Tacho: Memory-Scalable Task Parallel Sparse Cholesky Factorization

Kyungjoo Kim  
Center for Computing Research  
Sandia National Laboratories  
Albuquerque, USA  
kyukim@sandia.gov

H. Carter Edwards  
Center for Computing Research  
Sandia National Laboratories  
Albuquerque, USA  
hcedwar@sandia.gov

Sivasankaran Rajamanickam  
Center for Computing Research  
Sandia National Laboratories  
Albuquerque, USA  
srajama@sandia.gov

Abstract—We present a memory-scalable, parallel, sparse multifrontal solver for solving symmetric positive-definite systems arising in scientific and engineering applications. Factorizing sparse matrices requires memory for both the computed factors and the temporary workspaces for computing each frontal matrix - a data structure commonly used within multifrontal methods. To factorize multiple frontal matrices in parallel, the conventional approach is to allocate a uniform workspace for each hardware thread. In the manycore era, this results in increasing memory usage proportional to the number of hardware threads. We remedy this problem by using dynamic task parallelism with a scalable memory pool. Tasks are spawned while traversing an assembly tree and executed after their dependences are satisfied. We also use an idea to respawn the tasks when certain conditions are not met. Temporary workspace for frontal matrices in each task is allocated from a memory pool designed by us. If the requested memory space is not available in the memory pool, the task is respawned to yield the hardware thread to execute other tasks. The respawned task is executed after high priority tasks are executed. This approach allows to have robust parallel performance within a bounded memory space. Experimental results demonstrate the merits of our implementation on Intel multicore and manycore architectures.

Index Terms—Kokkos, Task Parallelism, Memory Pool, Sparse Direct Method, Cholesky, Multifrontal

I. INTRODUCTION

Sparse direct methods [1] based on Gaussian elimination are often used in solving large-scale sparse systems arising from scientific and engineering applications due to its robustness. Direct methods are computationally expensive and it requires a large amount of memory that rapidly increases with the problem size. Several direct techniques based on the properties of the linear systems exist. A recent survey covers a number of these techniques [2]. The focus of this paper is on symmetric, positive-definite linear systems that are generated by applications such as solid mechanics and structural dynamics. In particular, we focus on the multifrontal, sparse Cholesky factorization method and its task parallel implementation on multicore and manycore architectures.

Multifrontal method [3] is of special interest to us due to its efficient use of many small dense linear algebra operations exploiting highly optimized BLAS and LAPACK routines. The method is characterized by forming a set of small dense blocks called fronts in an assembly tree representing data dependence among the fronts during factorization. A sparse matrix is factorized by traversing the tree in a topological order performing partial factorization on each front. When the multifrontal method is used for sparse factorizations, typically two memory spaces are required. First, a space is needed to store the resultant factor(s) and second, a temporary workspace is needed to store the different Schur complements in the assembly tree that is later used for updating other frontal matrices. The sparsity pattern of the factors is pre-computed in the symbolic phase. When using the same ordering the space needed by the factors will be the same for different implementations of numeric factorization. However, the workspace required for parallel sparse factorization on a shared memory architecture varies dynamically based on the strategies used for parallelism. For a simple example, consider dynamic task scheduling with a uniform workspace per thread and a task-to-thread mapping strategy as dictated by a runtime. Obviously, this is not memory scalable as the total workspace increases with the number of threads. Furthermore, the dynamic approach maps any task to any idling thread; thus, the uniform workspace size for every thread may need to correspond to the largest workspace required during the multifrontal factorization. In the modern manycore era, this can incur a significant performance bottleneck for solving large sparse problems.

This memory scalability problem has been a research subject in distributed memory systems where several research projects focus on so-called memory-aware task mapping strategies. Most of the works target such an use case on distributed memory architectures. In [4], the authors introduced a scheduling heuristic based on the memory usage of each compute node. A master node monitors the memory usage of all compute nodes and tasks are scheduled not to increase the current peak memory usage of nodes. Jacquelin et al. [5] proposed heuristics to find optimal tree traversal while minimizing memory usage. This work is extended to parallel scheduling of tasks in a tree workflow [6]. A hybrid processor mapping algorithm for parallel multifrontal factorization with memory constraint is proposed in [7]. The approach first maps compute nodes to frontal matrices according to the associated workload within the constrained memory space. Near the root of an assembly tree, where the memory constraint is in general not satisfied, it serializes the tree traversal. Then, parallel dense
linear algebra is used to compute the sequence of frontal matrices. We could take this approach in shared-memory as well. However, the approach using parallel dense linear algebra introduces several global synchronizations for computing each frontal matrix and would perform suboptimal on the context of solving multiple dense problems unless each dense problem is sufficiently large enough so that such overhead can be negligible.

In this paper, we present a robust task parallel implementation of sparse multifrontal Cholesky factorization with a bounded memory constraint. We attempt to solve the memory scalability problem from a different perspective. Instead of finding the best parallel scheduling within a bounded memory space, we rely on a task runtime and the task runtime dynamically schedules tasks with the constraint of the bounded memory space. We use two-level task parallelism [8], [9]. First, tasks are created from a parallel traversal of the assembly tree. The induced parallelism from the tree tends to decrease with increasing workload close to the root. To improve the overall parallel efficiency, those tree-level tasks are subdivided via an algorithms-by-blocks technique [10], [11]. This is a matrix-level parallelism within a tree-level parallelism approach. For an efficient implementation, Kokkos tasking APIs [12], [13] are used for dynamic task scheduling. We also use the Kokkos memory pool to manage temporary memory allocations on a bounded memory space. Major challenges for Kokkos' memory pool design include supporting the highly irregular temporary workspace allocations required by the multifrontal factorization tasks and ensuring thread-scalability for large numbers of concurrently executing tasks.

The main contributions in this paper are:

- A scalable, memory-pool design for irregular temporary, workspace allocation
- non-waiting, dynamic, task parallel implementation of sparse multifrontal Cholesky factorization with a bounded memory constraint;
- performance evaluation of the sparse factorization on set test problems on an Intel Knights Landing and Intel Skylake processors

The rest of this paper is organized as follows. In Section II, we summarize Kokkos’ variable size block memory pool algorithm. Next, we present task parallel multifrontal factorization algorithms in Section III. Section IV presents experimental results comparing with Intel MKL Pardiso [14] on multicore and many core architectures. We conclude the paper in Section V.

II. KOKKOS VARIABLE SIZE BLOCK MEMORY POOL

In this section, we brieﬂy explain how Kokkos’ memory pool works. For a more detailed explanation, see [13]. The memory pool dynamically manages memory blocks of variable sizes and is designed to be efﬁcient within following application constraints:

- use ﬁnite memory capacity as speciﬁed by the application;
- perform thread scalable and low latency memory lookup, allocation and deallocation; and
- minimize memory fragmentation.

The memory pool allocates and hierarchically partitions a contiguous span of memory as depicted in Figure 1. This allocated memory is uniformly subdivided into \( p \) superblocks of \( 2^M \) bytes each. Each superblock \( S_i \) is partitioned into allocatable blocks of \( 2^{N_i} \) bytes each, resulting in \( 2^{M-N_i} \) allocatable blocks within superblock \( S_i \). The superblock size \( (2^M) \) and block size \( (2^{N_i}) \) are powers of two for simplicity and runtime efficiency. A superblock’s block size is dynamic; \( N_i \) may be reassigned at runtime in response to an application’s allocation requests.

The thread scalable process for allocating a block of memory is summarized in Algorithm 1. This algorithm uses the term attempt to indicate where atomic operations are used to avoid race conditions and deadlocks. The allocation algorithm maintains a list of superblocks, one for each block size \( 2^N \), in which a block is likely to be available for allocation. This is the initial superblock for the allocation attempt. If the allocation attempt fails then a suitable superblock must be chosen from among the entire set of \( p \) superblocks. The first choice is a non-full superblock that is already assigned to the desired \( 2^N \) block size. The second choice is an empty superblock that can be reassigned to the desired \( 2^N \) block size. The last resort is a partially full superblock \( S_i \) assigned to a larger block size, \( N < N_i \). If none of these superblocks are available the allocation fails.

Deallocation is relatively simple compared to the allocation process. The superblock and block are identiﬁed by the distance between the memory address to be deallocated and the base address of the memory pool. This block is marked as not-allocated.

The variable size block memory pool has performance tuning parameters to support a variety of use cases.

- \( S_{total} \) is the minimum size of allocated memory. The actual size is rounded up to \( p \times 2^M \) as per Figure 1.
- \( S_{SB} \) is the minimum superblock size. The actual size is rounded up to \( (2^M) \) as per Figure 1.
Algorithm 1 Memory Pool Block Allocation Algorithm.

1: procedure POOL_ALLOCATE(size)
2: \[ N \leftarrow 2^{N-1} < \text{size} \leq 2^N \]
3: \[ S \leftarrow \text{likely superblock assigned to } N \]
4: \[ \text{block} \leftarrow \text{NULL} \]
5: \[ \triangleright \text{iterate to handle concurrent allocations and deallocations} \]
6: while block = NULL and \( S \neq \text{undefined} \) do
7: \[ \text{attempt block} \leftarrow \text{claim a block from } S \]
8: if block \neq \text{NULL} then break; attempt succeeded
9: \[ \triangleright \text{attempt failed, search for suitable superblock} \]
10: \[ S \leftarrow \text{undefined} \]
11: \[ S_{\text{empty}} \leftarrow \text{undefined} \]
12: \[ S_{\text{larger}} \leftarrow \text{undefined} \]
13: for \( i \leftarrow \{0, 1, \ldots : p - 1\} \) do
14: \[ \text{if } N_i = N \text{ and } S_i \text{ not full then} \]
15: \[ S \leftarrow S_i \]
16: \[ \text{break: from search loop} \]
17: \[ \text{else if } S_{\text{empty}} = \text{undefined and } S_i \text{ empty then} \]
18: \[ S_{\text{empty}} \leftarrow S_i \]
19: \[ \text{else if } S_{\text{larger}} = \text{undefined and } N_i > N \text{ then} \]
20: \[ S_{\text{larger}} \leftarrow S_i \]
21: \[ \text{if } S = \text{undefined and attempt } N_{\text{empty}} \leftarrow N \text{ then} \]
22: \[ S \leftarrow S_{\text{empty}} \]
23: \[ \text{if } S = \text{undefined then} \]
24: \[ S \leftarrow S_{\text{larger}} \]
25: \[ \text{return block} \]

Algorithm 2 Sequential Multifrontal Factorization

1: procedure RECURSIVE_MULTIFRONTAL_CHOLO(node)
2: \[ \triangleright \text{recursive post order tree traversal} \]
3: for each \( \text{child} \in \text{node.children} \) do
4: \[ \text{RECURSIVE_MULTIFRONTAL_CHOLO(child)} \]
5: \[ \triangleright \text{factorize the frontal matrix associated with this supernode} \]
6: Partition node.A into \( 2 \times 2 \) blocks
7: \[ \left(\begin{array}{cc} A_{TL} & A_{TR} \\ A_{BR} & A_{BR} \end{array}\right) \leftarrow \text{node.A} \]

where the block row of \((A_{TL}, A_{TR})\) and \(A_{BR}\) correspond to the factor block and the contribution block respectively.

III. TASK PARALLEL SPARSE CHOLESKY FACTORIZATION

In this section, we first describe the standard multifrontal algorithm concisely. A more in depth description of the algorithm can be found in the direct methods related references [1], [2]. Next, we explain our proposed task parallel multifrontal factorization algorithm using a bounded memory pool.

A. Multifrontal Method

The multifrontal method factorizes a sparse matrix by forming a set of dense blocks, called fronts. Frontal matrices are related to each other and their dependence is expressed in an assembly tree. A frontal matrix consists of a factor block to be eliminated and a contribution block (Schur complement) where its elimination is deferred to parents. A simple example is illustrated in Figure 2. In general, the sparsity pattern of Cholesky factors and the corresponding assembly tree are determined by performing a symbolic factorization which ignores any numerical values. This allows to reuse the structure for several matrices that have the same structure but different values. In this example described in Figure 2, the markers \( \times \) and \( o \) represent an entry of Cholesky factors and an entry of a contribution block, respectively. The arrow in the tree represents the data dependence among frontal matrices. By traversing the assembly tree as depicted in Algorithm 2, the multifrontal method performs a sequence of partial factorization. We express the algorithm in a recursive style to be concise. The recursive (sequential) algorithm traverses the tree in a post-order. In each front, we consider a frontal matrix node.A as \( 2 \times 2 \) block matrix. The first block row of \((A_{TL}, A_{TR})\) corresponds to the factor block and \(A_{BR}\) represents the contribution block. A temporary workspace is allocated for computing hermitian rank-k updates on \(A_{BR}\) and the workspace is deallocated after the block’s parents in the tree are updated. For serial execution, the required workspace is determined by the maximum contribution block size in the assembly tree, which implies

\[ S_{\text{total}} = S_{sb} = S_{\max} = S_{\min} = \max(A_{BR}) \quad i \in \text{tree}. \]

The multifrontal method is relatively easy to parallelize as independent dense blocks are used for the partial factorization in each front. There are two sources of parallelism in the multifrontal method - tree-level and matrix-level parallelism. Tree-level parallelism is inherent from the assembly tree as independent subtrees can be processed in parallel. A typical assembly tree has large frontal matrices closer to the root where tree-level parallelism diminishes. Thus, the matrix-level parallelism should be exploited for computing such large frontal matrices using parallel dense linear algebra.

Fig. 2: A simple multifrontal Cholesky factorization. The marker \( \times \) represents a nonzero Cholesky factor and the marker \( o \) implies an entry of the Schur complement (rank-k update) computed in a temporary workspace.

- \( S_{\min} \) is the minimum size of an allocatable block. The actual size is \( 2^{N_{\min}} \) where \( 2^{N_{\min}} - 1 < S_{\min} \leq 2^{N_{\min}} \).
- \( S_{\max} \) is the maximum size of an allocatable block. The actual size is \( 2^{N_{\max}} \) where \( 2^{N_{\max}} < S_{\max} \leq 2^{N_{\max}} \).
- Tuning parameters are constrained as \( 0 < S_{\min} \leq S_{\max} \leq S_{sb} \leq S_{\text{total}} \).

\( A \) is the maximum size of an allocatable block. The actual size is \( 2^{N_{\max}} \) where \( 2^{N_{\max}} < S_{\max} \leq 2^{N_{\max}} \).
- Tuning parameters are constrained as \( 0 < S_{\min} \leq S_{\max} \leq S_{sb} \leq S_{\text{total}} \).

III. TASK PARALLEL SPARSE CHOLESKY FACTORIZATION

In this section, we first describe the standard multifrontal algorithm concisely. A more in depth description of the algorithm can be found in the direct methods related references [1], [2]. Next, we explain our proposed task parallel multifrontal factorization algorithm using a bounded memory pool.

A. Multifrontal Method

The multifrontal method factorizes a sparse matrix by forming a set of dense blocks, called fronts. Frontal matrices are related to each other and their dependence is expressed in
parallel performance of the factorization is dependent on how well these two levels of parallelism are exploited.

B. Tacho

Tacho [15] is a new thread parallel sparse direct solver developed using Kokkos [16], a performance portable on-node parallel programming model and its C++ library implementation. Tacho is available as a sub-package of ShyLU node level solvers in the Trilinos framework. We also provide an interface to Tacho through the Amesos2 library [17].

Lately, Kokkos is extended to support a task Directed Acyclic Graph (DAG) execution model. A set of tasks are generated with dependences and the tasks are executed asynchronously in parallel after their dependences are satisfied. To help understand our task parallel multifrontal algorithm, we briefly introduce the concepts and the notations of Kokkos tasking APIs.

- **Task**: A task is a user-defined function object to be executed by a task scheduler.
- **Dependence**: A task may have an execute-after dependence on other tasks. The execution of the task is deferred after its dependent tasks are completed.
- **Spawn**: The process of creating a new task and submitting it to a task scheduler with optional dependences and priority.
- **Respawn**: After a task executes, it is either complete or resubmitted to the task scheduler. Respawn is the process of resubmitting a task to a task scheduler, optionally with new dependences and priority. In Kokkos, a task A cannot wait for task B to complete; instead task A must respawn with a dependence on task B. Kokkos replaces the conventional task-wait mechanism with the task-respawn mechanism to enable portability to GPU architectures, and to eliminate the complexity and latency of blocking and context-switching executing tasks.
- **MemoryPool**: A memory pool that manages dynamic allocation of small blocks of memory from a large pre-allocated chunk of memory is an important foundation for task-based algorithms. Detailed description of the memory pool is given in Section II.

For a richer introduction of the task DAG features, see Edwards *et al.* [12] and for a complete reference of Kokkos task DAG capabilities, see the tasking reference [13].

1) Task Parallel Multifrontal Cholesky Algorithm: Algorithm 3 describes a single task object performing parallel multifrontal factorization by spawning other tasks and creating dependences or factoring a matrix when the dependences are satisfied. At the beginning of factorization, a single task is spawned with the root of the assembly tree as the input. The task is immediately executed and starts generating child tasks. After child tasks are spawned, the task respawns itself (puts itself in the task queue) with dependences on the child tasks. This process is recursive and it traverses the entire tree in a post-order. After the child tasks are executed, this task gets scheduled again and it performs partial factorization on the associated frontal matrix. This part requires temporary workspace to store Schur complement $A_{BR}$ resulting from the elimination process. The memory pool may or may not have enough memory for the factorization at this moment. If the memory pool fails to allocate the requested workspace, the execution of the task is stopped and the task is respawned with a low priority yielding the current thread to other active tasks. This non-waiting respawning mechanism is used so that multiple threads can share the bounded memory pool space. The minimum superblock allocation size is determined as

\[ S_{sb} = S_{max} = \max(A_{BR}^{i}, i \in \text{tree}). \]

For efficient parallel factorization, multiple superblocks are used. Then, the total memory capacity is determined as

\[ S_{total} = N_{sb}S_{sb}, \]

where $N_{sb}$ is the number of superblocks. A larger $N_{sb}$ allows more concurrent tasks. For instance, if the same number of superblocks is used as the number of threads $N_{th}$, a task does not respawn itself due to allocation failures in the memory pool as each thread essentially uses a private workspace. However, this approach is obviously not memory scalable as the total memory usage increases with the number of threads. When we use a smaller $N_{sb}$ than $N_{th}$, this may decrease task parallelism as tasks may need to respawn themselves depending on the instantaneous availability of blocks in the memory pool. The performance trade-off in using different $N_{sb}$ will be discussed in Section IV-A. Again, we would like to emphasize that

<table>
<thead>
<tr>
<th>Algorithm 3 Task Parallel Multifrontal Cholesky</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: procedure TASK.MULTIFRONTALCHOL(node)</td>
</tr>
<tr>
<td>2: ≻ state is zero when a task is generated</td>
</tr>
<tr>
<td>3: if task.state = 0 then</td>
</tr>
<tr>
<td>4: dep[] ← NULL</td>
</tr>
<tr>
<td>5: for each child ∈ node.children do</td>
</tr>
<tr>
<td>6: dep[child] ← SPAWN(TASK.MULTIFRONTALCHOL(child))</td>
</tr>
<tr>
<td>7: ≻ respawn this task with modified state and dependences</td>
</tr>
<tr>
<td>8: task.state ← 1</td>
</tr>
<tr>
<td>9: RESPAWN(this, dep)</td>
</tr>
<tr>
<td>10: if task.state = 1 then</td>
</tr>
<tr>
<td>11: ≻ serial Cholesky factorization</td>
</tr>
<tr>
<td>12: Partition $A$ into $2 \times 2$ blocks</td>
</tr>
<tr>
<td>$\begin{pmatrix} A_{TL} &amp; A_{TR} \ A_{BR} &amp; A_{BR} \end{pmatrix}$ ← node.A</td>
</tr>
</tbody>
</table>

where the block row of $(A_{TL}, A_{TR})$ and $A_{BR}$ correspond to the factor block and the contribution block respectively.

| 13: ≻ try to allocate a block from memory pool |
| 14: POOL.ALLOCATE($A_{BR}$, $A_{BR}$.NumRows, $A_{BR}$.NumCols) |
| 15: ≻ if allocation fails, respawn this task with low priority |
| 16: if $A_{BR} = NULL$ then                    |
| 17: RESPAWN(this, LowPriority)                 |
| 18: return                                      |
| 19: ≻ $A_{TL} := U_{TL}^{H}U_{TL}$ CHOL($A_{TL}$) |
| 20: ≻ $A_{TR} := \text{triu}(A_{TL})^{-1}A_{TR}$ |
| 21: TRSM($A_{TL}, A_{TR}$)                     |
| 22: ≻ $A_{BR} := A_{BR} - A_{TR}^{H}A_{TR}$    |
| 23: HERK($A_{TR}, A_{BR}$)                     |
| 24: UPDATE-TREE($A_{BR}$)                      |
| 25: POOL.DEALLOCATE($A_{BR}$)                  |
| 26: task.state ← done                         |
First, the matrix in the assembly tree and not the entire linear system.

Algorithm 4 Task Parallel Dense CholeskyByBlocks.

1: procedure TASK.CHOLBYBLOCKS(A)
2:   ▷ Loop over blocks instead of scalars
3:   for i = 0 to A.NumRows - 1 do
4:     ▷ Factorize ith block
5:     A11 ← A(i, i)
6:     A11.task ← SPAWN(CHOL(A11), A11.task)
7:   for j = i + 1 to A.NumCols - 1 do
8:     A12 ← A(i, j)
9:     dep[j] ← [A11.task A12.task]
10:    ▷ Spawn a task for triangular solve
11:     A12.task ← SPAWN(TRSM(A11, A12), dep)
12:     A12 ← A(j, j)
13:     dep[j] ← [A12.task A12.task]
14:    ▷ Spawn a task for the update with task handle stored in A22
15:     A22.task ← SPAWN(HERK(A12, A22), dep)
16:   for p = i + 1 to j - 1 do
17:     A22 ← A(p, i)
18:     A22 ← A(p, j)
19:     dep[j] ← [A22.task A22.task A22.task]
20:    A22.task = SPAWN(GEMM(A12, A12, A22), dep)

Algorithm 5 Task Parallel Dense TrsmByBlocks

1: procedure TASK.TRSMBYBLOCKS(A, B)
2:   for i = 0 to A.NumRows - 1 do
3:     A11 ← A(i, i)
4:   for j = 0 to B.NumCols - 1 do
5:     B1 ← B(j, j)
6:     B1.task ← A11.task B1.task
7:     B1.task ← SPAWN(TRSM(A11, B1), dep)
8:     for p = 0 to i - 1 do
9:     B0 ← B(p, j)
10:    B0 ← A0.task B0.task B1.task
11:    B0.task = SPAWN(GEMM(B1, A01, B0), dep)

this algorithm allows robust, parallel, multifrontal factorization with a fixed $N_{sb}$ independent of the number of threads $N_{th}$.

2) Algorithms-By-Blocks: The tree-level task parallelism described in Algorithm 3 is further refined using algorithms-by-blocks [10], [18] for matrix-level parallelism (also called as tiled algorithms [11]). This family of algorithms organizes a matrix as a collection of blocks and uses tasking to operate on the various blocks. For example, consider Cholesky factorization of a dense matrix $A$ assuming the $A$ here is the frontal matrix in the assembly tree and not the entire linear system. First, the matrix $A$ can be viewed as a collection of $N \times N$ block matrices where $A^{0,i,j}$ has compatible dimensions each other.

\[
A = \begin{pmatrix}
A^{(0,0)} & A^{(0,1)} & \cdots & A^{(0,N-1)} \\
A^{(1,0)} & A^{(1,1)} & \cdots & A^{(1,N-1)} \\
\vdots & \vdots & \ddots & \vdots \\
A^{(N-1,0)} & A^{(N-1,1)} & \cdots & A^{(N-1,N-1)}
\end{pmatrix}
\]

Next, the Cholesky factorization is reformulated by changing the computing unit from scalars to blocks. In the first iteration of Cholesky,

- $A^{(0,0)}$ is factored and overwritten with Cholesky factors (CHOL)

\[
A^{(0,0)} := U^H U.
\]

- A block row $A^{(0,1:N−1)}$ is computed in blockwise fashion (TRSM)

\[
A^{(0,j)} := U^{-1} A^{(0,j)} \text{ for } j \in \{1, \cdots, N−1\}.
\]

- Upper triangular blocks of $A^{(1:N−1,1:N−1)}$ are updated (HERK, GEMM)

\[
A^{(i,j)} := A^{(i,j)} H A^{(0,j)} \text{ for } i \leq j \in \{1, \cdots, N−1\}.
\]

This can be expressed as a set of tasks. Tasks are generated in the subsequent iterations and resulting tasks are related with input/output blocks as described in Algorithm 4. The algorithm loops over the blocks of $A$ and generates four kinds of tasks for the essential steps described above. When a task is spawned, a task handle is returned for it. The task handle is retained in the output block of the task to trace the correct data-flow for matrix-level parallelism (e.g., $A_{11}.task$). When the output block is used as an input or an output in a new task, there is an absolute dependence such that the new task should be executed after the task recorded on the output block. There is no separate book-keeping procedure but the algorithm naturally provides dependence among tasks. In the same way, the triangular solve by blocks is described in Algorithm 5. The two essential set of tasks here are TRSM and GEMM called on block rows. We leave the step-by-step description of triangular solve by blocks out for brevity. This matrix-level parallelism (using tasks) can be naturally blended with the induced tree-level parallelism when the frontal matrices are sufficiently large.

3) Task Parallel Multifrontal CholeskyByBlocks Algorithm:

Algorithm 6 illustrates a task-based, parallel multifrontal Cholesky factorization exploiting both tree-level and matrix-level parallelism via algorithms-by-blocks. First, a task is spawned with the root of an assembly tree similar to Algorithm 3. Second, the task recursively spawns child tasks and respawns itself with dependences on the child tasks (lines 4-9). After the child tasks complete, the frontal matrix associated with this tree node is partitioned with a block size $mb$ resulting in matrix-parallelism (lines 13-16). Third, the task spawns tasks by invoking CHOLBYBLOCKS and TRSMBYBLOCKS computing partial factorizations on the partitioned blocks in parallel (lines 17-23). Then, the task respawns itself with required dependences on the factor blocks. When rescheduled, the task performs multiple panel updates in parallel (lines 25-40) by spawning tasks described in Algorithm 7. In the update task described in Algorithm 7, a panel on the contribution block is allocated and used for updating parents in the tree. Recall that our memory pool has a constraint of the minimum superblock size to cover the maximum contribution block determined by the symbolic factorization. In this case, the required minimum superblock size is reduced by a panel size $nb$.

\[
S_{sb} = S_{max} = nb \cdot \max(A^{i,br}_{th} \cdot \text{NumRows} \quad i \in \text{tree}).
\]
Algorithm 6 Task Parallel Multifrontal CholeskyByBlocks
1: procedure TASK_MULTIFRONTAL_CHOLESKYYBLOCKS(node)
2: ▷ state is zero when a task is generated
3:  if task.state = 0 then
4:      dep[height] ← NULL
5:  for each Qchild ∈ node.children do
6:      dep[height] := SPAWN(TASK_MULTIFRONTAL_CHOLESKYBYBLOCKS(child))
7:  respawn this task with modified state and dependences
8:  task.state ← 1
9:  RESPAWN(this, dep)
10:  if task.state = 1 then
11:     parallel CholeskyByBlocks factorization
12:     Partition A into 2 x 2 blocks
13:     (A_{TL}, A_{TR}, A_{BR}) ← node.A

where the block row of (A_{TL}, A_{TR}) and A_{BR} correspond to the factor block and the contribution block respectively.
14:     organize matrices by blocks
15:     H_{TL} ← PARITION-BY-BLOCKS(A_{TL}, mb, mb)
16:     H_{TR} ← PARITION-BY-BLOCKS(A_{TR}, mb, mb)
17:     perform task parallel dense linear algebra
18:     CHOLBYBLOCKS(H_{TL})
19:     TRYMBLOCKS(H_{TR}, H_{BR})
20:  if A_{BR}, NumRows > 0 then
21:      task.state ← 2
22:      RESPAWN(this, H_{BR}, task)
23:  return
24:  if task.state = 2 then
25:     nb ← panel size used for panel update algorithm
26:     offsets ← 0
27:     i ← 0
28:     dep[height] ← NULL
29:     while offsets < A_{BR}, NumCols do
30:        spawn multiple panel updates
31:        dep[i] := SPAWN(GEMM AND UPDATE(
32:          A_{TR}(0 : end, 0 : offsets + nb),
33:          A_{BR}(0 : offsets + nb, offsets : offsets + nb)\))
34:        offsets ← offsets + nb
35:        i ← i + 1;
36:        respawn this to finish all gemm updates
37:        task.state ← done
38:        RESPAWN(this, dep)
39:  return

As a result, the total active workspace can be set as

\[ S_{total} = N_{ab} \cdot nb \cdot \max\{A_{BR}^i, \text{NumRows} \quad i \in \text{tree} \}. \]

Since \( A_{BR}^i, \text{NumRows} \) is inherent from the given sparse problem, we have two tunable performance parameters i.e., the number of superblocks \( N_{ab} \) and the panel size \( nb \). To improve concurrency with fixed memory constraints, one may want to increase \( N_{ab} \) while reducing \( nb \). The performance trade-off between \( N_{ab} \) and \( nb \) is discussed in the next section. We will use this algorithm for all our performance experiments as blending both tree-level parallelism and matrix-level parallelism provides significant advantage over just using tree-level parallelism as depicted in Algorithm 3.

IV. NUMERICAL EXPERIMENTS

In this section, we compare Tacho against the state-of-the-art parallel sparse direct solver on a shared memory architecture i.e., Intel MKL, Pardiso [14], [19]. To evaluate our proposed task parallel approach using a bounded memory pool, various test problems are selected from the SuiteSparse matrix collection [20], of which the number of rows ranges from 0.5 million to 1.5 million as shown in Table I. All numeric experiments are performed on Intel Xeon Skylake and Xeon Phi Knight Landing (KNL) architectures. Detailed specifications of the test machines are tabulated in Table II. In terms of architecture design, Skylake and KNL are similar as both of them has a high enough number of cores with a wide vector units (AVX512) representing the modern computing trend. However, their different memory system provides very distinct performance characteristics. The KNL system consists of 34 tiles where each tile has a pair of cores sharing 1MB L2

<table>
<thead>
<tr>
<th>Problem</th>
<th># rows</th>
<th>nnz(A)</th>
<th>Pardiso</th>
<th>Tacho</th>
</tr>
</thead>
<tbody>
<tr>
<td>af_0_k101</td>
<td>503,625</td>
<td>17,550,675</td>
<td>1,700</td>
<td>1,700</td>
</tr>
<tr>
<td>inline_1</td>
<td>503,712</td>
<td>36,816,170</td>
<td>2,988</td>
<td>3,645</td>
</tr>
<tr>
<td>af_shell3</td>
<td>504,855</td>
<td>17,562,051</td>
<td>1,610</td>
<td>1,700</td>
</tr>
<tr>
<td>af_shell7</td>
<td>504,855</td>
<td>17,579,155</td>
<td>1,560</td>
<td>2,225</td>
</tr>
<tr>
<td>parabolic_fem</td>
<td>525,825</td>
<td>3,674,625</td>
<td>1,022</td>
<td>1,027</td>
</tr>
<tr>
<td>Fault_639</td>
<td>638,802</td>
<td>27,245,944</td>
<td>14,679</td>
<td>19,433</td>
</tr>
<tr>
<td>apache2</td>
<td>715,176</td>
<td>4,817,870</td>
<td>3,509</td>
<td>3,203</td>
</tr>
<tr>
<td>tmt_sym</td>
<td>726,713</td>
<td>5,080,961</td>
<td>1,484</td>
<td>1,193</td>
</tr>
<tr>
<td>PFlow_742</td>
<td>742,793</td>
<td>37,138,461</td>
<td>7,507</td>
<td>6,973</td>
</tr>
<tr>
<td>bone10</td>
<td>914,898</td>
<td>40,878,708</td>
<td>3,735</td>
<td>3,357</td>
</tr>
<tr>
<td>Emilia_923</td>
<td>923,136</td>
<td>40,373,538</td>
<td>18,156</td>
<td>20,679</td>
</tr>
<tr>
<td>audklw_1</td>
<td>943,695</td>
<td>77,651,847</td>
<td>10,569</td>
<td>9,912</td>
</tr>
<tr>
<td>ldooor</td>
<td>952,203</td>
<td>42,493,817</td>
<td>2,173</td>
<td>1,727</td>
</tr>
<tr>
<td>bone010</td>
<td>986,703</td>
<td>47,851,783</td>
<td>5,895</td>
<td>9,633</td>
</tr>
<tr>
<td>ecology2</td>
<td>999,999</td>
<td>4,995,991</td>
<td>1,769</td>
<td>1,773</td>
</tr>
<tr>
<td>thermal2</td>
<td>1,228,045</td>
<td>8,580,313</td>
<td>938</td>
<td>946</td>
</tr>
<tr>
<td>Serena</td>
<td>1,391,349</td>
<td>64,131,971</td>
<td>18,936</td>
<td>23,019</td>
</tr>
<tr>
<td>Geo_1438</td>
<td>1,437,960</td>
<td>60,236,322</td>
<td>19,038</td>
<td>13,755</td>
</tr>
<tr>
<td>StockF-1465</td>
<td>1,465,137</td>
<td>21,005,389</td>
<td>7,090</td>
<td>9,026</td>
</tr>
<tr>
<td>Hook_1498</td>
<td>1,498,023</td>
<td>59,374,451</td>
<td>10,902</td>
<td>13,692</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Testbed</th>
<th>Skylake</th>
<th>KNL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Xeon 8160 @ 2.1GHz</td>
<td>Xeon Phi 7250 @ 1.4GHz</td>
</tr>
<tr>
<td>Cache</td>
<td>33 MB shared L3</td>
<td>1 MB private L2 per tile</td>
</tr>
<tr>
<td># cores</td>
<td>2x24</td>
<td>1x68 (34 tiles)</td>
</tr>
<tr>
<td>Compiler</td>
<td>Intel 18.1.163</td>
<td>Intel 18.0.128</td>
</tr>
</tbody>
</table>

TABLE I: Test problems selected from the SuiteSparse matrix collection. All matrices are symmetric positive definite and the number of rows of the matrices ranges from 0.5 million to 1.5 million. Size of the largest dense block as computed by the two codes is also listed.

TABLE II: Testbed specification.
cache. On the Skylake system, each tile has a single core with 1MB private L2 cache and 1.375MB L3 cache per tile (core). In dynamic task scheduling, each thread selects an active task from a task queue. After a thread executes the task, the task queue is updated atomically. This dynamic workflow allows good load balance but may incur frequent data movements across computing units. Tasking performance on KNL could suffer from this frequent data movements among the private L2 caches. Furthermore, when a larger task queue than the L2 cache size is required, it looks up the main memory, which can cause much higher latency overhead for the dynamic tasking process. On the other hand, the non-inclusive L3 cache on the Skylake system allows to reuse the data spills from L2, which can reduce the data access overhead effectively. We will evaluate these conjectures further in this section.

A. Performance Parameters Setup

To begin with, we study the performance impact of using different memory pool parameters. For a comparison purpose, we use parabolic_fem matrix. \(^1\) After symbolic factorization is performed, the following supernodal structure is used for the numeric factorization: 1) \# of supernodes is 299530, 2) height of the assembly tree is 44, 3) \# of leaf-level supernodes (max tree-level concurrency) is 105312, and 4) the largest supernode size (related to max matrix-level parallelism) is 1027. Figure 3 demonstrates the parallel performance of Tacho with a different number of superblocks \(N_{sb}\) on the Skylake architecture. In this experiment, we uses fixed \(nb = 64\) and the total memory capacity \(S_{total}\) increases with \(N_{sb}\). Some observations are as follows.

- Peak memory usage of Pardiso increases proportionally with the number of threads.
- Tacho performs parallel factorization keeping the bounded memory constraint specified by \(N_{sb}\).
- A smaller \(N_{sb}\) compared to the number of threads tends to limit parallel efficiency. This is expected as it is more likely to respawn tasks due to failures in the temporary memory allocation, but the performance is not significantly degraded.
- Kokkos dynamic task scheduling achieves good load balance, which results in higher performance for a higher number of threads.

Figure 4 shows the variation in numeric factorization time depending on the number of superblocks and their size as measured from the Skylake system. Different memory pool configurations that has the same \(S_{total}\) are tested with 48 threads. Clearly, when we have a limited memory budget, a bigger \(N_{sb}\) allows more concurrency resulting in improved performance. However, the performance gap is small after the memory pool is large enough to allow the concurrency inherent from the problem. We also note that selecting an optimal \(N_{sb}\) is very problem and architecture specific and the subject is beyond the scope of this paper.

We perform another case study on the KNL architecture. Figure 5 shows the strong scaling of time and peak memory usage as we vary \(N_{sb}\). On this architecture, Pardiso performance ramps up fast with an increasing number of threads, however the performance benefits is not seen beyond 32 threads. The other performance behavior is similar to what is observed in Skylake architecture (Figure 3). As Pardiso is a closed source code, we cannot identify the reason for the performance drop after 32 threads. Pardiso requires workspace proportionally increasing with the number of threads as can be observed in Figure 5. On the other hand, our solver successfully factorizes the problem with a bounded memory space specified by \(N_{sb}\).
In the following numerical experiments, we use nb as the problem size and we do not discuss details in this paper. nb is compared with Pardiso. Algorithm 6 is used with a panel size $nb = 128$ and superblock size is set $S_{sb} = nb \cdot \max(A_i^{TR} \cdot \text{NumRows}_i \mid i \in \text{tree})$.

Figure 6 illustrates the performance impact due to a different configuration of the memory pool. In this particular problem, the combination of $N_{sb} = 16$ and $nb = 128$ performs similar to the case of $N_{sb} = 32$ and $nb = 64$. The KNL node is featured with a slow core processor and shared L2 cache per tile. Thus, data locality is as important as concurrent task scheduling. Since we rely on dynamic task scheduling without considering the locality of the task data, it incurs more frequent data transfer among the separate L2 caches. Later, we will discuss this performance trade-off with other test problems (see Table IV and Table VI). One may consider to use coarse grain tasks by increasing the block size $nb$ and the panel size $nb$. Determining an optimal task granularity is another research problem and we do not discuss details in this paper.

In the following numerical experiments, we use $nb = 64$ and $N_{sb} = 4$ to refine matrix-level parallelism.

### B. Performance Benchmark

#### a) Skylake

We compare the numeric phase of our task parallel Cholesky factorization against Pardiso on the Skylake machine. Both solvers use the same nested dissection

<table>
<thead>
<tr>
<th>Problem</th>
<th>Skylake</th>
<th>Time (sec)</th>
<th>Pardiso</th>
<th>Tacho</th>
<th>Peak memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>parabolic_fem</td>
<td>0.17</td>
<td>0.13</td>
<td>10,793</td>
<td>9,642</td>
<td></td>
</tr>
<tr>
<td>apache2</td>
<td>0.74</td>
<td>0.82</td>
<td>2,407</td>
<td>1,424</td>
<td></td>
</tr>
<tr>
<td>fault_639</td>
<td>17.44</td>
<td>11.39</td>
<td>10,793</td>
<td>9,642</td>
<td></td>
</tr>
<tr>
<td>tmb_sym</td>
<td>0.24</td>
<td>0.17</td>
<td>1,548</td>
<td>481</td>
<td></td>
</tr>
<tr>
<td>PFlow_742</td>
<td>3.01</td>
<td>2.15</td>
<td>6,066</td>
<td>4,542</td>
<td></td>
</tr>
<tr>
<td>bone510</td>
<td>1.11</td>
<td>0.76</td>
<td>4,355</td>
<td>2,561</td>
<td></td>
</tr>
<tr>
<td>Emilia_923</td>
<td>29.51</td>
<td>14.40</td>
<td>16,398</td>
<td>15,199</td>
<td></td>
</tr>
<tr>
<td>audikw_1</td>
<td>12.39</td>
<td>7.37</td>
<td>13,056</td>
<td>11,102</td>
<td></td>
</tr>
<tr>
<td>Idoor</td>
<td>0.48</td>
<td>0.41</td>
<td>3,198</td>
<td>2,081</td>
<td></td>
</tr>
<tr>
<td>bone010</td>
<td>10.68</td>
<td>6.28</td>
<td>11,895</td>
<td>10,210</td>
<td></td>
</tr>
<tr>
<td>ecology2</td>
<td>0.32</td>
<td>0.32</td>
<td>2,081</td>
<td>617</td>
<td></td>
</tr>
<tr>
<td>thermal2</td>
<td>0.34</td>
<td>0.19</td>
<td>2,595</td>
<td>807</td>
<td></td>
</tr>
<tr>
<td>Serena</td>
<td>70.76</td>
<td>41.22</td>
<td>26,256</td>
<td>24,048</td>
<td></td>
</tr>
<tr>
<td>Geo_1438</td>
<td>32.35</td>
<td>21.27</td>
<td>22,231</td>
<td>20,493</td>
<td></td>
</tr>
<tr>
<td>StocF-1465</td>
<td>9.73</td>
<td>6.12</td>
<td>11,787</td>
<td>10,001</td>
<td></td>
</tr>
<tr>
<td>Hook_1498</td>
<td>15.05</td>
<td>11.38</td>
<td>14,538</td>
<td>11,697</td>
<td></td>
</tr>
</tbody>
</table>

#### TABLE IV: [Skylake] Performance trade-off for different memory pool configurations constrained by the same memory capacity.
ordering computed from Metis [21]. Default parameters are used in testing Pardiso. Tacho uses following parameters: 1) a block size $mb = 256$ for running algorithms-by-blocks; 2) a panel size $nb = 64$ for updating frontal matrices; 3) superblocks $N_{sb} = 12$ of the memory pool. In Figure 7, relative speedup and memory performance of Tacho is compared against Pardiso. The actual time and memory usage is listed in Table III. Our solver gains 1.43 geometric mean speedup and uses overall 42% less peak memory compared to Pardiso. To provide a detailed performance comparison, we now focus on two characteristic problems i.e., Fault_639 and thermal2. The Fault_639 has the largest frontal matrix among all test problems. This implies that efficient parallelization in computing the largest frontal matrix determines the overall performance of the parallel factorization. The contribution block required for factoring the front takes about 1.8 GB. Thus, it is essential to use the panel-update algorithm depicted in Algorithm 7 to reduce the superblock size of the memory pool. On the other hand, thermal2 has the smallest front size while the sparse problem is fairly large and has 1.2 million rows. This also implies that the matrix is featured with a large tree-level parallelism, which should be exploited efficiently. For this type of problems, the additional parallelism induced from algorithms-by-blocks and panel updates may incur more tasking overhead than performance. To investigate the performance trade-offs due to panel size (locality) and the number of superblocks (concurrency), we test Tacho for different memory pool configurations within the same memory pool capacity as shown in Table IV. For example, the case of $64 \times 48$ allows maximum concurrency and each thread has its own workspace allocated in the superblock. Since the panel size is small, many small tasks are created, which can cause more tasking overhead. With a larger panel size, frontal matrices can be computed with a small number of tasks generated by panel updates, which results in small tasking overhead. On the Skylake machine, Tacho performs the best for both problems when $N_{sb} = 48$ superblocks are used with a panel size $nb = 64$. In this setting, each thread has a private workspace so that a task is never respawned due to a failure of memory pool allocation.

b) KNL: We repeat the same numerical experiments on the KNL architecture. Our KNL node is configured to Quadrant mode and tested with flat MCDRAM. As both Tacho and Pardiso perform similar to DDR flat mode, we do not report separate performance benchmark on DDR flat mode. Table V describes the time and peak memory used in the numeric factorization performed by Tacho and Pardiso. In this evaluation, Tacho uses the following parameters: 1) a block size $mb = 256$ running algorithms-by-blocks; 2) a panel size $nb = 128$ for updating frontal matrices; 3) superblocks $N_{sb} = 34$ in the memory pool. Tacho demonstrates comparable performance to Pardiso with 0.9 geometric mean speedup and uses 47% less peak memory than Pardiso. Figure 8 illustrates relative speedup and peak memory performance of our solver compared with Pardiso. Unlike Figure 7, a large speedup is not observed on the KNL node. This is because our dynamic tasking approach includes inherent performance penalty from the slow core processor and separate L2 cache per compute tile. A task-dag runtime in general is implemented with a task queue and the queuing procedures are essentially sequential. Thus, a slow processor of the KNL increases the sequential portion of the execution time resulting in slower

### Table V: [KNL] Time and peak memory usage measured in the numeric phase of sparse Cholesky factorization using 68 threads.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Time (sec)</th>
<th>Peak memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pardiso</td>
<td>Tacho</td>
</tr>
<tr>
<td></td>
<td>Pardiso</td>
<td>Tacho</td>
</tr>
<tr>
<td>af_0_k101</td>
<td>0.33</td>
<td>0.31</td>
</tr>
<tr>
<td>inline_1</td>
<td>0.79</td>
<td>0.68</td>
</tr>
<tr>
<td>af_shell3</td>
<td>0.32</td>
<td>0.30</td>
</tr>
<tr>
<td>af_shell7</td>
<td>0.32</td>
<td>0.35</td>
</tr>
<tr>
<td>parabolic_fem</td>
<td>0.26</td>
<td>0.28</td>
</tr>
<tr>
<td>Fault_639</td>
<td>12.87</td>
<td>12.56</td>
</tr>
<tr>
<td>apache2</td>
<td>0.92</td>
<td>1.46</td>
</tr>
<tr>
<td>tm_sym</td>
<td>0.30</td>
<td>0.36</td>
</tr>
<tr>
<td>PFlow_742</td>
<td>3.01</td>
<td>2.82</td>
</tr>
<tr>
<td>boneS10</td>
<td>1.46</td>
<td>1.46</td>
</tr>
<tr>
<td>Emilia_923</td>
<td>20.42</td>
<td>20.70</td>
</tr>
<tr>
<td>auditkw_1</td>
<td>10.73</td>
<td>9.42</td>
</tr>
<tr>
<td>Idoor</td>
<td>0.63</td>
<td>0.78</td>
</tr>
<tr>
<td>bone010</td>
<td>9.27</td>
<td>9.96</td>
</tr>
<tr>
<td>ecology2</td>
<td>0.36</td>
<td>0.67</td>
</tr>
<tr>
<td>thermal2</td>
<td>0.46</td>
<td>0.50</td>
</tr>
<tr>
<td>Serena</td>
<td>52.65</td>
<td>57.76</td>
</tr>
<tr>
<td>Geo_1438</td>
<td>25.54</td>
<td>21.79</td>
</tr>
<tr>
<td>StocF=1465</td>
<td>8.66</td>
<td>9.93</td>
</tr>
<tr>
<td>Hook_1498</td>
<td>12.52</td>
<td>17.70</td>
</tr>
</tbody>
</table>

### Table VI: [KNL] Performance trade-off for different memory pool configuration constrained with the same total memory capacity.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Time (sec)</th>
<th>Peak memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>64 × 68</td>
<td>128 × 34</td>
</tr>
<tr>
<td></td>
<td>256 × 17</td>
<td>512 × 8</td>
</tr>
<tr>
<td>Fault_639</td>
<td>13.12</td>
<td>10.91</td>
</tr>
<tr>
<td>thermal2</td>
<td>0.84</td>
<td>0.81</td>
</tr>
</tbody>
</table>
performance. Another performance issue lies on the data-locality as discussed earlier. The dynamic task scheduling can map any idling thread to any task. This gives us good load balancing for higher thread count; however, such dynamic task scheduling can incur more data movements among threads and result in degraded performance from polluting caches of each other. The lack of shared cache on KNL can cause this significant performance loss. Table VI compares four different memory pool configurations of the almost same memory pool capacity. In this experiment, the panel size nb = 128 performs best with \( N_{x,y} = 34 \) superblocks allocated in the memory pool. Compared to the Skylake result depicted in Table IV, the KNL architecture requires bigger panel size and it reduces tasking overhead.

V. CONCLUSION

We have presented a task parallel implementation of multifrontal Cholesky factorization with a bounded memory space. Our solver uses Kokkos task DAG APIs and tasks are dynamically mapped to threads. This is advantageous as the dynamic task scheduling can achieve superior load balancing. However, the dynamic tasking also make a solver difficult to be memory scalable as each thread may need a uniform workspace. We remedy this memory scalability problem by using a variable block memory pool. The proposed task parallel factorization algorithm uses the panel-update strategy by using a variable block memory pool. The proposed task parallel factorization algorithm uses the panel-update strategy that only requires small workspace of panel on the Schur complement. This allows the solver to utilize the memory pool more effectively.

We demonstrate that our solver can perform robust parallel factorization with a memory constraint on the temporary workspace allocation. Parallel performance is compared with Intel MKL Pardiso on Intel Xeon Skylake and Xeon Phi Knight Landing architectures. Overall, our solver uses about 40% less peak memory compared with Pardiso. With respect to parallel performance, our solver outperforms Intel Pardiso on the Skylake machine with 1.43 geometric mean speedup.

On the KNL, our solver shows comparable performance (0.9 speedup) with Pardiso. This is because tasking overhead is relatively more dominant on the KNL. The KNL architecture has private L2 cache per tile which can be more polluted by dynamic tasking. Also, the slow core processor increases sequential tasking queueing overhead. On the other hand, the Skylake architecture has a large shared L3 cache that can effectively decrease the overhead from dynamic tasking. We plan to study task granularity and parameter tuning in the future to better optimize the tasking overhead over available concurrency.

ACKNOWLEDGMENT

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy’s National Nuclear Security Administration under contract DE-NA-0003525.

Tacho is developed as a subpackage in Trilinos and the source codes are available under the BSD license at https://github.com/trilinos/Trilinos/tree/master/packages/shylu/shylu_node/tacho.

REFERENCES