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 postive-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 sscientific 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 precomputed 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 of 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.

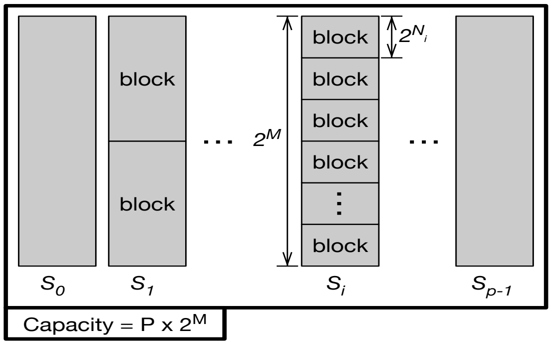

Fig. 1: Block hierarchy in the Kokkos memory pool. A contiguous memory span is allocated and uniformly partitioned into superblocks from S_0 to S_{p-1} where each superblock has 2^M allocatable space. Then, a superblock S_i is partitioned into blocks with a variable size 2^{N_i}.

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.
- 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 $0 < S_{min} \leq S_{max} \leq S_{sb} \leq S_{total}$.

Algorithm 1 Memory Pool Block Allocation Algorithm.

```
procedure POOL.ALLOCATE($size$)
    $N \leftarrow$ such that $2^{N-1} < size \leq 2^N$
    $S \leftarrow$ likely superblock assigned to $N$
    $block \leftarrow NULL$
    ▷ iterate to handle concurrent allocations and deallocations
    while $block = NULL$ and $S \neq$ undefined do
        attempt $block \leftarrow$ claim a block from $S$
        if $block \neq NULL$ then break; attempt succeeded
        ▷ attempt failed, search for suitable superblock
        $S \leftarrow$ undefined
        $S_{empty} \leftarrow$ undefined
        $S_{larger} \leftarrow$ undefined
        for $i \leftarrow \{0, 1, \cdots p-1\}$ do
            if $N_i = N$ and $S_i$ not full then
                $S \leftarrow S_i$
                break; from search loop
            else if $S_{empty} =$ undefined and $S_i$ is empty then
                $S_{empty} \leftarrow S_i$
            else if $S_{larger} =$ undefined and $N_i > N$ then
                $S_{larger} \leftarrow S_i$
        if $S =$ undefined and attempt $N_{empty} \leftarrow N$ then
            $S \leftarrow S_{empty}$
        if $S =$ undefined then
            $S \leftarrow S_{larger}$
    return block
```

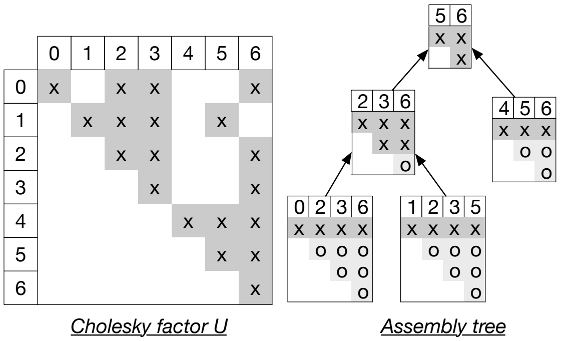

Fig. 2: A simple multifrontal Cholesky factorization. The marker × represents a nonzero Cholesky factor and the marker o implies an entry of the Schur complement (rank-k update) computed in a temporary workspace.

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 × 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 2×2 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

$$S_{total} = S_{sb} = S_{max} = S_{min} = \max(A^i_{BR} \quad i \in tree).$$

Algorithm 2 Sequential Multifrontal Factorization

```
procedure RECURSIVEMULTIFRONTALCHOL($node$)
    ▷ recursive post order tree traversal
    for each $child \in node.children$ do
        RECURSIVEMULTIFRONTALCHOL($child$)
    ▷ factorize the frontal matrix associated with this supernode
    Partition $node.A$ into $2 \times 2$ blocks
        $\begin{pmatrix} A_{TL} & A_{TR} \\ & A_{BR} \end{pmatrix} \leftarrow node.A$
    where the block row of $(A_{TL}\ A_{TR})$ and $A_{BR}$ correspond to the factor
    block and the contribution block respectively.
    ▷ Compute Cholesky Factorization $A_{TL} := U^H_{TL} U_{TL}$
    CHOL($A_{TL}$)
    ▷ Compute Triangular Solve $A_{TR} :=$ triu$(A_{TL})^{-1} A_{TR}$
    TRSM($A_{TL}$, $A_{TR}$)
    ▷ Compute Schur complement $A_{BR} := A_{BR} - A^H_{TR} A_{TR}$
    POOL.ALLOCATE($A_{BR}$, $A_{BR}.NumRows$, $A_{BR}.NumCols$)
    HERK($A_{TR}$, $A_{BR}$)
    UPDATE-TREE($A_{BR}$)
    POOL.DEALLOCATE($A_{BR}$)
```

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. The 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].

Algorithm 3 Task Parallel Multifrontal Cholesky

```
procedure TASK.MULTIFRONTALCHOL(node)
    ▷ state is zero when a task is generated
    if $task.state = 0$ then
        $dep[] \leftarrow NULL$
        for each $child \in node.children$ do
            $dep[child] \leftarrow$ SPAWN(TASK.MULTIFRONTALCHOL(child))
        ▷ respawn this task with modified state and dependences
        $task.state \leftarrow 1$
        RESPAWN(this, dep)
    if $task.state = 1$ then
        ▷ serial Cholesky factorization
        Partition $A$ into $2 \times 2$ blocks
            $\begin{pmatrix} A_{TL} & A_{TR} \\ & A_{BR} \end{pmatrix} \leftarrow node.A$
        where the block row of $(A_{TL}\ A_{TR})$ and $A_{BR}$ correspond to the factor
        block and the contribution block respectively.
        ▷ try to allocate a block from memory pool
        POOL.ALLOCATE($A_{BR}$, $A_{BR}.NumRows$, $A_{BR}.NumCols$)
        ▷ if allocation fails, respawn this task with low priority
        if $A_{BR} = NULL$ then
            RESPAWN(this, $LowPriority$)
            return
        ▷ $A_{TL} := U_{TL}^H U_{TL}$
        CHOL($A_{TL}$)
        ▷ $A_{TR} :=$ triu$(A_{TL})^{-1} A_{TR}$
        TRSM($A_{TL}$, $A_{TR}$)
        ▷ $A_{BR} := A_{BR} - A_{TR}^H A_{TR}$
        HERK($A_{TR}$, $A_{BR}$)
        UPDATE-TREE($A_{BR}$)
        POOL.DEALLOCATE($A_{BR}$)
        $task.state \leftarrow done$
```

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

$$S_{sb} = S_{max} = \max(A^i_{BR} \quad i \in tree).$$

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

$$S_{total} = N_{sb} S_{sb},$$

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 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

Algorithm 4 Task Parallel Dense CholeskyByBlocks.

```
procedure TASK.CHOLBYBLOCKS($A$)
    ▷ Loop over blocks instead of scalars
    for $i = 0$ to $A.NumRows - 1$ do
        ▷ Factorize $i^{th}$ block
        $A_{11} \leftarrow A(i, i)$
        $A_{11}.task \leftarrow$ SPAWN(CHOL($A_{11}$), $A_{11}.task$)
        for $j = i + 1$ to $A.NumCols - 1$ do
            $A_{12} \leftarrow A(i, j)$
            $dep[] \leftarrow [A_{11}.task\ A_{12}.task]$
            ▷ Spawn a task for triangular solve
            $A_{12}.task \leftarrow$ SPAWN(TRSM($A_{11}$,$A_{12}$), $dep$)
            $A_{22} \leftarrow A(j, j)$
            $dep[] \leftarrow [A_{12}.task\ A_{22}.task]$
            ▷ Spawn a task for the update with task handle stored in $A_{22}$
            $A_{22}.task \leftarrow$ SPAWN(HERK($A_{12}$,$A_{22}$), $dep$)
            for $p = i + 1$ to $j - 1$ do
                $A_{21} \leftarrow A(i, p)$;
                $A_{22} \leftarrow A(p, j)$;
                $dep[] \leftarrow [A_{21}.task\ A_{12}.task\ A_{22}.task]$
                $A_{22}.task$ = SPAWN(GEMM($A_{12}^H$,$A_{12}$,$A_{22}$), dep)
```

Algorithm 5 Task Parallel Dense TrsmByBlocks

```
procedure TASK.TRSMBYBLOCKS($A$, $B$)
    for $i = 0$ to $A.NumRows - 1$ do
        $A_{11} \leftarrow A(i, i)$
        for $j = 0$ to $B.NumCols - 1$ do
            $B_1 \leftarrow B(i, j)$
            $dep[] \leftarrow [A_{11}.task\ B_1.task]$
            $B_1.task \leftarrow$ SPAWN(TRSM($A_{11}$,$B_1$), $dep$)
            for $p = 0$ to $i - 1$ do
                $B_0 \leftarrow B(p, j)$;
                $A_{01} \leftarrow A(p, j)$;
                $dep[] \leftarrow [A_{01}.task\ B_0.task\ B_1.task]$
                $B_0.task$ = SPAWN(GEMM($B_1$,$A_{01}$,$B_0$), $dep$)
```

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 $A^{(i,j)}$ 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)
$$A^{(i,j)} \mathrel{-}= {A^{(0,i)}}^H A^{(0,j)} \text{ for } i \leq j \in \{1, \cdots, N-1\}.$$

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 automatic 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 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.

$$S_{sb} = S_{max} = nb \cdot \max(A^i_{BR}.NumRows \quad i \in tree).$$

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

$$S_{total} = N_{sb} \cdot nb \cdot \max(A^i_{BR}.NumRows \quad i \in tree).$$

Since $A^i_{BR}.NumRows$ is inherent from the given sparse problem, we have two tunable performance parameters *i.e.*, the number of superblocks N_{sb} and the panel size nb. To improve

Algorithm 6 Task Parallel Multifrontal CholeskyByBlocks

```
procedure Task.MultifrontalCholByBlocks($node$)
    ▷ state is zero when a task is generated
    if $task.state = 0$ then
        $dep[] \leftarrow NULL$
        for each $child \in node.children$ do
            $dep[child] \leftarrow$ spawn(TaskMultifrontalCholByBlocks($child$))
        ▷ respawn this task with modified state and dependences
        $task.state \leftarrow 1$
        respawn(this, $dep$)
    if $task.state = 1$ then
        ▷ parallel CholeskyByBlocks factorization
        Partition $A$ into $2 \times 2$ blocks
            $$\begin{pmatrix} A_{TL} & A_{TR} \\ & A_{BR} \end{pmatrix} \leftarrow node.A$$
  where the block row of $(A_{TL}\ A_{TR})$ and $A_{BR}$ correspond to the factor
  block and the contribution block respectively.
        ▷ organize matrices by blocks
        $mb \leftarrow$ block size used for byblocks algorithms
        $H_{TL} \leftarrow$ partition-by-blocks($A_{TL}$, $mb$, $mb$)
        $H_{TR} \leftarrow$ partition-by-blocks($A_{TR}$, $mb$, $mb$)
        ▷ perform task parallel dense linear algebra
        CholByBlocks($H_{TL}$)
        TrsmByBlocks($H_{TL}$, $H_{TR}$)
        if $A_{BR}.NumRows > 0$ then
            $task.state \leftarrow 2$
            respawn(this, $H_{TR}.task$)
            return
    if $task.state = 2$ then
        $nb \leftarrow$ panel size used for panel update algorithm
        $offset \leftarrow 0$
        $i \leftarrow 0$
        $dep[] \leftarrow NULL$
        while $offset < A_{TR}.NumCols$ do
            ▷ spawn multiple panel updates
            $dep[i] \leftarrow$ spawn(GemmAndUpdate(
                    $A_{TR}(0 : end, 0 : offs + nb)$,
                    $A_{TR}(0 : end, offs : offs + nb)$,
                    $A_{BR}(0 : offs + nb, offs : offs + nb)$))
            $offset \leftarrow offset + nb$
            $i \leftarrow i + 1$;
        ▷ respawn this to finish all gemm updates
        $task.state \leftarrow done$
        respawn(this, $dep$)
        return
```

Algorithm 7 Panel Update

```
procedure Task.GemmAndUpdate($A$, $B$)
    ▷ try to allocate a block from memory pool
    pool.allocate($C$, $A.NumCols$, $B.NumCols$)
    ▷ if allocation fails, respawn this task with low priority
    if $C = NULL$ then
        respawn(this, $LowPriority$)
        return
    ▷ $C := C - A^H B$
    GEMM($A^H$, $B$, $C$)
    update-tree($C$)
    pool.deallocate($C$)
    return
```

concurrency with fixed memory constraints, one may want to increase N_{sb} while reducing nb. The performance trade-off between N_{sb} 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

			Max dense block	
Problem	# rows	nnz(A)	Pardiso	Tacho
af_0_k101	503,625	17,550,675	1,700	1,700
inline_1	503,712	36,816,170	2,988	3,645
af_shell3	504,855	17,562,051	1,610	1,700
af_shell7	504,855	17,579,155	1,560	2,225
parabolic_fem	525,825	3,674,625	1,022	1,027
Fault_639	638,802	27,245,944	**14,679**	**14,933**
apache2	715,176	4,817,870	3,509	3,203
tmt_sym	726,713	5,080,961	1,484	1,193
PFlow_742	742,793	37,138,461	7,507	6,973
boneS10	914,898	40,878,708	3,735	3,357
Emilia_923	923,136	40,373,538	18,156	20,679
audikw_1	943,695	77,651,847	10,659	9,912
ldoor	952,203	42,493,817	2,173	1,727
bone010	986,703	47,851,783	5,895	9,633
ecology2	999,999	4,995,991	1,769	1,773
thermal2	1,228,045	8,580,313	**938**	**946**
Serena	1,391,349	64,131,971	18,936	23,019
Geo_1438	1,437,960	60,236,322	19,038	13,755
StocF-1465	1,465,137	21,005,389	7,090	9,026
Hook_1498	1,498,023	59,374,451	10,902	13,692

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.

Testbed	Skylake	KNL
Processor	Xeon 8160 @ 2.1GHz	Xeon Phi 7250 @ 1.4GHz
Cache	33 MB shared L3	1 MB private L2 per tile
# cores	2x24	1x68 (34 tiles)
Compiler	Intel 18.1.163	Intel 18.0.128

TABLE II: Testbed specification.

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 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

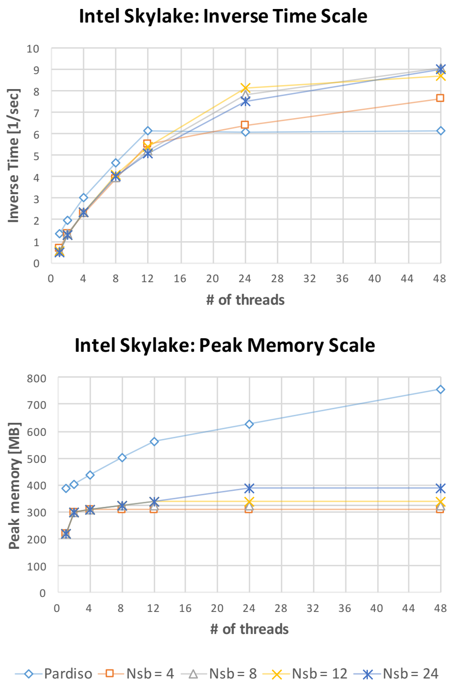

Fig. 3: [Skylake] Strong scale of time and peak memory for factorizing the parabolic_fem matrix. Tacho with a different number of superblocks N_{sb} is compared with Pardiso. Algorithm 6 is used with a panel size $nb = 64$ and superblock size is set $S_{sb} = nb \cdot \max(A^i_{BR}.NumRows \quad i \in tree)$.

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 causes much higher latency overhead in the dynamic tasking. On the other hand, the non-inclusive L3 cache on the Skylake sytem allows to reuse the data spills from L2, which can reduce the data access overhead. We will evalauate 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.

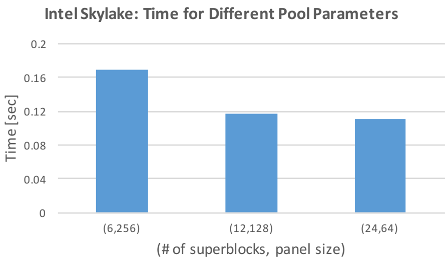

Fig. 4: [Skylake] Time complexity of the numberic factorization of the parabolic_fem matrix. Different memory pool configurations that use the same amount of memory is tested with 48 threads.

- 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}.

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

[1] Tacho is designed to be used as a subdomain solver in Trilinos domain decomposition solvers. The parabolic_fem is selected as the matrix is generated by the same discretization technique (finite element method) as one used in a Sandia application code and the matrix has the similar size to the typical subdomain size used in the application code.

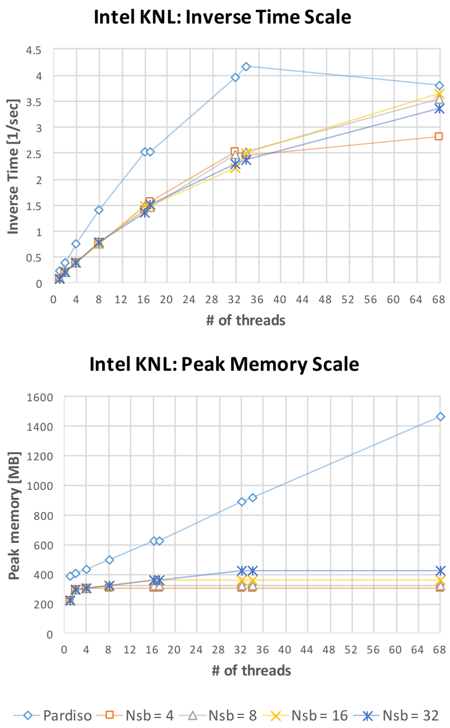

Fig. 5: [KNL] Strong scale of time and peak memory for factorizing the parabolic_fem matrix. Tacho with a different number of superblocks N_{sb} is compared with Pardiso. Algorithm 6 is used with a panel size $nb = 128$ and superblock size is set $S_{sb} = nb \cdot \max(A^i_{BR}.NumRows \quad i \in tree)$.

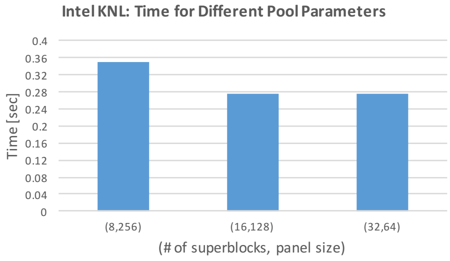

Fig. 6: [KNL] Time complexity of the numeric factorization of parabolic_fem matrix. Different memory pool configurations that use the same amount of memory is tested with 68 threads.

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 mb 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 $nb = 128$ for the Skylake and KNL testbeds respectively.

Skylake	Time (sec)		Peak memory (MB)	
Problem	Pardiso	Tacho	Pardiso	Tacho
af_0_k101	0.31	0.19	1,863	907
inline_1	0.61	0.51	2,872	1,618
af_shell3	0.31	0.19	1,811	847
af_shell7	0.30	0.24	1,825	862
parabolic_fem	0.17	0.13	1,143	388
Fault_639	17.44	11.39	10,793	9,642
apache2	0.74	0.82	2,407	1,424
tmt_sym	0.24	0.17	1,548	481
PFlow_742	3.01	2.15	6,066	4,542
boneS10	1.11	0.76	4,355	2,561
Emilia_923	29.51	14.40	16,398	15,199
audikw_1	12.39	7.37	13,056	11,102
ldoor	0.48	0.41	3,198	1,249
bone010	10.68	6.28	11,895	10,210
ecology2	0.32	0.32	2,081	617
thermal2	0.34	0.19	2,595	807
Serena	70.76	41.22	26,256	24,048
Geo_1438	32.35	21.27	22,231	20,493
StocF-1465	9.73	6.12	11,787	10,001
Hook_1498	15.05	11.38	14,538	11,697

TABLE III: [Skylake] Time and peak memory usage measured in the numeric phase of sparse Cholesky factorization using 48 threads.

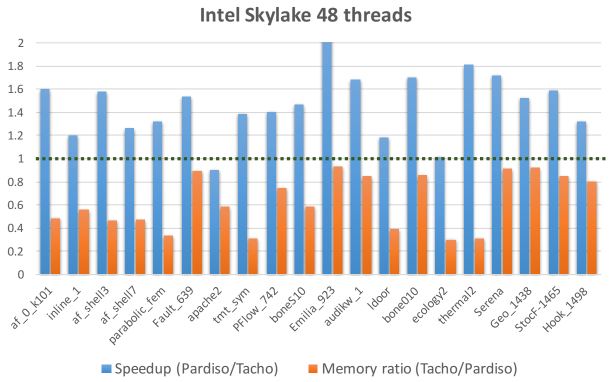

Fig. 7: [Skylake] Numeric factorization speedup (higher is better) and memory ratio (lower is better) of Tacho compared to Pardiso. Tacho uses a memory with 24 superblocks and a panel size 64 to refine matrix-level parallelism.

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 ordering computed from Metis [21]. Default parameters are

	Time (sec) with different nb × # superblocks			
Problem	64 × 48	128 × 24	256 × 12	512 × 6
Fault_639	9.01	10.54	14.57	48.02
thermal2	0.17	0.18	0.19	0.28

TABLE IV: [Skylake] Performance trade-off for different memory pool configurations constrained by the same memory capacity.

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 factorizing 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 × 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

KNL	Time (sec)		Peak memory (MB)	
Problem	Pardiso	Tacho	Pardiso	Tacho
af_0_k101	0.33	0.31	2,171	927
inline_1	0.79	0.68	3,212	1,706
af_shell3	0.32	0.30	2,120	867
af_shell7	0.32	0.35	2,134	950
parabolic_fem	0.26	0.28	1,464	428
Fault_639	12.87	12.56	11,544	9,994
apache2	0.92	1.46	2,844	1,464
tmt_sym	0.30	0.36	1,992	521
PFlow_742	3.01	2.82	6,641	4,718
boneS10	1.46	1.46	4,914	2,601
Emilia_923	20.42	20.70	17,426	15,903
audikw_1	10.73	9.42	13,826	11,454
ldoor	0.63	0.78	3,781	1,289
bone010	9.27	9.96	12,700	10,562
ecology2	0.36	0.67	2,692	657
thermal2	0.46	0.50	3,344	887
Serena	52.65	57.76	27,609	24,752
Geo_1438	25.54	21.79	23,445	20,845
StocF-1465	8.66	9.93	12,744	10,353
Hook_1498	12.52	17.70	15,653	12,049

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

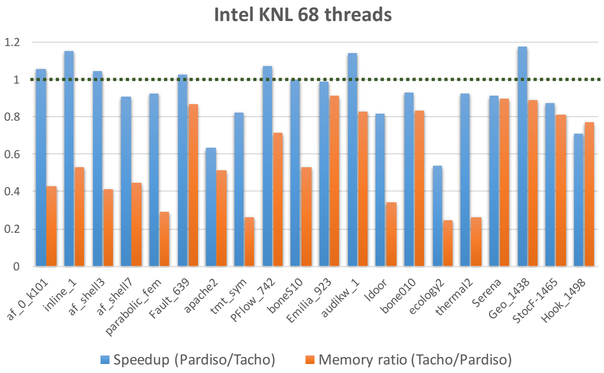

Fig. 8: [KNL] Numeric factorization speedup (higher is better) and memory ratio (lower is better) of Tacho compared to Pardiso. Tacho uses a memory with 34 superblocks and a panel size 128 to refine matrix-level parallelism.

	Time (sec) with different nb × # superblocks			
Problem	64 × 68	128 × 34	256 × 17	512 × 8
Fault_639	13.12	10.91	12.47	23.54
thermal2	0.84	0.81	0.85	1.03

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

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 $N_{sb} = 34$ 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 multi-frontal 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 dyanamic 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 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 architectures 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

[1] T. A. Davis, *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. SIAM, 2006.

[2] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar, "A survey of direct methods for sparse linear systems," *Acta Numerica*, vol. 25, pp. 383–566, 2016.

[3] J. W. H. Liu, "The multifrontal method for sparse matrix solution: theory and practice," *Siam Review*, vol. 34, no. 1, pp. 82–109, 1992.

[4] A. Guermouche and J.-Y. L'Excellent, "Memory-based scheduling for a parallel multifrontal solver," *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, vol. 33, no. April, pp. 71–80, 2004.

[5] M. Jacquelin, L. Marchal, Y. Robert, and B. Ucar, "On optimal tree traversals for sparse matrix factorization," in *International Parallel & Distributed Processing Symposium*, 2011, pp. 556–567.

[6] L. Eyraud-Dubois, L. Marchal, O. Sinnen, and F. Vivien, "Parallel Scheduling of Task Trees with Limited Memory," *ACM Trans. Parallel Comput.*, vol. 2, no. 2, pp. 13:1–13:37, 2015.

[7] E. Agullo, P. R. Amestoy, A. Buttari, A. Guermoche, J.-Y. L'Excellent, and F.-H. Rouet, "Robust Memory-Aware Mappings for ParallelL Multifrontal Factorizations," *SIAM J. Sci. Comput.*, vol. 38, no. 3, pp. 256–279, 2016.

[8] D. Irony, G. Shklarski, and S. Toledo, "Parallel and fully recursive multifrontal sparse Cholesky," *Future Generation Computer Systems*, vol. 20, no. 3, pp. 425–440, apr 2004.

[9] K. Kim and V. Eijkhout, "A Parallel Sparse Direct Solver via Hierarchical DAG Scheduling," *ACM Transactions on Mathematical Software*, vol. 41, no. 1, pp. 3:1–27, 2014.

[10] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. van de Geijn, F. G. van Zee, and E. Chan, "Programming matrix algorithms-by-blocks for thread-level Parallelism," *ACM Transactions on Mathematical Software*, vol. 36, no. 3, pp. 1–26, 2009.

[11] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures," *Parallel Computing*, vol. 35, no. 1, pp. 38–53, 2009.

[12] H. C. Edwards, S. Olivier, G. Mackey, K. Kim, M. Wolf, G. Stelle, J. Berry, and S. Rajamanickam, "Hierarchical Task-Data Parallelism Using Kokkos and QThreads," SAND2016-9613, Tech. Rep., 2016.

[13] H. C. Edwards and D. A. Ibanez, "Kokkos' Task DAG Capabilities," SAND2017-10464, Tech. Rep., 2017.

[14] O. Schenk and K. Gärtner, "Solving unsymmetric sparse systems of linear equations with PARDISO," in *Proceedings of the International Conference on Computational Science-Part II*. Springer-Verlag, 2002, pp. 335–363.

[15] K. Kim, S. Rajamanickam, H. C. Edwards, S. L. Olivier, and G. Stelle, "Task Parallel Incomplete Cholesky Factorization using 2D Partitioned-Block Layout," arxiv.org/pdf/1601.05871, Sandia National Labs, Albuquerque, NM, Tech. Rep., 2016.

[16] H. Carter Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[17] E. Bavier, M. Hoemmen, S. Rajamanickam, and H. Thornquist, "Amesos2 and Belos: Direct and iterative solvers for large sparse linear systems," *Sci. Program.*, vol. 20, no. 3, pp. 241–255, Jul. 2012.

[18] T. M. Low and R. A. van de Geijn, "An API for manipulating matrices stored by blocks," FLAME Working Note 12, TR-2004-15, The University of Texas at Austin, Tech. Rep., 2004.

[19] Intel, *Math Kernel Library*. http://software.intel.com.

[20] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1–25, 2011.

[21] G. Karypis and V. Kumar, "METIS : A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," University of Minnesota, Tech. Rep., 1998.