SAND2016-0637R

Task Parallel Incomplete Cholesky Factorization using 2D Partitioned-Block Layout

Kyungjoo Kim[1], Sivasankaran Rajamanickam[1], George Stelle[1], H. Carter Edwards[1], and Stephen L. Olivier[1]

[1]Center for Computing Research, Sandia National Laboratories
{kyukim,srajama,gwstell,hcedwar,slolivi}@sandia.gov

Abstract

We introduce a task-parallel algorithm for sparse incomplete Cholesky factorization that utilizes a 2D sparse partitioned-block layout of a matrix. Our factorization algorithm follows the idea of *algorithms-by-blocks* by using the block layout. The algorithm-by-blocks approach induces a task graph for the factorization. These tasks are inter-related to each other through their data dependences in the factorization algorithm. To process the tasks on various manycore architectures in a portable manner, we also present a portable tasking API that incorporates different tasking backends and device-specific features using an open-source framework for manycore platforms *i.e.*, Kokkos. A performance evaluation is presented on both Intel Sandybridge and Xeon Phi platforms for matrices from the University of Florida sparse matrix collection to illustrate merits of the proposed task-based factorization. Experimental results demonstrate that our task-parallel implementation delivers about 26.6x speedup (geometric mean) over single-threaded incomplete Cholesky-by-blocks and 19.2x speedup over serial Cholesky performance which does not carry tasking overhead using 56 threads on the Intel Xeon Phi processor for sparse matrices arising from various application problems.

***Keywords*—** Sparse Factorization, Algorithm-by-blocks, 2D Layout, Task Parallelism

---

This technical report is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that anyone wanting to cite or reproduce it ascertains that no published version in journal or proceedings exists.

The code described in this paper is publicly available at `https://github.com/trilinos/Trilinos/tree/master/packages/shylu/tacho`.

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the U.S. Department of Energy under contract DE-AC04-94-AL85000.

1 Introduction

Incomplete Cholesky factorization is effectively used for preconditioned iterative methods to solve large-scale Symmetric Positive Definite (SPD) linear systems. Computing incomplete factorizations scalably in shared-memory systems is an open problem for both multicore and manycore architectures, because incomplete factorizations are characterized by irregular data access patterns, frequent synchronizations, and dependences that limit the available parallelism when expressed in a data-parallel manner. First, incomplete factorizations, by definition, are much more sparse than their counterparts, the complete factorizations. This sparsity precludes the use of any Dense Linear Algebra (DLA) operations such as the Basic Linear Algebra Subprogram (BLAS) kernels and results in a sparse data access pattern that is very irregular in a traditional incomplete factorization algorithm. By traditional, we refer to the incomplete factorization algorithm (*e.g.*, left-looking and right-looking variants) that is implemented using a compressed sparse row/column format. Second, a traditional parallel incomplete factorization uses an ordering technique of rows or/and columns to expose some parallelism. However, the conventional sparse factorization algorithms still suffer from synchronization bottlenecks for rows or columns that cannot be factored in parallel. Third, when the only available parallelization option is using a simple `parallel for`, the traditional incomplete factorization cannot be expressed efficiently. In general, a matrix is reordered to explore parallelism and the substructure resulting from the reordering phase is more suitable for *task-parallel* algorithms, which require means of expression as such.

We propose a parallel incomplete sparse factorization algorithm and its implementation targeting multicore and manycore architectures, called *Tacho*. In particular, we focus on task-parallel sparse level(k) incomplete Cholesky factorization.[1] Our approach is based on a class of algorithms, called *algorithms-by-blocks*, originating from parallel out-of-core DLA algorithms [21]. This class of algorithms has been adopted for asynchronous thread-parallel execution in DLA libraries [11, 14, 35]. Applying this style of algorithms to sparse matrix factorization, there are several challenges in handling the irregular data structure of sparse matrices and blocking strategies that can expose most of parallelism.

In DLA, several Application Programming Interfaces (APIs) [22, 29, 37] are proposed to facilitate algorithms-by-blocks. These APIs are primarily developed to improve data locality by changing the standard columnwise storage format to a recursive block storage format. However, no attempt has been made to use a similar 2D block layout on sparse matrices for shared-memory factorizations. The sparse linear algebra community has considered block-based layouts for simple kernels such as sparse matrix vector multiply [5] or sparse matrix-matrix multiply [9]. Instead, 1D data layouts are often used for high performance computing libraries that utilize factorizations [24]. The use of 1D partitions can severely limit parallelism when a sparse matrix has a large bandwidth and it incurs synchronization bottlenecks. On the contrary, the 2D block layout based on Nested Dissection (ND) ordering is more suitable to expose fine-grained task parallelism and better load balance as it can create a number of tasks that can be concurrently executed. This enables the asynchronous task parallelism in the block level rather than in a row level. The usage of such a layout also regularizes the data access with respect to the blocks. The subblocking approach is mostly applied to sparse di-

[1] The level(k) incomplete factorization determines the location of additional nonzero factors, called $fills$, based on the sparsity pattern of a matrix. Initially, all nonzero entries of the matrix are set with a level 0. Then, a fill is created with an increasing level and restricted by the threshold k during Gaussian elimination.

rect factorizations for computing dense (supernodal) blocks [23, 26, 28, 36]. To the best of our knowledge, this 2D sparse partitioned-block layout, where *the blocks of a sparse matrix are themselves sparse*, has not been explored for complex kernels like sparse factorizations in shared-memory architectures. For brevity, we will use the term 2D layout or 2D block matrix for a two-dimensional sparse partitioned-block layout and a matrix that uses that layout respectively. From an implementation perspective, the 2D block, a light-weight object that describes a rectangular computing region on a sparse matrix, becomes an entry of a 2D block matrix (matrix of blocks). Note that the hierarchical representation of our block sparse matrix does not need to repack data associated with a block; instead, the block points to the base matrix with appropriate meta data (partition information) specifying the rectangular region. Further performance improvement can be achieved by repacking the corresponding data of blocks. However, repacking may carry additional overhead and the cost might be significant considering the light workload of incomplete factorization.

By applying algorithms-by-blocks on the 2D layout, a problem is reformulated in terms of block matrix computations; blocks become a computing unit and operations among blocks become tasks. Then, resulting tasks are scheduled, potentially *out-of-order*, to compute resources after satisfying task dependences. In short, the depedences expressed through the API define a partial order of possible task executions that the runtime system maps to available threads and the particular system architecture. This approach yields a clear separation of concerns by decoupling algebraic structure from runtime task scheduling.

We have implemented the task-parallel Cholesky factorization by extending the open-source Kokkos library [18] with a portable tasking interface and using it for the factorization. The Kokkos library provides a high-level programming abstraction pursuing portable performance on various manycore architectures. We have extended it to include a portable interface for task parallelism. Through the interface, developers write an application code once and the code is portable to heterogeneous device environments with device-specific programming models. Currently, our extensions include backends for Pthreads and Qthreads [39] to schedule task parallelism on host devices, *e.g.*, IBM POWER series, Intel Xeon multicore and Intel Xeon Phi manycore processors. Kokkos already provides support for data parallelism on the GPU, and we are on-track to develop a GPU backend for the Kokkos task interface in the coming year. Key features of the new interface include futures and dependences to enable general task Directed Acyclic Graphs (DAGs), and non-blocking semantics to accommodate devices such as GPUs.

The main contributions of this paper include:

- a high-level matrix abstraction for 2D sparse partitioned-block matrices that facilitates task parallelism with dependences using future references;
- a new task-parallel implementation for sparse level(k) incomplete Cholesky factorization that utilizes 2D layouts;
- a portable tasking interface and its implementation, designed to support different tasking backends for different hardware features and limitations;
- performance evaluation for several test problems that shows our task-parallel factorization-by-blocks delivers scalable and portable parallel performance on an Intel Sandybridge processor and an Intel Xeon Phi coprocessor.

The rest of the paper is organized as follows. Section 2 describes our extensions of the Kokkos library to support task parallelism. Section 3 explains sparse level(k) incomplete Cholesky-by-blocks. A performance evaluation is presented in Section 4. Section 5 discusses related work. Finally, we conclude in Section 6.

2 Kokkos portable tasking API

The programming model chosen for Tacho is an extension of Kokkos [18] to support dependency-driven task-parallel execution. Kokkos has been developed to address the challenge of performance portability across manycore architectures; *e.g.*, multicore CPU, Intel Xeon Phi, and NVIDIA GPU. Until recently, Kokkos was supported only for data parallelism. However, our extensions enable the specification of computational tasks together with the dependence relationships between them. These tasks and dependences form an implicit DAG that is scheduled by a run time system on behalf of the application.

2.1 Abstraction

A Kokkos task is created with a C++ *functor* (body of work) to execute and an optional number of *dependences*. Dependences are defined by handles to other tasks that must complete before the task scheduler will execute the newly created task. Upon successful creation of a task, a *future* is returned. A Kokkos future is the handle to a task that may be used to denote inter-task dependences, probe for task completion status, or obtain the return value of a task.

A Kokkos *execution policy* defines how and where "bodies of work" will execute in parallel. For the task interface, all run time task management occurs through a *task execution policy*. This policy is responsible for the creation, destruction, scheduling, and execution of a group of related tasks. Dependent tasks must be members of the same task execution policy so that their dependences can be enforced by the policy.

We succinctly illustrate how a user creates a task execution policy and tasks with dependences in Fig. 1. In this example, three tasks are implemented with C++ classes: `FunctorX`, `FunctorY`, and `FunctorZ`. The policy's `create` function allocates a task with the given functor implementation, but does not schedule the task. The `add_dependence` function introduces a dependence of the first task upon the second task; *i.e.*, the first task is not allowed to execute until the second task completes. The `spawn` function schedules the task for execution. If there are no dependences, the scheduled task may immediately execute, perhaps even completing before the `spawn` function returns. Finally, the `wait` function is called to wait for all ready tasks owned by the policy to complete, including tasks that become ready in the course of executing other tasks owned by the policy.

Properties of Kokkos tasks and data are derived from goals of both productivity and performance portability. Support for task DAG execution based on dependences and futures allow significant flexibility in the types of algorithms that can be expressed and the amount of parallelism that can be exposed. Tasks are non-preemptive and non-blocking because some architectures targeted by Kokkos; *e.g.*, tasks on GPUs cannot support blocking.

Kokkos data is expressed in the form of multidimensional arrays, called *views*. A Kokkos view defines where allocated data resides (*e.g.*, CPU vs. GPU memory) and the *layout* of that multidimensional array data. Layout is polymorphic with respect to the execution architecture. For example, on CPUs the default layout is row major (array of structures) and on

```
// using the Kokkos namespace
void foo() {
  using Space = /* where to execute */ ;

  // the policy is defined on a specific execution space
  // multiple policy objects on different execution spaces are allowed
  TaskPolicy<Space> policy ;

  // FunctorX,Y,Z are C++ classes containing tasks' code
  Future<Space> f_x = policy.create( FunctorX() );
  Future<Space> f_y = policy.create( FunctorY() );
  Future<Space> f_z = policy.create( FunctorZ() );
  policy.add_dependence( f_z , f_x ); // f_z depends on f_x
  policy.add_dependence( f_z , f_y ); // f_z depends on f_y
  policy.spawn( f_z ); // FunctorZ is now waiting on FunctorX and FunctorY
                       // to complete execution
  policy.spawn( f_x ); // may immediately execute
  policy.spawn( f_y ); // may immediately execute
  wait( policy ); // wait for all tasks to complete
}
```

Figure 1: Simple example of using a Kokkos task execution policy to create tasks, introduce dependences, spawn tasks for execution, and wait for a group of tasks to complete.

GPUs the default layout is column major (structure of arrays). The view abstraction and API allows an architecture-appropriate layout to be introduced into user code without requiring any modification of that code.

2.2 Implementation

Initially, we implemented a basic task execution policy using a Pthreads thread-pool to execute tasks on shared memory multicore and manycore platforms. We are also developing a task execution policy using the Qthreads [39] lightweight threading library for both task scheduling and execution. The Qthreads library is optimized for efficient and scalable node-level execution on CPU-like multicore and manycore architectures. Qthreads offers fast context switching between tasks and software-implemented Full Empty Bit (FEB) synchronizations, inspired by the Tera MTA / Cray XMT architecture [3], that map especially well to futures. We plan to develop a task execution policy for GPU architectures within the next year.

A task execution policy's scheduler has three primary responsibilities. First, it tracks which tasks are ready for execution and which tasks are waiting for inter-task dependences to be satisfied. Second, it selects and executes ready tasks on available cores. Third, as tasks complete it updates their associated dependent tasks to a ready state. The goal of a scheduler is to carry out these responsibilities as efficiently and thread-scalably as possible. Note that application code written using the Kokkos API does not need to be changed to benefit from new or improved backend implementations.

3 Sparse incomplete factorization

This section describes task-based level(k) incomplete Cholesky factorization using the 2D block layout. The factorization method in Tacho consists of symbolic factorization and nu-

meric factorization. Symbolic factorization (Section 3.1) results in a 2D sparse partitioned-block matrix (Section 3.2). Once the sparsity pattern of the Cholesky factors is determined, Cholesky-by-blocks numeric factorization (Section 3.3) generates tasks with dependences according to the data flow of the factorization process and computes factors via the portable tasking API.

3.1 Symbolic factorization

Algorithm 1 Symbolic factorization

1: Compute the ND ordering
2: Reorder the matrix with the ND ordering
3: Prune t levels of the ND tree to control minimum block sizes
4: Find the level-k fill
5: Construct 2D block matrix based on the ND tree

Algorithm 1 describes our symbolic factorization phase. As a first step, we use a Nested Dissection (ND) ordering [20] from the Scotch [32] library to expose a high degree of concurrency during the factorization. The algorithm recursively separates a matrix into two subproblems, providing a tree hierarchy. In addition to improving the concurrency, the ND ordering also reduces the amount of fill (zeros turning into non-zeros during the factorization), which corresponds to the amount of work in the factorization. To keep the same amount of work during scaling studies, we generate the same number of levels of the ND tree but optionally prune the tree to allow enough work for each task. Next, we determine the location of potential fill up to the given fill-level using graph analysis. Our implementation of the symbolic factorization follows the algorithm proposed by Hysom and Pothen [25]. Using the ND tree, we construct a 2D sparse partitioned-block matrix (a sparse matrix of sparse matrices). We devote the next subsection to this last step.

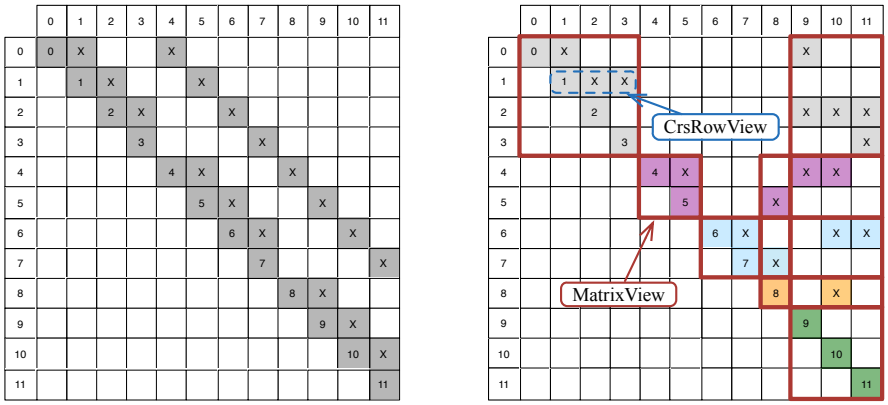
3.2 2D sparse partitioned-block matrix

The factorization algorithm in Tacho is uniquely characterized by its recursive definition of the sparse matrix structure. In this approach, a 2D sparse matrix consists of submatrices to define computational blocks on a scalar sparse matrix. As a result, our task-parallel Cholesky factorization has the same look-and-feel as the scalar Cholesky factorization, greatly improving programmability. We demonstrate this later in Section 3.3.

Scotch provides an array of ranges for columns (or rows) in the reordered matrix where a range corresponds to a group of variables (separator) that can be treated together. Based on the hierarchical relation of the ranges, we can construct a 2D sparse block layout over the scalar matrix by creating view objects that cover nonzero regions. For example, Fig. 2 illustrates the reordered sparse matrix and its corresponding block structure. We denote this collection of submatrices as a 2D sparse partitioned-block matrix.

To facilitate the 2D block layout, we propose the following hierarchy of views on a sparse matrix:

CrsMatrixBase a base matrix object that contains the standard data structure for sparse matrices *i.e.*, row-pointers, column-index array, and value array;



(a) Natural ordering

(b) Block reordering

Figure 2: An example of symmetric block nested dissection ordering permuted by Scotch. Left: a sparse matrix with natural ordering. Right: a hierarchical view of the block reordered matrix.

MatrixView a matrix view that defines a 2D rectangular data region overlaid on the base matrix, which is defined by offsets and view dimensions, see Fig. 2;

CrsRowView a sparse row view that defines the range of columns of a row associated with a `MatrixView`;

TaskView a derived class extended from the `MatrixView` to include a future associated with a corresponding 2D data region.

We assume that matrices are stored in `CrsMatrixBase` using the standard Compressed Sparse Row (CSR) format. This base matrix has template arguments for a value type which can be either a scalar (for a scalar matrix) or sparse block (for a 2D matrix). A light-weight matrix view is defined as `MatrixView` with partition information. As the matrix view is templated with an associated base matrix, it could be a block of scalars or block of 2D matrices. This view object becomes a basic computing unit in our task-parallel sparse matrix factorization. Task granularity is controlled by adjusting the size of a matrix view; a view can be split into many views or views can be merged into a single view. Since the matrix view only contains meta data, these operations do not carry overhead of data repacking. Our current work does not include precise blocksize tuning capabilities for generating optimal task granularity. Instead, we roughly control the task granularity by adjusting the Scotch tree hierarchy level.

In addition, an extension of the matrix view is used for the tasking interface, called `TaskView`. This class contains a `future` object to record a future state updated by tasks associated with a particular 2D block. Finally, the `CrsRowView` specifies a part of a row within a matrix view. The row view is used to access elements of a matrix view in which each element can be either a scalar or sparse block matrix itself.

Algorithm: $A := \text{CHOL-BLOCKED}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$

where A_{TL} is 0×0

while $length(A_{TL}) < length(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

where A_{11} is $b \times b$

$A_{11} := \text{CHOL}(A_{11})$ CHOL

$A_{12} := \text{TRIU}(A_{11})^{-1} A_{12}$ TRSM

$A_{22} := A_{22} - A_{12}^{T} A_{12}$ HERK

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

endwhile

Figure 3: Cholesky algorithm. The blocks in the 2×2 and 3×3 block matrices that correspond to each other are of the same color. CHOL and TRIU represent Cholesky factorization and the upper triangular part of an input matrix respectively.

Difference from other approaches. After we construct a 2D matrix based on ND block ordering, we no longer use the tree hierarchy to extract parallelism. Tasks are created by the algorithms-by-blocks based on the 2D block sparse layout. The approach differs from others in that we do not explicitly rely on the ND tree-hierarchy (or the elimination tree) in the numeric factorization phase. On the other hand, conventional approaches for task-parallel implementation explicitly use the tree-hierarchy to generate independent tasks and their dependences, as well as to distribute compute resources according to subtree structures [2]. Such implementations may not be performance-portable as the implementation is hard-wired to problem-specific sparse structures and hardware execution environments.

3.3 Numeric factorization: Cholesky-by-blocks

This section describes the Cholesky-by-blocks algorithm. The right-looking Cholesky algorithm is shown in Fig. 3. The algorithm is expressed with partitioned matrices using Formal Linear Algebra Methods Environment (FLAME) notations [34, 38]. A short description of the notation follows. First, note that the algorithm will work equally well on matrices with scalar entries (b=1) or sparse block entries (variable b). Second, A_{BR} is partitioned further and updated at each iteration of the `while` loop. In this particular algorithm, note that the computations only happen on the A_{BR} block. The algorithm consists of three different operations. Using the BLAS notation, the three operations in Fig. 3 are CHOL, TRSM, and HERK which correspond to a Cholesky factorization, triangular solve and hermitian rank-k update. Finally, the partition is redefined (see the thick partition line moving forward) for the next iteration of the algorithm.

We transform this algorithm into Cholesky-by-blocks by converting the basic computing

Algorithm: $A := \text{CHOL-BY-BLOCKS}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$

where A_{TL} is 0×0

while $length(A_{TL}) < length(A)$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

where A_{11} is 1×1

```
function genTaskChol:
  Future f = create( Chol, A11(0,0) )
  add_dependence( f, A11(0,0).getFuture() )
  A11(0,0).setFuture( f )
  spawn( f )

function genTaskTrsm:
  for j in A12.nnz()
    Future f = create( Trsm, A11(0,0), A12(0,j))
    add_dependence( f, A11(0,0).getFuture() )
    add_dependence( f, A12(0,j).getFuture() )
    A12(0,j).setFuture( f )
    spawn( f )

function genTaskHerk:
  for i in A12.nnz()
    for j in A12.nnz()
      if exist( A22(i,j) )
        Future f = create( i==j ? Herk : Gemm,
                           A12(0,i), A12(0,j),
                           A22(i,j) )
        add_dependence( f, A12(0,i).getFuture() )
        add_dependence( f, A12(0,j).getFuture() )
        add_dependence( f, A22(i,j).getFuture() )
        A21(i,j).setFuture( f )
        spawn( f )
```

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

endwhile

Figure 4: Cholesky-by-blocks algorithm on a 2D partitioned-block matrix. The blocks in the 2×2 and 3×3 block matrices that correspond to each other are of the same color.

unit from a scalar to a block. The blocked algorithm described in Fig. 3 is applied to A_{ij} elementwise. By doing so, the three different operations inside the while loop becomes three different opportunities to generate tasks. Fig. 4 describes the Cholesky-by-blocks algorithm using the Kokkos tasking interface. By running the Cholesky-by-blocks on a 2D matrix, tasks are created and spawned with dependences. A spawned task is recorded on a `future` of an output matrix view associated with the task. The dependence for each task is determined by the input/output blocks used in the task and any futures associated with them. Since blocks

record associated tasks, we do not need to keep track of the entire task dependences but only follow the loop body of the algorithm.

We demonstrate this algorithm with a small example matrix. Suppose that ND ordering provides a symmetric permutation matrix P, which leads to an upper triangular block matrix

$$P^T AP = \begin{pmatrix} A_{00} & & & & A_{04} \\ & A_{11} & & A_{13} & A_{14} \\ & & A_{22} & A_{23} & A_{24} \\ & & & A_{33} & A_{34} \\ & & & & A_{44} \end{pmatrix}$$

where all A_{ij} blocks are sparse and have block dimensions compatible with each other. Then, we apply the Cholesky-by-blocks algorithm as depicted in Fig. 4. As a result, a sequence of block matrix computations is generated as illustrated in Fig. 5. Task dependences are found from the input/output relations described in the loop body of the blocked Cholesky algorithm:

$$\text{CHOL} \rightarrow \text{TRSM} \rightarrow \text{HERK (or GEMM)}.$$

For example, a TRSM task will depend on a CHOL task that is an input for the task; a HERK task will depend on TRSM tasks that should be completed on the data region that is required as an input of the task. This dependence relationship is applied to individual blocks and a corresponding task DAG is shown in Fig. 6b. Contrary to the coarse grain tasks that correspond to the block ND tree depicted in Fig. 6a, our sparse Cholesky-by-blocks generates a larger number of asynchronous fine-grained tasks. In this particular example, the CHOL(A_{22}) in the third iteration can be executed before finishing tasks (*i.e.*, TRSM, HERK and GEMM) created in the first and second iterations. This is possible because the CHOL(A_{22}) has input dependences only to itself. Our Cholesky-by-blocks algorithm does not do any special book-keeping to determine when to launch a certain task. Instead, it loops through the entire 2D matrix and generates the tasks as it steps through the loop. Tasks such as CHOL(A_{22}) can be run immediately as they are created with no dependences. This is possible because our task-parallel approach is not strictly tied to the tree hierarchy derived from ND ordering. Furthermore, we see in the first iteration that TRSM(A_{04}) can begin immediately as its dependences are satisfied, whereas CHOL(A_{22}) won't be even created till the second iteration, which is the opposite of what a tree based algorithm would have done. This approach exposes much more fine-grained task parallelism. Also note that HERK(A_{33}) in the third iteration has taken the form of a sparse GEMM which is much more cache-friendly to compute than simple rank-1 updates.

4 Performance evaluation

In this section, we evaluate our task-parallel incomplete Cholesky factorization on problems selected from the University of Florida sparse matrix collection [17]. The matrix properties are tabulated in Table 1. The largest problem, `G3_circuit`, has about 1.5 million rows. Note that `bmwcra_1` and `pwtk` are relatively denser by an order of magnitude than other test problems (see the average number of non-zeros per row in the table). Later, we show that this property significantly changes both serial and parallel performance. All experiments are performed on a machine with a dual socket configuration of "Sandy Bridge" processors (2×8 Xeon E5-2670 2.6GHz cores) and two "Knights Corner" coprocessors (1×57 Xeon Phi with 1.1GHz cores)

$$\left(\begin{array}{ccccc} A_{00} & & & & A_{04} \\ & A_{11} & & A_{13} & A_{14} \\ & & A_{22} & A_{23} & A_{24} \\ & & & A_{33} & A_{34} \\ & & & & A_{44} \end{array}\right) \qquad \begin{array}{lcl} A_{00} & := & \textsc{Chol}(A_{00}) \\ A_{04} & := & \textsc{Triu}(A_{00})^{-1}A_{04} \\ A_{44} & := & A_{44} - A_{04}^T A_{04} \end{array}$$

(a) 1st iteration

$$\left(\begin{array}{ccccc} A_{00} & & & & A_{04} \\ & A_{11} & & A_{13} & A_{14} \\ & & A_{22} & A_{23} & A_{24} \\ & & & A_{33} & A_{34} \\ & & & & A_{44} \end{array}\right) \qquad \begin{array}{lcl} A_{11} & := & \textsc{Chol}(A_{11}) \\ A_{13} & := & \textsc{Triu}(A_{11})^{-1}A_{13} \\ A_{14} & := & \textsc{Triu}(A_{11})^{-1}A_{14} \\ A_{33} & := & A_{33} - A_{13}^T A_{13} \\ A_{34} & := & A_{34} - A_{13}^T A_{14} \\ A_{44} & := & A_{44} - A_{14}^T A_{14} \end{array}$$

(b) 2nd iteration

$$\left(\begin{array}{ccccc} A_{00} & & & & A_{04} \\ & A_{11} & & A_{13} & A_{14} \\ & & A_{22} & A_{23} & A_{24} \\ & & & A_{33} & A_{34} \\ & & & & A_{44} \end{array}\right) \qquad \begin{array}{lcl} A_{22} & := & \textsc{Chol}(A_{22}) \\ A_{23} & := & \textsc{Triu}(A_{22})^{-1}A_{23} \\ A_{24} & := & \textsc{Triu}(A_{22})^{-1}A_{24} \\ A_{33} & := & A_{33} - A_{23}^T A_{23} \\ A_{34} & := & A_{34} - A_{23}^T A_{24} \\ A_{44} & := & A_{44} - A_{24}^T A_{24} \end{array}$$

(c) 3rd iteration

$$\left(\begin{array}{ccccc} A_{00} & & & & A_{04} \\ & A_{11} & & A_{13} & A_{14} \\ & & A_{22} & A_{23} & A_{24} \\ & & & A_{33} & A_{34} \\ & & & & A_{44} \end{array}\right) \qquad \begin{array}{lcl} A_{33} & := & \textsc{Chol}(A_{33}) \\ A_{34} & := & \textsc{Triu}(A_{33})^{-1}A_{34} \\ A_{44} & := & A_{44} - A_{34}^T A_{34} \end{array}$$

(d) 4th iteration

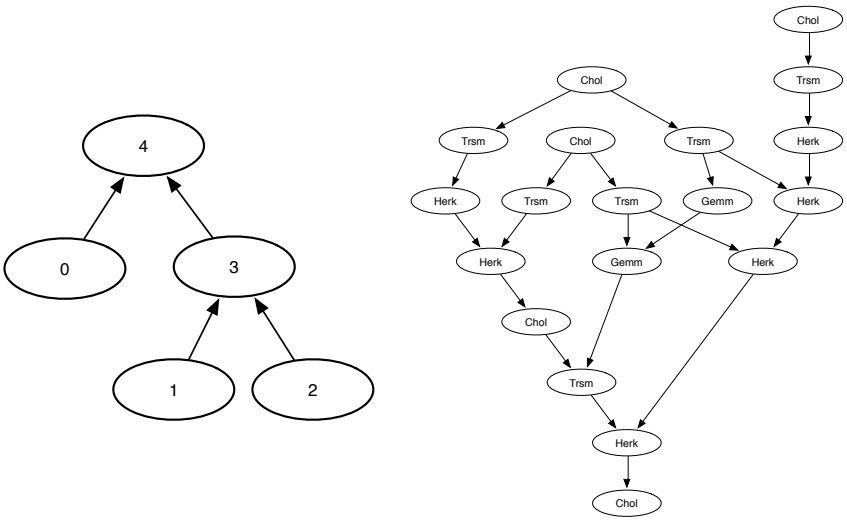
$$\left(\begin{array}{ccccc} A_{00} & & & & A_{04} \\ & A_{11} & & A_{13} & A_{14} \\ & & A_{22} & A_{23} & A_{24} \\ & & & A_{33} & A_{34} \\ & & & & A_{44} \end{array}\right) \qquad A_{44} := \textsc{Chol}(A_{44})$$

(e) 5th iteration

Figure 5: Generated block matrix computations while proceeding on Cholesky-by-blocks.

connected via PCI-Express. Each Intel Xeon Phi coprocessor has 57 cores with 4 hyperthreads per core. We turn off hyperthreading using a hardware locality library [8] and use up to 56 cores in a native mode as one core is reserved for the operating system. The GNU compiler (5.1.0) and Intel compiler (15.2.164) with -O3 and Kokkos Pthreads and Qthreads backends are used for the Sandy Bridge multicore processor and Xeon Phi coprocessor respectively.

To benchmark parallel performance, we compare Tacho to a parallel ILU package, Euclid [24] as the closest alternative. The Euclid library is a scalable implementation of the



(a) Block ND tree

(b) Task DAG

Figure 6: A task dependence graph for the example illustrated in Fig. 5.

Matrix ID	# of rows(n)	# of nonzeros(nnz)	nnz/n
ecology2	999,999	4,995,991	4.99
G3_circuit	1,585,478	7,660,826	4.83
parabolic_fem	525,825	3,674,625	6.98
thermal2	1,228,045	8,580,313	6.98
bmwcra_1	148,770	10,641,602	71.53
pwtk	217,918	11,524,432	52.88

Table 1: Test problems selected from the University of Florida sparse matrix collection.

parallel ILU factorization using Message Passing Interface (MPI). The Euclid parallel ILU algorithm consists of four phases: 1) a partitioning phase to minimize communication costs, 2) local reordering to separate interior nodes from boundary nodes, 3) global reordering to improve parallelism and 4) the numeric factorization. In this comparison, we use symmetric Reverse Cuthill McKee (RCM) ordering for Euclid to reduce the bandwidth of matrices as this is the best possible ordering for Euclid's performance. A matrix is distributed to processors such that each processor owns an equal number of nodes. As the two codes has different reordering strategies and symbolic factorization phases, we compare the numeric factorization phase of Euclid with our factorization code. Note that Euclid is an LU factorization code as opposed to Cholesky factorization that we do here. As a result, we divide Euclid's numbers by half to approximate the factorization costs. It is important not to place a huge emphasis on these performance numbers as we are comparing an MPI based code with a shared-memory code. However, this is the closest codebase that is publicly available for parallel incomplete factorization. We present these numbers just to demonstrate the difference between coarse-

	# of nonzeros in U[millions]				
Matrix ID	L0	L1	L2	L4	Chol(AMD)
`ecology2`	2.9	4.7	6.0	8.3	45.7
`G3_circuit`	4.6	7.4	9.6	14.5	189.1
`parabolic_fem`	2.1	3.5	4.8	6.9	36.1
`thermal2`	4.9	7.9	10.6	14.8	64.8
`bmwcra_1`	5.3	14.1	23.2	38.5	90.9
`pwtk`	5.9	10.9	15.0	21.4	60.0

Table 2: Number of nonzero U factors resulting from level(k) symbolic factorization; for comparison, the last column shows the number of fill from complete factorization with AMD ordering.

grained parallelism with traditional rowwise layouts and fine-grained parallelism with 2D layouts.

4.1 Symbolic factorization results

The symbolic factorization phase determines the location of fill for the level(k) incomplete Cholesky factorization. Similar to Hysom and Pothen [25], our symbolic factorization performs Breadth First Search (BFS) on the adjacency graph of a matrix in parallel for each node to specify level(k) fill structure. This is implemented in a scalable fashion using two Kokkos parallel patterns: `parallel_for` and `parallel_scan`. The numbers of nonzero U factors generated by the level(k) incomplete Cholesky factorization are summarized in Table 2. All matrices are reordered with the block ND algorithm provided by the Scotch library [32]. For comparison, we also provide size of the fill for complete Cholesky factorization in the last column of the table. We do not report the times for symbolic factorization as they are not significant.

4.2 Numeric factorization results

Parallel performance. We report strong scalability of our task-parallel level(k) incomplete Cholesky factorization and evaluate the parallel performance against the Euclid package. Since both Kokkos Pthreads and Qthreads backends report similar timing results, here we report only for Kokkos Pthreads results. First, we compare the time taken for factorization on the Intel Xeon multicore architecture. Fig. 7 and Fig. 8 show the numeric factorization time for our task-parallel Cholesky factorization and Euclid respectively. Euclid does not provide separate timing results for symbolic factorization.[2] As the symbolic factorization costs much less than the numeric factorization, we directly compare the numeric factorization time of our code to the time reported by the factorization phase of Euclid. As two codes report different ranges of timing results, we cannot plot all graphs in the same scale and some graphs are plotted with a different time scale. While Euclid scales well for some matrices, relatively denser problems such as `bmwcra_1` and `pwtk` prove harder to solve with Euclid-like algorithms. We conjecture

[2]Euclid reports timing results for subdomain graph setup, factorization, and solve setup.

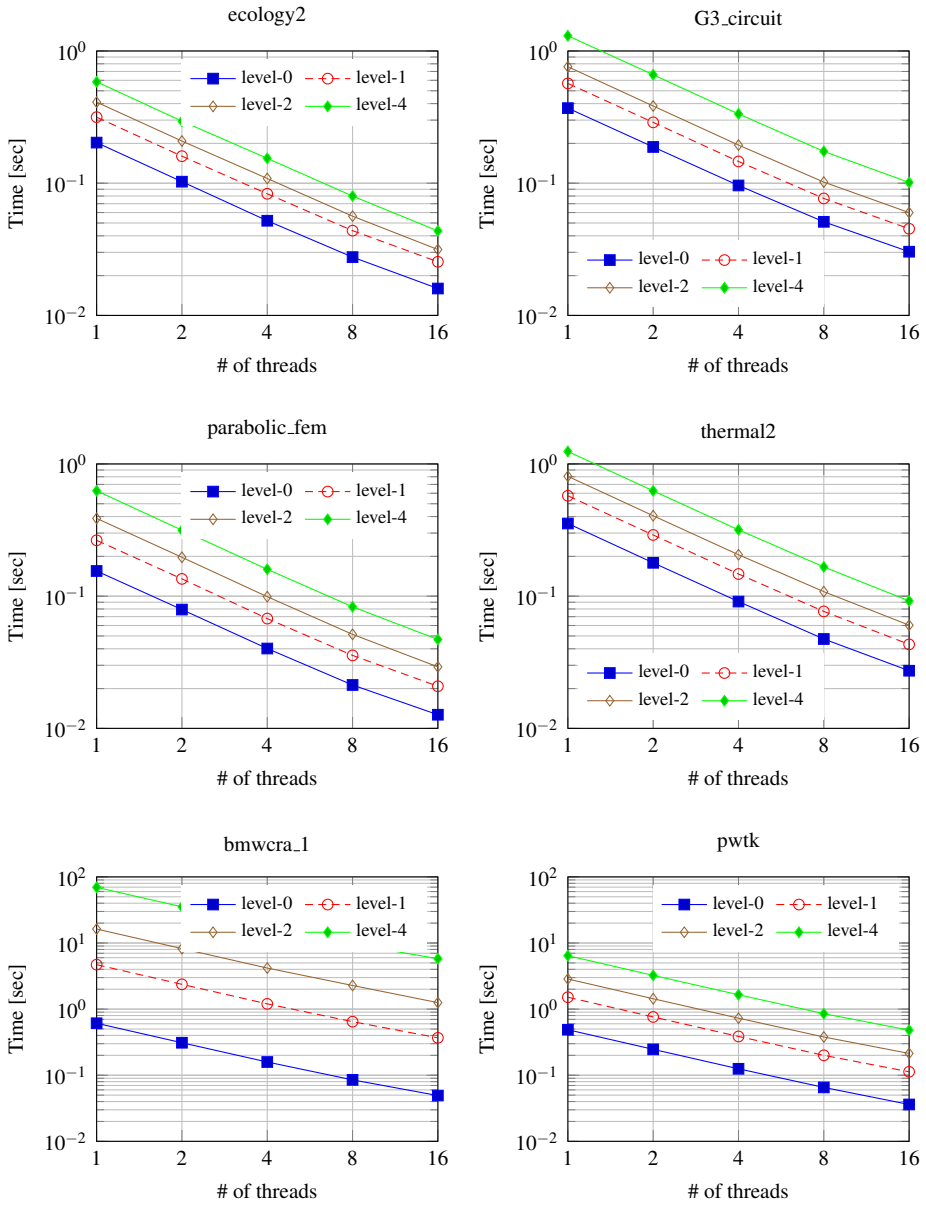


Figure 7: [Sandybridge] Time for level(k) incomplete Cholesky-by-blocks factorization with the Kokkos Pthreads backend (our method).

that the main reason that these matrices are harder to solve is the 1D rowwise matrix partitions. Although the 1D panels are globally reordered to increase parallelism, matrices with a large bandwidth (a higher number of nonzeros per row) are not ideal for such parallelism. The

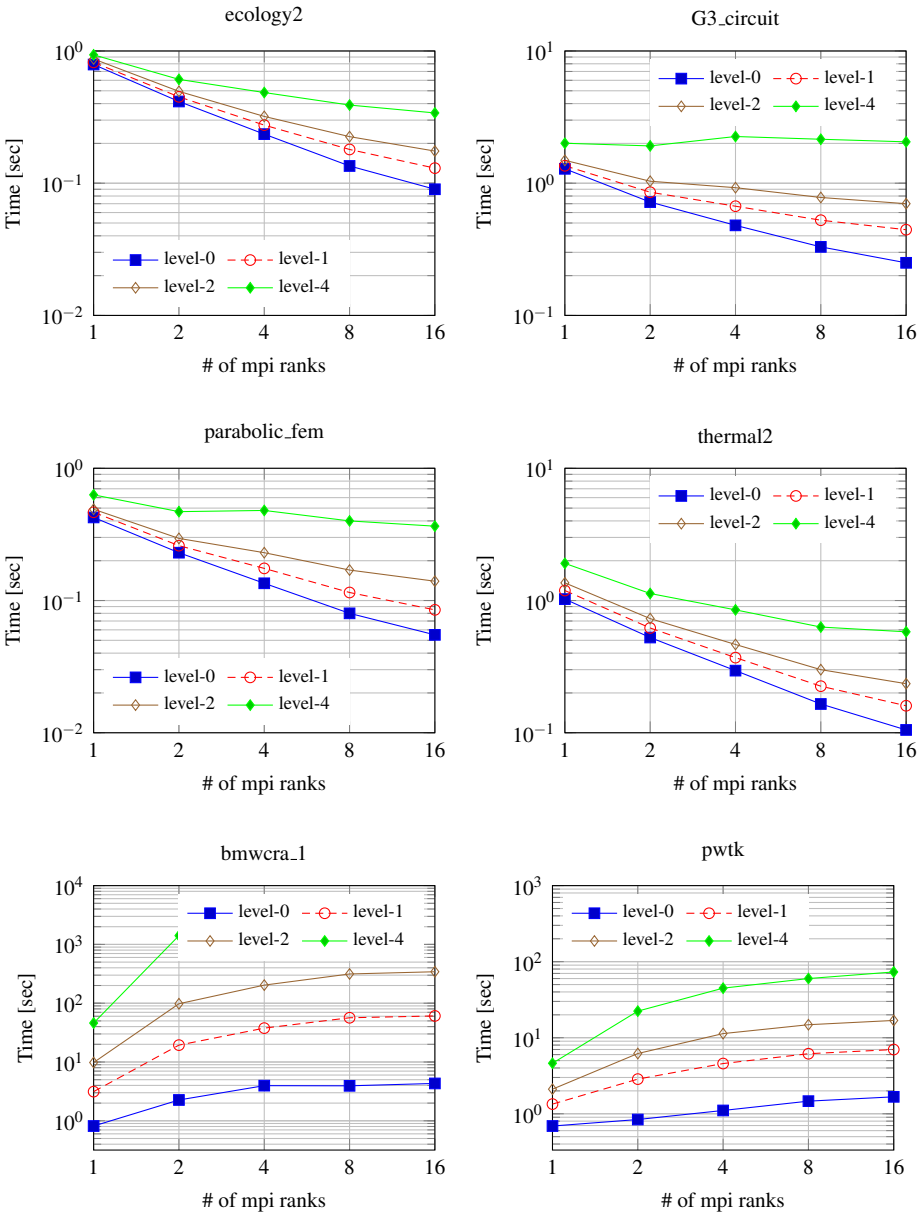


Figure 8: [Sandybridge] Time for Euclid level(k) incomplete LU factorization (for comparison). The time cost is divided by two to compare with the result of incomplete Cholesky factorization. Note that some plots have different ranges of values from the plots drawn in Fig. 7.

Matrix ID	prune level	# of ranges	# of blocks
ecology2	10	356	1,050
G3_circuit	10	572	2,082
parabolic_fem	8	668	2,437
thermal2	10	378	1,346
bmwcra_1	4	470	1,953
pwtk	4	778	2,933

Table 3: Structure of 2D sparse partitioned-block matrices. The prune level lists the height of pruned ND subtrees from the leaf level. The number of ranges is the total number of vertices in a ND tree. The number of blocks is the number of blocks in a 2D matrix.

same performance trend is commonly observed on other test problems with an increasing level of fills. The increased number of fills incurs more synchronization bottlenecks among 1D panels and results in the loss of concurrency. On the other hand, our task-parallel Cholesky factorization delivers robust parallel scalability for all test problems, as tasks are generated based on 2D block matrices and executed asynchronously.

Comparable parallel performance of our task-parallel Cholesky factorization on the Intel Xeon Phi coprocessor is illustrated in Fig. 9. For most test problems, our task-parallel Cholesky algorithm scales up to the largest number of available threads on the coprocessor. Our task-parallel implementation delivers about 26.6x speedup (geometric mean) over single-threaded Cholesky-by-blocks and 19.2x speedup over serial Cholesky factorization (which does not carry tasking overhead) using 56 threads on the Intel Xeon Phi processor.

Comparison of tasking overhead between Pthreads and Qthreads. Finding appropriate task granularity is very important to attain higher parallel performance. Multiple aspects of performance trade-offs should be considered to determine optimal task granularity:

- total number of generated tasks,
- level of concurrency expressed from sparse factorization,
- tasking overhead (context switching, task creation, scheduling and destruction),
- data access overhead for multiple sparse kernel launching,
- number of computing units and local cache sizes.

Using many fine-grained tasks results in a higher degree of concurrency which is more suitable for manycore computing environments. However, using such a large number of tasks may significantly increase tasking overhead and irregular data access cost, which decreases overall parallel performance of sparse factorization. On the other hand, generating coarse grained tasks can decrease tasking overhead but may not expose enough concurrency to use all available hardware resources.

To explore this performance trade-off, we plot the relative tasking overhead, T/T_{serial}, where T is the time cost of task-parallel Cholesky-by-blocks and T_{serial} is the time cost of serial sparse Cholesky factorization. Fig. 10 describes the relative tasking overhead on the Sandy

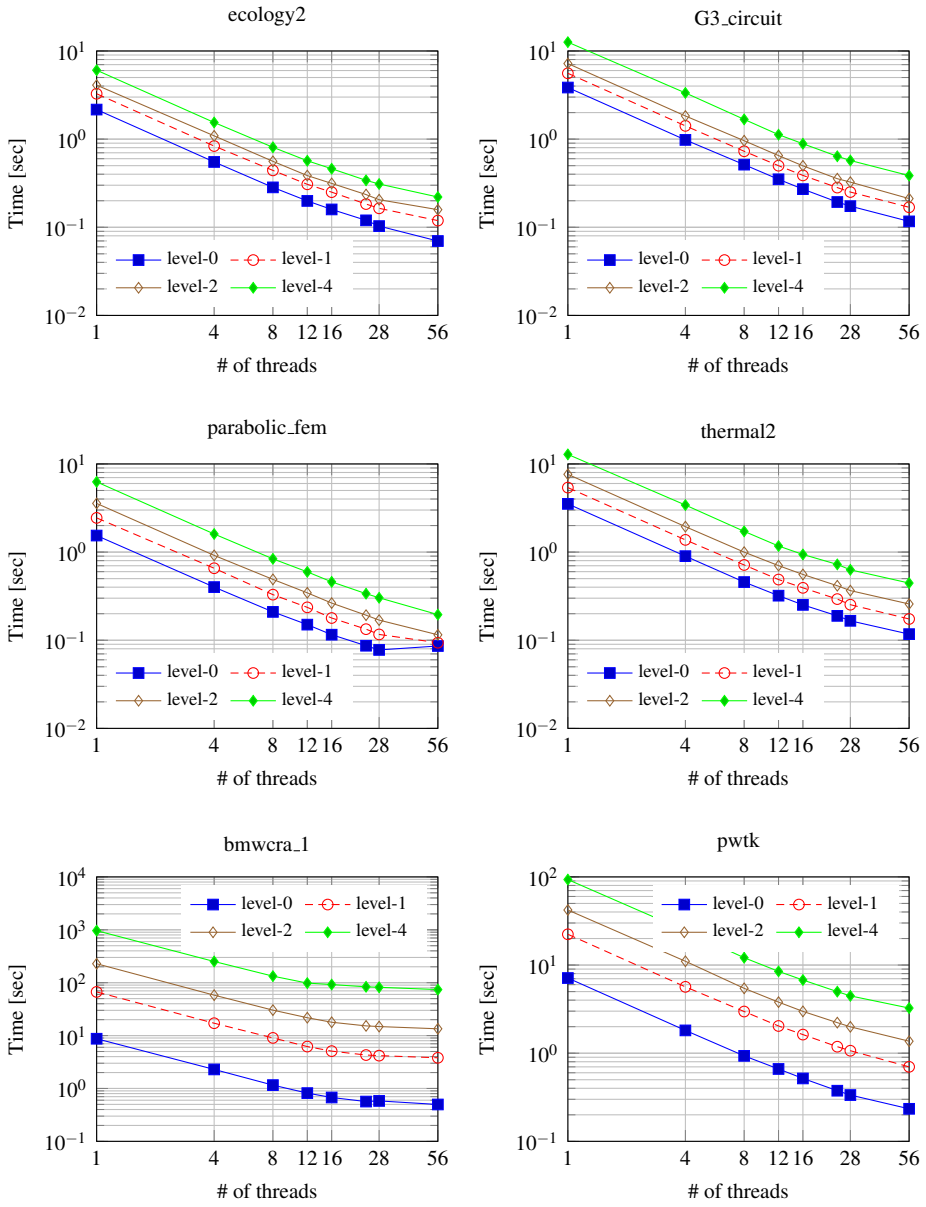


Figure 9: [Phi] Time for level(k) incomplete Cholesky-by-blocks factorization with the Kokkos Pthreads backend (our method).

Bridge multicore processor and the Xeon Phi manycore coprocessor. For the single-threaded case, this measure indicates that if the time ratio becomes close to one, our Cholesky-by-blocks runs with relatively small tasking overhead compared to the serial algorithm. The Cholesky-

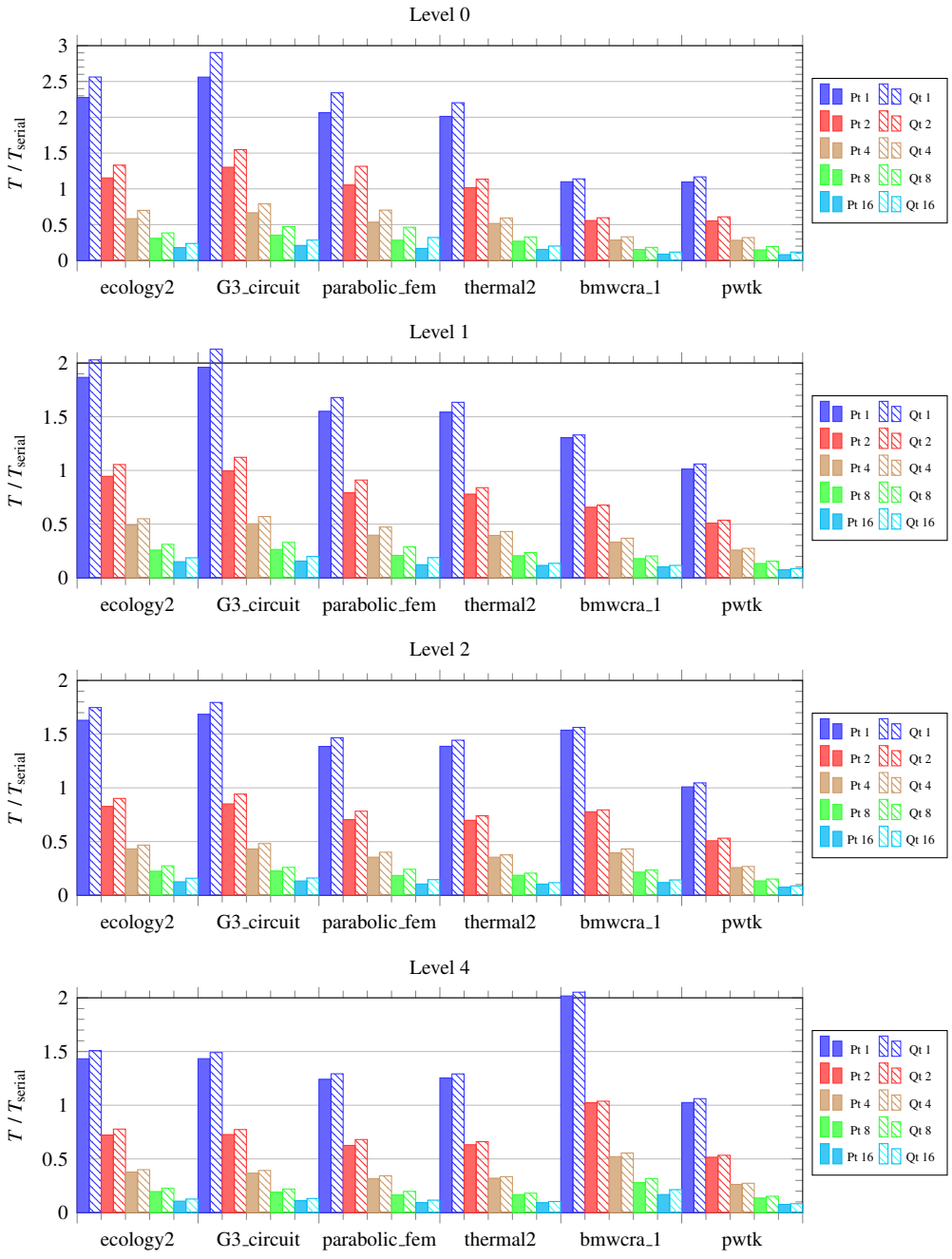


Figure 10: [Sandybridge] Time ratio between threaded incomplete Cholesky-by-blocks and serial version of incomplete Cholesky factorization.

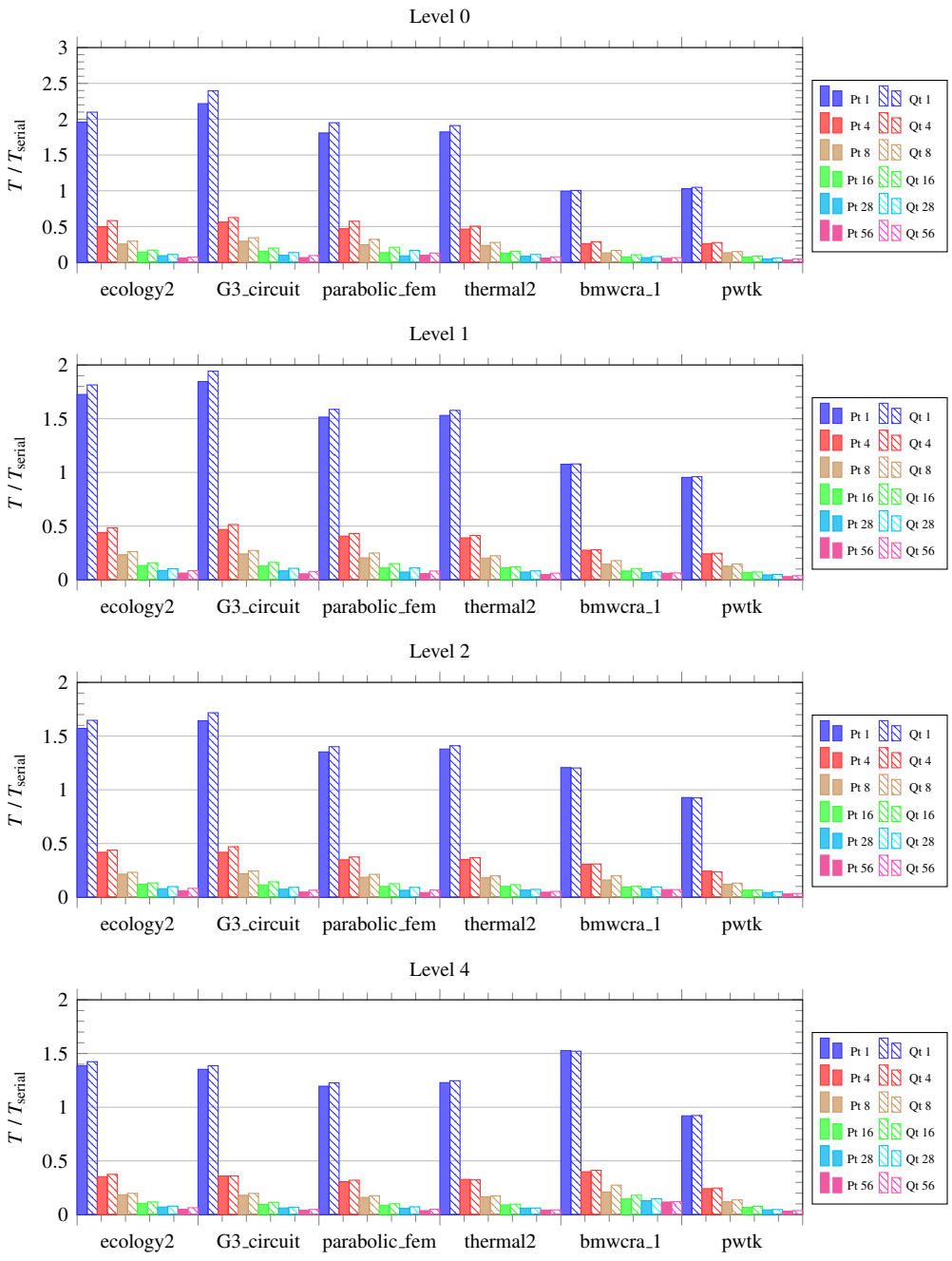


Figure 11: [Phi] Time ratio between threaded incomplete Cholesky-by-blocks and serial version of incomplete Cholesky factorization.

by-blocks factorization may include two different types of overhead: 1) task scheduling overhead that increases proportionally with the number of generated tasks and 2) overhead due to irregular data access during the asynchronous task execution. The tasking overhead from our Cholesky-by-blocks can be amortized by overlapping it during asynchronous task execution.

From the figures, those relatively less sparse matrices such as `bmwcra_1` and `pwtk` shows different performance trends from the others. For convenience, we use `ecology2` to represent the other sparse matrices and use `pwtk` to represent the less sparse matrices. Some key observations are:

- `ecology2` exhibits higher tasking overhead than `pwtk` due to its lower computational workload in each task (for the level 0 factorization, `ecology2` matrix carries almost the same amount of overhead as the numerical factorization while the tasking overhead in `pwtk` is almost negligible);
- with an increasing level of fill, relative tasking overhead of both test problems decreases as the workload associated with each task increases;
- the overhead is problem-specific; for example, the overhead of irregular data access patterns is more dominant for `bmwcra_1`, which may results in the increasing overhead with the factorization level.

The other test problems demonstrate similar performance behaviors to these two representative cases.

As depicted in Fig. 11, similar performance trends are observed on the Xeon Phi manycore coprocessor. However, the results are quantitatively very distinct from those obtained on the Sandy Bridge multicore architecture. The relative time ratio of the single-threaded Cholesky-by-blocks on the `pwtk` problem is smaller than one with an increased level of fill, which implies that the serial factorization that does not carries tasking overhead is slower than the single-threaded Cholesky-by-blocks. This counter-intuitive result is probably due to cache effects. Since `pwtk` is considerably less sparse than `ecology2`, the serial algorithm on this matrix can incur more cache misses as the effective working set size becomes the entire matrix. On the other hand, our Cholesky-by-blocks processes the factorization in terms of block computations. By doing so, we can effectively reduce cache misses and improve factorization performance similar to the BLAS level 3 operations in DLA. Consequently, the algorithm designed for task parallelism is also beneficial for modern manycore architectures by restricting the computation within a block. Also, the performance of the serial algorithm on the multicore architecture is less influenced by the increased amount of nonzeros because of the large shared L3 cache (20 MB).

5 Related work

We summarize other task-parallel implementations of sparse direct and incomplete factorizations as well as other task-parallel models and run time systems.

Task-parallel sparse factorization. Task-parallel sparse factorization has been implemented mostly along with multifrontal algorithms, as the algorithm is naturally parallelized using an elimination tree. This tree-level parallelism can be easily implemented using tasking APIs.

However, the tree-level parallelism decreases near the root of the tree. To remedy this inefficiency, nested parallelism within supernodal blocks is implemented for sparse multifrontal Cholesky [23, 26], LU [28] and QR [10, 16] factorizations. For an iterative method, ILUPACK [2] uses a runtime task scheduler, `OmpSs`, for Preconditioned Conjugate Gradient (PCG) algorithms. Their approach is similar to ours in that the parallelism is extracted from the ND ordering. However, the code uses a data-flow programming model, tasks are created using an elimination tree, and depedences are made through a row (contiguous memory region). Hence, their parallel tasks are very fine-grained task operations such as DOT and AXPY. By contrast, we use `future` as a task handle and dependences are made among future references associated with 2D partitioned-blocks (non contiguous memory region). Correct task dependences are derived from algorithms-by-blocks and tasks are generated separately from the elimination tree, enabling a more flexible tasking algorithm. With its 2D partitioned-block layout, our task-parallel Cholesky factorization exploits efficient sparse operations *i.e.*, CHOL, TRSM, HERK and GEMM, which are analogous to BLAS level 3 operations in DLA libraries.

Task-parallel models and run time systems. The need to exploit ever-increasing parallelism on emerging multicore and manycore architectures has motivated the development of numerous task-parallel languages, libraries, and run time systems. Our tasking model developed in the Kokkos framework supports both futures and dependences, allowing a large space of possible task DAGs, and its API and implementation use standard C++ with no specialized compiler support needed. By comparison, Cilk [19] and its successor Intel Cilk Plus [3] support only strict fork-join task DAGs with no dependences or futures, though an extension for arbitrary dependences has been explored [1]. OpenMP 4.0 [31] supports dependences between tasks but does not support futures. Cilk, Cilk Plus, and OpenMP all require language extensions, and thus, special compiler support. Beginning in version 4.0, Intel Threading Building Blocks (TBB) [4] includes a flow graph interface to represent "functional nodes" and edges between them. StarPU [4] is a C-based task-parallel framework for heterogeneous node architectures with tagged dependences that requires extensions to the C language through a GCC plug-in. High Performance ParalleX (HPX) [27] supports futures across distributed memory machines, as do the Chapel [13] language and Java-based X10 [15], and the related Habanero C and Habanero Java [12]. Although we use Kokkos to implement our algorithm, the techniques we use could be ported to these or similar programming models with support for futures and dependence-driven execution.

For DLA, several research projects have developed domain-specific runtime task schedulers: QUARK [40] and SuperMatrix [14] for shared memory architectures; DPLASMA [6] and PaRSEC [7] for distributed memory architectures. Recently, a distributed task-parallel Cholesky implementation [30] has been demonstrated using the SMPSs [33] programming model.

6 Conclusion

We have presented a novel algorithm for task-parallel incomplete Cholesky factorization that applies algorithms-by-blocks factorization to a 2D block matrix. We have shown that by en-

[3] https://www.cilkplus.org

[4] https://www.threadingbuildingblocks.org

coding the tree hierarchy in the 2D block matrix, the task DAG need not be restricted to a simple ND tree. This results in a much richer task DAG, leading to better performance. We believe this algorithm opens up a new direction of research in which other sparse factorizations such as LU and QR could also gain performance benefits by following the same pattern used here. We have also designed a simple tasking API and modified an open source library to support task parallelism with performance portable abstractions for heterogeneous computing devices using different backend libraries. While used for incomplete Cholesky here, these changes are much more general and we believe they will be useful to develop other task-parallel codes. We have also shown the performance of a task-parallel Pthreads-based backend with the incomplete Cholesky factorization as its driver. Our factorization has demonstrated robust parallel performance with several test problems both on Intel Xeon multicore and Intel Xeon Phi manycore architectures. We also evaluated tasking overhead associated with different task granularities and showed how the overhead costs impact parallel performance. As has been observed before, generating an excessive amount of tasks is prohibitive due to its tasking overhead, but we need a large enough number of fine-grained tasks order to effectively exploit manycore environments. We plan to remedy this granularity problem by exploiting data parallelism within the tasks in the future.

References

[1] K. Agrawal, C.E. Leiserson, and J. Sukha. Executing task graphs using work-stealing. In Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on, pages 1–12, April 2010.

[2] J I Aliaga, R M Badía, M Barreda, M Bollhöfer, and E S Quintana-Ortí. Leveraging Task-Parallelism with OmpSs in ILUPACK's Preconditioned CG Method. In 26th IEEE Intl. Symp. on Computer Architecture and High Performance Computing, SBAC-PAD '14, pages 262–269. IEEE, 2014.

[3] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. ACM SIGARCH Computer Architecture News, 18(3b):1–6, 1990.

[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-Andre\'{e} Wacrenier. STARPU: A unified platform for task scheduling on heterogeneous multicore architectures. Concurrency Computat. Pract. Exper., 23(2):187–198, 2011.

[5] Erik G Boman, Karen D Devine, and Sivasankaran Rajamanickam. Scalable matrix computations on large scale-free graphs using 2d graph partitioning. In Intl. Conf. on High Performance Computing, Networking, Storage and Analysis, page 50. ACM, 2013.

[6] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asim YarKhan, and Jack Dongarra. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. 2001 IEEE Intl. Parallel and Distributed Processing Symp. Workshops, pages 1432–1441, 2011.

[7] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Herault, and Jack J. Dongarra. PaRSEC: Exploiting heterogeneity to enhance scalability. Computing in Science and Engineering, 15(6):36–45, 2013.

[8] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. In 18th Euromicro Conf. on Parallel, Distributed and Network-Based Processing, pages 180–186, 2010.

[9] Aydin Buluc and John R Gilbert. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. SIAM J. Sci. Comput., 34(4):C170–C191, 2012.

[10] Alfredo Buttari. Fine-grained multithreading for the multifrontal QR factorization of sparse matrices. SIAM J. Sci. Comput., 35(4):323–345, 2013.

[11] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. Parallel Computing, 35(1):38–53, 2009.

[12] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-java: The new adventures of old X10. In Proc. 9th Intl. Conf. on Principles and Practice of Programming in Java, PPPJ '11, pages 51–61. ACM, 2011.

[13] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. Int. J. High Perform. Comput. Appl., 21(3):291–312, August 2007.

[14] Ernie Chan, Field G. van Zee, Enrique S. Quintana-Orti, Gregorio Quintana-Orti, and Robert A. van de Geijn. Satisfying your dependencies with Supermatrix. In 2007 Intl. Conf. on Cluster Computing, pages 91–99. IEEE, 2007.

[15] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In Proc. ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '05, pages 519–538. ACM, 2005.

[16] Timothy A Davis. Algorithm 915, SuiteSparseQR. ACM Trans. Math. Softw., 38(1):1–22, 2011.

[17] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. ACM Trans. Math. Softw., 38(1):1–25, 2011.

[18] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling many-core performance portability through polymorphic memory access patterns. J. Parallel. Distrib. Comput., 74(12):3202 – 3216, 2014.

[19] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. Implementation of the Cilk-5 multithreaded language. In 1998 ACM Conf. on Programming Language Design and Implementation, pages 212–223, June 1998.

[20] Alan George. Nested Dissection of a Regular Finite Element Mesh. SIAM J. Numer. Anal., 10(2):345–363, 1973.

[21] Brian C. Gunter and Robert A. van de Geijn. Parallel Out-of-Core Computation and Updating the QR Factorization. ACM Trans. Math. Soft., 31(1):60–78, March 2005.

[22] Fred G. Gustavson, Isak Jonsson, Bo Kågström, and Per Ling. Towards peak performance on hierarchical SMP memory architectures - new recursive blocked data formats and BLAS. In Parallel Processing for Scientific Computing, pages 1–4, 1999.

[23] J. D. Hogg, J. K. Reid, and J. A. Scott. Design of a multicore sparse Cholesky factorization using DAGs. SIAM J. Sci. Comput., 32(6):3627–3649, 2010.

[24] David Hysom and Alex Pothen. A Scalable Parallel Algorithm for Incomplete Factor Preconditioning. SIAM J. Sci. Comput., 22(6):2194–2215, 2001.

[25] David Hysom and Alex Pothen. Level-based incomplete LU factorization: Graph model and algorithms. Technical Report UCRL-JC-150789, Lawrence Livermore National Laboratory, 2002.

[26] Dror Irony, Gil Shklarski, and Sivan Toledo. Parallel and fully recursive multifrontal sparse Cholesky. Future Generation Computer Systems, 20(3):425–440, April 2004.

[27] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. HPX: A task based programming model in a global address space. In 8th Intl. Conf. on Partitioned Global Address Space Programming Models, PGAS '14, pages 6:1–6:11. ACM, 2014.

[28] Kyungjoo Kim and Victor Eijkhout. A Parallel Sparse Direct Solver via Hierarchical DAG Scheduling. ACM Trans. Math. Softw., 41(1):3:1–27, 2014.

[29] Tze Meng Low and Robert A. van de Geijn. An API for manipulating matrices stored by blocks. Technical report, FLAME Working Note 12, TR-2004-15, The University of Texas at Austin, 2004.

[30] Alberto F. Martín, Ruymán Reyes, Rosa M. Badia, and Enrique S. Quintana-Ortí. Leveraging task-parallelism in message-passing dense matrix factorizations using SMPSs. Parallel Computing, 40(5-6):113–128, 2014.

[31] OpenMP ARB. OpenMP API specification, v. 4.0, July 2013.

[32] Francois Pellegrini. Scotch and libScotch 6.0 User's Guide. Technical report, Universite Bordeaux I, 2012.

[33] Josep M Perez, Rosa M Badia, and Jesus Labarta. A Dependency-Aware Task-Based Programming Environment for Multi-Core Architectures. In IEEE Intl. Conf. on Cluster Computing, pages 142–151, 2008.

[34] Enrique Quintana-Ortí, Gregorio Quintana-Ortí, Xiaobai Sun, and Robert van de Geijn. A note on parallel matrix inversion. SIAM J. Sci. Comput., 22(5):1762–1771, 2001.

[35] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert A. van de Geijn, Field G. van Zee, and Ernie Chan. Programming matrix algorithms-by-blocks for thread-level Parallelism. ACM Trans. Math. Softw., 36(3):1–26, July 2009.

[36] Edward Rothberg and Anoop Gupta. An efficient block-oriented approach to parallel sparse cholesky factorization. SIAM J. Sci. Comput., 15(6):1413–1439, November 1994.

[37] Vinod Valsalam and Anthony Skjellum. A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. Concurrency Computat. Pract. Exper., 14(10):805–839, August 2002.

[38] Robert A. van de Geijn and Enrique S. Quintana-Ortí. The Science of Programming Matrix Computations. www.lulu.com, 2008.

[39] K.B. Wheeler, R.C. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads. In 2008 IEEE Intl. Parallel and Distributed Processing Symp. Workshops, pages 1–8, April 2008.

[40] Asim Yarkhan, Jakub Kurzak, and Jack Dongarra. QUARK Users' Guide. Technical report, EECS, University of Tenessee, 2011.