A Task-Based Distributed Parallel Sparsified Nested Dissection Algorithm

Léopold Cambier
Stanford University
Stanford, California, USA
leopoldcambier@gmail.com

Eric Darve
Stanford University
Stanford, California, USA
darve@stanford.edu

ABSTRACT

Sparsified nested dissection (spaND) is a fast scalable linear solver for sparse linear systems. It combines nested dissection and separator sparsification, leading to an algorithm with an $O(N \log N)$ complexity on many problems. In this work, we study the parallelization of spaND using TaskTorrent, a lightweight, distributed, task-based runtime in C++. This leads to a distributed version of spaND using a task-based runtime system. We explain how to adapt spaND's partitioning for parallel execution, how to increase concurrency using a simultaneous sparsification algorithm, and how to express the DAG using TaskTorrent. We then benchmark spaND on a few large problems. spaND exhibits good strong and weak scalings, efficiently using up to 9,000 cores when ranks grow slowly with the problem size.

CCS CONCEPTS

• **Mathematics of computing → Computations on matrices**; • **Computing methodologies → Parallel programming languages**.

ACM Reference Format:
Léopold Cambier and Eric Darve. 2021. A Task-Based Distributed Parallel Sparsified Nested Dissection Algorithm. In *Proceedings of Platform for Advanced Scientific Computing Conference (PASC21).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In this paper, we consider the parallelization a fast sparse hierarchical solver, sparsified nested dissection (spaND) [9]. spaND can be used to solve sparse linear systems

$$Ax = b \text{ where } A \in \mathbb{R}^{N\times N}, b \in \mathbb{R}^N, \text{ and } x \in \mathbb{R}^N.$$

It does so by computing an approximate factorization of A in $O(N \log N)$ and up to a controllable approximation error. To solve large problems, however, a distributed-memory parallel algorithm is required. While scientific codes are often written in a bulk synchronous or fork join approach, we consider instead a task-based approach. This lets the program exploit parallelism as much as possible and avoid unnecessary synchronization points. This can however be difficult to implement when relying only on MPI for instance. We do so with the help of a runtime system, TaskTorrent (`TTor`) [10]. `TTor` lets the user express computations using a parametrized task graph and explicit one-sided active messages between nodes. It is designed as a flexible library able to accommodate legacy MPI-based codes and arbitrary dependencies and user data structures.

1.1 Previous work

Task-based algorithms have already been used in linear algebra. Since the years 2010, most efforts have been focused on dense linear algebra algorithms (typically Cholesky, LU, and QR factorizations). The PLASMA (for multicore CPUs) and MAGMA (for hybrid CPU and GPU machines) projects [3, 23, 25] implement tiled task-based algorithms using a dynamic scheduler. DPLASMA [7] extends PLASMA to distributed memory machines and uses the PaRSEC runtime (closely related to the DAGuE compiler) [8]. More recently, the related SLATE project [15] aims at replacing ScaLAPACK's dense linear algebra algorithms with task-based algorithms. In [18] the authors introduced FLAME and the SuperMatrix data structure, to efficiently map dense linear algebra operations on heterogeneous systems.

Task-based parallelism has also been used in sparse direct solvers. In [20], the authors point out that DAG scheduling of sparse algorithms is challenging because of the large number of small and irregular tasks. Tasks are first created by traversing the elimination tree and using block algorithms. The algorithm only then executes tasks asynchronously using the previously computed dependencies. In [21] the authors integrated StarPU [5] and PaRSEC [8] in the PaStiX [16] sparse direct solver. Using a generic task-based runtime system led to similar performance compared to the original specialized scheduler used in PaStiX, with the additional benefit of leveraging accelerators such as GPUs. The authors in [1, 2] studied the use of a hybrid (CPU+GPU) STF task-based runtime system for a sparse QR algorithm, where task granularity has to be large enough to saturate the GPU, but small enough to exhibit enough parallelism. In [4] the authors took an MPI+task approach. Instead of expressing the entire DAG with PaRSEC or StarPU, only the local DAG is provided to a runtime system. This is well-suited to hybrid CPU+GPU machines, where each subdomain (owned by one MPI rank) is assigned to a heterogeneous machine. Each domain can then be scheduled on the multicore CPU, possibly extended by accelerators such as GPUs.

The only distributed memory implementation of a spaND-like algorithm can be found in [22] where the authors consider a distributed Hierarchical Interpolative Factorization (HIF) algorithm [17]. HIF is similar to spaND but is restricted to 7-point stencils and uses interpolative factorization. HIF can be extended to more general matrices but this has not been investigated so far. Our approach works on any sparse matrix, uses orthogonal transformations, and a task-based approach.

However, unlike in [22], we do not use distributed RRQR's to parallelize the large low-rank factorizations arising at the top of the tree. This will be the object a future publication. spaND is an $O(N \log N)$ algorithm when used over matrices coming from the discretization of 3D partial differential equations (PDEs). Every level in the algorithm has an $O(N)$ complexity.

In spaND, separators are split into interfaces, which form the basic blocking structure to operate on the matrix. The leaves have a lot of interfaces which naturally leads to significant concurrency. However, the top levels only have a few large interfaces. In our approach, we associate each task with an interface. This is a limitation that results primarily from our choice of a sequential RRQR. As such, concurrency is limited when the sparsified interfaces are large, as is typically the case on large problems coming from 3D PDEs. As such, we will focus on applications coming from 2D PDEs when the matrix size is very large. In this case, the top separator size grows slowly with N and the algorithm is expected to have an $O(N)$ complexity. Hence, a task-based approach should scale well. This is what we indeed observe.

1.2 Contributions

In this paper, we consider a distributed-memory, task-based implementation of spaND. In particular, we

- Explain how to perform partitioning, ordering, and clustering in a distributed-memory context;
- Modify the sparsification algorithm used in [9] to improve concurrency and mathematically prove that the accuracy is unchanged;
- Describe in details the task DAG, and how to formulate it using a PTG approach;
- Perform weak and strong scalings experiments up to 9,000 cores, demonstrating good scalability of spaND with `TTor`.

2 TASK-BASED SPAND ALGORITHM

2.1 The spaND algorithm

spaND is a hierarchical sparse algorithm to approximately factor sparse matrices. It starts by computing a multilevel partitioning and clustering of the graph of A. This partitioning is based on Nested Dissection (ND). At each level $0 \leq \ell < L$, un-eliminated degrees-of-freedom (dofs) of A are divided into two groups.

- Interiors are disconnected clusters of dofs, to be eliminated using a block LU.
- Separators are the remaining dofs, separating the interiors.

We then further cluster separators into clusters called *interfaces*. Each interface at level ℓ separates a pair of interiors at the same level. At every level, each interface is then scaled using block LU and then sparsified (i.e. compressed) using rank-revealing QR (RRQR). The algorithm (including the definition of elimination, block scaling, and sparsification) is formally described on Algorithm 1. Its output is an approximate factorization of A that is computationally cheap to solve and is then used as a preconditioner for CG or GMRES.

Algorithm 1 The spaND algorithm. The output is an implicit factorization of $A \approx P = \prod_i F_i$ into sparse triangular and orthogonal factors. P is then used as a preconditioner in CG or GMRES.

```
Require: Sparse matrix $A$, maximum level $L$
  Compute a ND ordering for $A$, infer interiors, separators and interfaces
  for all $\ell = 0, \dots, L-1$ do
      for all interior $p$ do
          % Eliminate $p$ using block LU
          Compute an LU factorization of $A_{pp}$, $A_{pp} = L_{pp}U_{pp}$
          for all $n$ neighbor of $p$ do
              Update $A_{pn} \leftarrow L_{pp}^{-1}A_{pn}$ and $A_{np} \leftarrow A_{np}U_{pp}^{-1}$
          for all $m, n$ neighbors of $p$ do
              Update $A_{mn} \leftarrow A_{mn} - A_{mp}A_{pn}$
      for all interface $p$ do
          % Scale $p$ using block LU
          Compute an LU factorization of $A_{pp}$, $A_{pp} = L_{pp}U_{pp}$,
          Update $A_{pp} \leftarrow I$
          for all $n$ interface neighbor of $p$ do
              Update $A_{pn} \leftarrow L_{pp}^{-1}A_{pn}$
              Update $A_{np} \leftarrow A_{np}U_{pp}^{-1}$
      for all interface $p$ between eliminated interiors do
          % Sparsify $p$ using RRQR
          Compute $Q_p = \begin{bmatrix} Q_p^f & Q_p^c \end{bmatrix}$ using RRQR on $\begin{bmatrix} A_{pn} & A_{np}^\top \end{bmatrix}$
          s.t. $\left\| Q_p^f \begin{bmatrix} A_{pn} & A_{np}^\top \end{bmatrix} \right\|_F \leq \varepsilon$ where $n = n_1, \dots, n_k$ are
          all the neighbors of $p$.
          for all $n$ interface neighbor of $p$ do
              Update $A_{pn} \leftarrow Q_p^{c,\top}A_{pn}$ and $A_{np} \leftarrow A_{np}Q_p^c$
      Merge all clusters and edges
```

2.2 Parallel partitioning, ordering, and clustering

We now describe the algorithm used to define separators, interfaces (a clustering of the vertices within separators), and to distribute the matrix across MPI ranks[1]. In the following, the notation $//$ denotes the integer division (i.e., $i//j = \text{floor}(i/j)$).

The algorithm is based on a recursive bisection (a partitioning) of the graph of A and proceeds in three steps. Let $L > 0$ be the number of desired levels and $P > 0$ (a power of 2) the number of MPI ranks. In the following, we say that a set $0, \dots, N-1$ is distributed based on a map $i \longrightarrow m_i$ with $0 \leq m_i < M$ over P ranks if rank $0 \leq p < P$ owns i such that $pQ \leq m_i < (p+1)Q$ with $Q = (M + P - 1)//P$.

Given $L > 0$, we consider a binary tree of separators as shown in Figure 1a. We denote a separator by (ℓ, k) where $0 \leq \ell < L$ is the level (with 0 for the leaves and $L-1$ for the top) and $0 \leq k < 2^{L-\ell-1}$.

For (s,t) a pair of separators, $\text{top}(s,t)$ is defined as the separator r on the shortest path $s \longrightarrow t$ the closest to the root $(L-1, 0)$. If $t = \texttt{none}$, then $\text{top}(s,t)$ returns s. For a separator $t = (\ell, k)$ we also define $\text{level}(t) = \ell$. Finally, let $N_i = \{j | A_{ij} \neq 0, j \neq i\}$ be the neighborhood of vertex i in A.

[1]In this work, we use the term rank for the ranks resulting from the low-rank approximations, and MPI rank for a processor's position within a MPI communicator.

The algorithm to build separators, interfaces, and to partition the matrix is the following.

- A recursive bisection (RB) of A is computed, assigning to every vertex i a partition p_i, $0 \leq p_i < 2^{L-1}$. The dissection is recursive, meaning the map $i \longrightarrow p_i // 2^\ell$ defines the partitioning at level ℓ. This can be done entirely algebraically or using geometrical information. In practice, if geometry information is available, it is preferable to use it. The matrix is then distributed based on this $i \rightarrow p_i$ partitioning. Edges in the matrix are distributed using a 1D (block-) column partitioning: A_{ij} is mapped to the MPI rank of vertex j.
- Nested dissection (ND) separators are computed, assigning to each vertex i a separator (ℓ, k). In our implementation, vertices on the left of a partition but adjacent to the right partition form a separator. The algorithm proceeds level by level, computing first the level $L-1$ separator (the top). It then communicates that information (using a halo exchange) and then proceeds with level $L-2$, etc., until level 0. Those halo exchanges are necessary since higher-level separator information is required to compute lower level separators.
- Interfaces are computed by assigning to each vertex i a tuple of separators (s_l, s_r). s_l (resp. s_r) is the highest separator on the left (resp. right) of i. s_l (resp. s_r) is computed by repeatedly applying top between s_l (resp. s_r) and the separator corresponding to the left (resp. right) neighbors of i. By construction, i always has at least one right-neighbor, so we initialize s_r to none. However, since i may have no left-neighbors, we initialize s_l to $(0, p_i)$. This step can be done in one pass over the data as it only depends on previously computed separator information.

The complete algorithm is available in Algorithm 2 and illustrated on Figure 1. Each MPI rank only holds the piece of s and c carrying local and adjacent (halo) nodes information.

The output of Algorithm 2 is a mapping from vertices i to a 5-tuples of integers and separators

$$\text{map} : i \longrightarrow \text{map}(i) = (p_i, s, s_l, s_r, 0)$$

defining interfaces at level 0. p_i is the partition of vertex i at level 0, s its ND separator, s_l and s_r its left and right ND separators neighbors, and 0 indicates this is the partitioning at the very first level. Vertices i, j such that $\text{map}(i) = \text{map}(j)$ are then clustered together and form an interface. Note that interfaces, by construction, belong to the same partition and, as such, to the same MPI rank.

From level ℓ to level $\ell+1$, merging is done by following the RB and ND trees up towards the root. Formally, a non-eliminated interface $\phi = (p_i, s, s_l, s_r, \ell)$ (i.e., s is not a leaf in the ND tree at level ℓ) is transformed into its parent interface $\psi = \text{parent}(\phi)$ using the map

$$\phi \longrightarrow \text{parent}(\phi) : (p_i, s, s_l, s_r, \ell) \longrightarrow (p_i//2, s, \text{up}(s_l, \ell), \text{up}(s_r, \ell), \ell{+}1)$$

If s is an ND leaf at level ℓ, $\text{up}(s, \ell)$ is the parent of s in the ND binary tree, otherwise $\text{up}(s, \ell) = s$. Different interfaces ϕ, ϕ' with the same parent ψ are then merged together to form an interface at the next level. Interfaces are then distributed across MPI ranks based on their partition.

Notice that

Algorithm 2 Partitioning algorithm. "Halo update" means communicating updated p, s, c to neighboring MPI ranks to update their halo values, and vice-versa.

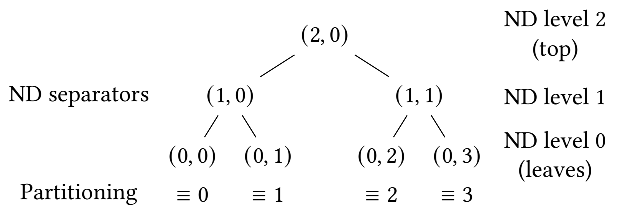
```
Require: $A$ (with symmetric sparsity pattern), distributed, $L > 0, P \geq 2^{L-1}$
Ensure: $0 \leq p_j < 2^{L-1}$
Ensure: $s_j = (\ell, k)$ with $0 \leq \ell < L, 0 \leq k \leq 2^{L-\ell-1}$
Ensure: $c_j = (s_l, s_r)$ are separators on the left and right of $j$
  $p \leftarrow RB(A, 2^{L-1})$                ▷ Recursively partition $A$ into $2^{L-1}$ pieces
  Distribute $A$ based on $j \rightarrow p_j$ map.        ▷ 1D column partitioning
  for $j$ local do                                ▷ Default values
      $s_j \leftarrow (0, p_j)$
  Halo_update$(A, p, s, c)$
  for $\ell = L-1, \ldots, 1$ do          ▷ Create separators, from top to bottom
      for $j$ local do
          if level$(s_j) = 0$ then                  ▷ If not yet a separator
              $p'_j \leftarrow p_j // 2^{\ell-1}$            ▷ Partitioning at level $\ell$
              if $p'_j \% 2 = 0$ then               ▷ Inside the left partition
                  for $i \in N_j$ do
                      $p'_i \leftarrow p_i // 2^{\ell-1}$    ▷ Partitioning at level $\ell$
                      if $p'_i = p'_j + 1$ then   ▷ Touches the right partition
                          $s_j \leftarrow (\ell, p'_j // 2)$
                          Break
      Halo_update$(A, p, s, c)$
  for $j$ local do                             ▷ Create interfaces
      if level$(s_j) > 0$ then                         ▷ If not a leaf
          $s_l, s_r, p'_j \leftarrow (0, p_j), \text{none}, p_j // 2^{\text{level}(s_j)-1}$
          for $i \in N_j$ do
              if level$(s_i) <$ level$(s_j)$ then    ▷ If $i$ is below $j$ in ND tree
                  $p'_i \leftarrow p_i // 2^{\text{level}(s_j)-1}$
                  if $p'_i \leq p'_j$ then                    ▷ If $i$ is left of $j$
                      $s_l \leftarrow \text{top}(s_i, s_l)$
                  else                                   ▷ If $i$ is right of $j$
                      $s_r \leftarrow \text{top}(s_i, s_r)$
          $c_i \leftarrow (s_l, s_r)$
      else
          $c_i \leftarrow ((0, p_i), (0, p_i))$
  Halo_update$(A, p, s, c)$
  return $p$ the partitioning, $s$ the ordering, $c$ the interfaces clustering
```

- By construction, an interface is always entirely contained in a partition and, as such, is resident on a given MPI rank.
- In the above formula, the ND separator s never changes.

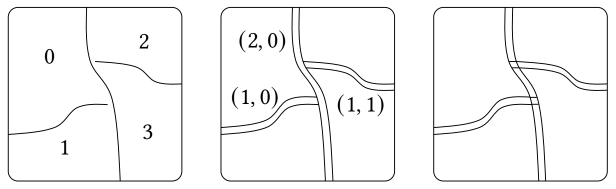
We re-emphasize that, in this algorithm, the load balancing (i.e., which interface is mapped to which MPI rank) is purely based on the recursive bisection of A. In particular, if interface ranks are uniform across the domain then the load balancing is exact. As a rule of thumb, this is usually the case on elliptic PDEs. However, if this is not the case, this load balancing strategy may be far from the optimum.

2.3 Simultaneous sparsification

Sparsification (see Algorithm 1) is the central part of spaND. In this section, we describe a parallel sparsification method. Let us consider a particular level, and assume that eliminations and block scalings have all been performed. What is left to do is to sparsify all interfaces.



(a) Nested Dissection ordering binary tree for $L = 3$



(b) Recursive bisection. The matrix is distributed across MPI ranks using a 1D column partitioning based on the RB.

(c) Separators creation. Separators are vertices in the left partition adjacent to the right partition.

(d) Interfaces definition. Interfaces are built by clustering vertices within separators based on the (p, s_l, s_r) tuple where p is the partition and s_l (resp. s_r) is the left (resp. right) separator.

Figure 1: Partitioning algorithm

Let us denote the remaining interfaces by $s_1, s_2, \ldots, s_k$. We also denote by $n_1, \ldots, n_k$ their respective complement (i.e., if I, $|I| = n$, denote all the remaining dofs, $n_i \cup s_i = I$, $n_i \cap s_i = \emptyset$). At a given step of the algorithm, denote the trailing matrix by A. We can write A in block form as $A = [A_{s_i s_j}]_{i=1,j=1}^{k} \in \mathbb{R}^{n \times n}$. In practice note that most blocks in this matrix are effectively zero and that $A_{s_i s_i} = I$. However, those do not affect the following analysis.

Regular sparsification algorithm. Now consider sparsification following the order $s_1 \to s_2 \to \cdots \to s_k$ (see Algorithm 3). Consider step i, where we sparsify s_i. This means computing, through a rank-revealing factorization, a basis $Q_i = \begin{bmatrix} Q_i^f & Q_i^c \end{bmatrix}$ such that $\left\| Q_i^{f,\top} \begin{bmatrix} A_{i-1,s_i n_i} & A_{i-1,n_i s_i}^\top \end{bmatrix} \right\|_F \leq \varepsilon$. Note the Frobenius norm, as this simplifies some derivations. Let A_i be the matrix with rows and columns to/from the fine dofs from/to n_i dropped. Also define U_i as a $k-$diagonal block matrix (with block j of size $|s_j|$) with ones on the diagonal, except for its i^{th} diagonal block equal to Q_i. We then have $\|U_i^\top A_{i-1} U_i - A_i\|_F \leq \varepsilon$. Now assume that $\|U_i^\top \cdots U_1^\top A_0 U_1 \cdots U_i - A_i\|_F \leq i\varepsilon$ and $\|U_{i+1}^\top A_i U_{i+1} - A_{i+1}\|_F \leq \varepsilon$. We then have

$$
\begin{aligned}
&\|U_{i+1}^\top \cdots U_1^\top A_0 U_1 \cdots U_{i+1} - A_{i+1}\|_F \\
&= \|U_{i+1}^\top (U_i^\top \cdots U_1^\top A_0 U_1 \cdots U_i - A_i + A_i) U_{i+1} - A_{i+1}\|_F \\
&\leq \|U_{i+1}^\top (U_i^\top \cdots U_1^\top A_0 U_1 \cdots U_i - A_i) U_{i+1}\|_F + \|U_{i+1}^\top A_i U_{i+1} - A_{i+1}\|_F \\
&= \|U_i^\top \cdots U_1^\top A_0 U_1 \cdots U_i - A_i\|_F + \|U_{i+1}^\top A_i U_{i+1} - A_{i+1}\|_F \leq (i+1)\varepsilon
\end{aligned}
$$

using the invariance of the Frobenius norm to square orthogonal transformations and the triangle inequality. Let $\widetilde{A} := A_k$ be the final sparsified matrix. We conclude that $\|U_k^\top \cdots U_1^\top A U_1 \cdots U_k - \widetilde{A}\|_F \leq k\varepsilon$. This shows that the *forward error* between the original matrix and the trailing matrix is of order $O(\varepsilon)$.

Algorithm 3 Sequential sparsification algorithm. Q_i is computed based on A_{i-1} and drop_fine zeroes-out entries corresponding to fine degrees of freedom

```
$A_0 \leftarrow A$
for $i = 1, \ldots, k$ do
    Sparsify $s_i$, i.e., compute $Q_i$ such that
        $\|Q_i^{f,\top} \begin{bmatrix} A_{i-1,s_i n_i} & A_{i-1,n_i s_i}^\top \end{bmatrix}\|_F \leq \varepsilon$
    $A_i \leftarrow$ drop_fine($U_i^\top A_{i-1} U_i$)
```

Simultaneous sparsification algorithm. We now consider an alternative algorithm (see Algorithm 4). Instead of sparsifying $s_1 \to s_2 \to \cdots \to s_k$, we sparsify all s_i $(1 \leq i \leq k)$ at the same time, given the original A. This means that for all i, we *simultaneously* compute $Q_i = \begin{bmatrix} Q_i^f & Q_i^c \end{bmatrix}$ such that $\left\| Q_i^{f,\top} \begin{bmatrix} A_{s_i n_i} & A_{n_i s_i}^\top \end{bmatrix} \right\|_F \leq \varepsilon$. Define U_i as a $k-$diagonal block matrix with Q_i in its i^{th} position and ones otherwise. Then consider $\overline{A} = U_k^\top \cdots U_1^\top A U_1 \cdots U_k$ and $\widetilde{A}$ defined as

$$
\widetilde{A}_{s_i s_j} = \begin{bmatrix} 0 & 0 \\ 0 & Q_i^{c,\top} A_{s_i s_j} Q_j^c \end{bmatrix} \text{ if } i \neq j \text{ and } \widetilde{A}_{s_i s_i} = \overline{A}_{s_i s_i} \text{ otherwise.}
$$

In short, off-diagonal block rows and columns corresponding to fine degrees of freedom are zeroed-out compared to $\overline{A}$, with the rest unchanged. We then find

$$
\|\overline{A} - \widetilde{A}\|_F^2 = \sum_{i \neq j} \left\| \begin{bmatrix} Q_i^{f,\top} A_{s_i s_j} Q_j^f & Q_i^{f,\top} A_{s_i s_j} Q_j^c \\ Q_i^{c,\top} A_{s_i s_j} Q_j^f & 0 \end{bmatrix} \right\|_F^2
$$

$$
\leq \sum_{i \neq j} \left\| \begin{bmatrix} Q_i^{f,\top} A_{s_i s_j} Q_j^f & Q_i^{f,\top} A_{s_i s_j} Q_j^c \end{bmatrix} \right\|_F^2 + \sum_{i \neq j} \left\| \begin{bmatrix} Q_i^{f,\top} A_{s_i s_j} Q_j^f \\ Q_i^{c,\top} A_{s_i s_j} Q_j^f \end{bmatrix} \right\|_F^2
$$

Then, since $\left\| \begin{bmatrix} Q_i^{f,\top} A_{s_i s_j} Q_j^f & Q_i^{f,\top} A_{s_i s_j} Q_j^c \end{bmatrix} \right\|_F^2 = \|Q_i^{f,\top} A_{s_i s_j}\|_F^2$ and $\left\| \begin{bmatrix} Q_i^{f,\top} A_{s_i s_j} Q_j^f \\ Q_i^{c,\top} A_{s_i s_j} Q_j^f \end{bmatrix} \right\|_F^2 = \|A_{s_i s_j} Q_j^f\|_F^2 = \|Q_j^{f,\top} A_{s_i s_j}^\top\|_F^2$, we find

$$
\begin{aligned}
\|\overline{A} - \widetilde{A}\|_F^2 &\leq \sum_{i \neq j} \left\| Q_i^{f,\top} A_{s_i s_j} \right\|_F^2 + \sum_{i \neq j} \left\| Q_i^{f,\top} A_{s_j s_i}^\top \right\|_F^2 \\
&= \sum_i \left\| Q_i^{f,\top} \begin{bmatrix} A_{s_i n_i} & A_{n_i s_i}^\top \end{bmatrix} \right\|_F^2 \leq k\varepsilon^2
\end{aligned}
$$

We conclude that $\|U_k^\top \cdots U_1^\top A U_1 \cdots U_k - \widetilde{A}\|_F \leq \sqrt{k}\varepsilon$.

We see that both approaches, Algorithm 3 and Algorithm 4, have a similar bound on the sparsification error. Namely, both approaches implicitly compute an orthogonal matrix U such that, if sparsification is done with tolerance ε, and if $\widetilde{A}$ is the final matrix without the fine edges, $\|U^\top A U - \widetilde{A}\|_F = O(\varepsilon)$ (note that the constant is smaller for Algorithm 4). Because transformations are orthogonal, there is no loss of accuracy by sparsifying all clusters at the same time. We note that other approaches, such as those using interpolative factorization [17, 22], don't use orthogonal transformations.

Algorithm 4 Simultaneous sparsification algorithm. Q_i is computed based on A, for all i. drop_fine zeroes-out entries corresponding to fine degrees of freedom.

```
for $i = 1, \dots, k$ do
    Sparsify $s_i$, i.e., compute $Q_i$ such that
        $\|Q_i^{f,\top} \begin{bmatrix} A_{s_i n_i} & A_{n_i s_i}^\top \end{bmatrix}\|_F \leq \varepsilon$
Compute $\overline{A} \leftarrow U_k^\top \cdots U_1^\top A U_1 \cdots U_k$
Compute $\widetilde{A} \leftarrow \text{drop\_fine}(\overline{A})$
```

The same idea could also be applied, but the error bound will now include a term directly related to A.

However, Algorithm 4 has a significant advantage: all sparsifications can happen in parallel. For Algorithm 3 to be parallel, one would have to compute a 1-coloring on the graph of A, mapping every interface i to a color c_i so that no two neighboring interfaces have the same color. Sparsification would then have to be ordered by color, i.e., s_i is sparsified before s_j if and only if $c_i < c_j$. This is similar to what is done in the parallel version of LoRaSp, see [11]. As a result, we used Algorithm 4 to maximize concurrency. We note that Algorithm 4 has a slightly increased flop count. For neighboring clusters i, j such that i would have been sparsified before j, the sparsification of s_j does not benefit from the reduced size of s_i. However, given the added concurrency and the simplicity (no coloring needed) of the simultaneous sparsification algorithm, this is overall the better approach.

2.4 Task-based algorithm using TaskTorrent

We now turn to the description of the computer implementation. We implemented the algorithm using TTor and a PTG approach. For simplicity, we parallelize each level independently from the others. This means there is a synchronization point *between* levels, and in the following, we consider a specific level in the algorithm.

Using a PTG approach requires identifying tasks and all their dependencies. We create tasks by transforming every block matrix operation (see Algorithm 1) into a task:

- Elimination of an interior creates three different types of tasks: pivot factorization (getrf), panel update (trsm), and Schur complement updates (gemm);
- Block scaling of an interface creates two types of tasks: pivot factorization (getrf) and panel update (trsm);
- Sparsification of an interface creates two types of tasks: rank-revealing factorization (geqp3), i.e., compute Q_i, and trailing matrix update (ormqr), i.e., updating $A_{ij} \leftarrow Q_i^{c,\top} A_{ij} Q_j^c$.

Dependencies depend directly on the data inputs and outputs of tasks. For instance, geqp3 requires as input the blocks located in the row and column of the associated diagonal block. Its output, Q_i, is then needed for the ormqr associated with every block in its row and column.

Figure 2 illustrates the PTG formulation for the elimination, Figure 3 for the scaling, and Figure 4 for the sparsification. Every figure shows the block operation (top), the local PTG (middle), and an illustration of the dependencies in terms of the trailing matrix (bottom). In the trailing matrix, the first three diagonal blocks are interiors (to be eliminated) and the last three are interfaces (to be scaled and sparsified).

Figure 5 shows a small section of the task DAG (arising in the Ice-Sheet benchmark, see Section 3.3). We recall that the parallelization is level-wise. Hence, the DAG is relatively shallow and wide. Figure 6 shows all the tasks at all levels of the algorithm ordered from bottom to top levels.

We finally present some of the TTor code regarding the geqp3 task flow (compare with Figure 4). We recall the following.

- geqp3(p) requires, as inputs, the blocks A_{pn_i} and A_{n_ip}, for all neighbor $n_i = n_1, \dots, n_k$ of p;
- geqp3(p) performs a low-rank approximation of $\begin{bmatrix} A_{pn} & A_{np}^\top \end{bmatrix}$ where $n = n_1 \cup \dots \cup n_k$, leading to an orthogonal tall-and-skinny Q_p^c;
- geqp3(p) outputs Q_p^c, which is the input of the tasks ormqr(p,n) and ormqr(n,p) for all neighbor n of p.

Now assume Taskflow<int> geqp3 has been defined. We first express the number of incoming dependencies for a given diagonal block k.

```
geqp3.set_indegree([&](int k) {
    Cluster* self = get_cluster_local(k);
    return 1 +
           self->edgesColNbrSparsification().size() +
           self->edgesRowNbrSparsification().size();
});
```

Then, we provide a function to map task k to a thread.

```
geqp3.set_mapping([&](int p) {
    return p % ttor_threads;
});
```

Then, we indicate the computational routine to perform when task k is ready. This function internally assembles $\begin{bmatrix} A_{pn} & A_{np}^\top \end{bmatrix}$ and computes Q_p^c using LAPACK's geqp3 routine.

```
geqp3.set_task([&](int p) {
    this->sparsify(p);
});
```

Finally, we provide the function used to fulfill dependencies. This is the most complex part of using TTor. This function first collects all the neighboring MPI ranks and which blocks A_{pn}, A_{np} are resident on those MPI ranks. Then, it sends one active message per neighboring MPI rank, fulfilling the dependencies of the ormqr task flow on those blocks (p, n) and (n, p). Note that the implementation computes Q_p^c both as a tall-and-skinny orthogonal matrix (Q in the code), as well as a set of householder reflectors (referred to as h and v in the code).

```
geqp3.set_fulfill([&](int p) {
    Cluster* self = get_cluster_local(p);
    // Collect a map rank -> tasks to fulfill
    map<int,vector<int2>> ff;
    for(auto erow: self->edgesRowNbrSparsification()) {
        int dest = edge2rank(erow);
        if(dest == my_rank) {
            ormqr.fulfill_promise(erow);
        } else ff[dest].push_back(erow);
    }
    for(auto ecol: self->edgesColNbrSparsification()) {
        int dest = edge2rank(ecol);
        if(dest == my_rank) {
            ormqr.fulfill_promise(ecol);
        } else ff[dest].push_back(ecol);
```

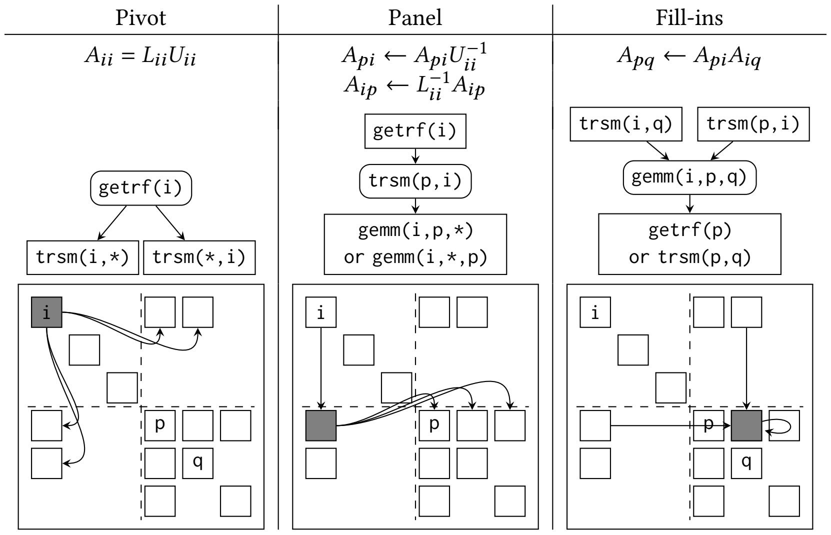


Figure 2: Task-based elimination. $L_{ii}U_{ii}$ represents a generic factorization where L_{ii} and U_{ii} can both be quickly inverted (Cholesky, partial pivoted LU, etc.). The `gemm(i,p,q)` tasks can happen in any order but are bound to a specific thread to prevent race conditions.

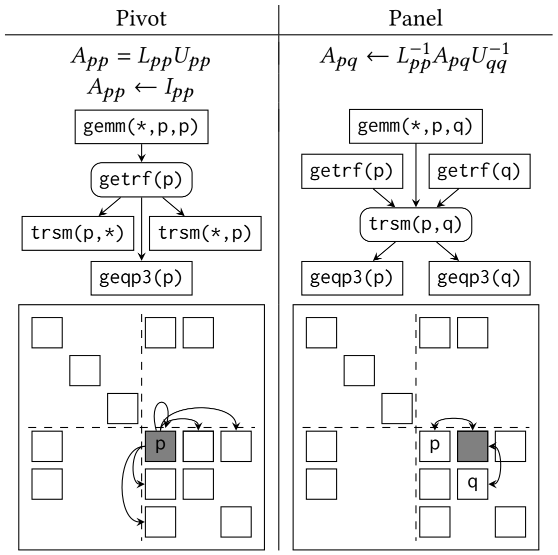
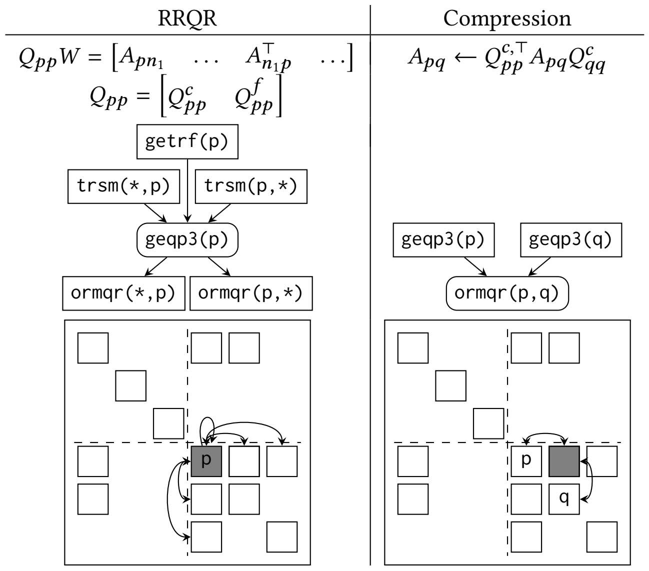


Figure 3: Task-based block scaling. $L_{pp}U_{pp}$ represents a generic factorization where L_{pp} and U_{pp} are easy to invert and where $\|U_{pp}^{-1}\| \approx \|L_{pp}^{-1}\|$.



RRQR
$Q_{pp}W = \left[A_{pn_1} \quad \ldots \quad A_{n_1p}^{\top} \quad \ldots\right]$
$Q_{pp} = \left[Q_{pp}^{c} \quad Q_{pp}^{f}\right]$
getrf(p)
trsm(*,p)
trsm(p,*)
geqp3(p)
ormqr(*,p)
ormqr(p,*)
Compression
$A_{pq} \leftarrow Q_{pp}^{c,\top}A_{pq}Q_{qq}^{c}$
geqp3(p)
geqp3(q)
ormqr(p,q)
p
q

Figure 4: Task-based sparsification. Q_{pp} is computed using a rank-revealing factorization such as QR with column pivoting (geqp3).

```
    }
    // Send to all neighbors
    for(auto& r_ff: ff) {
        int size = self->get_v()->rows();
        int rank = self->get_v()->cols();
        auto ff_view = view(&r_ff.second);
        auto Q_view = view(self->get_Q());
        auto v_view = view(self->get_v());
        auto h_view = view(self->get_h());
        geqp3_am->send(r_ff.first, p, size, rank,
            ff_view, Q_view, v_view, h_view);
    }
});
```

This function requires the definition of the geqp3_am active message, which we finally provide. This active message (1) stores Q_p^c on the receiver and (2) fulfills the local ormqr tasks.

```
auto geqp3_am = comm.make_active_msg(
    [&](int &p, int& size, int& rank,
        ttor::view<EdgeGID>& ff,
        ttor::view<double>& Q_data,
        ttor::view<double>& v_data,
        ttor::view<double>& h_data)
    {
        GhostCluster* s = get_cluster(p);
        GhostEdge* pp = get_pivot(p);
```

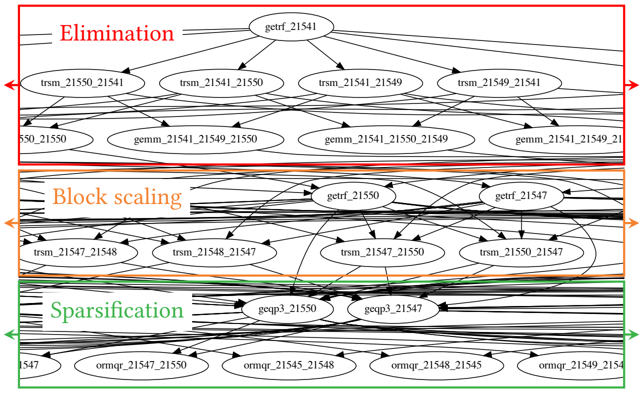


Figure 5: Example of task DAG for the ice-sheet problem (see Section 3) at a particular level. This shows only a portion of the DAG, which is relatively shallow but expands further on the left and right.

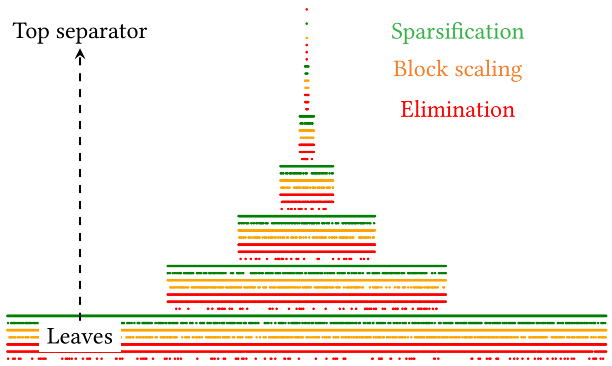


Figure 6: Illustration of all the tasks for the ice-sheet problem. From lower to top levels. Colors indicate the kind of task. We see that spaND exhibits many tasks at the lower levels but only a few large tasks at higher levels.

```
        // Copy Q, v, h
        s->set_Q(make_matrix(Q_data, size, rank));
        s->set_v(make_matrix(v_data, size, rank));
        s->set_h(make_vector(h_data, rank));
        // Make pivot identity
        pp->make_pivot_identity(rank);
        // Fulfill dependencies
        for(auto& e: ff) {
            ormqr.fulfill_promise(e);
        }
    });
```

This is the entire code for the geqp3 task flow in TTor.

3 NUMERICAL RESULTS

We now show some benchmarks of the proposed approach on large problems. The goal of this section is to demonstrate that the TTor approach scales well on sparse blocked linear algebra algorithms.

We implemented parallel spaND in C++ using TaskTorrent [10]. In all the following experiments, the matrices are SPD. As such, we use block Cholesky for pivot factorization, and a tolerance of $\varepsilon = 10^{-2}$. We use Zoltan [6] for algebraic and geometric recursive bisection, BLAS/LAPACK and the Eigen C++ library. Benchmarks were run on the LLNL Quartz machine (https://hpc.llnl.gov/hardware/platforms/Quartz). Each node has a dual-socket Intel(R) Xeon(R) E5-2695 v4 for a total of 36 cores with 128 GB of RAM. GCC (version 8.1.0) is used with OpenMPI (version 4.0.0) and Intel MKL (version 2020.0) for BLAS/LAPACK. Since every node is dual-socket, we use 1 MPI rank per socket. TTor uses threads for shared-memory parallelism. Experimentally, we notice that this is more efficient than 1 rank per node. We conjecture that this is related to memory affinity.

We recall that our parallelization approach is at the granularity of interfaces. We create one task for every block operation, and every block operation is associated to a (pair of) interfaces. spaND exhibits many tasks at the lower level of the hierarchy, but only a few tasks near the root. As such, a key quantity is how large those interfaces are. In 2D or near-2D problems, we expect them to grow slowly with the matrix size N. In 3D, however, the largest interfaces scale like $O\left(N^{1/3}\right)$. This will eventually prevent scalability in our current implementation. Improving scalability for large interfaces and would require parallelizing the geqp3 algorithm using distributed memory. Currently we are limited to a shared memory (multithreaded) parallelization of geqp3. In the following, we study a mix of 2D and 3D problems highlighting this characteristic.

3.1 Regular laplacians

We begin with simple benchmarks using 2D and 3D laplacians. In particular, we study the sizes of the interfaces (i.e., ranks) at various levels of the algorithm and for various problem sizes.

Figure 7 illustrates ranks (averages and maximums) throughout the factorization for a 2D 5-points stencil Poisson equation over a regular $n{\times}n$ grid. The matrix has size $N = n^2$. We can see that, as the matrix size increases, there is only a minor increase in ranks. The maximum ranks throughout the domain grow from 25 to 40 while the problem size grows from $N = 1M$ to $N = 268M$, throughout all levels. Note that a direct method, without sparsification, would see separators of size $O\left(N^{1/2}\right)$ instead of $O(1)$ here. Since tasks have a runtime proportional to the *cube* of the ranks, tasks will remain small throughout the problem. We can then expect good scalability.

We then study ranks of 3D problems on Figure 8 with a 7-points stencil Poisson equation over a regular $n \times n \times n$ grid. The matrix has size $N = n^3$. Unlike in the previous case, there is now a clear increase of ranks with the problem size: when N is multiplied by 8, ranks are multiplied roughly by 2. This shows that ranks grow like $O\left(N^{1/3}\right)$. This is a reduction compared to direct methods with scalings like $O\left(N^{2/3}\right)$. Since the runtime of tasks is proportional to the cube of the ranks, we expect less ideal parallel scaling as we get closer to the top of tree where ranks are the largest.

Given this, Figure 9 shows weak and strong scalings on 2D and near-2D 5 and 7-points stencil Poisson equations. In this setting, ranks grow slowly with N and we see that the algorithm has no problems scaling over many cores.

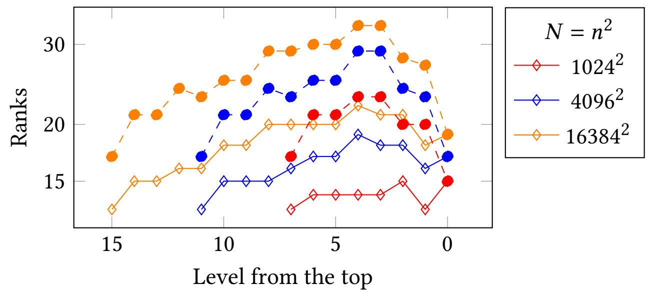


Figure 7: 2D $n \times n$ Poisson equations. Distribution of ranks: averages (solid line, open diamonds) and maximums (dashed line, closed circles). Slowly growing ranks are characteristic of 2D or near-2D problems, where ranks are close to $O(1)$ throughout the factorization. This leads to an $O(N)$ algorithm where most of the work is located at the leaves.

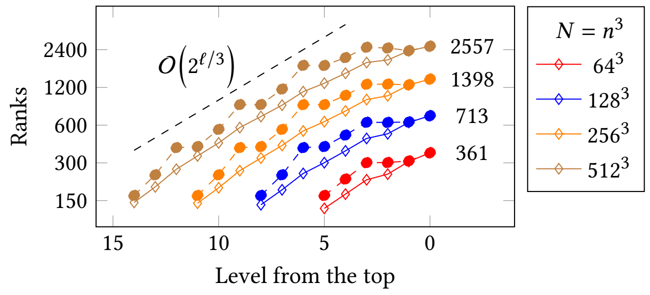
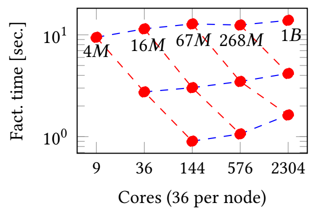
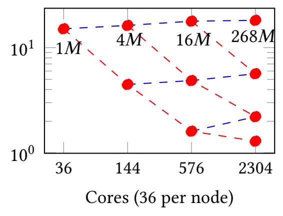


Figure 8: 3D $n \times n \times n$ Poisson equation. Distribution of ranks: averages (solid line, open diamond) and maximums (dash line, solid circle). This shows the expected $O\left(N^{1/3}\right)$ growths of ranks (and the related $O\left(2^{\ell/3}\right)$ growth with the levels), leading to an $O(N \log N)$ algorithm where each level has a comparable amount of flops.



(a) 2D $n \times n$ Poisson equation



(b) Quasi-2D $n \times n \times 16$ Poisson equation

Figure 9: Poisson equation, weak and strong scalings. The numbers indicate the total matrix size N. In those problems, ranks are small throughout the factorization.

3.2 SPE

We continue with the SPE benchmark [12]. This problem corresponds to the discretization of a scalar elliptic PDE in 3D with a 7-points stencil over a regular cube. As such, separators near the top of the tree are relatively large (as mentioned in the previous section), with size $\approx$ 1000 for problems of size 8 M. In addition, those get larger as the problem becomes larger. As such, we perform strong scalings on 1 node with 1 rank, using from 1 to 16 TTor threads (i.e. cores) with N = 8 M. Figure 10 shows the time spent on each level with various numbers of cores (i.e., TTor's threads).

Experimentally, we observe that it is more efficient to use multithreaded BLAS and LAPACK compared to TTor threads towards the end of the factorization. Around level 9, there are still enough tasks to fill all TTor threads. However, we observe that the task duration is slower with 16 TTor threads than multithreaded BLAS/LAPACK. We suspect this is due to suboptimal memory locality, blocking, and vectorization. Multithreaded BLAS/LAPACK exhibit better scaling at this point and are what is used at these levels and above. Furthermore, towards the very end, there are very few tasks left. Multithreaded BLAS and LAPACK are then the only way to leverage multiple cores with our current implementation of the DAG.

Overall, since this problem has a moderate size, most of the time is spent in the first few sparsification levels. Hence, there is enough concurrency available, and the algorithm strong scales well. Larger test cases would exhibit larger ranks near the top, with worst strong scalings. Eventually, the top levels will start dominating.

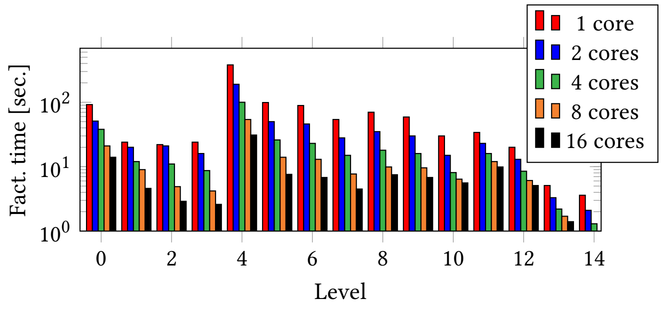


Figure 10: SPE profile, strong scalings, from 1 to 16 cores, N = 8 M. This is a 3D problem so the work is fairly evenly distributed across levels.

3.3 Ice-Sheet

We finish with some results on an Ice-Sheet modeling problem. Those problems come from the modelization of the movement of ice in Antarctica [24]. Stokes equations are used, and a modeling assumption leads to a non-linear equation. The mesh is a vertically extruded 2D mesh. Finally, the combination of a continuation and Newton method is used, and we consider the matrices arising in the first and fourth step, respectively. We consider a mesh with 10 vertical layers and with horizontal x and y resolutions from 16 km (1 M problem size) to 1 km (296 M problem size). As such, the problem is only refined in the x and y dimensions, and can be considered as nearly-two dimensional.

We perform weak scalings and compare spaND to Hypre [14] using Boomer AMG. We use Hypre as a reference point, since it usually scales very well on such elliptic problems. We note that it is a pure MPI code and is not actively managing a task graph. For AMG, the matrix is distributed using ParMetis [19] k-way and Boomer AMG uses a strong threshold of 0.9. For spaND we use one MPI rank per socket (with TTor using all cores through threads), while for Hypre, we use one MPI rank per core. Both use CG with a residual tolerance of 10^{-8}. We use from 36 to 9,216 cores, with

		spaND					AMG (Hypre)	
cores	N	t_{fact}	t_{app}	t_{CG}	t_{tot}	n_{CG}	t_{tot}	n_{CG}
36	1M	6.2	0.14	0.9	7.1	6	15	427
144	4M	7.3	0.15	1.2	8.5	6	16	456
576	18M	8.9	0.15	1.6	10.5	7	22	527
2304	74M	9.8	0.17	1.9	11.7	8	29	627
9216	296M	13.2	0.21	3.7	16.9	12	39	623

Table 1: Ice-sheet results, weak scalings, from 36 to 9,216 cores. t_{fact} is the factorization time, t_{app} is the time to apply the preconditioner once, t_{CG} is the total CG time, $t_{tot} = t_{fact} + t_{CG}$ and n_{CG} is the number of CG steps.

a matrix size from $N = 1.1$ M to $N = 296$ M. Table 1 shows the result. We observe that both the factorization and the preconditioner application time scale well, albeit with a small uptick in time for the largest test case. The number of CG steps is also close to constant, only doubling from the smallest to largest test case. Overall spaND is scaling similarly to Hypre and is more competitive in terms of time-to-solution. Figure 11 shows time spent at every level. Since the problem is nearly 2D, ranks are growing slowly and most of the time is spent in the first levels.

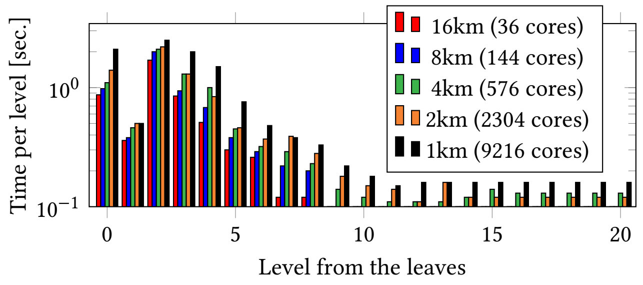


Figure 11: Ice-sheet profile, weak scalings, from 36 to 9,216 cores. From 12 levels (16 km, $N = 1$ M) to 20 levels (1 km, $N = 296$ M). Since this is a thin 3D problem, most of the work is located in the first levels.

Figure 12 shows the ranks across the physical domain. Our load balancing heuristic assigns vertices to MPI ranks based on the initial partitioning of A. In particular, it does not try to predict ranks, which are directly related to the number of flops. However, in this problem, we observe that the ranks are very uniform. Strong localization of high ranks would lead to poor load balancing.

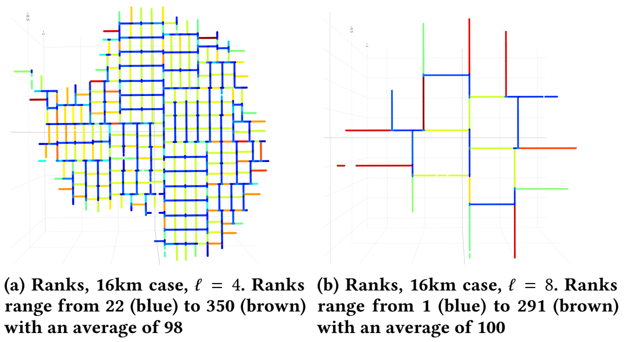

(a) Ranks, 16km case, $\ell = 4$. Ranks range from 22 (blue) to 350 (brown) with an average of 98 **(b) Ranks, 16km case, $\ell = 8$. Ranks range from 1 (blue) to 291 (brown) with an average of 100**

Figure 12: Ice-sheet ranks at various levels of the hierarchy.

Figure 13 shows the rank across levels for all problem sizes. We display both the average and maximum ranks. As expected on 2D-like problems, we see that ranks grow very slowly with the problem size. Average ranks show less than a 10% increase between problems of size N and $4N$. We however observe that the maximum ranks show large values towards the leaves. This is due to imperfections in the partitioning at the boundary of the domain. However, this effect is limited to the first few levels and does not impact the overall runtime of the algorithm.

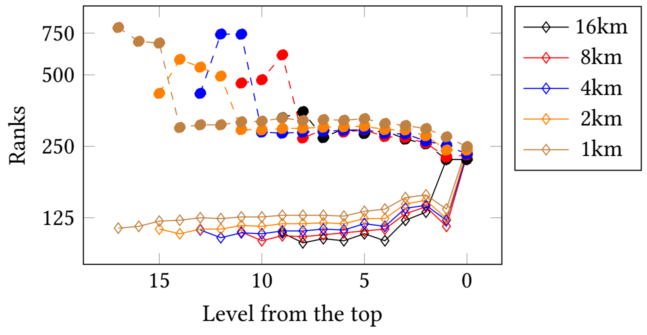


Figure 13: Ice-sheet ranks for all problem sizes. Averages (solid line, open diamond) and maximums (dash line, solid circle). The y-scale is logarithmic.

4 BENEFITS AND LIMITATIONS OF THE `TTOR` APPROACH

In this work, we parallelized spaND using `TTor`. This presents a couple of advantages but has some limitations. We here discuss some of those.

- The approach is well suited for (sparse) tiled matrix algorithms, where every block operation generates a corresponding task in the DAG. This makes writing the PTG code simple since there is a one-to-one mapping between tasks and block operations. In this work, for example, every block operation between interiors and/or interfaces generates a task.
- The code is simpler if all shared data structures are preallocated. Tasks can then just allocate and fill individual dense blocks with the appropriate values at runtime. Changing shared data structures at runtime may introduce data races and require mutexes.
- `TTor` is easy to use when dependencies are simple to compute. In spaND, dependencies depend only on the matrix graph (in which there is an edge between i and j if block A_{ij} is non-zero). As such, knowledge of the neighbors is enough to compute the in-degree of every task and to know what tasks to fulfill at the end of a task. This is similar to a classical MPI-based code for numerical partial differential equation (PDE) solvers, where every processor needs to know about processors that own neighboring vertices of the mesh.
- In the largest test cases studied, the DAG is very large near the leaves. In this case, the PTG+AMs approach taken by `TTor` works well since the DAG is entirely distributed and explored in parallel.

- Since TTor is based around message passing and MPI, it is fully compatible with any MPI-based code & library. In related work, we integrated spaND into the SU2 [13] codebase without any particular hurdle. In terms of programming difficulty, it was equivalent to integrating an MPI-based library into SU2. This feature is important to accelerate the adoption of task-based runtime systems into legacy codes.
- TTor's approach is based on message passing. This means every block in the matrix is resident on a particular MPI rank. There is no built-in dynamic load balancing capability that migrates tasks and data across the system while computations are happening. This may pose problems when the computational load is difficult to predict. In this case, the runtime system does not provide any specific solution to address this issue.
- Composing codes in PTG runtime systems is typically difficult. For example, in spaND we introduced an unnecessary synchronization point between levels to simplify the management of dependencies between levels. Similarly, if we have for example a PTG RRQR library, we have the option of introducing a synchronization point before and after the call to RRQR, or we need to account for the dependencies for all the inputs and outputs of RRQR. This can make it difficult to develop modular libraries while avoiding synchronization points. Further developments in TTor can make this process of coupling independently developed task-based libraries more seamless.

5 CONCLUSIONS

In this work, we designed a task-based distributed version of the spaND algorithm using TTor. We explained the implementation and performed both strong and weak scaling experiments. Strong scaling benchmarks exhibit good performances on problems where the top ranks are not too large (e.g., 2D or quasi-2D problems). Weak scaling benchmarks exhibit excellent performances on very large problems where ranks grow slowly with N. This demonstrates the good performances of a task-based approach using TTor, and show that TTor scales very well on large numbers of cores & nodes.

Future work for this project should be centered around addressing the large ranks arising on large 3D problems. This is the main limitation of our approach at the moment. A naive RRQR is difficult to parallelize in a distributed setting. However, results such as [22] indicate that random-sampling based approaches should work well.

ACKNOWLEDGMENTS

Léopold Cambier was supported by a fellowship from Total S.A. and Eric Darve by grant 80NSSC18M0152 from NASA, "Scalable Hierarchical CFD Solvers for Future Exascale Architectures". The authors thank Mauro Perego for help with the ice-sheet test cases.

REFERENCES

[1] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. 2015. Task-based multifrontal QR solver for GPU-accelerated multicore architectures. In *22nd international conference on high performance computing (HiPC)*. IEEE, 54–63.

[2] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. 2016. Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems. *ACM Trans. Math. Software* 43, 2 (2016), 1–22.

[3] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. 2009. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. In *Journal of Physics: Conference Series*, Vol. 180. IOP Publishing, 012037.

[4] Emmanuel Agullo, Luc Giraud, and Stojce Nakov. 2016. Task-based sparse hybrid linear solver for distributed memory heterogeneous architectures. In *European Conference on Parallel Processing*. Springer, 83–95.

[5] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198.

[6] E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine. 2012. The Zoltan and Isorropia Parallel Toolkits for Combinatorial Scientific Computing: Partitioning, Ordering, and Coloring. *Scientific Programming* 20, 2 (2012), 129–150.

[7] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, et al. 2011. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In *International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE, 1432–1441.

[8] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra. 2013. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science Engineering* 15, 6 (2013), 36–45.

[9] Léopold Cambier, Chao Chen, Erik G. Boman, Sivasankaran Rajamanickam, Raymond S. Tuminaro, and Eric Darve. 2020. An algebraic sparsified nested dissection algorithm using low-rank approximations. *SIAM J. Matrix Anal. Appl.* 41, 2 (2020), 715–746.

[10] Léopold Cambier, Yizhou Qian, and Eric Darve. 2020. TaskTorrent: a Lightweight Distributed Task-Based Runtime System in C++. In *2020 IEEE/ACM 3rd Annual Parallel Applications Workshop: Alternatives To MPI+ X (PAW-ATM)*. IEEE, 16–26.

[11] Chao Chen, Hadi Pouransari, Sivasankaran Rajamanickam, Erik G Boman, and Eric Darve. 2018. A distributed-memory hierarchical solver for general sparse linear systems. *Parallel Comput.* 74 (2018), 49–64.

[12] MA. Christie, MJ. Blunt, et al. 2001. Tenth SPE comparative solution project: A comparison of upscaling techniques. In *SPE reservoir simulation symposium*. Society of Petroleum Engineers.

[13] Thomas D. Economon, Francisco Palacios, Sean R. Copeland, Trent W. Lukaczyk, and Juan J. Alonso. 2016. SU2: An open-source suite for multiphysics simulation and design. *AIAA Journal* 54, 3 (2016), 828–846.

[14] Robert D. Falgout and Ulrike Meier Yang. 2002. Hypre: A library of high performance preconditioners. In *International Conference on Computational Science*. Springer, 632–641.

[15] Mark Gates, Jakub Kurzak, Ali Charara, Asim YarKhan, and Jack Dongarra. 2019. SLATE: design of a modern distributed and accelerated linear algebra library. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–18.

[16] Pascal Hénon, Pierre Ramet, and Jean Roman. 2002. PASTIX: a high-performance parallel direct solver for sparse symmetric positive definite systems. *Parallel Comput.* 28, 2 (2002), 301–321.

[17] Kenneth L. Ho and Lexing Ying. 2016. Hierarchical interpolative factorization for elliptic operators: differential equations. *Comm. Pure Appl. Math.* 69, 8 (2016), 1415–1451.

[18] Francisco D. Igual, Ernie Chan, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, Robert A. Van De Geijn, and Field G. Van Zee. 2012. The FLAME approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations. *J. Parallel and Distrib. Comput.* 72, 9 (2012), 1134–1143.

[19] George Karypis, Kirk Schloegel, and Vipin Kumar. 1997. *Parmetis: Parallel graph partitioning and sparse matrix ordering library.* Technical Report. University of Minnesota.

[20] Kyungjoo Kim and Victor Eijkhout. 2014. A parallel sparse direct solver via hierarchical DAG scheduling. *ACM Trans. Math. Software* 41, 1 (2014), 1–27.

[21] Xavier Lacoste, Mathieu Faverge, George Bosilca, Pierre Ramet, and Samuel Thibault. 2014. Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. In *International Parallel & Distributed Processing Symposium Workshops*. IEEE, 29–38.

[22] Yingzhou Li and Lexing Ying. 2017. Distributed-memory hierarchical interpolative factorization. *Res. Math. Sci.* 4, 1 (2017), 12.

[23] Fengguang Song, Asim YarKhan, and Jack Dongarra. 2009. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. IEEE, 1–11.

[24] Irina K. Tezaur, Mauro Perego, Andrew G. Salinger, Raymond S. Tuminaro, and Stephen F. Price. 2015. Albany/FELIX: a parallel, scalable and robust, finite element, first-order Stokes approximation ice sheet solver built for advanced analysis. *Geo. Model Dev. (Online)* 8, 4 (2015).

[25] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. 2010. Dense linear algebra solvers for multicore with GPU accelerators. In *International Symposium*

on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW). IEEE, 1–8.