TACC Technical Report TR-12-05

A Parallel Sparse Direct Solver via Hierarchical DAG Scheduling

Kyungjoo Kim* Victor Eijkhout*

October 2, 2012

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.

Permission to copy this report is granted for electronic viewing and single-copy printing. Permissible uses are research and browsing. Specifically prohibited are *sales* of any copy, whether electronic or hardcopy, for any purpose. Also prohibited is copying, excerpting or extensive quoting of any report in another work without the written permission of one of the report's authors.

The University of Texas at Austin and the Texas Advanced Computing Center make no warranty, express or implied, nor assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed.

* The University of Texas at Austin, Austin, TX 78712

Abstract

We present a scalable parallel sparse direct solver for multi-core architectures based on Directed Acyclic Graph (DAG) scheduling. Recently, DAG scheduling has become popular in advanced Dense Linear Algebra libraries due to its efficient asynchronous parallel execution of tasks. However, its application to sparse matrix problems is more challenging as it has to deal with an enormous number of highly irregular tasks. This typically results in substantial scheduling overhead both in time and space and causes overall parallel performance to be suboptimal. The described parallel solver is based on the *multi-frontal* method exploiting two-level parallelism: coarse grain task parallelism is extracted from the assembly tree, then those tasks are further refined in the matrix-level computation using *algorithms-by-blocks*. Resulting fine grain tasks are asynchronously executed after their dependencies are analyzed. Our approach is distinct from others in that we adopt a two-level scheduling strategy to mirror the two-level parallelism. As a result we reduce scheduling overhead, and increase efficiency and flexibility. The proposed parallel sparse direct solver is evaluated for the particular problems arising from the *hp*-Finite Element Method where conventional sparse direct solvers do not scale well.

Keywords

Gaussian elimination, Directed Acyclic Graph, Direct method, LU, Multi-core, Multi-frontal, OpenMP, Sparse matrix, Supernodes, Task parallelism, Unassembled Hyper-Matrix

1 Introduction

Many scientific applications require a considerable amount of time to solve a linear system of equations $Ax = b$, where A is usually a large and sparse matrix. Large sparse systems can be solved by either direct or iterative methods. The major disadvantage of iterative methods is that the solution may not converge: usually the success of iterative methods depends on the construction of good preconditioners that boost the convergence rate. On the other hand, direct methods based on Gaussian elimination are robust but expensive. Required memory for the solution in 2D problems is $O(NlogN)$ and 3D problems increases the space complexity $O(N^{5/3})$ when a matrix is permuted by nested dissection ordering [23, 25]. The performance of sparse direct methods generally varies according to the sparsity of problems [27]. No single approach is the best in solving all types of sparse matrices. The approach selected is based on characteristics of the sparsity pattern such as banded, (un)symmetric, and/or (un)structured. Thus, a sparse direct solver should be customized to the type of sparse matrices derived from a class of numerical algorithms. In our previous work [10], a new sparse direct solver using Unassembled HyperMatrices (UHMs) was designed and developed for the Finite Element Method (FEM) with hp-mesh refinements. In the adaptive context, the solver effectively uses the application information in solving a sequence of linear systems that are locally updated, and the solver stores partial factors previously computed and exploits them to factorize the current sparse system. In this paper, extending our previous work, we present a fully asynchronous parallel sparse direct solver targeting large-scale sparse matrices associated with the hp-FEM that range from 100k to a few million unknowns [1].

There are two important aspects to hp-adaptive FEM matrices that we use in our solver. First of all, the structure of the factorization is inherit from the refinement nature of the problem. Our overall factorization approach is similar to the multi-frontal method [22, 28, 29, 35]: we recursively find multiple parallel entry points into the factorization and eliminate them. Unlike traditional multi-frontal factorizations, we do not need to discover this parallelism from a matrix (or finite element) structure, but we derive it from the refinement history of the mesh, giving us an *assembly tree* that also becomes the factorization tree (see our paper [10] for more details).

Secondly, matrices from hp-adaptive problems have dense subblocks – and this is more so in the unassembled case – implying that there are essentially no scalar operations: all operations are dense, which allows highly efficient Basic Linear Algebra Subprogram (BLAS) level 3 functions [21] to be utilized.

Since we target our solver to a multicore and shared memory model, we develop a so-

1. The advanced hp-FEM typically provides an order of magnitude higher solution resolution than conventional linear FEM by using both variable element size (h) and higher order of approximation (p).

lution based on Directed Acyclic Graph (DAG) scheduling. DAG scheduling in Dense Linear Algebra (DLA) has been developed over the last decade [12, 15, 39]. Unlike the classic *fork-join* parallel model, this approach adopts asynchronous parallel task scheduling using the DAG of tasks, where nodes stand for tasks and edges indicate dependencies among them.

While these methods have been successful in the context of dense matrices, the application of DAG scheduling to sparse matrices is not trivial for the following reasons:

1. There are two types of parallelism: task parallelism that decreases as the factorization progresses, and finer grained parallelism in handling the dense blocks that increases accordingly.
2. The overall factorization has a large number of tasks, which increases scheduling overhead.
3. Tasks are inherently irregular, owing to the sparse matrix structure, the *hp*-adaptivity, and the growing block size during the factorization.
4. Numerical pivoting of sparse matrices may create additional *fills* and cause dynamic changes to the workflow. Consequently, *out-of-order* scheduling based on a DAG becomes less efficient.

Possibly due to the above difficulties, we have found only one case which uses the DAG scheduling in sparse Cholesky factorization [32]. To the best of our knowledge, no implementation based on DAG scheduling exists for sparse LU factorization with pivoting.

In our scheme, irregular coarse grain tasks are decomposed into regular fine-grain tasks using *algorithms-by-blocks*. Refined tasks are scheduled in a fully asynchronous manner via *multiple* DAG schedulers. A key aspect of this solver is that a DAG scheduler locally analyzes a series of block operations associated with a set of dense matrices. The DAG scheduler developed here and its interface have the following features that are distinct from other advanced DLA libraries such as SuperMatrix [15] and Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) [1]:

- *OpenMP framework.* Our DAG scheduling is implemented based on the OpenMP framework. Users can create and insert their own tasks directly using OpenMP environments or our tasking objects. PLASMA also allows user-defined tasks through QUeuing And Runtime for Kernels (QUARK) [44]. However, users need to redesign their codes for adoption in QUARK environments. OpenMP is considered the *de facto* standard in parallel processing on the shared-memory processor. OpenMP, using high-level compiler directives, has superior portability and programmability.
- *Storage format.* The proposed DAG scheduler uses the traditional column-major matrix format while both SuperMatrix and PLASMA use a locally packed stor-

age scheme called *storage-by-blocks*. Compared to storage-by-blocks, the column-major format may incur more cache misses in computation [30]. On the other hand, storage-by-blocks generally increases overhead in the assembly procedure as the entries of a matrix need to be hierarchically indexed. In addition, matrix repacking overhead occurs when a solver interfaces to an application which does not use the same blockwise matrix format. Later, we argue that the benefit of the blockwise storage is low compared to this repacking cost.

- *Task scheduler object.* While both SuperMatrix and PLASMA are designed as an infrastructure, which has a global scope to wrap a parallel region, our DAG scheduler is designed as a local object. Users can create a scheduler object, and select tasks to be associated with a specific scheduler. This feature enables nested DAG scheduling; in effect we are *scheduling schedulers*. For multiple dense problems, multiple schedulers can be used for different sets of independent dense problems. Although tasks are separately analyzed, they are asynchronously executed together through the unified tasking environment in OpenMP.

Together, these insights advance the state-of-the art.

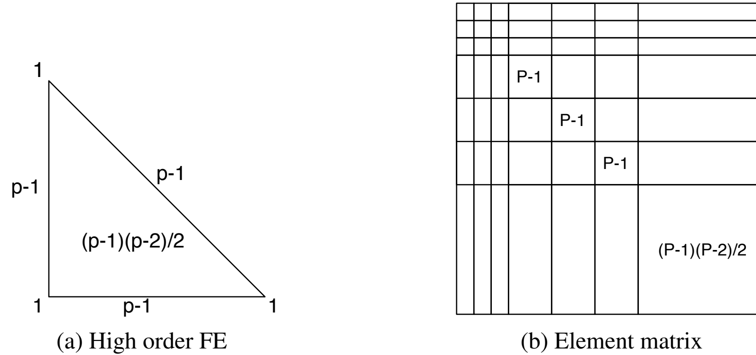
The paper is organized as follows. In Section 2, we discuss characteristic sparse patterns created from *hp*-FEM. Section 3 explains the basic methods adopted to implement our parallel solver. Section 4 explains the design and components of the proposed parallel sparse direct solver. In Section 5, the solver interfaces with *hp*-FEM and is evaluated against other state-of-the-art parallel sparse direct solvers. Finally, we summarize achievements and future works in Section 7.

2 Sparse Matrices from the *hp*-Finite Element Method

The FEM is widely used for solving engineering and scientific problems. The method approximates solutions using piecewise polynomial basis functions in a discrete domain which creates a large sparse system of equations. The system of equations must be solved, and the numerical cost is a trade-off with the quality of the solution. For an efficient solution process it is essential to understand the characteristics of the sparse system.

The advanced *hp*-FEM uses an adaptive strategy to deliver highly accurate solutions while keeping the cost low. In *hp*-FEM, the adaptive procedure controls both mesh size (h) and polynomial order of approximation (p). A large body of mathematical literature [8, 9, 42] on the theoretical analysis of *hp*-FEM proves its superior performance compared to the conventional linear FEM or with fixed p.

To solve problems formulated with hp-FEM, a direct method is often preferred to iterative methods. The difficulty with iterative methods is an increasing condition number with the approximation order [7, 13], which leads to a slow convergence or failure to find solutions.



(a) High order FE (b) Element matrix

Figure 1: The left figure shows the number of DOFs related to topological nodes with a polynomial order p. The figure on the right illustrates the shape of an unassembled element matrix corresponding to the finite element depicted on the left.

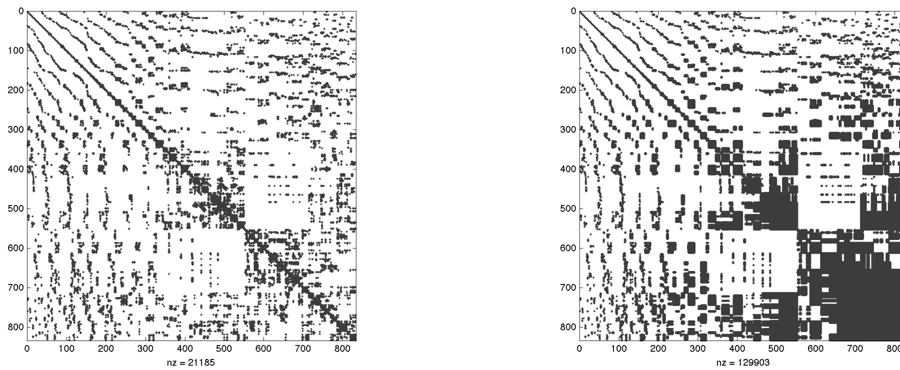
Compared to other linear methods, hp-discretization creates a more complex sparsity pattern due to the variable order of approximation. In hp-discretization, multiple DOFs, corresponding to the polynomial order, are associated with the same topological node, as illustrated in Fig. 1a. The number of DOFs increases with the order of approximation [2] as follows:

	Edge	Face	Volume
# of DOF	$O(p)$	$O(p^2)$	$O(p^3)$

Thus, the sparse system derived from hp-discretization can be characterized in terms of topological nodes and associated DOFs rather than individual DOFs. However, a general purpose sparse direct solver does not recognize the presence of such a structure. In our solver, the hp-discretization is directly associated with a weighted graph structure. The connectivity of element matrices can be illustrated by blocks, as depicted in Fig. 1b. Mirroring the hp-mesh leads to a more efficient solver because it significantly reduces the size of the graph that represents the sparse system, as well as giving increased reliance on level 3 BLAS operations.

By assembling element matrices, a global sparse system can be formed; a typical sparse matrix created by hp-FEM is shown in Fig. 2. Our solver is based on keeping element

2. In general, the order of approximation selected by hp-FEM is 3-4 for 2D and 5-6 for 3D problems.



(a) A sparse pattern based on *hp*-discretization (b) Factors

Figure 2: The left figure shows a sparse matrix based on *hp*-discretization, which is ordered by nested dissection. The figure on the right includes additional fill resulting from the factorization.

matrices unassembled as long as possible during the factorization. Since element matrices are characterized by (almost) dense blocks, this leads to a very efficient factorization; see our earlier work [10]. This is somewhat similar to existing work on *supernodes* [6, 17, 19, 28]; it differs in that supernodes are not discovered in our solver, but rather given *a priori*.

3 Factorization Algorithms

A general procedure for direct methods consists of four phases: ordering, analysis, factorization, and forward/backward substitution. In the first phase, a fill-reducing order of the sparse matrix is constructed. Next, a symbolic factorization is performed in the analysis phase to determine workspaces and supernodes. The heaviest workload is encountered within the numerical factorization. Once factors are computed, the solution is obtained by forward/backward substitution.

Our factorization adheres to the same schema, but several aspects differ from the general case because of our application area. For an overview of existing algorithms and literature, we refer to the book [16]. In the next two sections, we first consider the ordering stage, which is fairly simple in *hp*-FEM context: we can construct a nested dissection ordering based on the refinement tree. Then, we focus on various aspects of the numerical factorization stage, where again we benefit from our application context.

3.1 Ordering strategy

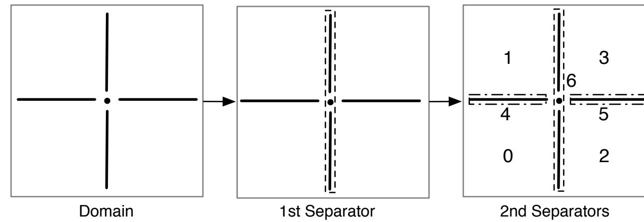


Figure 3: A nested dissection ordering is applied to a 2×2 quadrilateral mesh. Nodes are divided and numbered by "separators".

Typically, the factorization order of the unknowns is determined through graph analysis. In our application we derive it from the way the matrix is constructed hierarchically. This hierarchy is either the refinement history, or an *a posteriori* reconstruction from the *hp*-mesh.

Let a given sparse matrix A be associated with the mesh depicted in Fig. 3. The figure describes how a sequence of nested dissections is applied to the domain. For convenience, a set of nodes is amalgamated by a separator, and numbered. Factorization can

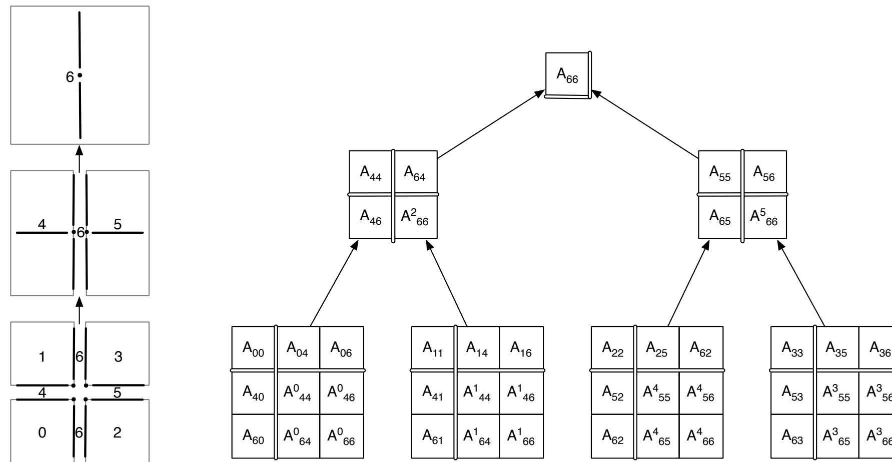


Figure 4: An assembly tree can be organized by coarsening a mesh. The superscript of blocks represents temporary storage for Schur complements computed in each UHM.

proceed either within an assembled form or alternatively with unassembled matrices. In *hp*-FEM, storing unassembled matrices is advantageous in the adaptive context because this format provides a potential opportunity to reuse partial factors previously

computed for the solutions of a new updated system. For notation consistent with our previous work [10], we denote the unassembled frontal matrices as UHMs.

The ordering shown in this example can decouple the system into several subsystems that are hierarchically related, as depicted for a simple examples in Fig. 4. The UHM solver organizes unassembled matrices as a tree structure, which can be derived from either graph analysis or mesh refinements. The factorization is then a recursive application of the following cycle:

- We can eliminate interior nodes within UHMs, giving a partial elimination.
- Joining together such partially eliminated elements, the nodes on interior boundaries become fully assembled interior nodes.

This procedure is recursively driven by post-order tree traversal; slightly more formally we describe this recursion as follows:

1. *Update.* A UHM assembles its children.

$$A := merge\left(A_{BR}^{left}, A_{BR}^{right}\right)$$

   where A_{BR}^{left} and A_{BR}^{right} represent the Schur complements from children. Leaf-level UHMs do not need this step.

2. *Partition of UHM.* The UHM is partitioned into four blocks, where A_{TL} contains the fully assembled nodes.

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

3. *Partial factorization.* We compute the block LU factorization of the partitioned matrix A as shown below, where L_{TL} and L_{BR} are lower triangular matrices with unit diagonal entries, and U_{TL} and U_{BR} are upper triangular matrices.

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right)\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right)$$

   Interior nodes are eliminated by a standard right-looking LU algorithm computing factors L_{TL} and U_{TL}. Respectively, interface matrices L_{BL} and U_{TR} are computed and overwritten on $\hat{A}_{BL}$ and $\hat{A}_{TR}$. Next, the Schur complement $\hat{A}_{BR} = A_{BR} - L_{BL}U_{TR}$ is computed for the following update to its parent.

$$\left(\begin{array}{c|c} \hat{A}_{TL} & \hat{A}_{TR} \\ \hline \hat{A}_{BL} & \hat{A}_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c} L_{TL}U_{TL} & U_{TR} := L_{TL}^{-1}A_{TR} \\ \hline L_{BL} := A_{BL}U_{TL}^{-1} & L_{BR}U_{BR} := A_{BR} - L_{BL}U_{TR} \end{array}\right)$$

   The recursive factorization finishes at the root, where A_{BR} is an empty matrix.

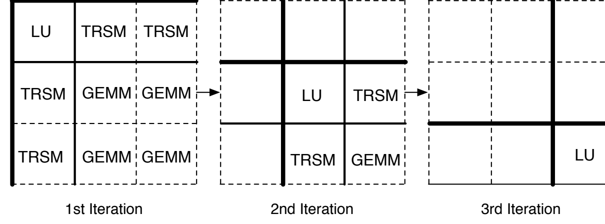


Figure 5: Block LU factorization without pivoting is applied to a 3×3 block matrix, and the corresponding workflow produces a series of fine-grain tasks.

3.2 Algorithms-by-blocks

The multi-frontal method has a natural parallelism from its recursive post-order tree traversal; tasks on separate branches can be processed simultaneously. However, closer to the root, task parallelism decreases, while the blocks to be factored grow. Thus, further parallelism in processing these dense blocks is essential to improve the efficiency of the overall algorithm.

For processing these blocks, we use so-called *algorithms-by-blocks* [39], which reformulate DLA algorithms in terms of blocks. Consider for example the partial factorization described above. The factorization algorithm is presented in terms of blocks, but only two independent tasks are available:

$$\begin{array}{ccl} & A_{TL} := \{L \backslash U\}_{TL} & \text{LU} \\ & \downarrow & \\ A_{BL} := L_{TL}^{-1} A_{BL} & A_{TR} := A_{TR} U_{TL}^{-1} & \text{TRSM} \\ & \downarrow & \\ & A_{BR} := A_{BR} - A_{TR} A_{BL} & \text{GEMM} \end{array}$$

Further task-level parallelism can be pursued by organizing a matrix by blocks (submatrices). For instance, consider a matrix A_{TL} with 3×3 blocks, where each block A_{ij} has conforming dimensions with adjacent blocks:

$$A_{TL} = \begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix}$$

The LU factorization can be reformulated as an algorithm-by-blocks changing the unit of data from a scalar to a block. A number of tasks are identified from the resulting workflow. For example, Fig. 5 describes a block LU factorization without pivoting. In the first iteration of the algorithm, four independent TRSM and four independent GEMM tasks are created after the unblocked LU factorization is performed on the first

diagonal block. The algorithm generates tasks by repeating this process. In the same way, coarse-grain TRSM and GEMM tasks encountered within the elimination tree are decomposed into fine-grain tasks. These fine-grain tasks are mostly regular and are related through input/output dependencies. After their dependencies are analyzed, tasks are scheduled asynchronously, which leads to highly efficient task parallelism on modern multi-core architectures. This has been explored in the past for parallelizing a sequential workflow of dense matrices [12, 39]. For the case of SuperMatrix [15], two separate task queues (*i.e.*, a task queue and a waiting queue) are used to store enqueued tasks and tasks ready to execute.

1. A task in the waiting queue is executed on an idle thread selected by a data (thread) affinity policy.
2. After the task is completed, task dependencies are updated. Then, tasks for which dependencies are satisfied are moved into the waiting queue.

This procedure repeated until the task queue is empty. In the next section, we explore how to incorporate such algorithms in the context of a sparse solver.

4 A Fully Asynchronous Parallel Sparse Direct Solver

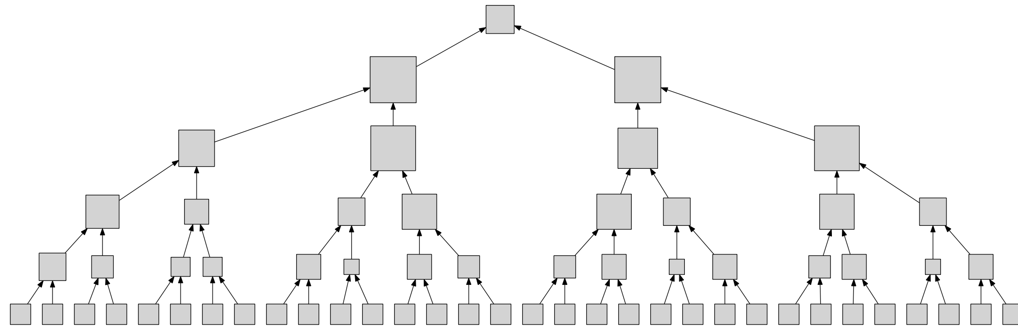


Figure 6: Macro tasks are created by recursive multi-frontal factorization on the assembly tree. Tasks are irregular and hierarchically related.

In the course of a sparse factorization, as depicted in Fig. 6, we have two oppositely behaving types of parallelism: on the one hand a decreasing amount of task parallelism as the factorization progresses from the leaves to the root of the tree; on the other hand, an increasing opportunity for parallelism inside the blocks as their sizes grow. The question is how to exploit the two-level parallelism in harmony to extract the near-optimal computing power from multi-core architectures avoiding excessive complexity in the implementation.

4.1 The limits of existing approaches

There are various ways of solving a linear system with the sparse matrix of an *hp*-adaptive problem that rely on, or at least leverage, existing software [18]. First of all, we note that existing sparse direct solvers do not do a good job of finding supernodes, so they are suboptimal on our type of matrices.

Next, we recognize that the two-level factorization leads to a large number of tasks, which we could schedule through a package like SuperMatrix or QUARK. The problem here is that the number of tasks is very large. Scheduling them in full would lead to large scheduling overhead, and the use of a window would make the scheduler less efficient. A further objection is that the task list is dynamic because of numerical pivoting, and these packages can not yet deal with that.

We could also use an approach that recognizes the two-level structure, such as using multi-threaded BLAS or a DAG scheduler such as QUARK for the nodes, coupled with a simple Breadth First Search (BFS) tree traversal. This approach suffers from imperfect load balance, for instance because each node in the graph has to be assigned to a fixed number of cores [24, 2], which can then not participate in processing other nodes. Also, completion of a node subtask is typically a synchronization point, diminishing the parallel efficiency.

For these reasons we advocate an approach where a single task list is formed from the subtasks of all the nodes. Our approach does not suffer from excessive scheduling overhead, since we do not analyze global dependencies: we combine the natural post-order node ordering in the tree, with a dependency analysis on the subtasks from each tree node factorization. One might say we use our *a priori* knowledge of the factorization algorithm for the large scale scheduling, and only use a runtime scheduler where the algorithm does not dictate any particular ordering.

Furthermore, our approach can deal with pivoting and its dynamic insertions and deletions in the task queue. as will be discussed below.

4.2 Parallelization strategy

Our solver strategy is driven by the post-order traversal of the elimination tree: each node in the tree is a macro task that handles one UHM, and it can not proceed until its children have been similarly processed. Recursively, this defines the overall algorithm.

We illustrate a macro task in Fig. 7. It consists of a merge of the Schur complement of the children factorization, which carries an $O(n^2)$ cost, followed by level 3 BLAS operations for the factorization, all of which are $O(n^3)$ in the macro block size.

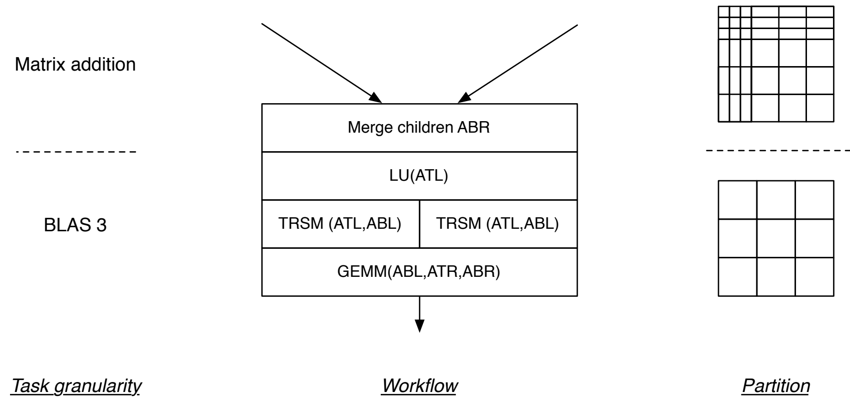


Figure 7: Internal workflow in a macro task associated with the partial factorization includes mixed task granularity supported by different block partitions.

The block algorithms in each macro task lead to fine-grain tasks related by a DAG; see Fig. 8. Rather than including the union of all fine-grain tasks in all blocks in a

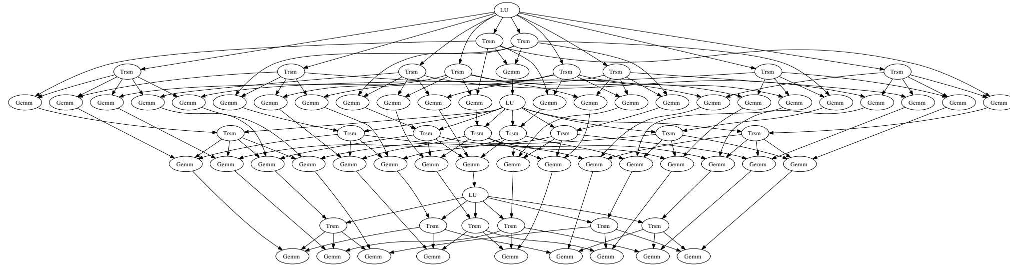


Figure 8: A DAG represents the relation of fine-grain tasks generated by block LU, TRSM, and GEMM which are applied to 3×3 block matrices.

single parallel region with a global DAG, we first schedule macro tasks related to the assembly tree in a BFS manner. Next, each group of fine-grain tasks, and the associated local DAG generated within a macro task, is scheduled in a Depth First Search (DFS) order. Since no global DAG is used, scheduling overhead is considerably reduced.

4.3 OpenMP tasking

We use OpenMP to schedule all the fine-grained tasks. However, we do not use an explicit data structure to reflect the schedule according to which tasks are to be executed. Instead, we rely on OpenMP mechanisms: by declaring tasks in the right execution order with OMP pragmas, they are entered into the internal OMP scheduler.

Note that we do not use the `omp parallel for` pragma around loops; rather, we sequentially let the loop generate OMP tasks. One advantage of this approach is that it prevents the creation and dismissal of thread pools around each loop. Another reason for not using OMP parallel loops is that they are not suited to nested parallelism.

The mechanisms we use are part of the explicit task management that was added to OpenMP 3.0, released in 2008 [37]. The new task scheme includes two compiler directives:

1. `#pragma omp task` creates a new task
2. `#pragma omp taskwait` is used to synchronize invoked (nested) tasks.

In this work, nested parallelism is supported by OpenMP tasking; a task can recursively create descendent tasks. Invoked tasks are scheduled based on a BFS order. When a task spawns descendent tasks, `#pragma omp taskwait` can suspend the task until those tasks are completed. For example, Fig. 9 outlines the parallel multi-frontal factorization through post-order tree traversal with OpenMP tasking.

For low-level thread binding, we rely on OpenMP primitives. OpenMP utilizes a thread pool to execute tasks; when a task is ready to execute, an idle thread picks it up to process the task.

4.4 Scheduling strategy of the block matrix operations

The post order scheduling of the UHM macro tasks is clearly recognized in Fig. 9, in particular in the structure of the function `post_order_macro_task`. We do not use the same mechanism for scheduling the fine-grained tasks, since a DFS strategy is more appropriate here. The reason for this is that a DFS strategy gives higher priority to tasks on the critical path.

However, DFS task scheduling is not supported by OpenMP native task scheduling. To realize DFS scheduling on a DAG, we designed a custom out-of-order task scheduler and implemented this using the OpenMP framework. For example, calls such as `create_gemm_task` in Fig. 9 add a task to a local scheduler queue (see Fig. 10) for the UHM macro task.

Details of the scheduler are given in Fig. 11; notably, the `execute` call causes the DAG nodes of the dense block to be declared in the proper order as OpenMP tasks. Since these local schedulers are scheduled in the proper post-order in Fig. 9, we find that all tasks are executed in the right global order with only local analysis.

```
// ** Sparse factorization via UHMs
int factorize_sparse_matrix(Tree::Node *root) {
  // begin with the root node
  post_order_macro_task(root, &factorize_uhm));
  return SUCCESS;
}

// ** Post-order tree traversal
int post_order_macro_task(Tree::Node *me, int (*op_func)(Tree::Node*)) {
  for (int i=0;i<me->get_n_children();++i) {
    // macro task generation for child tree-nodes
    #pragma omp task firstprivate(i)
    post_order_macro_task(me->get_child(i), op_func));
  }

  // BFS macro task scheduling
  #pragma omp taskwait

  // process the function (local scheduling)
  op_func(me);

  return SUCCESS;
}

// ** Partial factorization in UHM
int factorize_uhm(Tree::Node* nod) {
  // merge the Schur complements from child tree-nodes
  nod->merge();

  // Local DAG scheduling for LU factorization
  Scheduler s;

  // tasks are created using algorithms-by-blocks
  // and they are associated with a local scheduler
  create_lu_tasks(nod->ATL, s);
  create_trsm_tasks(nod->ATL, nod->ABL, s);
  create_trsm_tasks(nod->ATL, nod->ATR, s);
  create_gemm_tasks(nod->ABL, nod->ATR, nod->ABR, s);

  // parallel execution of tasks in a DFS-like manner
  s.flush();

  return SUCCESS;
}
```

Figure 9: Macro tasks are generated through post-order tree traversal within the OpenMP framework.

```
// ** Fine-grain task generation using algorithms-by-blocks
int create_gemm_task(int transa, int transb,
                     FLA_Obj alpha, FLA_Obj A, FLA_Obj B,
                     FLA_Obj beta,  FLA_Obj C,
                     Scheduler s) {
  // no transpose A, no transpose B
  // Matrix objects A, B, and C are hierarchically partitioned
  for (p=0;p<A.width();++p)
    for (k2=0;k2<C.width();++k2)
      for (k1=0;k1<C.length();++k1)
        // a task is enqueued with in/out arguements
        s.enqueue(Task(name=''Gemm'', op=blas_gemm,   // function pointer
                       n_int_args=2, transa, transb,  // 2 - integer variables
                       n_fla_in=4, alpha, A, B, beta, // 4 - FLA input matrices
                       n_fla_out=1, C);               // 1 - FLA output matrix
}
```

Figure 10: Fine-grain tasks are gathered for out-of-order scheduling when the operation is equipped with a valid scheduler object.

4.5 Storage scheme for dense problems

Certain current DLA libraries (for instance SuperMatrix and PLASMA) have moved from the traditional column-major matrix format used in the classic BLAS and Linear Algebra PACKage (LAPACK) libraries to a format of storage-by-blocks (also called tile layout); a matrix is divided in blocks, and each block is contiguously laid out in memory. While this may have certain performance advantages, we argue that it is not appropriate in our application.

Block formats are motivated by the consideration that, if a block fits in the cache of a single core, it provides a better data locality. Additionally, the storage scheme improves the concurrency of multi-threaded operations as false-sharing is reduced. The increased complexity of global addressing elements in this storage-by-blocks can be resolved by using high-level programming interfaces [26, 43]. To interface a matrix in column-major format to such a library, the matrix can be repacked into blocks to exploit the hierarchy of the memory structure [1, 36].

Our main reason for using column-major storage throughout, and not adopting a block format, derives from the workflow depicted earlier in Fig. 7. One sees that we have to reconcile two different partitioning layouts; one is based on an irregular *hp*-mesh for the subassembly procedure, and the other is partitioned by a fixed computational block size for the numerical factorization. While the packed format is computationally advantageous, it may incur significant overhead in merging the Schur complements unless data enumeration is carefully managed.

Fig. 12a shows that our choice of storage format does not adversely affect performance:

```
// ** Schedule all tasks
int Scheduler::flush() {
  while (tasks_not_empty()) {
    end = begin + window_size;       // set a range of tasks for analysis
    analyze();                       // construct a DAG for those tasks
    execute();                       // execute tasks in parallel
  }
  tasks.clear();                     // clean-up task queue

  return SUCCESS;
}

// ** Execute tasks in an active window
int Scheduler::execute() {
  for (i=begin;i<end;++i) {
    #pragma omp task firstprivate(i)
    schedule_tasks(&tasks[i]);       // schedule fine-grain tasks using OpenMP
  }

  #pragma omp taskwait               // complete execution of a DAG
  begin = end;                       // close the window

  return SUCCESS;
}

// ** Recursive DFS-like task scheduling
int Scheduler::schedule_tasks(Task *t) {
  // if this task is alreay processed, then skip it
  if (t->get_status() == COMPLETED)
    return SUCCESS;

  for (i=0;i<t->get_n_dependent_tasks();++i) {
    // recursively schedule dependent tasks first
    #pragma omp task firstprivate(i)
    schedule_tasks(t->get_dependent_task(i));
  }

  #pragma omp taskwait

  // after dependent tasks are executed, then the current task is executed
  t->execute();

  return SUCCESS;
}
```

Figure 11: To execute tasks in a DFS-like order, dependent tasks are recursively executed before processing a task.

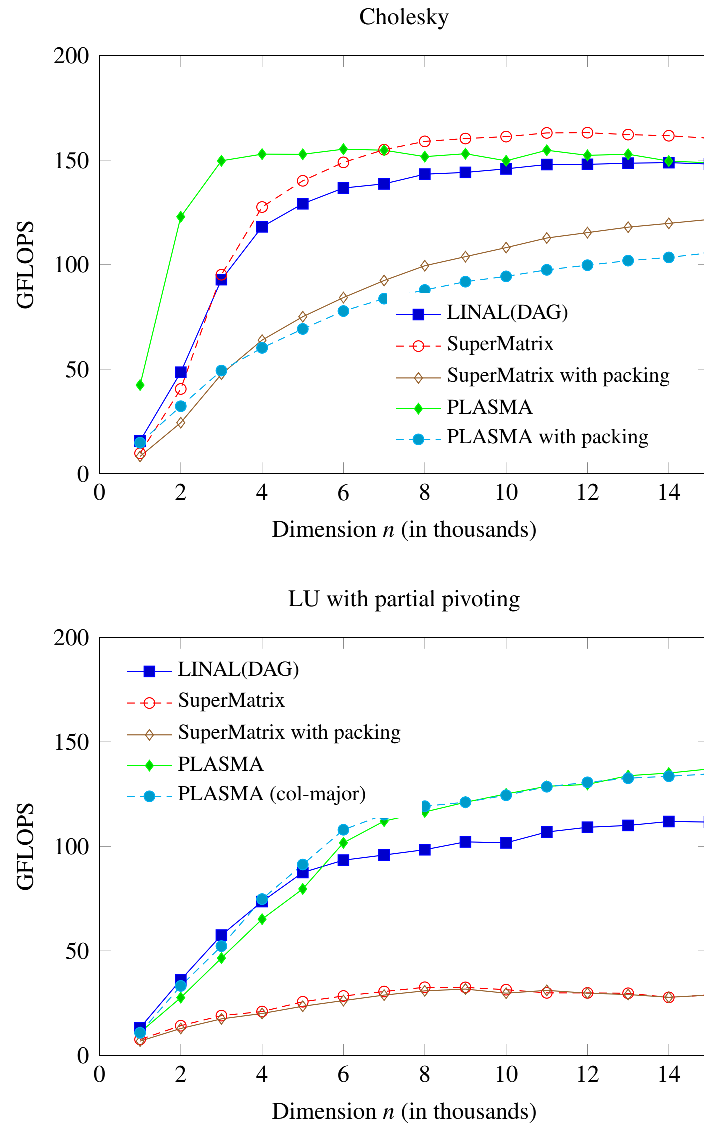


Figure 12: [Clarksville 24 cores] Dense matrix factorizations are evaluated with a fixed blocksize 256. Our dense solver adopts the column-major storage format for both factorizations, and SuperMatrix uses the storage-by-blocks for both factorizations. PLASMA is evaluated with storage-by-blocks and column-major storage format respectively for Cholesky and LU factorization with partial pivoting.

our software performs similar to, or slightly less than SuperMatrix and PLASMA on Cholesky factorization. However, if these packages interface to software using column format, and repacking needs to be included, our performance is far superior.

4.6 LU factorization

A major difficulty for the efficient parallelization of LU factorization lies in the design of the pivoting operation, which requires an entire column vector. While the traditional column-major matrix format is well-suited for this purpose, the storage-by-blocks is problematic since column vectors are scattered among blocks.

Both Supermatrix [14] and PLASMA [20] have strategies for dealing with this problem. The comparison of LU factorization with partial pivoting depicted in Fig. 12b shows the performance of our task scheduler matches that of PLASMA for small matrices, but PLASMA achieves a higher performance asymptotically. Both implementations do not pay an extra cost in converting the format to detect pivots. By contrast, Supermatrix does not scale well, as it requires copying blocks for each panel factorization.

Alternatively, traditional partial pivoting can be modified into the incremental pivoting scheme [39]. The scheme updates the column blocks by performing LU factorization with partial pivoting in a pair of blocks including the upper triangular matrix of a corresponding diagonal block. Since the pivoting operation is executed separately in each column block, this approach carries a number of pivoting operations which may cause excessive element growth [38]. In practice, the incremental pivoting scheme provides more efficient task parallelism as it removes synchronizations due to pivoting on column vectors [14].

5 Performance

We present the performance of the proposed solver against state-of-the-art parallel sparse direct solvers *i.e.*, MUltifrontal Massively Parallel sparse direct Solver (MUMPS) and PARDISO:

- MUMPS [4, 5] has been developed for distributed architectures via Message Passing Interfaces (MPI) since 1996. For this comparison, MUMPS version 4.10.0 is interfaced to Scalable Linear Algebra PACKage (ScaLAPACK) with Basic Linear Algebra Communication Subprograms (BLACS) provided by Intel Math Kernel Library (MKL) version 10.2.
- PARDISO [40, 41] solver was developed for multi-core architectures in 2004, and is part of Intel MKL version 10.2 and higher.

Clarksville	
Processors	24 Processing Cores, Intel Dunnington 2.66 GHZ (4x hex-core chips)
Memory	96 GB ccNUMA
Cache	16 MB L3, three 3 MB shared L2 caches
Compiler & OpenMP	Intel 11.1
DLA	FLAME ver 6883, PLASMA ver 2.0
BLAS	Intel MKL 10.2
Theoretical Peak	256 GFLOPS

Figure 13: Experimental setup on the *Clarksville* machine installed at the Texas Advanced Computing Center

Test problems		
Order p	# of DOFs	# of non-zeros
1	6,017	524,288
2	45,825	3,276,800
3	152,193	13,107,200
4	357,889	40,140,800
5	695,681	102,760,448
6	1,198,337	231,211,008
7	1,898,625	471,859,200

Figure 14: Sparse matrices are obtained from tetrahedral meshes varying an approximation order from 1 to 7. The maximum # of DOFs reaches 1.9 million, and the smallest problem includes 6 thousand unknowns.

For all cases, we use a 24-core machine and test setup described in Fig. 13.

Test problems are based on the same tetrahedral *hp*-discretization varying the polynomial order of approximation, p, from 1 to 7; problem sizes and sparsity are described in Fig. 14. Test problems produce double precision and structurely symmetric matrices. Test problems are reordered based on the nested dissection by Metis version 4.0 [34], and LU factorization is performed with partial pivoting.

5.1 Scalability

We assess the strong scalability of our solver by performing an analysis in terms of Amdahl's law. Let the normalized workload be divided into serial and parallel fractions $s+p=1$. The serial part is unavoidable for several reasons: mutual dependencies, sequential workflows, shared resources, and start-up overhead, etc. The time cost performed by a single thread for the normalized workload is

$$T_1 = s + p$$

and the time cost needed by N workers for the same workload is

$$T_N = s + \frac{p}{N}.$$

This type of analysis is called *strong scaling*: we consider the speed-up by increasing the number of processing units for a fixed problem.

The speed-up of a parallel application is then (Amdahl's law [3])

$$S = T_1/T_N = \frac{1}{s + \frac{1-s}{N}}.$$

The law expects that the speed-up of parallel applications eventually converges to $1/s$ as N goes to infinity.

Through curve-fitting to Amdahl's law, the serial part, s, of the solvers can be quantitatively estimated as shown in Fig. 15. Some observations are:

- As shown in the graph, our solver follows Amdahl's expectation up to 16 cores with $s=0.030$.
- Meanwhile, MUMPS shows increasing speed-up to 12 cores, and its serial part is estimated as $s=0.195$.
- The graph also shows that the UHM solver does not scale well when it does not exploit fine-grain tasks. This implies that the contribution from using algorithms-by-blocks and asynchronous task execution are pivotal in gaining parallel performance.

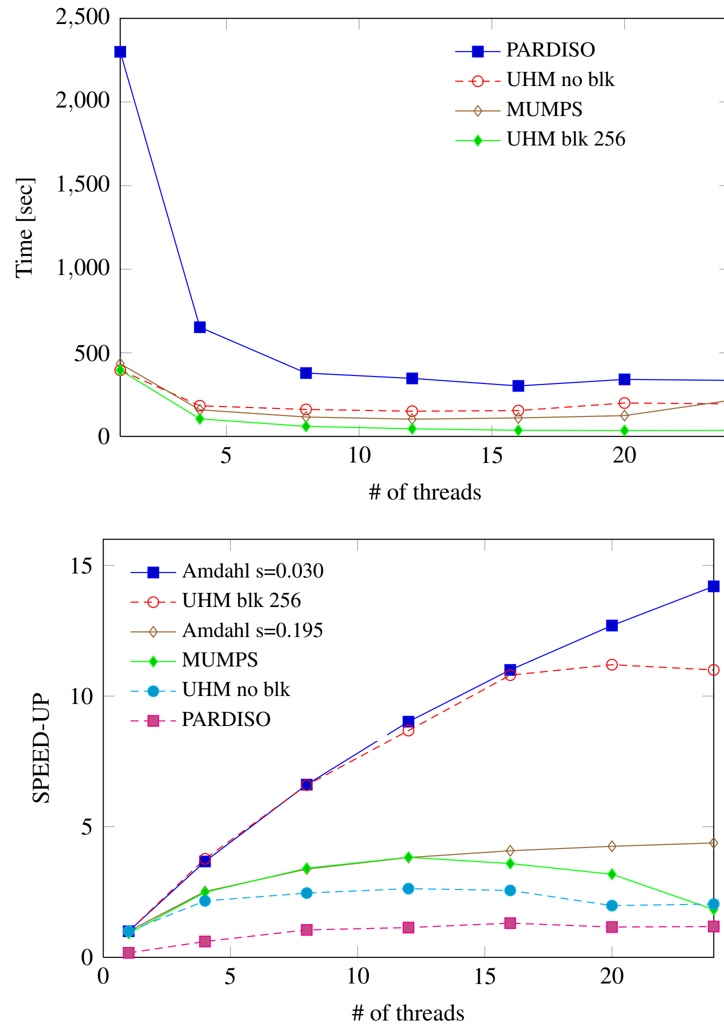


Figure 15: Factorization phase for fixed $p = 4$ by increasing the number of threads from 1 to 24.

We also note that the number of scalable cores is dependent on the problem size; hence, the number of effective cores shown in this figure is limited for this case study only.

Based on this evaluation, the two most competitive direct solvers are UHM and MUMPS. In the next part of this performance comparison, we will evaluate these two solvers.

5.2 Analysis phase

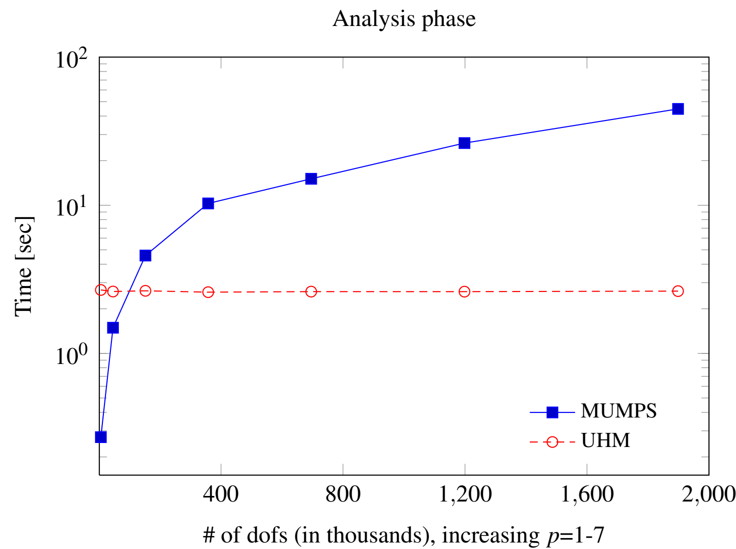


Figure 16: Time (lower is better) measured in the analysis phase with increase in the polynomial order from 1 to 7.

The efficiency of our solver in the reordering and analysis phases can be attributed to a unique interface that takes into account the *hp*-discretization process. Instead of partitioning the matrix graph itself, we let Metis partition a weighted quotient graph based on the element connectivity, where the weights are the associated number of DOFs. Hence, the time complexity does not vary if we only vary the polynomial orders. On the other hand, Fig. 16 shows that MUMPS spends a considerable amount of time reordering and analyzing the matrix that increases with higher p. Most conventional direct solvers with an entry-wise interface would suffer the same problem because this format loses such high-level application information.

On the other hand, for lower order FEM we incurs additional overhead in the conversion from a sparse matrix to a weighted graph.

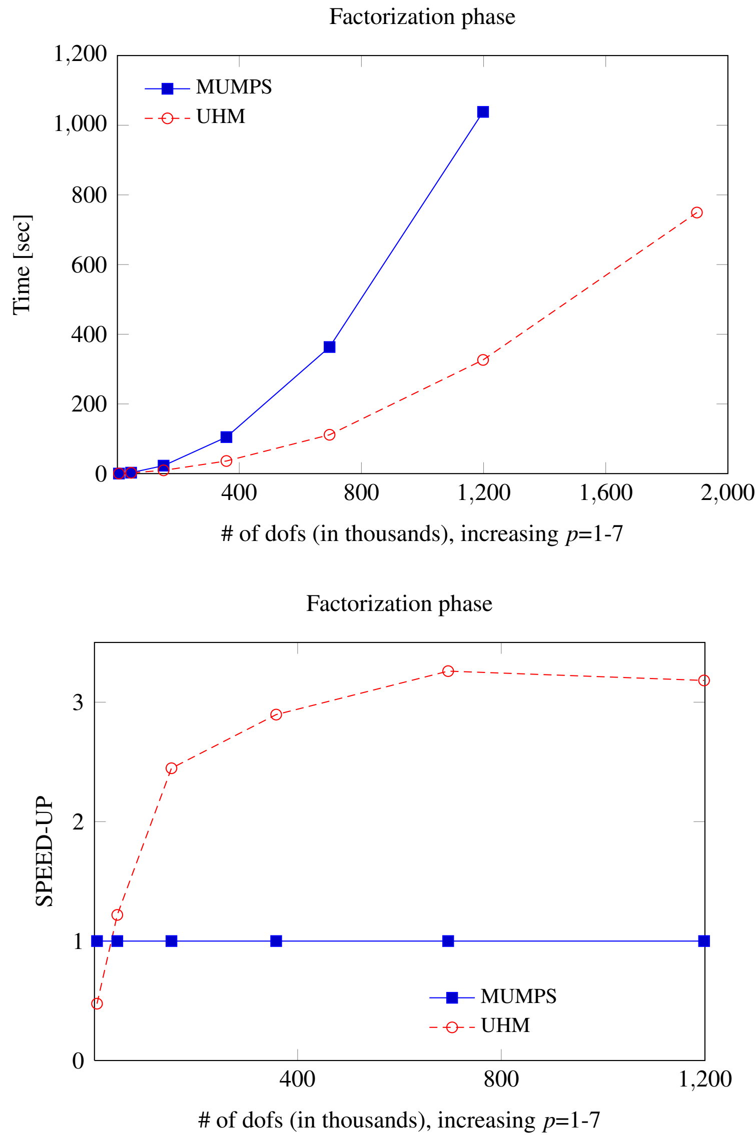


Figure 17: Time in the factorization phase; problem size increase is induced by the increase in the polynomial order from 1 to 7.

5.3 Factorization

Both solvers report roughly the same FLoating Point Operation (FLOP) estimates for the factorization, which is probably caused by the fact that in both cases we use the same nested dissection algorithm provided by Metis. The overall performance gain of our solver is mainly due to its efficient parallel execution of fine-grain tasks for the entire sparse matrix factorization. Fig. 17 shows that our solver is more efficient than MUMPS for problems with higher p values. The graph also shows that our solver is slower than MUMPS for the matrices based on linear discretization.

Fig. 18 compares the space complexity of both solvers: the graph records the maximum amount of memory required for solving the problems. Our solver uses less memory than MUMPS, which is estimated at more than 100 GB for $p = 7$. Because of this, we could not use MUMPS to solve the $p = 7$ case problem on our test machine. The reason for the fact that MUMPS requires more memory than our solver, even if both compute the similar number of FLOPs, is probably that MUMPS includes extra workspaces for communication as the solver is designed for distributed memory architectures.

6 Related Work

We mention two packages that have similarities to our solver; however, we did not test these as they are designed for Cholesky factorization only.

TAUCS [33] uses recursive block storage in its multi-frontal Cholesky factorization accompanying parallel recursive BLAS. The combination of recursive format and dense subroutines naturally schedules fine-grain tasks exploiting the memory hierarchy of modern computing architectures. The basic approach is similar to ours in that the fine-grain tasks are created within the harmony of two-level parallelism. The difference is that the solver is parallelized through Cilk [11], and the lookahead based on task dependencies is not directly used in scheduling tasks.

MA87 [32] uses DAG-based task scheduling for left-looking sparse Cholesky factorization. A global DAG is implicitly created and loosely guides task scheduling. A task, of which dependencies are relieved, is moved from the task pool to a local thread stack. After a task in the thread stack is executed, dependencies are updated, and the procedure is repeated. The task pool maintains tasks with a priority based on type; for example, factorization on diagonal blocks has the highest priority, and updating blocks has a lower priority. Additionally, MA87 interfaces in-house dense kernels implemented in MP54 [31], which also uses DAG-based scheduling. However, the scheme for DAG-based task scheduling is tied to the numerical algorithms, and requires a large task pool to avoid global synchronization.

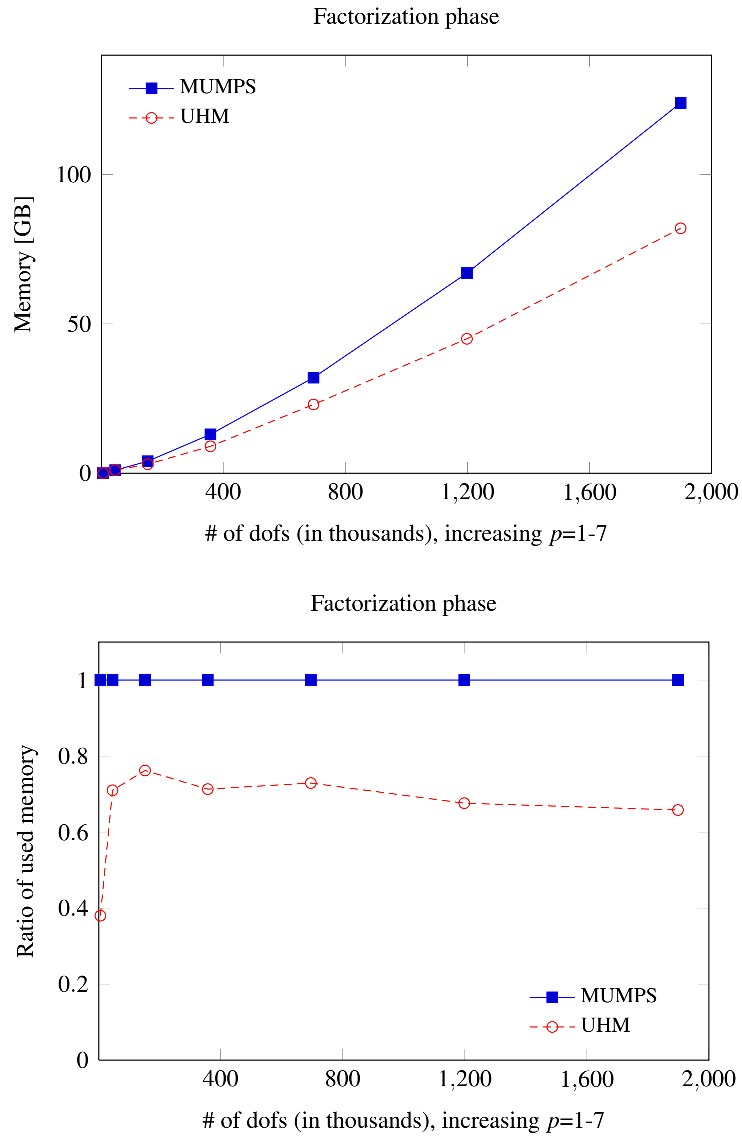


Figure 18: Memory used in the factorization phase with increase in the polynomial order from 1 to 7.

7 Conclusion

This paper presents a novel design for a scalable parallel sparse direct solver that exploits the features of *hp*-adaptive FEM problems, outperforming the MUMPS solver. The proposed direct solver uses two-levels of tasks; macro tasks are associated with the assembly tree, and fine-grain tasks are related to the partial factorization of each UHM. Correspondingly, we have treewise scheduling of the macro tasks, where each task uses a local DAG scheduler. Together, we get efficient scheduling, without the need for constructing a global DAG. For the assignment of tasks to threads, we fully rely on the OpenMP task mechanism.

The multi-level task scheduling does not only provides a lookahead opportunity but also effectively controls the overhead resulting from out-of-order task scheduling as DAGs are locally built. In addition, the multi-level task scheduling enables a smooth transition between tree-level and matrix-level parallelism. A macro task dispatched in a BFS order can naturally associate its subbranches to the same thread, which provides better locality. Meanwhile, fine-grain tasks can be dispatched to other available resources, which enables better load balancing.

The high performance of the proposed solver is mainly attributed to the fine-grain tasks produced by means of algorithms-by-blocks applied to dense subproblems. By contrast, the currently available advanced DLA libraries strictly control all computing resources in exploiting DAGs, and do not allow application-level resource management. The lack of such application-level resource management in those DLA libraries may significantly limit overall efficiency when they are interfaced with other applications that require solutions of multiple dense problems such as a multi-frontal solver.

Our experiments on dense factorization demonstrate that the column-major matrix format does not significantly lower parallel performance. The experiments also show that matrix repacking is expensive compared to the benefits of locality obtained from the blockwise packed format. In fact, the blockwise packed format is not required in driving algorithms-by-blocks; however, the format is used in SuperMatrix and QUARK because they analyze data dependencies in terms of continuous array objects. Instead, we use the column-major matrix format, which allows efficient irregular access to matrix members, and use associated view objects as basic computing units for fine-grain tasks.

The highly asynchronous parallel performance comes from the blend of all design points described above. The solver especially aims at the solution of problems based on *hp*-discretization. Our experiments shows that the solver outperforms the other state-of-the-art sparse direct solvers (PARDISO and MUMPS) when a domain is discretized with a higher order of approximation. This solver is also effective if problems are

formulated with a group of variables, which results in similar sparse patterns to high order discretization.

Acknowledgement

This research was sponsored by National Science Foundation (NSF) under grant no. 0904907. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF. We thank Texas Advanced Computing Center (TACC) at the University of Texas at Austin for allowing to use their equipments in this work.

The code that this paper describes has been developed based on `libflame` and OpenMP. Codes are available under the GNU Lesser General Public License (LGPL) for the non-commercial use at `http://code.google.com/p/uhm`.

References

[1] Emmanuel Agullo, Julie Langou, and Piotr Luszczek. *PLASMA Users Guide 2.0.* `http://icl.cs.utk.edu/plasma`, 2010.

[2] Pothen Alex and Chunguang Sun. A mapping algorithm for parallel sparse Cholesky factorization. *SIAM Journal on Scientific Computing*, 14(5):1253–1257, 1993.

[3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67, Spring Joint Computer Conference*, pages 483–485, May 1967.

[4] P. R. Amestoy, I. S. Duff, J. Y. L'Excellent, and J. Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2002.

[5] P. R. Amestoy, Abdou Guermouche, and J. Y. L'Excellent. Hybrid scheduling for the parallel solution of linear systems. *Parallel Comput.*, 32(2):136–156, 2006.

[6] Cleve Ashcraft and Roger Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Transactions on Mathematical Software*, 15(4):291–309, December 1989.

[7] I. Babuška, M. Griebel, and J. Pitkäranta. The problem of selecting the shape functions for a p-type finite element. *International Journal for Numerical Methods in Engineering*, 28(8):1891–1908, August 1989.

[8] Ivo Babuška and Manil Suri. The p and hp versions of the finite element method, basic principles and properties. *SIAM Review*, 36(4):578–632, 1994.

[9] Ivo Babuška, B. A. Szabó, and I. N. Katz. The p-version of the finite element method. *SIAM journal on numerical analysis*, 18(3):515–545, 1981.

[10] Paolo Bientinesi, Victor Eijkhout, Kyungjoo Kim, Jason Kurtz, and Robert A. van de Geijn. Sparse direct factorizations through Unassembled Hyper-Matrices. *Computer Methods in Applied Mechanics and Engineering*, 199(9-12):430–438, December 2010.

[11] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 37, pages 207–216, August 1995.

[12] 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.

[13] P. Carnevali, R. B. Morris, Y. Tsuji, and G. Taylor. New basis functions and computational procedures for p-version finite element analysis. *International journal for numerical methods in engineering*, 36(22):3759–3779, 1993.

[14] Ernie Chan, Robert A. van de Geijn, and Andrew Chapman. Managing the complexity of lookahead for LU factorization with pivoting. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures - SPAA '10*, pages 200–208, New York, New York, USA, 2010. ACM Press.

[15] 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 IEEE International Conference on Cluster Computing*, pages 91–99. IEEE, 2007.

[16] Timothy A. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Number 0898716136. SIAM, Philadelphia, PA, USA, 2006.

[17] Timothy A. Davis and William W. Hager. Dynamic supernodes in sparse Cholesky update / downdate and triangular solves. *ACM Transactions on Mathematical Software (TOMS)*, 35(4):1–23, 2009.

[18] Leszek Demkowicz, Jason Kurtz, David Pardo, Maciej Paszynski, Waldemar Rachowicz, and Adam Zdunek. *Computing with Hp-Adaptive Finite Elements, Vol. 2: Frontiers Three Dimensional Elliptic and Maxwell Problems with Applications*. Chapman & HallCRC, 2007.

[19] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications*, 20(3):720–755, 1999.

[20] Jack Dongarra, Mathieu Faverge, Hatem Ltaief, and Piotr Luszczek. Achieving numerical accuracy and high performance using recursive tile LU factorization. Technical report, LAPACK Working Note 259, 2011.

[21] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[22] I. S. Duff. Parallel implementation of multifrontal schemes. *Parallel Comput.*, 3(3):193–204, 1986.

[23] I. S. Duff, A. M. Erisman, and J. K. Reid. On George's nested dissection method. *SIAM Numerical Analysis*, 13(5):686–695, 1976.

[24] G. A. Geist and E. Ng. Task scheduling for parallel sparse Cholesky factorization. *International Journal of Parallel Programming*, 18(4):291–314, 1989.

[25] J. R. Gilbert and R. E. Tarjan. The analysis of a nested dissection algorithm. *Numer. Math.*, 50(4):377–404, 1987.

[26] Peter Gottschling, David S. Wise, and Michael D. Adams. Representation-transparent matrix algorithms with scalable performance. In *ICS '07 Proceedings of the 21st annual international conference on Supercomputing*, pages 116–125, 2007.

[27] Nicholas I. M. Gould, Jennifer A. Scott, and Yifan Hu. A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations. *ACM Transactions on Mathematical Software*, 33(2):10:1–32, June 2007.

[28] A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):502–520, May 1997.

[29] Anshul Gupta and Vipin Kumar. A scalable parallel algorithm for sparse Cholesky factorization. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 793–802. ACM, 1994.

[30] Fred G. Gustavson, Isak Jonsson, Bo Kagström, 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.

[31] J. D. Hogg. A DAG-based parallel Cholesky factorization for multicore systems. Technical Report December, SFTC Rutherford Appleton Laboratory, RAL-TR-2008-029, Harwell Science and Innovation Campus, 2008.

[32] J. D. Hogg, J. K. Reid, and J. A. Scott. A DAG-based sparse Cholesky solver for multicore architectures. Technical report, SFTC Rutherford Appleton Laboratory, RAL-TR-2009-004, Harwell Science and Innovation Campus, 2009.

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

[34] George Karypis and Vipin Kumar. METIS : A software package for partitioning unstructured graphs , partitioning meshes , and computing fill-reducing orderings of sparse matrices. Technical report, University of Minnesota, 1998.

[35] Joseph W. H. Liu. The multifrontal method for sparse matrix solution: theory and practice. *Siam Review*, 34(1):82–109, 1992.

[36] 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, Depar, 2004.

[37] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 3.0*. `http://www.openmp.org`, 2008.

[38] Enrique S. Quintana-Ortí and Robert A. van de Geijn. Updating an LU factorization with pivoting. *ACM Transactions on Mathematical Software*, 35(2):1–16, July 2008.

[39] 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 Transactions on Mathematical Software*, 36(3):1–26, July 2009.

[40] Olaf Schenk and Klaus Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. In *Proceedings of the International Conference on Computational Science-Part II*, pages 335–363. Springer-Verlag, 2002.

[41] Olaf Schenk and Klaus Gärtner. On fast factorization pivoting methods for sparse symmetric indefinite systems. *Electronic Transactions on Numerical Analysis*, 23:158–179, 2006.

[42] B. A. Szabó. The p and hp versions of the finite element method in solid mechanics. *Computer Methods in Applied Mechanics and Engineering*, 80(1-3):185–195, 1990.

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

[44] Asim Yarkhan, Jakub Kurzak, and Jack Dongarra. QUARK Users' Guide. Technical Report April, Electrical Engineering and Computer Science, Innovative Computing Laboratory, University of Tenessee, 2011.