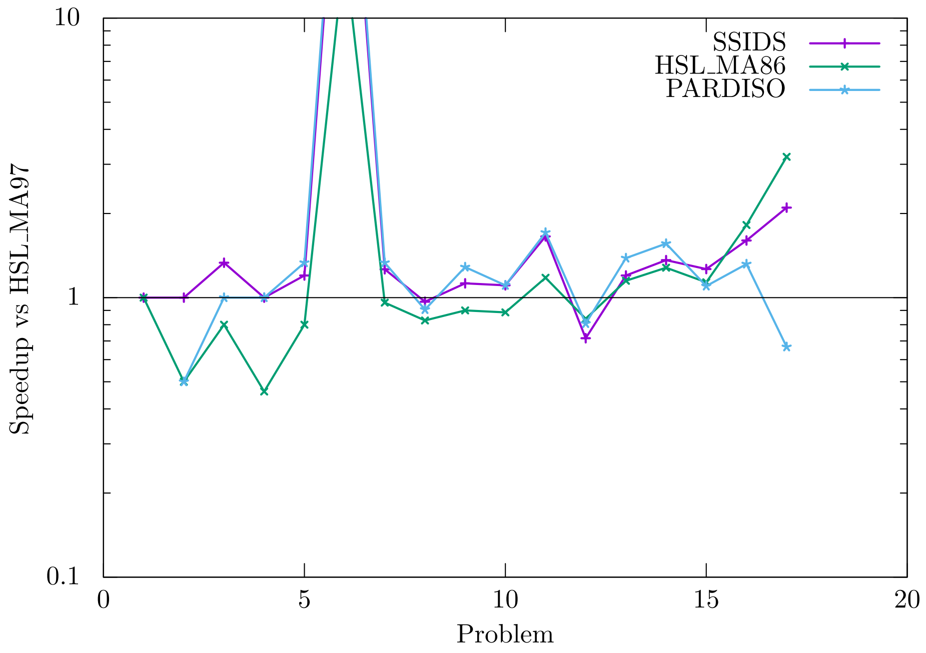


Table 3: Test Set 3: Positive Definite. Statistics as reported by the analyse phase of SSIDS with default settings.

Problem	n	$nz(A)$	$nz(L)$	$flops$
	$\times 10^3$	$\times 10^6$	$\times 10^6$	$\times 10^9$
CEMW/tmt_sym	726.71	2.90	55.90	11.70
McRae/ecology2	1000.00	3.00	72.50	18.50
Schmid/thermal2	1228.05	4.90	94.80	18.60
DNVS/m_t1	97.58	4.93	37.70	23.40
Boeing/pwtk	217.92	5.93	57.90	25.10
GHS_psdef/crankseg_1	52.80	5.33	36.00	33.90
Rothberg/cfd2	123.44	1.61	44.20	34.30
DNVS/thread	29.74	2.25	25.40	35.70
GHS_psdef/crankseg_2	63.84	7.11	47.00	48.80
Schenk_AFE/af_shell3	504.86	9.05	116.00	57.20
AMD/G3_circuit	1585.48	4.62	171.00	67.30
DNVS/ship_003	121.73	4.10	69.80	87.20
GHS_psdef/ldoor	952.20	23.74	189.00	87.50
GHS_psdef/apache2	715.18	2.77	177.00	183.00
Koutsovasilis/F1	343.79	13.59	193.00	228.00
Oberwolfach/boneS10	914.90	28.19	326.00	297.00
JGD_Trefethen/Trefethen_20000	20.00	0.29	95.60	669.00
ND/nd24k	72.00	14.39	326.00	2080.00
Oberwolfach/bone010	986.70	36.33	1140.00	3910.00
GHS_psdef/audikw_1	943.70	39.30	1310.00	5840.00

Figure 4: Timing results on Haswell desktop for Test Set 2 (hard indefinite).

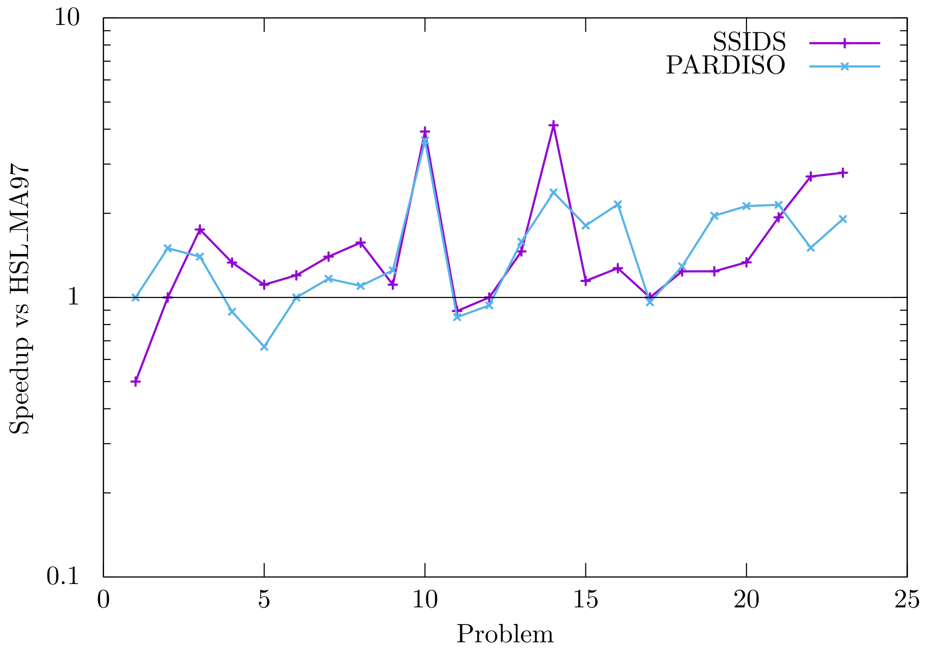


Figure 5: Timing results on Haswell desktop for Test Set 3 (positive definite).

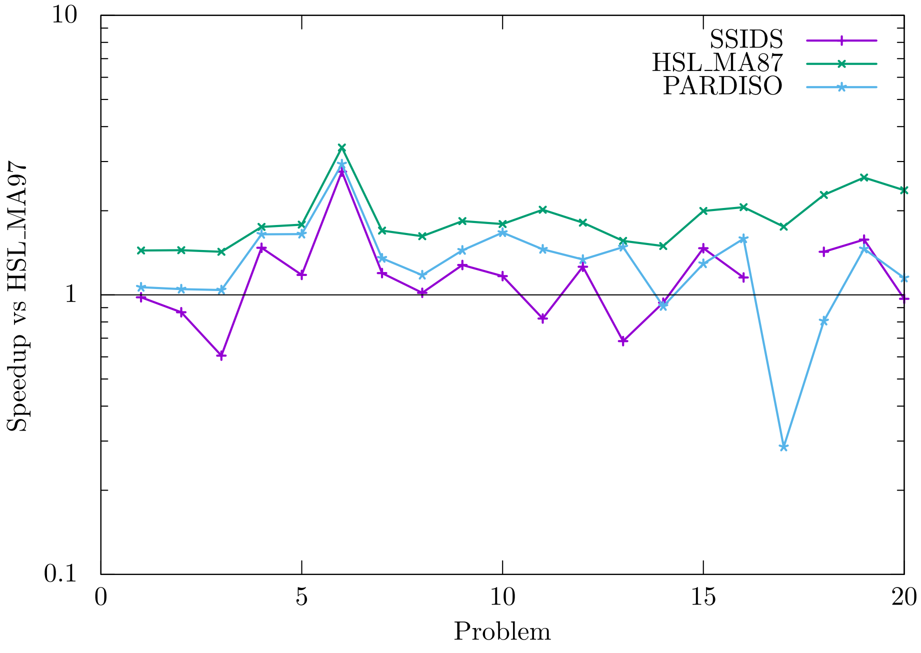


Figure 6: Timing results on Haswell HPC node for Test Set 1 (easy indefinite).

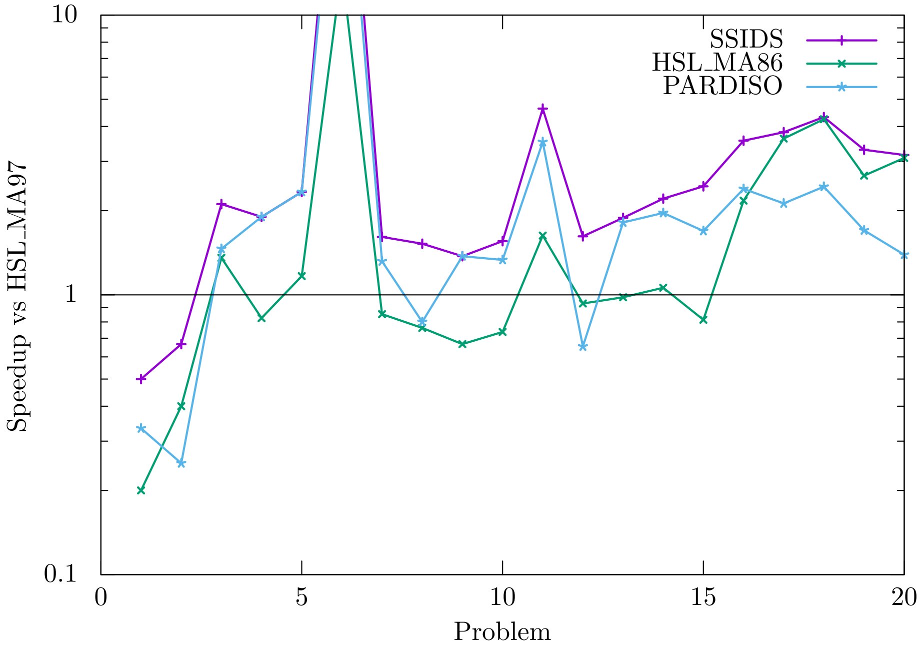


Figure 7: Timing results on Haswell HPC node for Test Set 2 (hard indefinite).

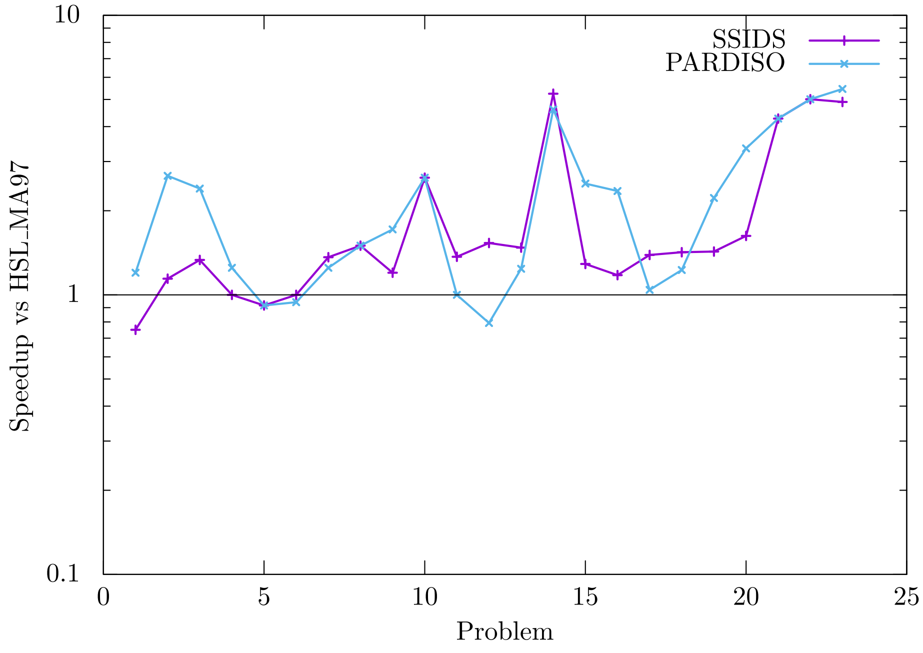


Figure 8: Timing results on Haswell HPC node for Test Set 3 (positive definite).

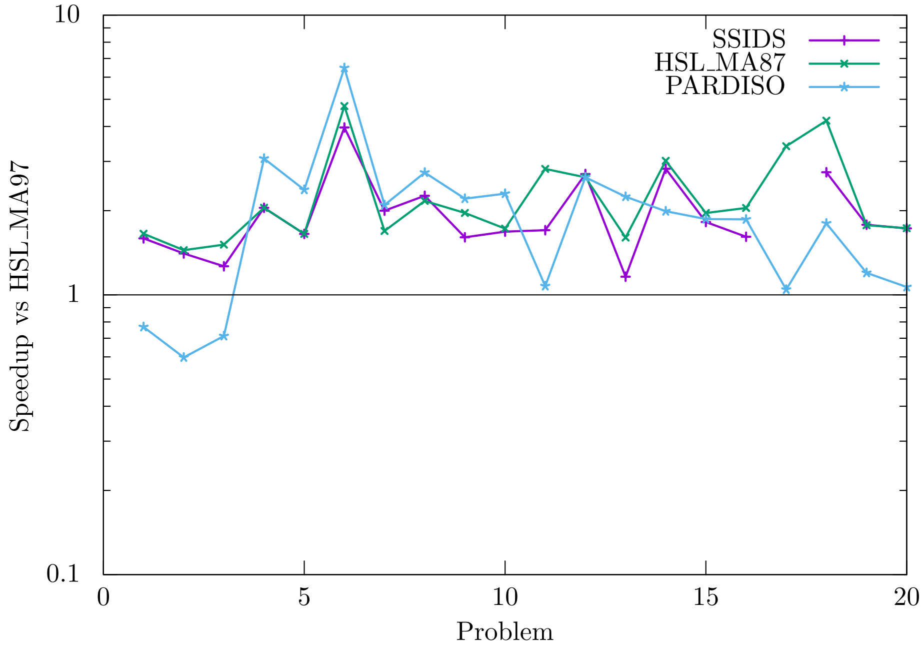


Table 4: Failed pivot handling comparison. tfact is the factorization time, ndelay is the number of delayed pivots, fail1 is the number of pivots that failed the first pass, and fail2 is the number that failed the second (i.e. after tpp). Factorization times that are significantly better are in bold.

Problem	tfact		ndelay		fail1	fail2
	tpp	pass	tpp	pass		
Boeing/pct20stif	0.17	0.17	13	108	192	4
GHS_indef/sparsine	8.06	**6.44**	24	941	1562	23
PARSEC/Ga10As10H30	22.62	22.62	0	33	35	0
TSOPF/TSOPF_FS_b39_c7	0.08	0.08	997	1378	1248	965
QY/case39	0.09	0.11	2312	3334	2963	2182
TSOPF/TSOPF_FS_b39_c19	0.10	0.11	3464	4691	4175	3198
TSOPF/TSOPF_FS_b39_c30	0.12	0.14	5031	7013	6277	4818
GHS_indef/ncvxqp1	0.24	0.20	78	222	97	76
GHS_indef/cvxqp3	0.29	**0.23**	26	45	37	26
TSOPF/TSOPF_FS_b300	**0.31**	7.36	1235	10703	5285	1235
TSOPF/TSOPF_FS_b300_c1	**0.34**	15.11	1599	12015	6122	1599
GHS_indef/bratu3d	0.21	0.22	27	199	172	3
GHS_indef/ncvxqp5	0.59	**0.42**	60	1058	1164	46
TSOPF/TSOPF_FS_b162_c1	0.07	0.08	401	788	603	401
GHS_indef/ncvxqp7	1.67	**1.11**	164	830	458	157

6.2.1 Effect of pivoting method and failed pivot handling

There are a number of options available that control pivot handling in the code. As mentioned previously, an aggressive variant of APTP can be used that assumes no pivots ever fail. Our numerical experiments showed this offered little benefit: it is likely that for small matrices there was insufficient work outside the critical path, and for large matrices there was already sufficient work to keep all resources busy. Another option was to use our implementation of traditional TPP instead of APTP, however our implementation is not designed for large matrices (it does not exploit BLAS3 calls), and as such performs poorly on any except the smallest matrices.

Another option is whether to attempt elimination of failed pivots after the first APTP pass, or whether to just pass straight to the parent. At present the second pass is implemented using TPP, so will perform poorly in the presence of many failed pivots. Table 4 shows the effects on a selection of matrices where there are sufficient failed pivots to make a difference. For some problems there is a significant (up to 30%) reduction in factorization time, however for other problems there is a much bigger increase due to the resulting larger fronts. The columns fail1 and fail2 give the number of pivots that failed APTP and subsequent TPP respectively. It is clear that for some problems, a significant number are being eliminated by the second TPP pass.

6.2.2 Using separate NUMA regions

The option exists in the code to use each NUMA region as a separate resource to handle its own leaf subtree. Keeping memory local to a region should improve performance by putting less stress on the processor to processor link (e.g. QuickPath Interconnect on Intel systems). However the additional cost of this is the additional synchronisation before starting the root subtree. In our tests this could cause significant delays, so the default is to treat all cores as belonging to a single region.

6.3 Ivy bridge/GPU compute node

Results on a HPC node with 2×E5-2650 v2 processors and a K40 GPU.

Table 5: Effect of gpu α performance coefficient on selected problems.

Problem	Factorization time				GPU flops			
$\alpha =$	0.75	1.00	1.25	1.50	0.75	1.00	1.25	1.50
Koutsovasilis/F2	0.17	0.17	0.14	0.14	0	0	5.80×10^{9}	5.80×10^{9}
Cunningham/qa8fk	0.20	0.20	0.18	0.18	6.40×10^{9}	6.40×10^{9}	1.04×10^{10}	1.04×10^{10}
Oberwolfach/gas_sensor	0.21	0.21	0.19	0.19	5.36×10^{9}	5.36×10^{9}	1.15×10^{10}	1.15×10^{10}
McRae/ecology1	0.40	0.40	0.29	0.28	0	0	8.98×10^{9}	8.98×10^{9}
Oberwolfach/t3dh	0.43	0.44	0.45	0.45	2.51×10^{10}	2.51×10^{10}	2.57×10^{10}	2.57×10^{10}
Lin/Lin	1.16	1.16	1.02	1.00	1.32×10^{11}	1.32×10^{11}	1.40×10^{11}	1.40×10^{11}
GHS_indef/sparsine	8.89	8.69	11.05	11.07	3.77×10^{11}	3.77×10^{11}	2.68×10^{11}	2.68×10^{11}

- gcc 4.9.2 with flags "-g -O2 -march=native"
- nvcc 7.5.18 with flags "-arch=compute_35 -code=sm_35"
- Intel MKL BLAS 11.2.0
- metis 4.0.3

6.4 Parameter selection

Based on our test runs, we recommend that the minimum amount of work in a leaf subtree to be factorized, $gpu_{\min}$ on the GPU is at least 5 Gflop. We found that the load balance parameter $b_{\min}$ had little effect as most often the splitting of the tree was limited by the maximum number of iterations or minimum size of tree for the GPU. For a few problems setting $b_{\min} = 1.1$ reduced performance, but $b_{\min} = 1.2$ did not, so we choose a default value of 1.2.

Using these settings, we experimented with the GPU performance coefficient α_{gpu}. The results are shown in Table 5 for some problems selected from Test Set 1 (many problems were too small to benefit from the GPU, other ones ran out of memory). Based on these results we have chosen $\alpha_{gpu} = 1.0$ as the default.

6.5 Comparison versus other codes

We provide a comparison against the same CPU-only codes as the previous section, but add the GPU-only SSIDS v1 code (essentially the same as used for the GPU leaf subtrees in our new code), and WSMP v16.06.01 with GPU acceleration. Unfortunately the CPU-only version of WSMP relies on a newer version of GLIBC than was available on our cluster.

We note that the new code does not cope well in constrained memory environments. It seems the OpenMP runtime can become quite hungry and further offers no graceful exit when memory is exhausted.

7 Conclusions and Future Work

We present a new open source sparse symmetric indefinite solver that provides performance comparable to existing software on desktop architectures and provides best in class performance for large problems on manycore shared memory architectures. Performance on positive definite systems still lags behind dedicated sparse Cholesky factorization codes such as HSL_MA87.

During the course of our work, we found a number of "rough edges" to current OpenMP implementations. The gcc implementation seems to be the most feature complete, with the only flaw of crashing when it generates too many tasks (this could presumably be rectified by switching to a different task when a given task is generating too many). The intel implementation at present crashes when running our code, though we have been unable to determine the cause of this error (some form of stack corruption), and also suffers

Figure 9: Timing results on GPU HPC node for Test Set 1 (easy indefinite).

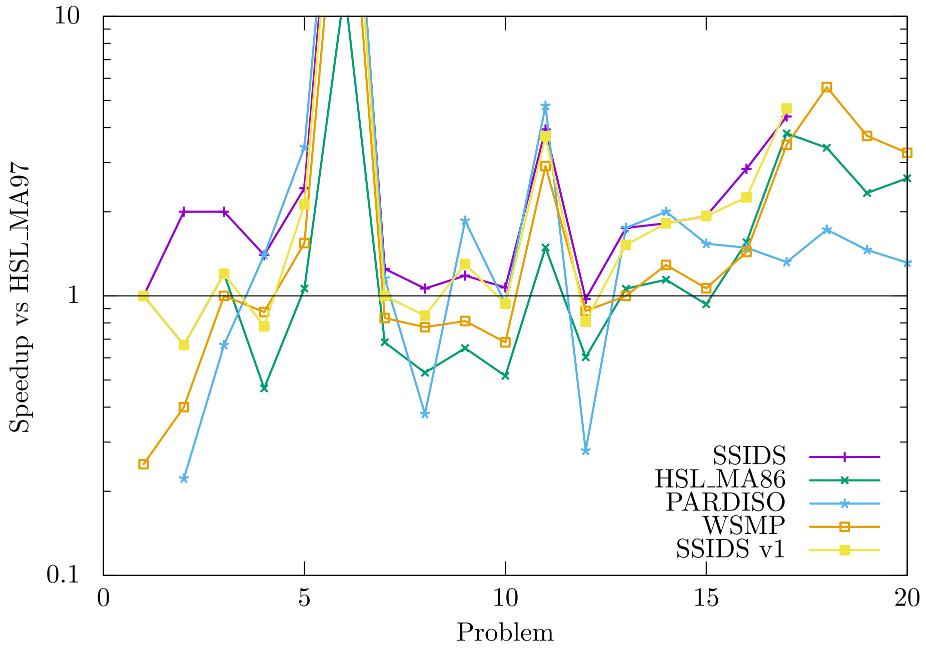


Figure 10: Timing results on GPU HPC node for Test Set 2 (hard indefinite).

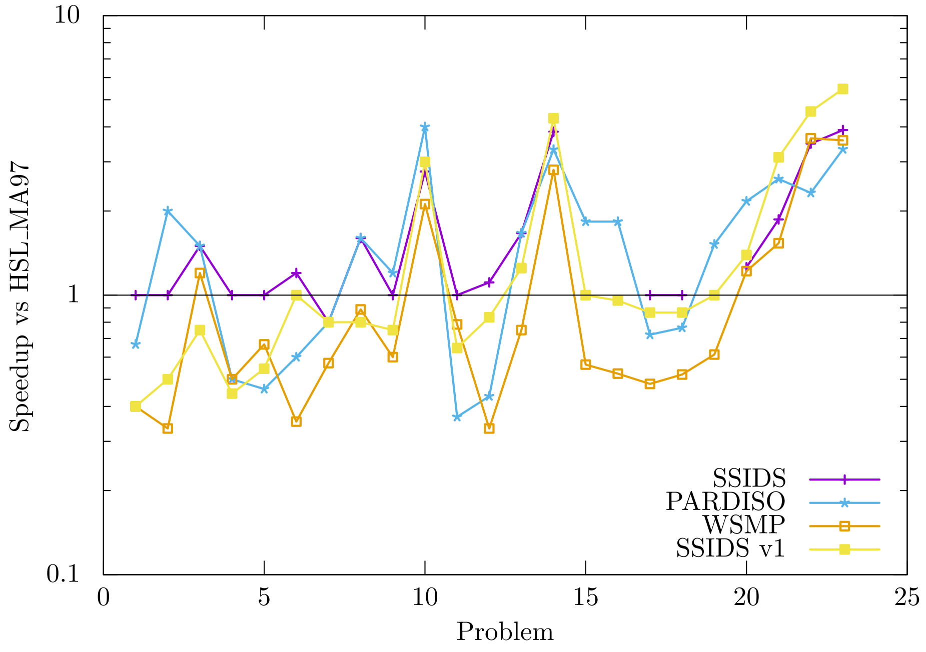
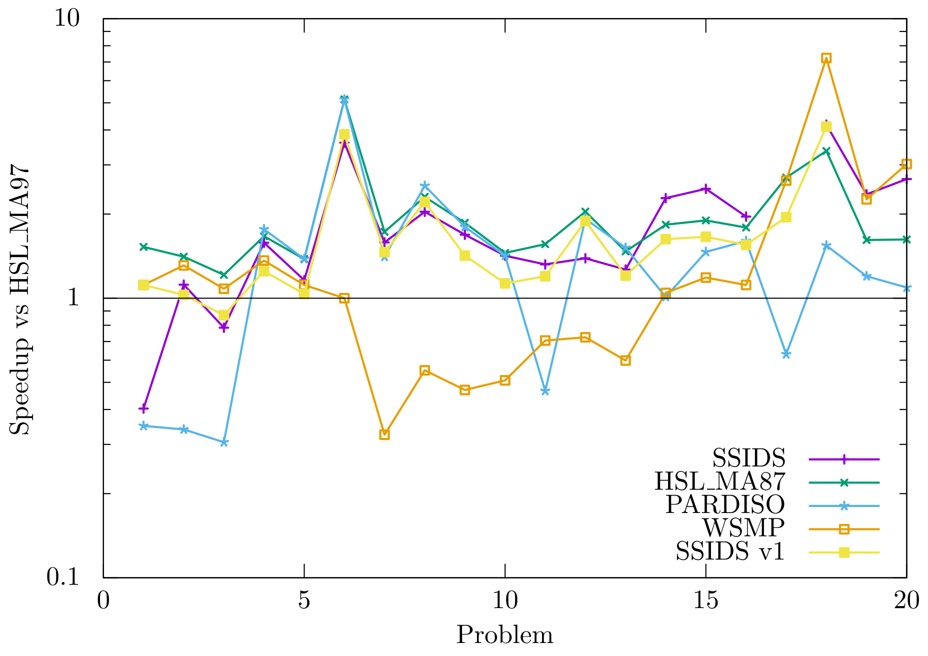


Figure 11: Timing results on GPU HPC node for Test Set 3 (positive definite).



from bugs around its implementation of the cancel construct. As such we recommend other authors approach OpenMP 4.0 tasking with caution until these bugs are ironed out.

There are a number of areas where performance could be significantly improved:

Tree partitioning It seems likely that the straightforward approach taken in determining leaf subtrees could be enhanced. There is significant work in the literature that could be reviewed to find alternative algorithms, and a better performance model could be found.

Supernode amalgamation The current algorithm dates back to MA57 and before. It seems likely that a better algorithm could be created, with different parameters for different subtrees (this may mean delaying final generation of row lists until after subtree partitioning is decided).

Grouping of small nodes mid-tree The grouping of small nodes at the leaves could be expanded to include a similar approach for mid-tree to reduce the number of tasks in flight.

Small node kernels Performance of these kernels could probably be significantly improved.

Fallback TPP kernel Add some blocking so BLAS3 can be used to boost performance. This will significantly improve speeds where significant numbers of pivots fail the first pass of APTP and/or are delayed.

It remains to be said that moving to a full task parallel approach as in HSL_MA87 is also likely to yield significant performance gains, but would require fundamental redesign of the code, and is probably not easily achieved within the current OpenMP tasking framework.

References

[1] C. Augonnet, *Scheduling Tasks over Multicore machines enhanced with Accelerators: a Runtime System's Perspective*, PhD thesis, Université Bordeaux 1, 12 2011.

[2] J. R. Bunch and L. Kaufman, *Some stable methods for calculating inertia and solving symmetric linear systems*, Mathematics of Computation, 31 (1977), pp. 1634–179.

[3] I. S. Duff, *MA57—a code for the solution of sparse symmetric definite and indefinite systems*, ACM Transactions on Mathematical Software, 30 (2004), pp. 118–144.

[4] J. D. Hogg, E. Ovtchinnikov, and J. A. Scott, *A sparse symmetric indefinite direct solver for GPU architectures*, ACM Transactions on Mathematical Software, 42 (2016), pp. 1:1–1:25.

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

[6] *OpenMP 4.0 Complete Specifications*, 2013. OpenMP Architecture Review Board.

[7] O. Schenk and K. Gärtner, *On fast factorization pivoting methods for symmetric indefinite systems*, Electronic Transactions on Numerical Analysis, 23 (2006), pp. 158–179.