SIAM J. SCI. COMPUT. © 2020 Society for Industrial and Applied Mathematics
Vol. 42, No. 2, pp. C23–C42

A NEW SPARSE LDL^T SOLVER USING A POSTERIORI THRESHOLD PIVOTING*

IAIN DUFF†, JONATHAN HOGG†, AND FLORENT LOPEZ†

Abstract. The factorization of sparse symmetric indefinite systems is particularly challenging since pivoting is required to maintain stability of the factorization. Pivoting techniques generally offer limited parallelism and are associated with significant data movement hindering the scalability of these methods. Variants of the threshold partial pivoting (TPP) algorithm, for example, have often been used because of its numerical robustness but standard implementations exhibit poor parallel performance. On the other hand, some methods trade stability for performance on parallel architectures such as the supernode Bunch–Kaufman used in the PARDISO solver. In this case, however, the factors obtained might not be used to accurately compute the solution of the system. For this reason we have designed a task-based LDL^T factorization algorithm based on a new pivoting strategy called a posteriori threshold pivoting (APTP) that is much more suitable for modern multicore architectures and has the same numerical robustness as the TPP strategy. We implemented our algorithm in a new version of the SPRAL sparse symmetric indefinite direct solver, which initially supported GPU-only factorization. We have used OpenMP 4 task features to implement a multifrontal algorithm with dense factorizations using the novel APTP, and we show that it performs favorably compared to the state-of-the-art solvers `HSL_MA86`, `HSL_MA97` and PARDISO both in terms of performance on a multicore machine and in terms of numerical robustness. Finally we show that this new solver is able to make use of GPU devices for accelerating the factorization on heterogeneous architectures.

Key words. sparse linear systems, LDLT factorization, multifrontal, multicore, GPU

AMS subject classifications. 68W10, 65F50, 65F05

DOI. 10.1137/18M1225963

1. Introduction.

In this paper we describe the a posteriori threshold pivoting (APTP) algorithm and its implementation within the new version of the `SSIDS` solver [15] for solving the linear system

$$Ax = b, \tag{1.1}$$

where A is a large, sparse, symmetric indefinite matrix. Such systems arise in many applications in science and engineering for which computing the solution often represents a bottleneck. This is achieved through factorizing the matrix A so that

$$PAP^T = LDL^T, \tag{1.2}$$

where L is a lower triangular matrix, D is block diagonal with 1×1 and 2×2 blocks, and P is a permutation matrix for maintaining sparsity in L and for ensuring the stability of the factorization process. In this work, we introduce a new pivoting strategy that we call APTP, and we show its efficiency for exploiting the capabilities of modern multicore architectures while preserving the stability of the factorization. The efficiency of this new algorithm has been made possible by exploiting a *fail-in-place* approach enabling a reduction of data movement and the use of *speculative execution*

*Submitted to the journal's Software and High-Performance Computing section November 12, 2018; accepted for publication (in revised form) November 8, 2019; published electronically March 10, 2020.
https://doi.org/10.1137/18M1225963

†Scientific Computing Department, STFC Rutherford Appleton Laboratory, Didcot, Oxfordshire, OX11 0QX (iain.duff@stfc.ac.uk, jhogg41@gmail.com, florent.lopez@stfc.ac.uk).

to increase parallelism in the factorization algorithm. The APTP strategy presented in this work is inspired by that implemented in the first version of SSIDS, designed to run on GPU devices [15]. In the new version of SSIDS, the APTP strategy allows us to design a task-based LDL^T factorization algorithm that shows better performance than state-of-the-art solvers while preserving numerical robustness. We have implemented this algorithm using the OpenMP 4 tasking system. One of the reasons for using OpenMP over a more extensive runtime system such as StarPU [1] is to avoid having an additional dependency in the SPRAL library.[1] In addition, common implementations of the OpenMP standard such as that available in the GNU and Intel compilers have been shown to be efficient for implementing task-based algorithms on multicore architectures [6, 10].

In the context of heterogeneous CPU-GPU architectures, we show, in section 8, that the new version of SSIDS is capable of exploiting the GPU device by offloading part of the computations to the GPU. Experimental results show that this approach allows us to dramatically accelerate the factorization phase compared to the CPU-only version.

2. Background and related work.

We follow a now standard procedure for the direct solution of sparse symmetric linear systems by basing our factorization on an assembly tree where a partial factorization of a dense matrix is performed at each node of the tree. See, for example, [8] for a detailed description of this process. The algorithms and code for this work are based on a multifrontal approach [11]. Parallelism can be exploited within the context of the tree where the dense factorizations at different nodes can be performed at the same time if the nodes are not on the same path of the tree. Although we do exploit this tree parallelism, it is crucially important to exploit parallelism for the factorization operations on the dense matrices at each node (called node parallelism). This is particularly difficult in the case of indefinite matrices because the dense matrix is of the form

$$F = \begin{pmatrix} F_{11} & F_{21}^T \\ F_{21} & F_{22} \end{pmatrix}, \tag{2.1}$$

where the entries in F_{22} are not fully summed, meaning that data from ancestor nodes of the tree are needed before they have their correct value. For that reason, a simple implementation of standard algorithms for symmetric indefinite matrices, such as the Bunch–Kaufman algorithm [3], is not possible since we cannot choose a 2×2 pivot with a diagonal entry in F_{22}.

If the matrix in (2.1) is of order m and F_{11} is a square, symmetric indefinite matrix of order p and F_{21} is rectangular of order $(m-p) \times p$, then the factorization is only done on the first p columns and F_{22} is updated using the computed factors but is not factorized itself. For positive-definite matrices, a Cholesky factorization can be used and an implementation of this for multicore architectures is available in the PLASMA library [18]. When the matrix is indefinite, 1×1 and 2×2 pivots are needed to guarantee the numerical stability of the factorization [12] and, as noted earlier, pivots can only be chosen from within F_{11}. For robustness and accuracy, the *threshold partial pivoting* (TPP) strategy is generally employed. This technique, used in the MA57 solver [7], enables stability in the factorization of indefinite matrices by bounding the entries in the factor L, i.e., $|l_{ij}| < u^{-1}$ for a certain threshold u. Note that, because pivoting is restricted to the diagonal part of the matrix, the algorithm

[1]www.numerical.rl.ac.uk/spral.

may only find $q < p$ suitable pivots for a given value of u. When this occurs, the uneliminated $p - q$ columns are *delayed*, which means that they are eliminated later at an ancestor node in the tree.

Because of its numerical robustness, the TPP strategy is widely used in sparse LDL^T solvers. However, at each step of the factorization, determining whether a pivot is numerically acceptable requires the availability of all the coefficients in the column being processed. The parallelization of the TPP algorithm is based on a block-column partitioning approach. The use of two-dimensional (2D) block partitioning, as used by the Cholesky algorithm in the case of positive-definite matrices, incurs significant overheads and does not give any more parallelism than the 1D partitioning. In the HSL[2] library, the two state-of-the-art parallel sparse direct solvers for indefinite systems are `HSL_MA86` [13] and `HSL_MA97` [14]. They both implement a LDL^T factorization with TPP and exploit parallelism by using task-based algorithms based on a 1D partitioning of the matrix F. The main difference in the dense kernels of these two solvers is that `HSL_MA97` uses a recursive algorithm and is a bitwise reproducible implementation.

Another technique for handling instability in the LDL^T factorization of indefinite systems is to perturb the system whenever a potential case of instability is detected, such as when a pivot is too small during the factorization. The PARDISO solver [17], available in the MKL library,[3] implements a so-called supernodal Bunch–Kaufman (SBK) algorithm which uses 1×1 and 2×2 pivoting as in the Bunch–Kaufman algorithm [3] on the F_{11} submatrix together with a pivot perturbation strategy to prevent pivots from being too small compared to a threshold value. The need for using a perturbation technique is related to the fact that pivots can only be chosen from the F_{11} block which can be arbitrarily close to singular in the indefinite case. In addition coefficients in the F_{21} block can be arbitrarily large relative to entries in the F_{11} block. This can be partially alleviated by prescaling and reordering using algorithms similar to that of [9]. However, the algorithm is still unstable and the addition of perturbations can make it difficult to preserve inertia, which is essential for some optimization applications. Note that even when using the perturbation technique, the SBK algorithm is unstable and might lead to inaccurate results when computing the solution. For this reason, this algorithm is not as robust as the TPP algorithm and must generally be associated with an iterative refinement step. In some very ill-conditioned systems even this may not be enough to obtain an accurate solution [14].

3. A posteriori threshold pivoting algorithm.

In this section, we introduce our new LDL^T factorization algorithm based on a pivoting strategy called APTP that is applied to each dense frontal matrix. The factorization is based on a 2D partitioning of the matrix into square blocks that increases the parallelism compared to the traditional TPP algorithm using block-column partitioning. Our new algorithm is made possible by exploiting the following two techniques:

Fail-in-place approach that consists in keeping the failed columns in place and handling them at the end of the factorization. In this case, these columns must be updated during the factorization.

Speculative execution that consists in speculatively running a task assuming that no numerical issues have occurred in other tasks that might affect the current one. This requires doing a backup of entries and implementing a backtracking strategy if numerical instability is detected.

[2]www.hsl.rl.ac.uk.

[3]https://software.intel.com/mkl.

Using these two techniques it is possible to design a task-based LDL^T algorithm that offers more parallelism than the normal TPP strategy. The use of speculative execution makes it possible to process block columns in parallel using 2D partitioning, and the fail-in-place approach greatly reduces complexity and data movement and avoids communication because we do not permute rows and columns outside their blocks.

Note that using speculative execution comes at the cost of storing backups for the tasks that are speculatively executed. This overhead is limited because it is generally negligible compared to the workload associated with each block. Also, the fail-in-place strategy has two main drawbacks: first, we need a fallback strategy for handling the failed columns; second, keeping the failed columns up-to-date introduces small granularity tasks in the DAG. This, however, has low impact on the performance due to the small amount of failed pivots encountered in practice during the factorization.

Algorithm 3.1 A posteriori threshold pivoting LDL^T factorization.

```
1: Let nblk = $\lceil n/nb \rceil$ be the number of block columns
2: Let mblk = $\lceil m/nb \rceil$ be the number of block rows
3: for $j = 1$ to nblk do
4:   Factor( $A_{jj}$, $nelim_j$ ) ▷ Factorize block on diagonal
5:   for $i = 1$ to $j - 1$ do
6:     ApplyT( $A_{ji}$, $nelim_j$; $A_{jj}$ ) ▷ Apply pivot to superdiagonal blocks
7:   end for
8:   for $i = j + 1$ to mblk do
9:     ApplyN( $A_{ij}$, $nelim_j$; $A_{jj}$ ) ▷ Apply pivot to subdiagonal blocks
10:  end for
11:  Adjust($nelim_j$; All blocks $A_{:j}$ and $A_{j:}$ ) ▷ Reduce $nelim_j$
12:  for $k = 1$ to $j - 1$ do
13:    for $i = k$ to $j - 1$ do
14:      UpdateTN( $A_{ik}$; $A_{ji}$, $A_{jk}$, $nelim_j$ ) ▷ Update previously failed columns
15:    end for
16:    for $i = j$ to mblk do
17:      UpdateNT( $A_{ik}$; $A_{ij}$, $A_{jk}$, $nelim_j$ ) ▷ Update previously uneliminated
         columns
18:    end for
19:  end for
20:  for $k = j$ to nblk do
21:    for $i = k$ to mblk do
22:      UpdateNN( $A_{ik}$; $A_{ij}$, $A_{kj}$, $nelim_j$ ) ▷ Update trailing submatrix
23:    end for
24:  end for
25: end for
```

The LDL^T factorization with the APTP strategy is shown in Algorithm 3.1. 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 parameters after it. The variable $nelim_j$ is special as we do not express task dependencies involving it, but instead use atomic updates. The kernels are as follows:

Factor($A_{jj}, nelim_j$**)** computes the LDL^T factorization of a block on the diagonal as follows:

1. Store a backup of block A_{jj}.
2. Perform a pivoted factorization $P_j A_{jj} P_j^T = L_{jj} D_{jj} L_{jj}^T$, where P_j is a permutation matrix.
3. Initialize $nelim_j$ to the block size of A_{jj}.

ApplyN($A_{ij}, nelim_j$**;** A_{jj}**)** applies the LDL^T factorization from the diagonal block A_{jj} to a subdiagonal block A_{ij}. This is done by

1. storing a backup of block A_{ij};
2. performing the operation $L_{ij} = A_{ij} P_j (L_{jj} D_{jj})^{-T}$;
3. finding the first column $nelim_{ij}$ in L_{ij} that contains a failed entry, i.e., $l_{pq} > u^{-1}$ and performing the atomic reduction $nelim_j = \min(nelim_j, nelim_{ij})$.

ApplyT($A_{ji}, nelim_j$**;** A_{jj}**)** applies the LDL^T factorization from the block A_{jj} to the columns containing the failed entries in the block A_{ji} on its left. This is done by

1. storing a backup of block A_{ji};
2. performing the operation $L_{ji} = (L_{jj} D_{jj})^{-1} P_j^T A_{ji}$ on the columns corresponding to the failed pivots;
3. finding the first row $nelim_{ji}$ in L_{ji} that contains a failed entry, i.e., $l_{pq} > u^{-1}$ and performing the atomic reduction $nelim_j = \min(nelim_j, nelim_{ji})$.

Note that if there are no failed entries in block A_{ji}, then this operation becomes a no-op.

Adjust($nelim_j$**; all blocks** $A_{:j}$ **and** $A_{j:}$**)** ensures $nelim_j$ has been atomically reduced in all the blocks of $A_{:j}$ and $A_{j:}$. Decrements $nelim_j$ by one if we would otherwise accept only the first column of a 2×2 pivot.

UpdateNN(A_{ik}**;** A_{ij}**,** A_{kj}**,** $nelim_j$**)** performs the update operations on a block that is on the right of the eliminated block column j.

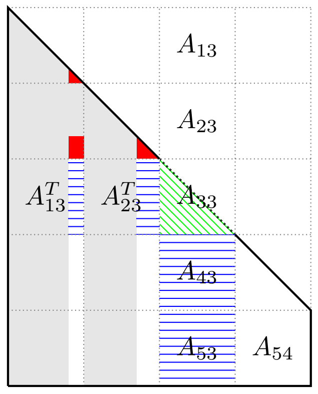
1. If block A_{ik} is in the same column as the pivotal column, i.e., $k = j$, then restores the failed entries from the backup.
2. Performs the update operation $A_{ik} = A_{ik} - L_{ij} D_{jj} L_{kj}^T$. If the task operates on column j only the uneliminated entries (that have just been restored) are updated; otherwise the whole block is updated.

UpdateNT(A_{ik}**;** A_{ij}**,** A_{jk}**,** $nelim_j$**)** performs the update operations on a block that is at the bottom left of the eliminated block column j.

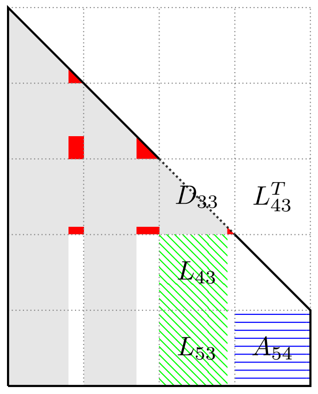
1. If block A_{ik} is in the same row as the eliminated column, i.e., $i = j$, then restores the failed entries from the backup.
2. Performs the update operation: $A_{ik} = A_{ik} - L_{ij} D_{jj} L_{jk}$ on the uneliminated entries.

UpdateTN(A_{ik}**;** A_{ji}**,** A_{jk}**,** $nelim_j$**)** performs the update operations on a block that is at the top left of the eliminated block column j. The operation $A_{ik} = A_{ik} - L_{ji}^T D_{jj} L_{jk}$ is performed on the uneliminated entries.

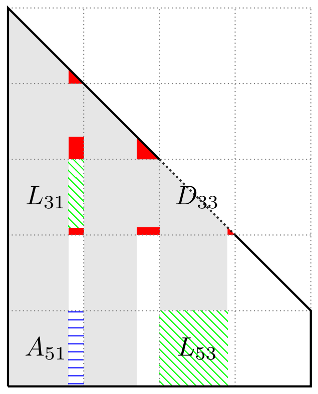
The APTP algorithm is illustrated in Figure 1 on a 2D block-partitioned matrix with 5 block-rows and 4 block-columns. In this example, iterations $j = 1, 2$ have completed and we show the processing of the third block-column. In Figure 1(a) we show the ApplyN and ApplyT operations, respectively, on subdiagonal blocks and superdiagonal blocks stored as transpose in third block-row. Upon completion of the *apply* kernels, we perform the update operations on the uneliminated entries represented in white, and the failed entries represented in red, using the three different



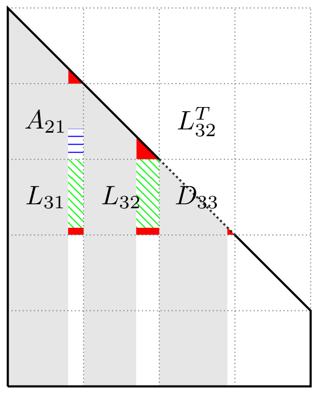
(a) ApplyN and ApplyT operations, respectively, on subdiagonal and superdiagonal blocks.



(b) UpdateNN operation on block A_{54}.



(c) UpdateNT operation on block A_{51}.



(d) UpdateTN operation on block A_{21}.

FIG. 1. *Illustration of the APTP strategy presented in Algorithm* 3.1*, where iterations* $j = 1, 2$ *have completed. Eliminated entries are represented in gray, uneliminated entries in white, failed entries in red, and updates in blue (horizontal lines). Following the factorization of block* A_{33}*, we show ApplyN and ApplyT operations* 1(a)*. Upon completion of these tasks, the trailing submatrix is updated using an UpdateNN kernel* 1(b)*, as well as the uneliminated entries from previous iteration using an UpdateNT kernel* 1(c) *and the failed entries using UpdateTN kernel* 1(d)*.*

kernels described above depending on the position of the updated block with respect to the block on the diagonal. The trailing submatrix is updated using kernel UpdateNN with the example of block A_{54} 1(b) updated using factors L_{53} and factor L_{34} stored as transpose. The uneliminated entries at the previous steps are updated with kernel UpdateNT, with the example of block A_{51} being updated using the factors L_{31} and L_{53} 1(c). Finally, the failed entries are updated with kernel UpdateTN with the example of block A_{21} being updated using the factors L_{31} and L_{23} stored as transpose 1(d).

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 usually been updated since they failed) using either APTP or TPP, or passed directly to the parent.

Note that there are several options for performing the pivoted factorization of the block A_{jj} in the *Factor* kernel including using complete pivoting or an implementation

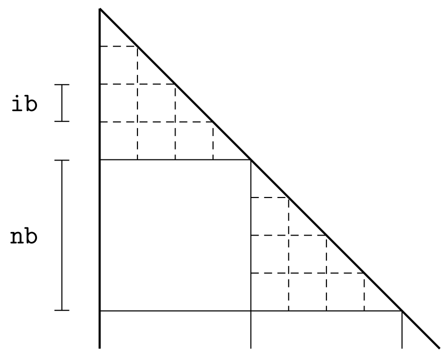


FIG. 2. *Blocking strategy for the APTP factorization. The input matrix is partitioned by blocks using an* nb *parameter and the various kernels operate on these blocks. In the Factor task, the block on the diagonal block is partitioned using a block size ib.*

of TPP. In our implementation, we use a two-level approach involving an inner block size *ib* in addition to the outer block size *nb*. This hierarchical strategy is illustrated in Figure 2. APTP is performed in parallel using the outer block size *nb* and recurses to perform APTP in serial with the smaller inner block size *ib* so that the factorization of the blocks on the diagonal is serial. The blocks of order *ib* on the diagonal of these are factorized using complete pivoting. We chose the values of 256 for *nb* and 32 for *ib* based on machine characteristics and some trial experiments.

4. Relation between full pivoting and threshold pivoting.

As described in the previous section, at the most fine-grained level we use complete pivoting on small dense blocks of order *ib*. The reason for doing so is that such a block fits in cache, so there is little penalty in 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).

The complete pivoting algorithm proceeds columnwise in a traditional fashion, but the choice of the next pivot is determined by the location of the maximum entry as shown in Algorithm 4.1, where the threshold δ is used to define singularity. 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 to 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.

4.1. Stability.

Algorithm 4.1 is at least 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 note that a value of u greater than 0.5 may result in it being impossible to complete the pivoting [12]. We now study the stability of Algorithm 4.1 for each of the three nonsingular execution paths. For ease of reference, we relabel $a_{11} = a_{mm}, a_{21} = a_{tm}, a_{22} = a_{tt}$.

Algorithm 4.1 Complete pivoting algorithm.

```
1: Find uneliminated entry with the maximum magnitude in position $(t, m)$ with
   value $a_{tm}$.
2: if $|a_{tm}| < \delta$ then
3:    All remaining pivots are considered to be zero.
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
```

Immediate 1 × 1 pivot case. Trivially

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

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

$$|a_{21}| \geq |a_{11}| \geq |a_{22}|, \tag{4.1}$$

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

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

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

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

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

The first condition ensures that the pivot is nonsingular, while the latter two ensure any cancellation in the calculation of Δ is insignificant. Given condition (4.1), observe that (c)⇒(b). Furthermore, since $|a_{21}| \geq \delta$, it follows that (c)⇒(a). We assume that condition (c) holds and 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 $|a_{21}| \geq |a_{i1}|, |a_{i2}|$, and the triangular inequality we get

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

By condition (c) we have $\frac{1}{\Delta} \leq 2\frac{1}{|a_{21}||a_{21}|}$, combined with (4.1) to give

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

This corresponds to a traditional pivoting threshold of $u = 0.25$.

Failed 2×2 pivot case. Note that it is possible for condition (c) to be violated; consider, for example,

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

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. Further, from (4.1) we have $|a_{21}a_{21}| \geq |a_{11}a_{22}|$. Combining these allows us to write

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

Without loss of generality, assume that $|a_{11}| \geq |a_{22}|$. After rearrangement 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:

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

5. Multifrontal LDL^T factorization in `SSIDS`.

In `SSIDS`, we have implemented a sparse LDL^T factorization using the APTP algorithm presented in section 3 within a multifrontal scheme [11]. The code for the factorization is written in C++ and uses the OpenMP tasking features from Version 4 of the standard.

In the multifrontal algorithm, the factors and the dependencies between the columns are represented by an *assembly tree*. As discussed in section 2, at each node we have a dense $m \times m$ matrix, called a *frontal matrix* F, that consists of a set of p fully summed columns, ready to be eliminated, and $m-p$ non-fully-summed columns called *contribution blocks* that are passed to the parent. The multifrontal factorization is done by traversing the assembly tree using a topological order, and at each frontal matrix we

1. assemble the contribution blocks coming from the children;
2. factorize the fully summed columns eliminating $q \leq p$ pivots;
3. form the contribution block.

As discussed in section 2, several levels of parallelism can be exploited in the multifrontal factorization including *tree-level* parallelism due to the fact that frontal matrices in independent branches of the assembly tree can be processed in parallel and *node-level* parallelism due to the parallelization of the operations done on the frontal matrices.

In our implementation, the fully summed columns are stored directly into the matrix factors, while the $(m-p) \times (m-p)$ contribution block is stored separately in temporary memory. The routines operating on the frontal matrix are the following:

- **assemble_pre()**
  1. Allocates memory for both the fully summed columns and the contribution block ensuring that the memory for the fully summed columns includes the space needed to accommodate the delayed columns.
  2. Copy the delayed columns from the children.
  3. Assembles contributions from children into the fully summed columns only.
- **factor()** Factorizes the fully summed columns and calculates the contribution block.
- **assemble_post()** Assembles contributions from children into the contribution block.

Note that, in the `assemble_pre()` routine, the memory associated with the fully summed columns should be zeroed before being assembled. Although the contribution block should be zeroed as well before forming the contribution blocks in the `factor` routine, it is done directly since the `_gemm()` operations can be told to treat memory as zero before the calculation. This removes the need for explicitly zeroing, which is an expensive operation.

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 precleared 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 within `SSIDS`:

- Entries of the factors are managed using a *stack allocator* as they are never released (until the user releases them). 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 are allocated using a *buddy system allocator* [16]. This allows memory to be recovered without loss, at the expense of requiring more space than some other methods.

Figure 3 shows a pseudocode for the parallel implementation of the multifrontal factorization in `SSIDS` using OpenMP 4. In this code, `nfronts` denotes the number of fronts (nodes) in the assembly tree, and an OpenMP task is created for each node. These are processed in a topological ordering. When nodes are large enough, the operations are parallelized via nested OpenMP tasks. We make use of the `taskgroup` directive to ensure that all the tasks submitted in each routine are completed before the next operation begins. The `factor` routine is parallelized using the LDL^T algorithm with APTP pivoting as presented in section 3, whereas the assembly operations use a 1D parallelization across block columns of each child node.

In Figure 3, the two `depend` clauses associated with the `task` directive ensure that a front is processed only when all of its child nodes have been processed. In this pseudocode this is done using `front(f)`, which is a symbolic representation of front `f`, and `parent(f)`, which is a symbolic representation of its parent. Note that, for the parent front we use an *in* dependency even though the dependency goes the other way around, which means that in this case an *out* dependency would have been semantically correct. However, this would serialize the processing of these child fronts even though they can be processed in parallel. Alternatively, the same dependency could have been enforced with an *in* dependency on the child fronts but as

```
#pragma omp taskgroup
{
  for (f = 0; f < nfronts; f++) {

     #pragma omp task depend(inout: front(f)) depend(in: parent(f))
    {
      // Allocates memory for both fully-summed columns and contribution
      // block and assembles contributions into fully-summed columns
      assemble_pre(f, children(f));
      // Factorizes fully-summed columns in parallel using APTP
      // strategy
      factor(f);
      // Assembles contributions into contribution block
      assemble_post(f, children(f));

      // Wait for the completion of the frontal matrix factorization
    }
  }
}
```

FIG. 3. *Pseudocode for parallel multifrontal factorization implemented in* SSIDS *using OpenMP* 4.

OpenMP doesn't allow an arbitrary number of dependencies, this solution cannot be used.

6. High-level parallel strategy. The vast majority of multicore architectures are composed of several NUMA nodes, which means that for each processor, data access times vary depending on the memory location of the data. In order to get the best performance out of these architectures it is therefore important to take into account data locality to minimize memory access times, especially when increasing the number of cores involved in the computations. This can be done by prioritizing intraresource communications over the more expensive interresource communications.

In the SSIDS solver, this is achieved through tree parallelism. The assembly tree is split into leaf subtrees that are mapped onto the different NUMA nodes such that the workload is well balanced between the resources and the remaining root subtrees are executed on the combined resources.

Assignment of leaf subtrees to resources is done in a round robin fashion following Algorithm 6.1. First, the leaf subtrees are ordered in descending order of floating-point operation count. The largest subtree is assigned to the first resource, the next largest subtree to the next resource, and so forth until all subtrees are assigned. If there are more subtrees than resources, we loop back around again.

Algorithm 6.1 `find_subtree_partition()`.

```
Compute the workload associated with the subtrees rooted at each node.
Initialize partitions with the independent trees in the assembly forest.
Assign trees to resources using a round robin algorithm and calculate *balance*.
while  $balance > b_{\min}$ and # iterations < max_itr  do
  Find largest subtree in partition.
  Create a new partitions from the child subtrees of the root node for this subtree.
  Sort subtrees in descending order of flops.
  Assign subtrees to resources using a round robin algorithm and calculate *balance*.
end while
Return partition with least *balance* value encountered.
```

```
!$omp parallel proc_bind(spread) num_threads(num_regions)

  ! Represents a NUMA region
  !$omp parallel proc_bind(close) num_threads(region_num_threads)

  ! Factor subtree
  call factor_leaf_subtrees()

  !$omp end parallel

!$omp end parallel

!$omp parallel num_threads(total_threads)

  call factor_root_subtrees()

!$omp end parallel
```

FIG. 4. *Pseudocode for high-level parallel factorization implemented in* SSIDS *using OpenMP* 4.

For a tree partitioning, we define the load balance as

$$balance = \frac{\max_i x_i}{\frac{1}{n_{\text{res}}} \sum_j x_j},$$

where n_{res} is the number of resources and x_i is the total number of floating point operations assigned to resource i. In a perfectly balanced situation, $balance = 1.0$. In Algorithm 6.1, the minimum value of *balance* we are willing to accept is denoted by $b_{\min}$ and we bound the number of iterations using the `max_itr` variable to prevent the root subtrees from growing too large.

Within `SSIDS`, the detection of the machine topology information is achieved through the hwloc library [2]. The OpenMP implementation of the subtree mapping resulting from the the tree partitioning is shown in Figure 4. We use the `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. Synchronizations are done by locating the leaf subtree and root subtree calculations in different OpenMP parallel regions: one must complete before the other can start.

7. Experimental results.

For testing the new version of `SSIDS` we use a large set of test matrices from the SuiteSparse matrix collection[4] [5]. This test set is divided into two subsets: the *easy-indefinite* test set, described in Table 1, for matrices that require few delayed pivots during the factorization, and the *hard-indefinite* test set, described in Table 2, for matrices that require significant pivoting during the factorization. In the case of *easy-indefinite* matrices, no scaling is performed prior to the factorization whereas *hard-indefinite* matrices are scaled and ordered using a matching-based ordering and scaling.

Our tests were performed on a compute node of the Kebnekaise system located at the High Performance Computing Center North (HPC2N), Umeå University.[5] The compute node contains 28 Intel Xeon E5-2690 v4 (Broadwell) cores, clocked at 2.6 GHz, organized into two NUMA islands with 14 cores in each. Each compute node thus has a total of 28 cores and a theoretical peak of 1164.8 GFlop/s in double

[4]https://sparse.tamu.edu/.
[5]https://www.hpc2n.umu.se/resources/hardware/kebnekaise.

TABLE 1
Easy indefinite test set. Statistics as reported by the analyze phase of SSIDS *with default settings, assuming no delays. Ordered by increasing number of flops for factorization.*

#	Problem	n $\times 10^3$	$nz(A)$ $\times 10^6$	$nz(L)$ $\times 10^6$	$flops$ $\times 10^9$
1	Oberwolfach/t2dal	4.26	0.02	0.28	0.02
2	GHS_indef/dixmaanl	60.00	0.18	1.58	0.05
3	Oberwolfach/rail_79841	79.84	0.32	4.43	0.33
4	GHS_indef/dawson5	51.54	0.53	5.69	0.90
5	Boeing/bcsstk39	46.77	1.07	9.61	2.66
6	Boeing/pct20stif	52.33	1.38	12.60	5.63
7	GHS_indef/copter2	55.48	0.41	12.70	6.10
8	GHS_indef/helm2d03	392.26	1.57	33.00	6.16
9	Boeing/crystk03	24.70	0.89	10.90	6.26
10	Oberwolfach/filter3D	106.44	1.41	23.80	8.71
11	Koutsovasilis/F2	71.50	2.68	23.70	11.30
12	McRae/ecology1	1000.00	3.00	72.30	18.20
13	Cunningham/qa8fk	66.13	0.86	26.70	22.10
14	Oberwolfach/gas_sensor	66.92	0.89	27.00	22.10
15	Oberwolfach/t3dh	79.17	2.22	50.60	70.10
16	Lin/Lin	256.00	1.01	126.00	285.00
17	PARSEC/H2O	67.02	2.22	234.00	1290.00
18	GHS_indef/sparsine	50.00	0.80	207.00	1390.00
19	PARSEC/Ge99H100	112.98	4.28	669.00	7070.00
20	PARSEC/Ga10As10H30	113.08	3.11	690.00	7280.00
21	PARSEC/Ga19As19H42	133.12	4.51	823.00	9100.00

TABLE 2
Hard indefinite test set. Statistics as reported by the analyze phase of SSIDS *with default settings, using matching-based ordering, assuming no delays. Ordered by increasing number of flops for factorization.*

#	Problem	n $\times 10^3$	$nz(A)$ $\times 10^6$	$nz(L)$ $\times 10^6$	$flops$ $\times 10^9$
1	TSOPF/TSOPF_FS_b39_c7	28.22	0.37	2.61	0.26
2	TSOPF/TSOPF_FS_b162_c1	10.80	0.31	1.89	0.36
3	QY/case39	40.22	0.53	3.87	0.40
4	TSOPF/TSOPF_FS_b39_c19	76.22	1.00	7.28	0.75
5	TSOPF/TSOPF_FS_b39_c30	120.22	1.58	11.10	1.10
6	GHS_indef/cont-201	80.59	0.24	7.12	1.11
7	GHS_indef/stokes128	49.67	0.30	6.35	1.16
8	TSOPF/TSOPF_FS_b162_c3	30.80	0.90	6.37	1.41
9	TSOPF/TSOPF_FS_b162_c4	40.80	1.20	7.32	1.43
10	GHS_indef/ncvxqp1	12.11	0.04	3.56	2.52
11	GHS_indef/darcy003	389.87	1.17	23.20	3.01
12	GHS_indef/cont-300	180.90	0.54	17.20	3.58
13	GHS_indef/bratu3d	27.79	0.09	7.49	4.72
14	GHS_indef/cvxqp3	17.50	0.07	6.33	5.27
15	TSOPF/TSOPF_FS_b300	29.21	2.20	13.40	6.92
16	TSOPF/TSOPF_FS_b300_c1	29.21	2.20	13.50	7.01
17	GHS_indef/d_pretok	182.73	0.89	24.80	7.42
18	GHS_indef/turon_m	189.92	0.91	24.70	7.60
19	TSOPF/TSOPF_FS_b300_c2	56.81	4.39	27.00	14.10
20	TSOPF/TSOPF_FS_b300_c3	84.41	6.58	40.50	21.40
21	GHS_indef/ncvxqp5	62.50	0.24	22.90	24.30
22	GHS_indef/ncvxqp3	75.00	0.27	39.30	63.70
23	GHS_indef/ncvxqp7	87.50	0.31	51.00	101.00
24	Schenk_IBMNA/c-big	345.24	2.34	110.00	295.00

Table 3
Factorization times (in seconds) for SSIDS, HSL_MA86, HSL_MA97, and PARDISO for matrices taken from the easy indefinite test set run on the Kebnekaise compute node. For SSIDS, column loc. indicates that we use data locality whereas flat corresponds to the case where NUMA effects are ignored.

Problem	SSIDS		HSL_MA86	HSL_MA97	PARDISO
	loc.	flat			
Oberwolfach/t2dal	0.03	0.02	0.04	0.01	0.01
GHS_indef/dixmaanl	0.07	0.04	0.09	0.02	0.06
Oberwolfach/rail_79841	0.06	0.05	0.12	0.08	0.07
GHS_indef/dawson5	0.37	0.08	0.22	0.08	0.04
Boeing/bcsstk39	0.11	0.10	0.31	0.21	0.05
Boeing/pct20stif	0.16	0.14	1.02	18.88	0.12
GHS_indef/copter2	0.16	0.16	0.28	0.16	0.12
GHS_indef/helm2d03	0.17	0.18	0.49	0.20	0.25
Boeing/crystk03	0.16	0.16	0.32	0.16	0.09
Oberwolfach/filter3D	0.15	0.17	0.43	0.14	0.12
Koutsovasilis/F2	0.18	0.17	0.68	0.69	0.16
McRae/ecology1	0.30	0.36	0.96	0.46	0.78
Cunningham/qa8fk	0.26	0.23	0.44	0.40	0.19
Oberwolfach/gas_sensor	0.26	0.24	0.41	0.42	0.19
Oberwolfach/t3dh	0.47	0.39	1.05	0.78	0.42
Lin/Lin	1.24	1.17	1.52	3.43	1.35
PARSEC/H2O	4.55	5.38	6.23	28.50	7.39
GHS_indef/sparsine	7.88	8.71	9.00	43.90	12.29
PARSEC/Ge99H100	21.24	23.43	24.48	141.31	38.60
PARSEC/Ga10As10H30	19.28	22.06	27.99	106.53	38.70
PARSEC/Ga19As19H42	23.89	28.85	29.30	175.36	61.63

precision arithmetic. Each CPU core has 32 KB L1 data cache, 32 KB L1 instruction cache, and 256 KB L2 cache. Moreover, for every NUMA island there is 35 MB of shared L3 cache. The total amount of RAM per compute node is 128 GB.

The factorization times obtained with SSIDS and three state-of-the-art solvers HSL_MA86, HSL_MA97, and PARDISO (from MKL library) are reported in Table 3 for the easy indefinite test set and in Table 4 for the hard indefinite test set. There are several parameters in the codes and in each case we have used the defaults set by the code developer. In particular, PARDISO uses two steps of iterative refinement to improve the accuracy of the computed solution when pivot perturbation is used in the factorization. We use the default value for pivot perturbation in PARDISO, which is 1×10^{-8}. In addition, it is possible to use a two-level factorization algorithm in PARDISO, which can improve the scalability of the factorization, but this algorithm is not compatible with matching-based ordering and scaling so we use the default algorithm in our experiments. Note that, in the case of hard indefinite matrices, we did not use HSL_MA86 as it does not offer the capability to use a matching-based ordering without significant additional work. In each table, we show the SSIDS timings using two different configurations: *loc.*, referring to the case where we use the tree partitioning strategy as presented in section 6 to improve the exploitation of data locality, and *flat*, corresponding to the case where we consider the machine topology as flat and thus ignore the data locality during the factorization. In all these tests, SSIDS uses the default threshold parameter $u = 0.01$. In the runs with PARDISO, we follow the default strategy that uses iterative refinement after computing the solution. This is needed to improve its accuracy, which can be badly affected by possible instabilities occurring in the factorization as discussed in section 2. In Table 4 we also report the

TABLE 4
Factorization times (in seconds) for SSIDS*,* HSL_MA86*,* HSL_MA97*, and PARDISO for matrices taken from the hard indefinite test set run on the Kebnekaise compute node. The residual is also reported for both* SSIDS *and PARDISO.*

Problem	SSIDS			HSL_MA97	PARDISO	
	loc.	flat	res.			res.
TSOPF/TSOPF_FS_b39_c7	0.05	0.05	2.01×10^{-14}	0.04	0.02	2.50×10^{-15}
TSOPF/TSOPF_FS_b162_c1	0.06	0.05	7.55×10^{-15}	0.05	0.02	1.32×10^{-15}
QY/case39	0.08	0.07	1.49×10^{-14}	0.10	0.04	9.02×10^{-14}
TSOPF/TSOPF_FS_b39_c19	0.08	0.06	9.72×10^{-15}	0.07	0.06	3.65×10^{-15}
TSOPF/TSOPF_FS_b39_c30	0.12	0.10	2.77×10^{-14}	0.08	0.09	6.25×10^{-15}
GHS_indef/cont-201	0.08	0.09	3.32×10^{-14}	0.07	0.08	1.85×10^{-14}
GHS_indef/stokes128	0.07	0.08	1.09×10^{-15}	0.06	0.04	3.88×10^{-3}
TSOPF/TSOPF_FS_b162_c3	0.08	0.07	1.21×10^{-14}	0.09	0.05	2.36×10^{-15}
TSOPF/TSOPF_FS_b162_c4	0.09	0.09	1.20×10^{-14}	0.08	0.05	2.02×10^{-15}
GHS_indef/ncvxqp1	0.15	0.19	3.16×10^{-14}	0.44	0.09	2.81×10^{-17}
GHS_indef/darcy003	0.15	0.16	3.48×10^{-15}	0.15	0.16	1.31×10^{-16}
GHS_indef/cont-300	0.13	0.14	3.88×10^{-14}	0.11	0.16	7.00×10^{-14}
GHS_indef/bratu3d	0.67	0.15	8.31×10^{-14}	0.23	0.10	8.87×10^{-15}
GHS_indef/cvxqp3	0.18	0.22	1.70×10^{-11}	0.95	0.18	8.92×10^{-7}
TSOPF/TSOPF_FS_b300	0.31	0.29	3.64×10^{-15}	0.29	0.13	5.90×10^{-13}
TSOPF/TSOPF_FS_b300_c1	0.33	0.33	1.54×10^{-15}	0.26	0.13	3.43×10^{-15}
GHS_indef/d_pretok	0.16	0.16	4.65×10^{-16}	0.18	0.14	3.25×10^{-16}
GHS_indef/turon_m	0.16	0.17	5.73×10^{-16}	0.16	0.12	1.13×10^{-16}
TSOPF/TSOPF_FS_b300_c2	0.51	0.41	3.57×10^{-15}	0.34	0.21	3.87×10^{-14}
TSOPF/TSOPF_FS_b300_c3	0.72	0.73	5.04×10^{-15}	0.70	0.29	1.83×10^{-14}
GHS_indef/ncvxqp5	0.61	0.57	5.66×10^{-13}	1.80	0.34	9.97×10^{-12}
GHS_indef/ncvxqp3	1.04	1.01	2.42×10^{-10}	10.39	0.79	1.17×10^{-9}
GHS_indef/ncvxqp7	1.64	1.57	5.50×10^{-9}	11.04	1.19	1.95×10^{-7}
Schenk_IBMNA/c-big	2.99	2.34	1.75×10^{-16}	19.66	2.95	1.41×10^{-16}

scaled residual (backward error) defined as

$$\frac{\|Ax - b\|_2}{\|A\|_1 \|x\|_2 + \|b\|_2}$$

for SSIDS and PARDISO. In our experiments we observed that HSL_MA97 gave similar residuals to SSIDS, and therefore we do not report it for the sake of conciseness in this table.

The experiments on the easy indefinite test set show that SSIDS performs better than the state-of-the-art HSL solvers HSL_MA86 and HSL_MA97 for the large majority of the tested problems. Although PARDISO seems to be marginally better for the smallest problems, SSIDS compares favorably when the problem size increases and the performance gap grows with the problem size. The results also show that the exploitation of data locality as described in section 6 is generally effective above a certain problem size but can reduce the performance on smaller problems. This behavior can be expected as the tree partitioning comes at the cost of additional synchronizations that can be penalizing when the factorization time is bounded by the critical path or when the workload between the partitions is not well balanced (more likely on small problems). Note that, although we observed that most of these easy indefinite problems are associated with small residuals, PARDISO could not reach a solution for matrices Oberwolfach/t2dal and Cunningham/qa8fk as the iterative refinement step failed to converge. PARDISO is only able to solve these two matrices if they are preprocessed with scaling and ordering but this is unnecessary for SSIDS.

This is advantageous because the scaled problem may come at a higher computational cost for the factorization as well as larger number of entries in the factors.

Experiments on the hard indefinite matrices show that SSIDS performs favorably compared to HSL_MA97 and, as in the case of easy indefinite matrices, the performance gap increases with the problem size, which shows that SSIDS is better at exploiting parallelism on these numerically challenging problems than HSL_MA97. On this test set, SSIDS and PARDISO achieve similar performance although PARDISO seems marginally better on smaller problems as we observed in the case of easy indefinite problems. The classification for hard indefinite problems was based on the difficulty found during numerical pivoting, particularly on the number of delayed pivots. However, for the runs of all codes on these problems we first scale them using a matching based algorithm [9]. The effect is that some of the matrices are not so "hard" after the scaling. Thus the fact that the residuals for these easier problems appear quite good for PARDISO is a little deceptive since the code is still using iterative refinement that is not possible to switch off. Indeed, we note that due to the unstable nature of the factorization algorithm used in PARDISO, iterative refinement is used by default in the solver when solving the sparse linear system (1.1) but the timings reported in Table 4 do not include this cost which can become relatively large for such numerically challenging problems. In other work, which specifically addresses parallelism in the back and forward substitution steps [4], we discuss these aspects in more detail. On the other hand, the values for the residuals (for the default threshold parameter $u = 0.01$) show that, on nearly all the tested problems, SSIDS achieves small backward errors. This is not always the case with PARDISO. For example, with matrix GHS_indef/stokes128 PARDISO achieves a residual of 3.88×10^{-3} despite using scaling and iterative refinement whereas SSIDS gives a residual close to machine precision. Similarly, with matrix GHS_indef/cvxqp3, PARDISO achieves a residual of 8.92×10^{-7} whereas SSIDS gives a residual several orders of magnitude smaller equal to 1.70×10^{-11}. These experimental results show that our APTP algorithm allows us to achieve similar performance to PARDISO but with the same robustness provided by the traditional TPP algorithm implemented in HSL_MA97 and HSL_MA86 on the tough indefinite problems. On the other hand, the APTP algorithm is fast when solving easy indefinite problems on multicore architectures.

In Table 5 we report, for a selection of test matrices, the factorization time, the number of number of delayed pivots, and the number of failed columns after the APTP pass for a value of the threshold parameter varying from 10^{-1} to 10^{-3}. We first observe that the number of delayed pivots can be significant for hard indefinite matrices despite the use of scaling techniques. In these cases reducing the value of the threshold parameter decreases the number of delayed columns leading to a reduction in the factorization time. For example, in the factorization of matrix GHS_indef/ncvxqp7, reducing u from 10^{-1} to 10^{-3} allows us to dramatically reduce the number of delayed columns and to halve the factorization time. This behavior can be observed on many other problems and is expected since we know that delaying columns is associated with more floating-point operations and more data movement. However, this is not the only factor impacting the performance of the factorization. As explained in section 3, the APTP algorithm is followed by a fallback routine that tries to eliminate the columns that failed in the first pass. The uneliminated columns could be directly passed to the parent nodes and be eliminated later at an ancestor node in the assembly tree but this was deemed to be potentially too costly. For this reason, the default fallback strategy consists in using a sequential TPP strategy on the remaining columns. A larger value of u naturally leads to more columns being uneliminated at the end of

Table 5

Factorization times (in seconds) and number of delayed and failed columns with SSIDS *on a subset of our test matrices from both the easy indefinite and the hard indefinite test sets. These results are obtained for different values of the threshold parameter u. Note that the factorization times for the default value in* SSIDS *($u = 0.01$) are already reported in Tables* 3 *and* 4.

Problem	Time (s)		delays			failed col.		
$u =$	10^{-1}	10^{-3}	10^{-1}	10^{-2}	10^{-3}	10^{-1}	10^{-2}	10^{-3}
Boeing/pct20stif	0.20	0.16	239	13	2	1974	192	5
GHS_indef/sparsine	61.33	5.38	358	24	1	9547	1562	104
PARSEC/Ga10As10H30	22.69	18.69	3	0	0	1157	35	0
TSOPF/TSOPF_FS_b39_c7	0.18	0.06	4672	997	174	4341	1248	297
QY/case39	0.27	0.05	8744	2318	1094	7462	2960	1480
TSOPF/TSOPF_FS_b39_c19	0.28	0.06	14270	3464	416	11758	4175	720
TSOPF/TSOPF_FS_b39_c30	0.61	0.08	23166	5031	663	18613	6277	1324
GHS_indef/stokes128	0.08	0.08	0	0	0	0	0	0
GHS_indef/ncvxqp1	0.16	0.15	92	78	77	122	97	92
GHS_indef/cvxqp3	0.23	0.28	469	26	14	670	37	19
TSOPF/TSOPF_FS_b300	0.41	0.22	3276	1235	143	11541	5290	1446
TSOPF/TSOPF_FS_b300_c1	0.90	0.22	3685	1599	141	12038	6142	1562
GHS_indef/bratu3d	0.57	0.58	27	27	27	239	172	77
GHS_indef/ncvxqp5	1.15	0.46	6977	60	5	11190	1164	124
TSOPF/TSOPF_FS_b162_c1	0.08	0.05	1476	401	10	2098	603	18
GHS_indef/ncvxqp7	2.97	1.18	2777	164	21	5763	458	35

the APTP factorization and therefore having more columns eliminated by the TPP kernel, which is generally associated with higher factorization times. In the case of the PARSEC/Ga10As10H30 matrix, for example, the factorization time is strongly impacted by the value u but not the number of delayed pivots. In this case, the decrease in the factorization time when the parameter u goes from 10^{-1} to 10^{-3} is related to the fact that the number of failed columns after using the APTP algorithm drops from 1157 to 0. The main reason for the fallback strategy being so penalizing lies in the fact that our implementation of TPP in SSIDS is not only sequential but is also unblocked, which means that it cannot make use of Level-3 BLAS kernels. We computed the backward error for all the runs. With $u = 10^{-1}$ it was normally close to rounding, and we were pleased to see that the degradation when increasing the threshold parameter was only around one order of magnitude.

8. Accelerating the factorization using GPUs.

As mentioned in the introduction, SSIDS was originally designed to run on only a GPU. This means that the factorization was done entirely on the GPU, without exploiting the capability of the CPU cores available on the architecture. In addition, this approach had the disadvantage of being constrained by the memory available on the GPU, which can be quite limited compared to the amount of available RAM.

In order to exploit GPU devices on heterogeneous CPU-GPU architectures in the new version of SSIDS, we enable the factorization of subtrees on the GPU. For assigning subtrees between NUMA regions and GPUs, we use the same partitioning strategy as presented in Algorithm 6.1, but we introduce a parameter α_i corresponding to the speed of the resource i, using GFlop/s as a metric and compute the balance as follows:

$$balance = \frac{\max_i(x_i/\alpha_i)}{\frac{1}{n_{\text{res}}}\sum_j(x_j/\alpha_j)},$$

where we set $\alpha_i = 1.0$ for NUMA regions and set it as the performance ratio between GPU and NUMA regions for GPU resources. For example, if the GPU device is

Table 6

Factorization times (s) obtained with `SSIDS` *on a subset of test matrices when using both CPUs and GPUs compared to the CPU-only version.*

Problem	t (s)	t (s) CPU-GPU				
$\alpha =$		0.75	1.0	1.25	2.0	10.0
GHS_indef/helm2d03	0.15	0.15	0.15	0.10	0.10	0.10
GHS_indef/copter2	0.15	0.12	0.11	0.07	0.07	0.07
Boeing/crystk03	0.15	0.11	0.11	0.05	0.05	0.05
Oberwolfach/filter3D	0.14	0.13	0.13	0.08	0.08	0.08
Boeing/pct20stif	0.13	0.10	0.11	0.06	0.07	0.06
Koutsovasilis/F2	0.16	0.16	0.16	0.11	0.11	0.12
Cunningham/qa8fk	0.21	0.19	0.18	0.14	0.15	0.15
Oberwolfach/gas_sensor	0.23	0.20	0.19	0.15	0.20	0.19
McRae/ecology1	0.30	0.33	0.34	0.22	0.22	0.22
Oberwolfach/t3dh	0.39	0.34	0.34	0.34	0.39	0.38
Lin/Lin	1.22	0.76	0.76	0.67	0.79	0.80
PARSEC/H2O	5.80	3.42	3.44	2.81	2.93	2.91
GHS_indef/sparsine	8.64	5.66	5.66	6.99	7.03	3.86
PARSEC/Ge99H100	27.76	14.15	14.35	16.73	18.35	18.45
PARSEC/Ga10As10H30	25.89	17.35	17.51	17.95	18.08	18.26
PARSEC/Ga19As19H42	33.24	19.19	19.20	18.76	17.42	17.42

twice as fast as a NUMA region we set $\alpha_i = 2.0$ for GPUs. Finding a suitable value for α_i is not trivial in practice and using theoretical peaks might not be optimal. This is due to the fact that the speed of the various CPUs and GPUs depends on the workload associated with each subtree partition. In `SSIDS`, the value α for GPU resources is chosen empirically via an optional parameter whose default value is set to 1.0. In addition, we use a parameter gpu_{min} corresponding to the minimum number of flops required for a subtree to be processed on a GPU. This threshold prevents the solver from running small subtrees on the GPUs for which the performance would be relatively low on the GPU and would certainly not compensate for the start-up latency and the time to transfer data from the main memory to the GPU memory. Note that in the tree partitioning, the root subtree is run on the CPUs because the GPU kernel cannot receive contributions from children subtrees in the current implementation of `SSIDS`. Note that, in all of our experiments in this section, we use 28 cores on the CPUs and we have the same threshold parameter, u, on the GPU as on the CPUs.

For testing our approach, we ran `SSIDS` on a heterogeneous machine with two NUMA nodes equipped with a 10 cores Intel Xeon CPU E5-2650 v3 (Haswell) clocked at 2.3 GHz, plus one GPU device P100 (Pascal). For this experiment we use the matrices from the easy indefinite test set. In Table 6 we compare the factorization times using CPUs only with using both the CPUs and the GPU for various values of α. In Table 7 we present the floating-point operations associated with the factorization of each problem and the floating-point operations performed by the GPU for the values of α used in Table 6. Note that we do not give the factorization times for the five smallest matrices in the easy indefinite test set because the workload associated with them was below our threshold gpu_{min} and therefore the GPU was not exploited for these problems.

For the first five matrices in Table 6, we observe that, for $\alpha \leq 1.0$, the factorization is entirely done on the CPUs whereas for $\alpha > 1.0$ it is done entirely on the GPU. This is due to the fact that in these cases, if the GPU is faster than a NUMA region, the whole tree is mapped to the GPU. For these problems the factorization times are halved when using the GPU compared to the CPU only execution. For the larger

TABLE 7
Total floating point operations associated to the test problems with the amount of floating point operations performed on the GPU for the various value of α used in Table 6.

Problem	Flops ($\times 10^9$)	GPU Flops ($\times 10^9$)				
$\alpha =$		0.75	1.0	1.25	2.0	10.0
GHS_indef/helm2d03	6.1	0.0	0.0	6.2	6.2	6.2
GHS_indef/copter2	6.1	0.0	0.0	6.1	6.1	6.1
Boeing/crystk03	6.3	0.0	0.0	6.3	6.3	6.3
Oberwolfach/filter3D	8.7	0.0	0.0	8.7	8.7	8.7
Boeing/pct20stif	5.6	0.0	0.0	5.6	5.6	5.6
Koutsovasilis/F2	11.3	0.0	0.0	5.8	5.8	5.8
Cunningham/qa8fk	22.1	6.4	6.4	10.4	10.4	10.4
Oberwolfach/gas_sensor	22.1	5.4	5.4	11.5	5.4	5.4
McRae/ecology1	18.2	0.0	0.0	9.0	9.0	9.0
Oberwolfach/t3dh	70.1	25.1	25.1	25.7	23.3	23.3
Lin/Lin	285.0	132.0	132.0	140.0	131.0	131.0
PARSEC/H2O	1290.0	559.0	559.0	634.0	611.0	611.0
GHS_indef/sparsine	1390.0	377.0	377.0	268.0	268.0	882.0
PARSEC/Ge99H100	7070.0	3000.0	300.0	3210.0	2680.0	2680.0
PARSEC/Ga10As10H30	7280.0	2370.0	2370.0	2280.0	2260.0	2260.0
PARSEC/Ga19As19H42	9100.0	4110.0	4110.0	4230.0	4710.0	4710.0

problems, both CPUs and GPU are involved in the factorization and in all cases we manage to reduce the factorization by up to a factor of 2.2 compared with a CPU-only execution. Note that for these problems, the best value for α varies and although for some problems, such as PARSEC/Ge99H100, the factorization runs faster with a lower value of α, for other problems, such as PARSEC/Ga19As19H42, it performs better for large values. This shows that the best α value that controls the workload balance between the GPU and the CPUs depends on the problem.

9. Conclusions.

Although the widely used traditional *threshold partial pivoting* strategy has good numerical performance, we show that it may not be ideal for exploiting parallelism on modern multicore architectures. In order to overcome this limitation some methods, such as the SBK algorithm implemented in PARDISO, trade stability for performance. However, the unstable nature of these methods can lead to inaccurate results when solving sparse linear systems. For these reasons we have developed a new pivoting strategy, namely, *a posteriori threshold pivoting* for the LDL^T factorization of indefinite matrices that offers more parallelism without sacrificing numerical robustness. We showed that this method, implemented in a new version of the SSIDS solver, outperforms the state-of-the-art solvers on modern multicore machines and, in addition, is able to exploit heterogeneity in the context of GPU-accelerated architectures.

REFERENCES

[1] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, *StarPU: A unified platform for task scheduling on heterogeneous multicore architectures*, Concurrency Comput. Pract. Exper., 23 (2011), pp. 187–198, https://doi.org/10.1002/cpe.1631.

[2] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, *hwloc: A generic framework for managing hardware affinities in HPC applications*, in Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing, IEEE, ed., Pisa, Italy, 2010, https://doi.org/10.1109/PDP.2010.67.

[3] J. R. Bunch and L. Kaufman, *Some stable methods for calculating inertia and solving symmetric linear systems*, Math. Comp., 31 (1977), pp. 162–179, https://doi.org/10.1090/S0025-5718-1977-0428694-0.

[4] S. Cayrols, I. S. Duff, and F. Lopez, *Parallelization of the Solve Phase in a Task-Based Cholesky Solver Using a Sequential Task Flow Model*, Technical report RAL-TR-2018-008, Rutherford Appleton Laboratory, Oxfordshire, England, 2018.

[5] T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Trans. Math. Software, 38 (2011), pp. 1–25, http://doi.acm.org/10.1145/2049662.2049663.

[6] I. Duff, J. Hogg, and F. Lopez, *Experiments with sparse Cholesky using a sequential task-flow implementation*, Numer. Algebra Control Optim., 8 (2018), pp. 237–260, https://doi.org/10.3934/naco.2018014.

[7] I. S. Duff, *MA57—A code for the solution of sparse symmetric definite and indefinite systems*, ACM Trans. Math. Software, 30 (2004), pp. 118–144, https://doi.org/10.1145/992200.992202.

[8] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*, 2nd ed., Oxford University Press, Oxford, 2017, https://doi.org/10.1093/acprof:oso/9780198508380.001.0001.

[9] I. S. Duff and J. Koster, *On algorithms for permuting large entries to the diagonal of a sparse matrix*, SIAM J. Matrix Anal. Appl., 22 (2001), pp. 973–996, https://doi.org/10.1137/S0895479899358443.

[10] I. S. Duff and F. Lopez, *Experiments with sparse Cholesky using a parametrized task graph implementation*, in Parallel Processing and Applied Mathematics, R. Wyrzykowski, J. Dongarra, E. Deelman, and K. Karczewski, eds., Lecture Notes in Comput. Sci. 10777, Springer, Cham, Switzerland, 2018 pp. 197–206, https://doi.org/10.1007/978-3-319-78024-5_18.

[11] I. S. Duff and J. K. Reid, *The multifrontal solution of indefinite sparse symmetric linear*, ACM Trans. Math. Software, 9 (1983), pp. 302–325, https://doi.org/10.1145/356044.356047.

[12] I. S. Duff, J. K. Reid, N. Munksgaard, and H. B. Neilsen, *Direct solution of sets of linear equations whose matrix is sparse, symmetric and indefinite*, J. Inst. Maths. Applics., 23 (1979), pp. 235–250, https://doi.org/10.1093/imamat/23.2.235.

[13] J. Hogg and J. Scott, *An Indefinite Sparse Direct Solver for Large Problems on Multicore Machines*, Technical report RAL-TR-2010-011, Rutherford Appleton Laboratory, Oxfordshire, England, 2010, http://purl.org/net/epubs/work/56163.

[14] J. Hogg and J. Scott, *HSL_MA97: A Bit-Compatible Multifrontal Code for Sparse Symmetric Systems*, Technical report RAL-TR-2011-024, Rutherford Appleton Laboratory, Oxfordshire, England, 2011, http://purl.org/net/epubs/work/61445.

[15] J. D. Hogg, E. Ovtchinnikov, and J. A. Scott, *A sparse symmetric indefinite direct solver for GPU architectures*, ACM Trans. Math. Software, 42 (2016), pp. 1–25, https://doi.org/10.1145/2756548.

[16] K. C. Knowlton, *A fast storage allocator*, Commun. ACM, 8 (1965), pp. 623–624, https://doi.org/10.1145/365628.365655.

[17] O. Schenk and K. Gärtner, *On fast factorization pivoting methods for sparse symmetric indefinite systems*, Electron. Trans. Numer. Anal., 23 (2006), pp. 158–179.

[18] A. YarKhan, J. Kurzak, P. Luszczek, and J. Dongarra, *Porting the PLASMA numerical library to the OpenMP standard*, Internat. J. of Parallel Programming, 45 (2017), pp. 612–633, https://doi.org/10.1007/s10766-016-0441-6.