2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)

Evaluation of Programming Models to Address Load Imbalance on Distributed Multi-Core CPUs: A Case Study with Block Low-Rank Factorization

Yu Pei*, George Bosilca*, Ichitaro Yamazaki†, Akihiro Ida‡, Jack Dongarra*
*University of Tennessee, Innovative Computing Laboratory, USA
†Sandia National Laboratories, Albuquerque, New Mexico, USA
‡University of Tokyo, Information Technology Center, Japan

***Abstract*—To minimize data movement, many parallel applications statically distribute computational tasks among the processes. However, modern simulations often encounters irregular computational tasks whose computational loads change dynamically at runtime or are data dependent. As a result, load imbalance among the processes at each step of simulation is a natural situation that must be dealt with at the programming level. The de facto parallel programming approach, flat MPI (one process per core), is hardly suitable to manage the lack of balance, imposing significant idle time on the simulation as processes have to wait for the slowest process at each step of simulation.**

One critical application for many domains is the LU factorization of a large dense matrix stored in the Block Low-Rank (BLR) format. Using the low-rank format can significantly reduce the cost of factorization in many scientific applications, including the boundary element analysis of electrostatic field. However, the partitioning of the matrix based on underlying geometry leads to different sizes of the matrix blocks whose numerical ranks change at each step of factorization, leading to the load imbalance among the processes at each step of factorization.

We use BLR LU factorization as a test case to study the programmability and performance of five different programming approaches: (1) flat MPI, (2) Adaptive MPI (Charm++), (3) MPI + OpenMP, (4) parameterized task graph (PTG), and (5) dynamic task discovery (DTD). The last two versions use a task-based paradigm to express the algorithm; we rely on the PaRSEC runtime system to execute the tasks. We first point out programming features needed to efficiently solve this category of problems, hinting at possible alternatives to the MPI+X programming paradigm. We then evaluate the programmability of the different approaches, detailing our experience implementing the algorithm using each of the models. Finally, we show the performance result on the Intel Haswell–based Bridges system at the Pittsburgh Supercomputing Center (PSC) and analyze the effectiveness of the implementations to address the load imbalance.

I. Introduction

Scientific simulations from many domains are utilizing high-performance computers to parallelize the workload and speed up knowledge discovery. Traditionally, these applications are implemented with a Flat MPI model coupled in the vast majority of cases with a static data distribution. A static mesh partition or domain decomposition could lead to imbalanced workloads, especially when the workload can change dynamically. Moreover, the explicit synchronization introduced in the Message Passing Interface (MPI) programming model invariably results in significant idle time under dynamically imbalanced workload.

The computational and storage costs of the dense matrix operation can be reduced significantly using a low-rank format. More precisely, Block Low-Rank (BLR) partitions the matrix in 2-D blocks and compresses the off-diagonal blocks using their low-rank representations, leading to a smaller need for storage space and a lower computational intensity. Thus, the use of a low-rank format can drastically shorten the factorization time, a highly desirable property for critical algorithms for as long as the error can be bound. Solution of a large-scale, diagonal, dominant dense linear system of equations is needed for a number of scientific and engineering simulations, and BLR format enables simulation of larger scale, which would not have been practical using the dense format, either due to the storage or to the computational costs.

One such application is the LU factorization of a dense matrix stored in the BLR format [1]. We have observed that geometry-based matrix partitioning compresses the matrix well, leading to many off-diagonal blocks with small numerical ranks, and therefore a lower computational cost. In a 2-D block-cyclic dense distribution, data is mostly evenly distributed across participating processes, leading to well balanced—both in terms of memory and computation—factorizations [2]. However, the compressed format does not inherit the even balance of the dense algorithm, leading to an algorithm that, while similar to the dense counterpart, is unbalanced and dynamic in memory needs, communications, and computation. An implementation of this algorithm using MPI exacerbated this imbalance due to its tightly coupled nature, where an explicit synchronization is necessary at each factorization step. It also highlighted that the accumulated idle time due to the explicit synchronization at each step of factorization can be significantly greater than the load imbalance in the total local computation time among the

†Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy National Nuclear Security Administration under contract de-na0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

processes. Moreover, the dynamic nature of each block rank during execution makes it difficult to statically distribute the blocks among the processes to reduce the load imbalance. Alternative, more dynamic, approaches are necessary to cope with the imbalance, and deliver efficient executions in distributed environments.

In this paper, we explore the computer science aspect of this highly dynamic problem, and try to understand how different programming approaches compare while supporting such an imbalanced application. We look simultaneously at the metric of programmability and the more objective metric of performance. More precisely, we evaluate five different programming models for implementing the BLR LU factorization of a dense matrix, arising from the boundary element analysis of electrostatic field:

1) The **Flat MPI** model with blocking collective operation, which leads to synchronization at each step of factorization,
2) The Adaptive MPI (**AMPI**) model, an implementation of the MPI standard on top of Charm++ that supports over-decomposition and dynamic load balancing [3],
3) The **MPI+OpenMP** tasking model, where both the computational and communication tasks are dynamically scheduled in order to remove the synchronization points of our flat MPI implementation,
4) The Dynamic Task Discovery (**DTD**) model [4] where the algorithm is described sequentially as a series of tasks and the runtime build the data dependency graph dynamically, and
5) The Parametrized Task Graph (**PTG**) model where the algorithm has a dataflow description as a parameterized graph of tasks.

For DTD and PTG, we use the distributed-memory runtime system PaRSEC [5], a runtime that can dynamically move data among processes to satisfy dependencies and schedule the available tasks.

We evaluate the programmability of each model, commenting on the experience of transitioning from the original flat MPI BLR LU implementation to task-based programming models. We then analyze in detail the performance, focusing on the effectiveness of each programming model to address the load imbalance, overlap communications and computations, and, more globally, reduce the factorization time. We intend this work to be a guide for parallel application developers to provide a path to avoid performance pitfalls with the MPI+X programming model, while describing a possible path to alternate programming models. Simultaneously, the data movement patterns and dependencies we expose represent the backbone of a large class of algorithms, and can be used by parallel programming researchers when developing new features on their next-generation programming models.

II. Related Works

Besides the BLR format, several other low-rank formats have been proposed, including $\mathcal{H}$-matrix [6] and Hierarchical Off-Diagonal Low-Rank (HODLR) [7] formats, and their nested variants $\mathcal{H}^2$-matrix [8] and HSS [9] formats. There are also multi-level low-rank formats with the lattice structures [10], [11]. Among those formats, the $\mathcal{H}$-matrix has the most general low-rank format, leading to the near-linear complexity of the factorization. However, its irregular hierarchical block structure poses a challenge when parallelizing the factorization on a distributed-memory computer. To simplify the parallelization and improve the scalability, BLR abandons the hierarchy, but comes with the price of higher storage and computational complexities (e.g., $\mathcal{O}(n^{1.5})$ storage and $\mathcal{O}(n^2)$ computational complexities for the BLR factorization of a dense matrix of dimension n [12], compared with $\mathcal{O}(n \log n)$ and $\mathcal{O}(n \log^2 n)$ complexities with the $\mathcal{H}$-matrix format [6]). Nevertheless, for factorizing a small-scaled matrix in practice (e.g., $n = \mathcal{O}(10^5)$), the BLR and $\mathcal{H}$-matrix formats often have similar costs of factorization.

The BLR's simpler flat low-rank format brings the potential for higher computational performance. However, for solving a practical problem with an irregular partitioning of the matrix, the parallel scalability of the BLR factorization can be greatly limited by the load imbalance among the processes, even on a small number of processes (e.g., tens or hundreds of processes). It is then the responsibility of the programming paradigm to provide developers with the means to efficiently handle such imbalance, either by shifting it around the participating processes or by overlapping multiple, possibly partially dependent, iterations.

The BLR format has been used for distributed multi-frontal sparse factorization [13]. In the Hierarchical Computations on Manycore Architectures (HiCMA) library, the StarPU runtime [14] was used to improve the performance of the distributed BLR Cholesky factorization [15]. However, the load balance issues of the low-rank factorization have not been explicitly studied. Previously, load balancing issues in generating and performing the matrix vector multiply with the $\mathcal{H}$-matrix have been studied [16]. Compared to matrix generation and multiplication, the factorization has more complex dataflow, and for matrix multiplication, the numerical ranks of the blocks do not change.

In terms of comparing programming models, [17] compared UPC++ with a Partitioned Global Address Space (PGAS) implementation of direct linear solvers for sparse symmetric matrices with two state-of-the-art ones and showed favorable results. [18] directly compares several task-based runtime system using a set of benchmarks to help application developers make informed decisions on the transition from MPI+X models. However, all these efforts dealt with regular and certainly less dynamic applications, and this study will complement their findings using a BLR factorization. More recently, a more comprehensive benchmarking suite comparing multiple parallel programming approaches has been proposed [19].

III. Block Low-rank Factorization Algorithm

To store the matrix in BLR format, our implementation uses a geometric-based partitioning algorithm [20] (to obtain high compression rate of the matrix) and tolerance-based

```
for $k = 1, 2, \ldots, n_t$ do
    //Factorize diagonal block
    $[P_k, L_{k,k}, U_{k,k}] := \mathrm{LU}(B_{k,k})$
    for $i = k+1, \ldots, n_t$ do
        //Compute blocks in panel column
        $L_{i,k} := B_{i,k} U_{k,k}^{-1}$
    end for
    for $j = k+1, \ldots, n_t$ do
        //Compute blocks in panel row
        $U_{k,j} := L_{k,k}^{-1} P_k B_{k,j}$
    end for
    for $i = k+1, \ldots, n_t$ do
        for $j = k+1, \ldots, n_t$ do
            //Update trailing block
            $B_{i,j} := B_{i,j} - L_{i,k} U_{k,j}$
        end for
    end for
end for
```

(a) LU factorization, where n_t is the numbers of the blocks in the matrix row or column.

```
$\Pi_{\text{row}} = \emptyset$, $\Pi_{\text{col}} := \emptyset$, $r := 0$, $\pi_1 := 1$
while not converged do
    // increment numerical rank
    $r := r + 1$
    // generate pivot row
    $\mathbf{y}_{:,r} := \mathbf{b}_{\pi_r,:}^T - \mathbf{y}_{:,1:r-1} \mathbf{v}_{\pi_r,1:r-1}^T$
    // pick pivot column
    $\pi_r := \arg\max_j (|y_{j,r}| : j \notin \Pi_{\text{col}})$
    $\Pi_{\text{col}} := \Pi_{\text{col}} \cup \{\pi_r\}$
    // generate pivot column
    $\mathbf{v}_{:,r} := \mathbf{b}_{:,\pi_r} - \mathbf{v}_{:,1:r-1} \mathbf{y}_{\pi_r,1:r-1}^T$
    // pick pivot row
    $\pi_r := \arg\max_i (|v_{i,r}| : i \notin \Pi_{\text{row}})$
    $\Pi_{\text{row}} := \Pi_{\text{row}} \cup \{\pi_r\}$
    // convergence check
    $\|E\| := \|V_{:,1:r}\| \|Y_{:,1:r}\|$
    if $r == 1$ then $\|A\| := \|E\|$
    if $\|E\| \le \tau \|A\|$ then break;
end while
```

(b) ACA compression to compute a low-rank VY^T form of a block B.

Fig. 1. Low-rank matrix factorization and compression algorithms.

recompression [21], [22] during the factorization, for all the low-rank off-diagonal blocks. For the applications of interest, with LU, when the matrix is properly ordered and partitioned, many of the off-diagonal blocks can be well approximated using small ranks. As a result, when n is the dimension of the coefficient matrix, BLR has the potential to reduce the storage and computational complexities of factorization to $\mathcal{O}(n^{1.5})$ and $\mathcal{O}(n^2)$ from $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$ when using the dense matrix format, respectively [12]. All diagonal blocks are stored in the dense format and treated as dense with regard to computations.

At each step of factorization, we first compute the LU factorization of the leading dense diagonal block using the LAPACK subroutine `dgetrf`. Then, the off-diagonal blocks of the leading block row and column, commonly known as *panels*, are factorized using the BLAS triangular solve `dtrsm` with the lower- and the upper-triangular factors of the diagonal block, respectively. These panel blocks are then used to update the trailing submatrix block by block. Figure 1(a) shows the resulting factorization algorithm.

In BLR format, the off-diagonal blocks can be either low-rank or dense. Thus, when updating the trailing blocks $B_{i,j}$ on Figure 1(a), each of the three blocks involved, $B_{i,j}$, $L_{i,k}$, and $U_{k,j}$ can be either dense or low-rank, giving eight potential configurations for the updating kernel. We update these blocks according to the approach that would minimize the floating-point operation (FLOP) count (see Figure 2 for an illustration). Compared with the dense-block update that requires $O(n_i n_j n_k)$ FLOPs, the low-rank update only requires $O(n_i n_j \min(r_{i,k}, r_{k,j}))$ FLOPs, where $r_{i,k}$ and $r_{k,j}$ are the respective numerical ranks of the blocks $L_{i,k}$ and $U_{k,j}$, and n_k is the dimension of the k-th diagonal block. As a result, when the blocks have small ranks (i.e., $r_{i,k}, r_{k,j} \ll n_k$), low-rank compression can significantly reduce the FLOP count.

To avoid the increase in the numerical rank while maintaining the user-specified accuracy, we use Adaptive Cross Approximation (ACA) [21], [22] to recompress the low-rank block after each update. As shown in Figure 1(b), at each step of ACA, we compute the pivot row (and column) by multiplying the corresponding row of $\widehat{Y}_{i,j}$ (and $\widehat{V}_{i,j}$) with $\widehat{V}_{i,j}$ (or $\widehat{Y}_{i,j}^T$). Thus, we do not explicitly form the dense representation of the whole low-rank block. The algorithm terminates when the user-specified accuracy of the approximation is obtained. As a result, the numerical rank of each block may change at each step of the factorization.

Our LU implementation seeks pivots only within the diagonal block, ignoring the potential pivots outside the diagonal blocks. This pivoting scheme (combined with the matrix balancing) was sufficient to maintain the numerical stability of the factorization for matrices arising from the applications we are interested in.

IV. Required Features

We highlight some of the most critical features needed in order to implement an efficient BLR factorization algorithm. Most of these requirements are generic enough to be applied disregarding the programming model, but some are particular for task-based models.

1) Address load imbalance at each step due to variable task granularities (different sizes of blocks whose numerical ranks change dynamically).
2) Allow dynamic reallocation of the data that define the data dependencies among the tasks (to store the low-rank block whose numerical rank changes, e.g., the numerical rank could increase).
3) Handle the dynamically changing size of the data to be sent or received (to send the low-rank block whose numerical rank is known only at run time).

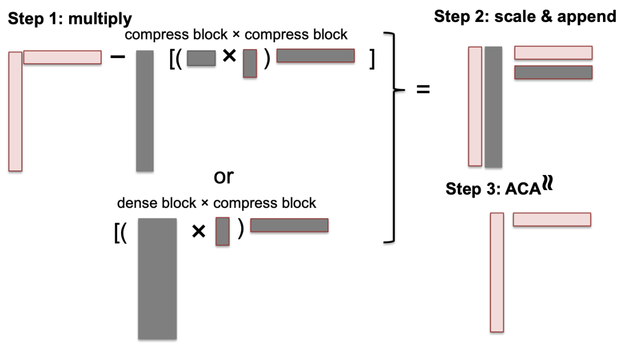


Fig. 2. Illustration of algorithm updating a low-rank block. When a dense block $B_{i,j}$ is updated using two low-rank blocks, $L_{i,k} = V_{i,k} Y_{i,k}^T$ and $U_{k,j} = V_{k,j} Y_{k,j}^T$, we first compute the small matrix $T := Y_{i,k}^T V_{k,j}$. We then multiply T with either $V_{i,k}$ or $Y_{k,j}^T$, depending on the required FLOP counts. Finally, $B_{i,j}$ is updated with the low-rank matrix, e.g., $B_{i,j} := B_{i,j} - V_{i,k}(T Y_{k,j}^T)$. Similarly, to update a dense block using a low-rank block and a dense block, we first merge the dense block into the low-rank block, e.g., $B_{i,j} := B_{i,j} - V_{i,k}(Y_{i,k}^T B_{k,j})$. On the other hand, if $B_{i,j}$ is a low-rank block, we can then directly merge the low-rank representation of the update with the original low-rank representation of $B_{i,j}$, i.e., $B_{i,j} := \widehat{V}_{i,j} \widehat{Y}_{i,j}^T$, where $\widehat{V}_{i,j} = [V_{i,j}, -\bar{V}_{i,j}]$ and $\widehat{Y}_{i,j} = [Y_{i,j}, \bar{Y}_{i,j}]$, and $V_{i,j} Y_{i,j}^T$ is the original low-rank representation of $B_{i,j}$ before the update, while $-\bar{V}_{i,j} \bar{Y}_{i,j}^T$ is its low-rank update to be applied.

4) Provide the means to overlap communication with computation (e.g., using a communication thread). A fork–join programming model (e.g., with MPI + OpenMP) without dedicated communication tasks or threads may not be sufficient.
5) The ability to specifically highlight the critical path of the algorithm, and prioritize its execution.
6) Define task or data dependencies at runtime (e.g., depends on the input matrix due to empty blocks, though the dependencies are not changed during the factorization).
7) Support heterogeneous systems (e.g., the ability to offload work to GPUs) and can manage devices tasks automatically

Most of the target programming models have some level of support for these features, even if in some instances the burden of handling concurrency (or potential parallelism) is on the developer. All MPI-based approaches (flat-MPI, MPI+OpenMP, and AMPI) claim support for asynchronous, or non-blocking, communications and for collective communications. In addition, AMPI supports load balancing via migration of computations to a less busy peer and communication computation overlap by a highly oversubscribed approach, but behaves the same as MPI for other features.

At the current stage, PaRSEC supports all but (3), where on the receiver a fixed size temporary buffer is used. In addition, we did not use feature (2) in our current PaRSEC implementations of BLR factorization (we only send the required data, but need a larger buffer). Thus, our current PaRSEC implementation requires two additional parameters for specifying the maximum numerical rank of each block, and specifying the size of the buffer (i.e., the minimum rank r_{min} and the ratio r_{rate} with respect to the block sizes such that the maximum rank for the (i, j)-th block is given by $\max(r_{\text{min}}, r_{\text{rate}} \cdot \min(n_i, n_j)))$. While these parameters could have been the target of an autotuning campaign, we use in this study default values that are selected by the developers; they might not be optimal but they should be relatively close. In our experiments, the maximum rank is set such that it is larger than the ranks chosen during the factorization, leading to a larger memory requirement for the PaRSEC implementation compared with the other implementations.

V. MPI-based Implementations

In the following sections we describe the design and distributed-memory implementation of an optimized version of the BLR LU algorithm using different programming models. First, we explore an MPI-based version (Flat MPI), and then extend it with the integration of OpenMP (MPI+OpenMP). To facilitate the handling of the imbalance and minimize the waiting time, we also explored an oversubscribed model for the Flat MPI approach using Charm++ AMPI layer.

A. Flat MPI Programming Model

To parallelize the BLR factorization on distributed-memory computers, our first implementation follows the ScaLAPACK LU implementation and is based on the Flat MPI programming model. We arrange the MPI processes on a p-by-q 2-D grid and distribute the blocks in a Two-Dimensional Block-Cyclic (2DBC) fashion among the processes (each block is stored in a contiguous memory region). Then, to factorize the matrix, each process updates and factorizes only its local blocks.

To gather the nonlocal blocks that are needed to update the local blocks from another process, each process creates two MPI sub-communicators: one for the processes in the same column of the process grid and the other for the processes in the same row. Then, at each factorization step, the blocks in the current panel are broadcasted using these two sub-communicators. Since the numerical rank of a low-rank block can change after each update, the processes involved in the broadcast must be informed of the size of the data prior to the broadcast, so the communication of the low-rank block is divided into two messages: the first message propagates the current numerical rank, and the second message the low-rank block data.

The LU with local pivoting is relatively simple to implement in the Flat MPI programming model especially with the 2DBC distribution. However, the collective communications required for the panel update that executes within the panel sub-communicators introduce a synchronization at each factorization step. When load imbalance exists among the processes at each factorization step (e.g., for the trailing submatrix update due to the different sizes and types of the blocks), many processes will idle waiting for the slowest process at these synchronization points, leading to a significant performance lost. We evaluated the effects of standard techniques (e.g., lookahead, accumulated update with multiple panels, balanced block sizes) without significant benefits.

B. Flat MPI with Charm++/AMPI

Charm++ is a runtime system that builds on three main concepts: 1) over-decomposition, where the work and data are decomposed to more than the number of available processing elements; 2) asynchronous message-driven execution, where a ”process” (chare in the Charm++ lingo) never wastes the physical resources while waiting on communication completions, by allowing other ”processes” to take over the physical cores and progress their own work; and 3) migratability, where the data and work can move among the processing elements. Combining these three features provides the potential to dynamically balance the load and hide the communication.

AMPI provides an MPI implementation that is built on top of the Charm++ framework. It uses user-level threads instead of OS processes to allow several MPI processes on a single physical core, providing the benefits mentioned above to the MPI code. It has been shown that AMPI can improve the performance of the Flat MPI implementation for many imbalanced applications and benchmarks [3] [23].

Porting the Flat MPI implementation to use AMPI requires minimal effort. We only need to change the name of the main routine to `mpi_main`, and to switch the compiler and linker to the ones required by AMPI. Setting the oversubscription factor could be challenging, but in our case the imbalance was reproducible and relatively enough to allow us to tune the oversubscription parameter manually. Our expectation was that the oversubscription would be highly beneficial, as the MPI processes are spending a significant amount of their execution time blocked on `MPI_Bcast`, and thus another process on the same node could then utilize the physical core for computations—thus reducing the idle time of the core.

C. OpenMP Task Programming Model

To manually remove the synchronization points, our second implementation relies on the OpenMP task programming model. Then, at run time, the OpenMP scheduler executes both computational and communication tasks of the factorization as their dependencies are resolved. In our implementation, we cannot use the memory pointers to the required data to track the data dependencies among tasks because the compressed blocks are dynamically freed and reallocated as their numerical ranks change after each update. Instead, we used a separate n_t-by-n_t integer array to keep track of the task dependencies.

With our implementation, the OpenMP runtime manages the dependency graph of only the local tasks, and does not form the global dependency graph of the factorization. Hence, the tasks are scheduled for the execution once all the local dependencies are resolved. However, when the task needs to communicate blocks with other processes, the thread will call `MPI_Bcast` either to send the local block (its current numerical rank and then the data) or to receive the non-local block. Thus, these tasks may block until the corresponding communication task is scheduled on other processes, contributing to the idling time of the core.

In order to reduce the number of tasks that are blocked due to the call to `MPI_Bcast` and are keeping the core idle, we implement nested parallelization. In this implementation, a single task updates all the blocks in one block column, but once it is scheduled to execute the update, it launches the child tasks, each of which updates one of the blocks in the column. To integrate nicely and maximize the performance of MPI in a multi-threaded environment, we applied some of the techniques described in [24]. We create a separate communicator for each thread (to minimize the cost of MPI matching and the potential for message overtaking) and use the communicators in a round-robin fashion on the block columns at each step of factorization. We place a higher priority on factorizing the panel column and updating the next panel column since all the tasks updating the blocks depend on the panel (see Figure 3).

In order to factorize a large matrix, the MPI buffers used to store the non-local blocks need to be deallocated once all the tasks that require the blocks have completed. Thus, we insert the tasks that set and decrement the counter for each non-local block, and once the counter becomes zero, the task deallocates the block.

The BLR factorization has a relatively simple dependency graph, and the computational kernel, which each task executes, has been already separated into its own subroutine for our

```
#pragma omp parallel
#pragma omp master
{
  // start pipeline (factor 1st panels)
  factorPanel(0, A);
  for (int k = 1; k < A.getNt(); k++) {
    lookaheadUpdateA(k-1, A);

    // factor next panel
    factorPanel(k, A);

    // update remaining submatrix
    // using current (k-1)th panel
    remainingUpdateA(k-1, A);
  }
}
```

(a) BLR factorization.

```
int *tileA = A.getTile(k, k);
int *tileB = k == 0 ? A.getTile(k, k) : \
                      A.getTile(k-1, k);

#pragma omp task priority(1) \
                 depend(in:tileB[0:1]) \
                 depend(inout:tileA[0:1])
{
 // factor diagonal
 if (A.isLocalRow(k) || A.isLocalCol(k)) {
     A.factorDiagBlock( k );
 }
 // compute off-diagonal L
 if (A.isLocalCol( k )) {
     for (int i = k+1; i < A.getMt(); i++) {
         if (A.isLocalRow( i )) {
             #pragma omp task priority(1)
             {
                 A.computeL(i, k);
             }
         }
     }
     #pragma omp taskwait

     if (!A.isLocal(k, k)) {
         A.freeBuffer(k, k);
     }
 }
 // broad cast tiles in panel along the rows
 A.iBcastL(k);
}
```

(b) Factor diagonal block and nested tasks for panel column update

Fig. 3. OpenMP task implementation of BLR factorization: `depend` clause is used to specify the data dependencies among the tasks, where `A.getTile(k, k)` returns the pointer to keep track of the (k, k)-th block. Line 11 is a blocking call that factors the diagonal and broadcast the data to panel row/column. lookaheadUpdateA and remainingUpdateA have similar structure, where we create an OMP task for the update column. In that task, we solve the panel for U, broadcast it down the column, then create nested tasks to compute individual updates.

Flat MPI implementation. Thus, it did not present a significant challenge to integrate OpenMP tasks to the sequential code. Furthermore, since many of the application codes already use OpenMP, this OpenMP implementation does not require any change to compile the code. Overall, the tasking improved the performance of Flat MPI by removing the synchronization points and reducing the idling time of the cores due the load imbalance. However, correctly scheduling the communication tasks for optimal performance remained a challenge. As the process count increases, it becomes more difficult to coordinate these communication tasks, and some tasks may be blocked on the communication, keeping the cores idle.

VI. PaRSEC Implementation

PaRSEC [5] is a distributed runtime system capable of scheduling tasks on heterogeneous resources and handle data movements internally. Tasks are placed based on distributed data placement, and task migration across nodes is not supported. Multiple domain specific languages (DSLs) can be built on top of the runtime system, sharing a common set of infrastructures (scheduler, communication engine, data representation). Out of the currently available DSLs we use for this study two: the Parameterized Task Graph and the Dynamic Task Discovery.

A. Dynamic Task Discovery Model

DTD allows the sequential task insertions into the PaRSEC runtime, hence providing a simpler to use API, capable of describing distributed algorithms. To use DTD, a user must specify the distribution of the data that the tasks operate on, the dependency among the tasks through their data usage, and the code that the task executes once all the required data becomes available. DTD can deliver reasonable levels of performance on small- and medium-sized platforms [4], but has scalability issues when the number of participating processes become too large. Recent study in StarPU [25] has demonstrated that by pruning the task graph it is possible to delay the task insertion bottleneck, allowing sequential task insertion model to scale to a larger number of processes. Although DTD could benefit from such optimization, we did not implement it in our current version.

In order to free us from manually moving data among the processes and managing a temporary buffer for the non-local data, we ported our OpenMP implementation to use the DTD interface in PaRSEC. Since DTD provides the sequential insert task interface, as can be seen in Figure 4, our DTD implementation resembles our OpenMP implementation. Thus, it was straightforward to implement. At each step, we first insert the diagonal factorization task `dgetrf`. We then compute the off-diagonal blocks of the panel by inserting the triangular solve tasks `dtrsm_l` and `dtrsm_u` for each off-diagonal block in the lower and upper triangular factors, respectively. Finally, we insert the tasks to update each block in the trailing sub-matrix. To recycle the temporary buffer for the non-local data, we call `data_flush` when the non-local data is no longer needed.

In order to transition our MPI implementation to use the PaRSEC runtime, we had to describe the data distribution in the PaRSEC data descriptor format. Though the PaRSEC data collections can be more dynamic and support non-regular, non-2DBC distributions, we decided to restrict the PaRSEC data collection to a regular 2DBC distribution, which our MPI implementation uses, using the API provided by PaRSEC.

BLR factorization requires the runtime system to dynamically change the size of the data being sent or received since the numerical rank of the block changes during the factorization. DTD provides this capability by enabling us to specify the size of the data in the task body. We use this feature such that our implementation sends only the required amount of data specified by the current numerical rank. Similar to Flat MPI, the dynamic size of the blocks imposes an increased communication load, as the size of the blocks must be propagated before sending the block data.

In order to maintain the minimum amount of the memory usage, we would also like to reallocate the data as the low-rank block is recompressed. PaRSEC provides a flexible data descriptor that supports irregular data sizes, which allows the reallocation of the data to accommodate rank changes. Our current implementation does not use this functionality. Instead, we specify a maximum rank for each low-rank block to avoid the reallocation at the cost of higher memory consumption.

```
for(k = 0; k < NT; k++){
  // diagonal DGETRF
  insert_task(taskpool, parsec_dgetrf,
    1, "getrf",
    sizeof(int)  , &k                 ,VALUE,
    PASSED_BY_REF, TILE_OF(A, k, k) ,INOUT | AFFINITY,
    PASSED_BY_REF, TILE_OF(IP, k, 0),OUTPUT,
    PARSEC_DTD_ARG_END);
  if(k < NT-1){
    for(int i = k+1; i < NT; i++){
      insert_task(taskpool, parsec_dtrsm_l,
                  ...);
      insert_task(taskpool, parsec_dtrsm_u,
                  ...);
    }
    data_flush(dtd_tp, TILE_OF(A, k, k));
    data_flush(dtd_tp, TILE_OF(IP, k, 0));

    for(int i = k+1; i < NT; i++){
      for(int j = k+1; j < NT; j++){
        insert_task(taskpool, parsec_dgemm,
                    ...);
      }
    }
  }
}

int parsec_dgemm(parsec_execution_stream_t *es,
                 parsec_task_t *this_task) {
  int k, i, j;
  double *A, *B, *C;
  parsec_dtd_unpack_args(this_task, &k, &i, &j,
                         &descA, &A, &B, &C);
  int rankA = (int)A[0]; // rank of non-local block A
  int rankB = (int)B[0]; // rank of non-local block B
  int mb = descA->super.nbi[i]; // # of rows in block C
  int nb = descA->super.nbi[j]; // # of cols in block C
  // perform update
  ...
  // update the output message size
  new_count = rank * (mb + nb) + 1;
  dtd_update_count_of_flow(this_task, 2, new_count);
}
```

Fig. 4. PaRSEC DTD implementation of BLR factorization, insertion of the tasks in sequential order with the data usage information provided. We show how the data usage is specified only for the `dgetrf` task: it executes the code `parsec_dgetrf` that takes three arguments `k`, `A`, and `IP`, where the diagonal block `A` and pivoting `IP` are passed in by references. The `INOUT` flag indicates that the data needs to be read and will be written. `AFFINITY` flag indicates that this task will be executed on the process that owns the k-th diagonal block. `PARSEC_DTD_ARG_END` signals the end of parameters list. DTD provides an API `dtd_update_count_of_flow` to update the size of the data to be sent in the task body.

B. *Parameterized Task Graph Model*

PTG uses a concise, parametrized task graph description known as Job Data Flow (JDF) to represent the dependencies between tasks. As we will show in the later section, the developer needs to specify for each task class: 1) the data distribution, 2) the possible input parameter values, 3) the process that will execute the task based on the data distribution, 4) the data dependency between the task classes and 6) the actual code body of the task. Based on these pieces information, PaRSEC can discover and execute all the available tasks at run time, moving the data as the tasks are completed—without exploring the whole task graph at once. Previous results have shown that PTG can deliver a significant percentage of the hardware peak performance on heterogeneous distributed machines [26].

In the dataflow description of the PTG DSL, each computational task is defined by a set of parameters and a number of input and output flows of data. Unlike in the DTD implementation, the PTG model requires the programmer to express the data dependencies between tasks as mathematical relationships between the tasks' parameters. These data dependencies, along with the shape and size of the data, must be specified and agreed on by the pair of tasks that is sending and receiving the data.

Figure 5 shows the JDF specification of the diagonal factorization task, where the parameter k defines the task (for factorizing the k-th diagonal block). The second line specifies the range of the parameter, showing that all integer values between 0 and the last diagonal index, NT, are legal for the parameter k. On the third line, the locality statement specifies that the k-th diagonal factorization task will be executed by the process that owns the specified data (i.e., the k-th diagonal block). Finally, the data dependencies for the tasks are defined (the data can be initialized by reading from the memory, outputted to the memory, or passed in or to another task).

For the computational task to be executed, once all the input flows are locally available, we can simply call (in the BODY) the computation kernels developed for the Flat MPI implementation.

Given this dataflow expression in JDF format, the PTG preprocessor generates the C/C++ code that encodes the symbolic task representation. Then, at run time, the PaRSEC runtime explores the task graph, moves the specified data between the tasks, and executes the tasks as all the required data become available—without the overhead of task discovery, which our DTD implementation has to pay.

Our PTG and DTD implementations use the same data distribution descriptor, allowing a smooth transition from the DTD to PTG implementation. From programmability perspective, PTG introduces a completely different parallelization philosophy, driven by data dependencies and not by control dependencies. For most of HPC users, converting their parallel applications (e.g., parallelized with MPI and OpenMP) might require decent amount of effort. However, the description provides enough information to the runtime itself to allow for automatic communication and computation overlap, as well as collective pattern description, providing a strong base for more scalable and more efficient implementations.

```
dgetrf(k)
k = 0 .. NT

: descA(k, k) //locality

RW A  <- (FIRST) ? descA(k,k)
      <- (!FIRST) ? C dgemm(k_prev, DIAG, DIAG)

      -> (END>=START) ? A dtrsm_l(k, START..END)
      -> (END>=START) ? A dtrsm_u(k, START..END)

RW IP <- IP ipiv_in(k)  [type = PIVOT count = NB]
      -> IP ipiv_out(k) [type = PIVOT count = NB]

/* Priority */
;1

BODY
{
  // Factorizing diagonal block (k, k)
  int mb = descA.nbi[k];
  double *dA = &(((double*)A)[1]);
  iinfo = LAPACKE_dgetrf(LAPACK_COL_MAJOR,
                         mb, mb, dA, mb, ipiv);
}

dgemm(k, i, j)
...
  RW C <- (k == 0) ? descA(i, j) : C dgemm(k-1, i, j)
                         [count = COUNT_C]
       -> (k == lastk && i == j) ? A dgetrf(m)
                         [count = COUNT_C]
...

/* Priority */
;(j == k+1 ? 1 : 0)

BODY
{
  ...
  // update the output message size
  this_task->locals.COUNT_C.value = 1 + ranks * (mb + nb);
}
```

Fig. 5. PaRSEC PTG specification of the diagonal factorization tasks: defining the parameter space, data locality, and data dependencies, written in JDF. In the figure, "RW" specifies that these diagonal factorization tasks both read and write the data (equivalent to "INOUT" in DTD), while the left-arrow and right-arrow show where the data is read from and written to at the completion of the task, respectively: e.g., for reading the data A, "descA(k,k)" indicates that the data is read from the memory at the initialization, while "C dgemm(k_prev, DIAG, DIAG)" indicates that the task dgemm(k_prev, DIAG, DIAG) will send the data C, which the diagonal factorization uses as A. The "type" combined with "count" indicates the temporary buffer size for sending and receiving the data. It is possible to change the size of the data to be sent (e.g., when the numerical rank changes after the recompression), by changing the local value passed to "count" in BODY.

VII. Performance Evaluation

In addition to evaluating the effort needed for each implementation qualitatively, here we compare the performance of the models quantitatively.

A. *Experimental Setup*

For our experiments, we used a software package called ppohBEM [27] that numerically solves the integral equations for simulating the electrostatic field based on the boundary

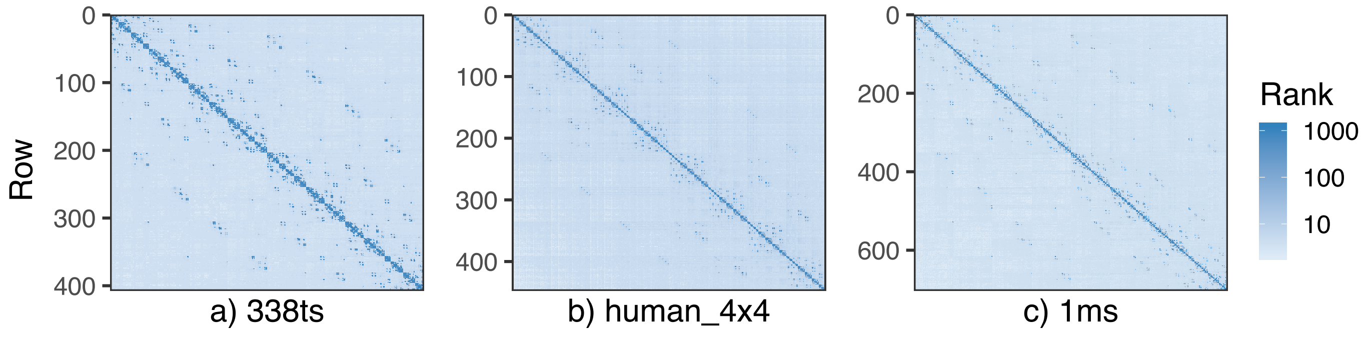
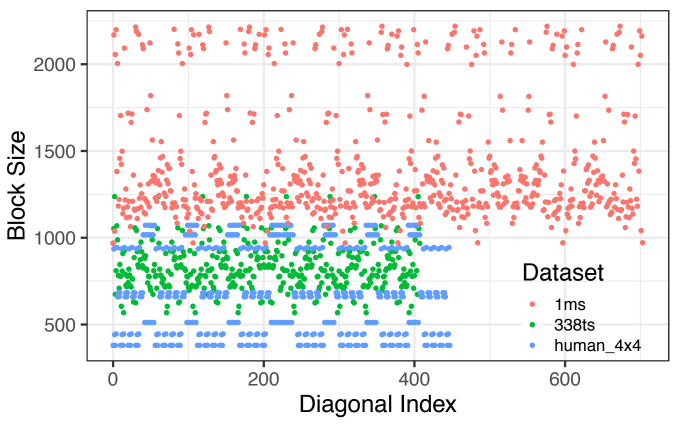


Fig. 6. Initial block ranks for each test matrix, all have dense tiles near diagonal, but different off-diagonal low rank patterns



(a) Diagonal square blocks sizes, and the off-diagonal blocks will be rectangular

	matrix dimension	# of block columns
338ts	338000	408
human_4x4	314624	448
1ms	10004400	704

(b) Test matrices

Fig. 7. Test matrices information

element method. In particular, we used the BLR matrices generated by the software package called HACApK [20], which uses the low-rank matrix format for solving dense linear systems of equations. To compute the appropriate matrix permutation and partition for generating the low-rank matrix, HACApK uses the geometrical information associated with the underlying physical problem such that the off-diagonal blocks of large dimensions become low-rank. Figure 7 shows the size information of our test matrices, and their initial numerical ranks are shown in Figure 6.

We compiled the entire software stack using Intel Parallel Studio XE 2019 suite and linked with the corresponding MPI and OpenMP library. We used PaRSEC library in the master branch as of June 2019 with DTD `dtd_update_count_of_flow` API in development branch, and the release version 6.9.0 of Charm++. Our experiments were conducted on Bridge cluster located at Pittsburgh Supercomputing Center (PSC). Each compute node has 2 Intel Haswell (E5-2695 v3) CPUs with 14 cores per CPU, running at 2.3–3.3 GHz, and are interconnected using Intel Omni-Path.

Experiments were run using all 28 cores per node starting from one node through up to 16 nodes (448 cores), which were enough to show the overall performance trend. Results for the `1ms` dataset starts from 4 nodes due to memory constraint. For the Flat MPI model, each core has one process; for MPI+OMP, the best configuration we observed is with two processes per socket (with the socket cores evenly divided between the processes), so a total of four MPI processes per node. For the AMPI model we use the SMP mode, with two processes per node, each with 14 threads, and set the virtual process number to be three times the physical core counts. Finally, for PaRSEC implementations we have one process per node with one core dedicated to communication thread, and the rest as computation threads. To avoid non-uniform memory access (NUMA) effect for data accesses, PaRSEC data must be initialized from all the threads.

B. Experimental Results

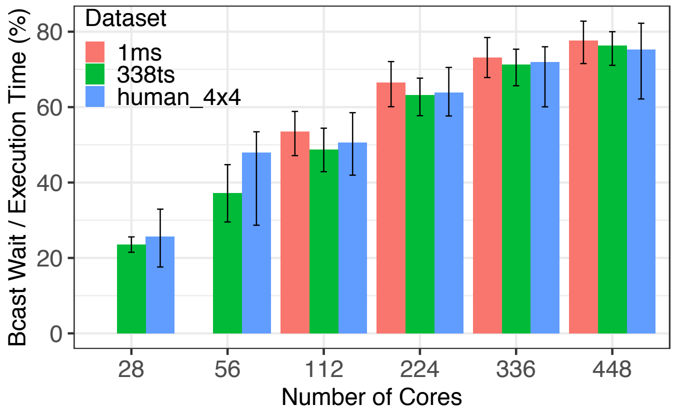


Fig. 8. The average wait time of a MPI process in a collective call for the flat MPI model, shown as percentage of total execution time. Minimum and maximum shown as well

1) Flat MPI: In our experiments with the 2DBC distribution of the blocks, the total computational load was well balanced among the processes. However, at each step of the factorization, the different sizes and ranks of the blocks created significant load imbalances among the processes. Since our Flat MPI implementation introduces global synchronization, all the processes have to wait for the slowest processes, and the accumulated idle time due to the load imbalance can become significant in the total factorization time. This observation had motivated us to explore alternative programming models besides Flat MPI.

Figure 8 illustrates these load imbalances for the three test matrices. To measure the imbalance, we put a barrier before each broadcast and accumulated this wait time for each process. As shown in the figure, the average idle time can be as high as 77% of the execution time, while the error bars indicate that the total computational load among the processes has a much smaller variation for most cases.

Thus, the existence of such large imbalance opens opportunities for oversubscription-based approaches to translate this wasted waiting time into useful computation time for another thread. Moreover, in the case where over 50% of the time is wasted in average on all processes, it seems extremely plausible that oversubscription could drastically reduce the wasted time and therefore minimize the time-to-solution.

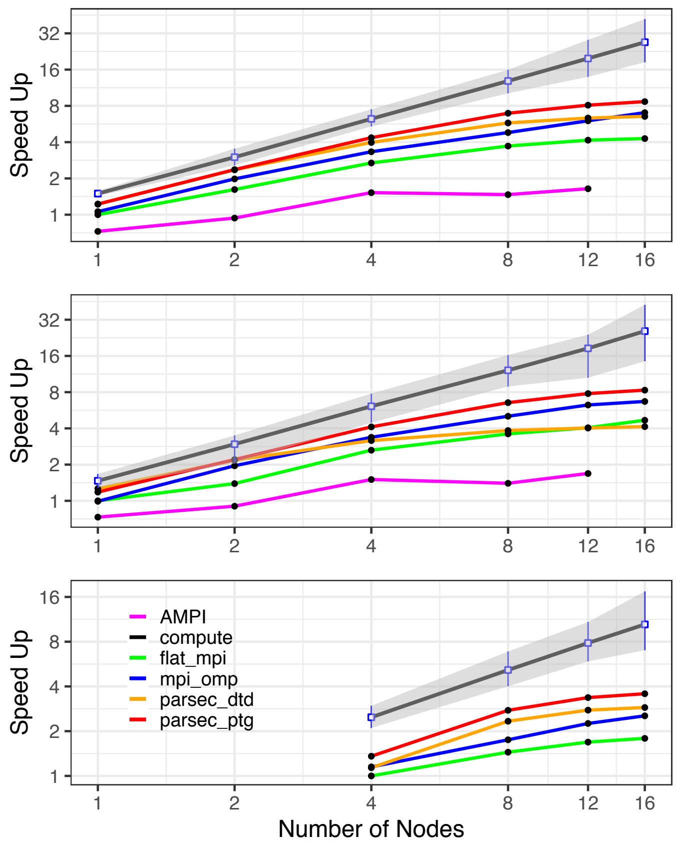


Fig. 9. Execution time of each model on different datasets, top) 338ts, middle) human_4x4, bottom) 1ms. Flat MPI performances on 1 node (28 cores)/4 nodes are used as base to show speed up of the models. They are 317, 221 and 1050 seconds respectively. Both X- and Y-axis are plotted on log2 scale. PTG speed up over MPI+OMP are 1.23, 1.24 and 1.40 at 16 nodes

2) AMPI: In Figure 9 the green and the pink lines show the results comparing AMPI with Flat MPI. By oversubscribing the cores, we expect AMPI to be able to reduce the idle time, thus achieving better performance than the Flat MPI model. Unfortunately, the result contradicts our expectation. To investigate why the AMPI is taking more time to execute, we timed the different sections in the Flat MPI/AMPI implementation using a smaller test dataset on a single node (28 processes, 84 virtual processes). Figure 10 shows the trace for one process. FactorDiag includes `dgetrf` and the resulting broadcast to panel row and column. PanelUpdate computes the panel, BCastPanel broadcasts the panel blocks to the corresponding column or row processes. UpdateRemain is the computation of the update kernel on trailing submatrix.

Since we oversubscribe by 3:1, AMPI's UpdateRemain time is roughly 1/3 of Flat MPI's time. But the AMPI BCastPanel time is much larger and is the reason for the longer execution time. We varied the oversubscription factor from 1 to 5, and 3 was the best configuration.

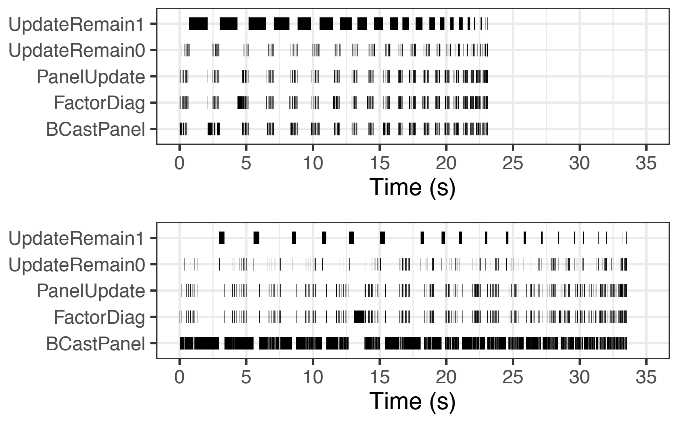


Fig. 10. Execution stream of the different sections for one selected process, top) Flat MPI, bottom) AMPI. Most of the BCastPanel time are likely idle time

3) MPI+OMP: In Figure 9, in addition to the factorization time, the black line shows the average total compute time for each process, as obtained from the Flat MPI model result. It also has ticks for the minimum and maximum among the processes as well. The line serves as an unattainable lower bound of the execution time, as it represent the most favorable scenario, one that only accounts for computational costs and completely disregard all costs related to data movements.

The first thing we notice is that Flat MPI model performs the worst among the remaining tested programming models, all other models perform better at some degree. This is expected as the strongest point of all the other approaches is to somewhat relax the strong synchronization inherent to the Flat MPI model. Second, MPI+OMP scales well as the number of nodes increases, but there are still limitations to prevent it from obtaining better performance, as we will analyze later.

4) PaRSEC DTD: The DTD implementation performs better than MPI+OMP at the beginning, which can be attributed to its finer-grain dependency, further removing the synchronization imposed on block columns. But PaRSEC DTD has it's own issues, mainly with regard to the scalability of the sequential task insertion. As the node count increases, the performance begins to deviate from that of the PaRSEC PTG version to finally become worse than the Flat MPI result for the human 4×4 dataset on 16 nodes. We believe that as we strong scale, the overhead of PaRSEC DTD task discovery starts to take a bigger portion of the execution time, and the discovery and insertion of local tasks being slower than the execution of already inserted tasks. DAG trimming technique proposed in [25] likely can help mitigate the problem but is not implemented in this case.

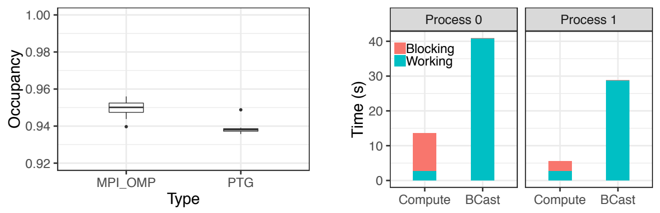


Fig. 11. left) Computation kernels occupancy summary of all the threads, right) Detail breakdown of the diagonal factorization task for MPI+OMP model

5) PaRSEC PTG: The PTG implementation performs consistently better than the other models. Not only it removes all global synchronizations (replacing them with fine grain synchronizes at the task level), but also creates more opportunity for communication computation overlap; it has the features allowing us to specify higher priorities to diagonal tasks. Given the disparity between diagonal/off-diagonal computation loads, this capability ensures high levels of parallel workloads and cores occupancy by directing the runtime to follow, even losely, the algorithmic critical path [28].

To understand the improvement from MPI+OMP to PaRSEC PTG, we profiled the executions on a single node. MPI+OMP was run with two processes, each on a socket. The left plot in Figure 11 shows a summary of each thread's occupancy information, defined as the summation of all the computation kernels' time on a thread divided by the total execution time. On a single node, both models achieve over 90%, and we can attribute the rest roughly to runtime overhead.

But a closer look at the diagonal factorization kernel in MPI+OMP reveals that all the processes in the current panel will call this kernel in order to receive the actual diagonal factorization. Root process will thus complete the computation then block on the broadcast to panel row and column, while the receiving processes will directly block on the broadcast. The right plot in Figure 11 shows that if the kernel is doing only broadcast (not the root), it takes as long as needed to complete the broadcast and exit. But on root process it will block on the broadcast for additional time after the computation.

On the other hand, PaRSEC delegates all the communication to a single thread dedicated to communications and uses non-blocking communications. It removes this synchronization point and likely provided the performance benefits we observe in Figure 9 as we scale.

VIII. Conclusion

In this paper, we implemented BLR LU factorization as a test case using five programming models: Flat MPI, MPI+OMP and alternative models AMPI/Charm++, PaRSEC DTD, and PaRSEC PTG. We summarized our experience implementing the algorithm using these models, and evaluated their respective performance. The results indicate the potential for the task based approach to address the load imbalance and outperform Flat MPI. Overall PaRSEC PTG achieved the best execution time and scalability, with a certain cost on the programming effort. PaRSEC DTD provides a smoother transition to task-based runtime but face scalability issues as the number of nodes grows. MPI+OMP can obtain reasonable results and might be more familiar and easier to implement. AMPI's result for this test case is unexpected and warrants further investigation.

Several features needed for an efficient BLR factorization are highlighted, including necessary capabilities to address load imbalance, handle dynamic data sizes, reduce synchronization, and provide the ability to highlight the algorithm critical path. We hope that this work can motivate future adoption of alternative programming models to tackle the irregular workloads arising from the system or the application.

Our current implementation is designed as a benchmark to compare different programming models. It is possible to further optimize some of the implementations. For example, instead of using the 2DBC distribution, we may evenly distributed the dense tiles close to the diagonal among the processes, which may greatly improve the load balance and improve performance. Since PaRSEC handles all the data movement, the user just needs to define a new data distribution, making it easy to use a different data distribution. Other programming models like Task-aware MPI (TAMPI) [29] implements the interoperability services between MPI and OpenMP tasks, and can be further investigated. We observed a higher memory consumption of PaRSEC based approaches due to the temporary buffer used in the runtime, a quantitative evaluation in this aspect will also be interesting in the future. Finally, GPU kernels can also be added to offload work and speed up computation.

Acknowledgments

This work was supported in part by Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures and High Performance Computing Infrastructure in Japan (Project ID: jh180012), JSPS KAKENHI Grant Numbers #17H01749 and #17K19962, the National Science Foundation under Grant No. 1740250, and the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, under prime contract #DE-AC05-00OR22725, and UT Battelle subaward #4000152412 and #4000151974.

References

[1] P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L'Excellent, C. Weisbecker, Improving multifrontal methods by means of block low-rank representations, SIAM Journal on Scientific Computing 37 (2015) A1451–A1474.

[2] J. Demmel, L. Grigori, M. Hoemmen, J. Langou, Communication-optimal parallel and sequential qr and lu factorizations, SIAM J. Sci. Comput. 34 (1) (2012) 206–239. doi:10.1137/080731992.
URL `http://dx.doi.org/10.1137/080731992`

[3] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, L. Kale, Parallel programming with migratable objects: Charm++ in practice, in: SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2014, pp. 647–658.

[4] R. Hoque, T. Herault, G. Bosilca, J. Dongarra, Dynamic Task Discovery in PaRSEC: A Data-flow Task-based Runtime, in: Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA '17, ACM, New York, NY, USA, 2017, pp. 6:1–6:8.

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

[6] W. Hackbusch, A sparse matrix arithmetic based on $\mathcal{H}$-matrices, part I: Introduction to $\mathcal{H}$-matrices, Computing 62 (1999) 89–108.

[7] S. Ambikasaran, E. Darve, An O(N log N) fast direct solver for partial hierarchically semi-separable matrices, Journal of Scientific Computing 57 (2013) 477–501.

[8] W. Hackbusch, B. Khoromskij, S. A. Sauter, On $\mathcal{H}^2$-matrices, in: H. Bungartz, R. Hoppe, C. Zenger (Eds.), Lectures on Applied Mathematics, Springer Berlin Heidelberg, 2000.

[9] S. Chandrasekaran, M. Gu, T. Pals, A fast ULV decomposition solver for hierarchically semiseparable representations, SIAM Journal on Matrix Analysis and Applications 28 (3) (2006) 603–622.

[10] P. Amestoy, A. Buttari, J.-Y. L 'excellent, T. Mary, Bridging the gap between flat and hierarchical low-rank matrix formats: the multilevel BLR format, Tech. Rep. hal-01774642, University of Manchester (2018).

[11] I. Yamazaki, A. Ida, R. Yokota, J. Dongarra, Distributed-memory lattice h-matrix factorization, The International Journal of High Performance Computing Applications (2019).

[12] P. Amestoy, A. Buttari, J. L'Excellent, T. Mary, On the complexity of the block low-rank multifrontal factorization, SIAM Journal on Scientific Computing 39 (4) (2017) A1710–A1740.

[13] P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, T. Mary, Performance and scalability of the block low-rank multifrontal factorization on multicore architectures, ACM Trans. Math. Softw. 45 (1) (2019) 2:1–2:26.

[14] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: a unified platform for task scheduling on heterogeneous multicore architectures, Concurrency and Computation: Practice and Experience 23 (2) (2011) 187–198.

[15] K. Akbudak, H. Ltaief, A. Mikhalev, A. Charara, A. Esposito, D. Keyes, Exploiting data sparsity for large-scale matrix computations, in: M. Aldinucci, L. Padovani, M. Torquati (Eds.), Euro-Par 2018: Parallel Processing, Springer International Publishing, Cham, 2018, pp. 721–734.

[16] T. Hiraishi, K. Munakata, S. Bai, A. Ida, Dynamic load balancing for construction and arithmetic of hierarchical matrices, presented at SIAM Conference on Parallel Processing for Scientific Computing (2018).

[17] J. Bachan, D. Bonachea, P. H. Hargrove, S. Hofmeyr, M. Jacquelin, A. Kamil, B. van Straalen, S. B. Baden, The UPC++ PGAS Library for Exascale Computing, in: Proceedings of the Second Annual PGAS Applications Workshop, PAW17, ACM, New York, NY, USA, 2017, pp. 7:1–7:4.

[18] R. Hoque, P. Shamis, Distributed Task-Based Runtime Systems - Current State and Micro-Benchmark Performance, in: 2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2018, pp. 934–941.

[19] E. Slaughter, W. Wu, Y. Fu, L. Brand enburg, N. Garcia, W. Kautz, E. Marx, K. S. Morris, W. Lee, Q. Cao, G. Bosilca, S. Mirchandaney, S. Treichler, P. McCormick, A. Aiken, Task Bench: A Parameterized Benchmark for Evaluating Parallel Runtime Performance, arXiv e-prints (2019) arXiv:1908.05790arXiv:1908.05790.

[20] A. Ida, T. Iwashita, T. Mifune, Y. Takahashi, Parallel hierarchical matrices with adaptive cross approximation on symmetric multiprocessing clusters, Journal of Information Processing 22 (2014) 642–650.

[21] S. Kurtz, O. Rain, S. Rjasanow, The adaptive cross-approximation technique for the 3-D boundary-element method, IEEE Trans. Magn. 38 (2002) 421–424.

[22] M. Bebendorf, S. Rjasanow, Adaptive low-rank approximation of collocation matrices, Computing 70 (2003) 1–24.

[23] S. Bak, H. Menon, S. White, M. Diener, L. V. Kalé, Multi-level load balancing with an integrated runtime approach, in: 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018, Washington, DC, USA, May 1-4, 2018, 2018, pp. 31–40.

[24] T. Patinyasakdikul, D. Eberius, G. Bosilca, N. Hjelm, Give MPI Threading a Fair Chance: A Study of Multithreaded MPI Designs, in: 2019 IEEE International Conference on Cluster Computing, 2019, pp. 1–11.

[25] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, S. P. Thibault, Achieving high performance on supercomputers with a sequential task-based programming model, IEEE Transactions on Parallel and Distributed Systems (2017).

[26] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, J. Dongarra, Hierarchical DAG Scheduling for Hybrid Distributed Systems, in: 2015 IEEE International Parallel and Distributed Processing Symposium, 2015, pp. 156–165.

[27] T. Iwashita, A. Ida, T. Mifune, Y. Takahashi, Software framework for parallel BEM analyses with $\mathcal{H}$-matrices using MPI and OpenMP, in: Proceedings of the International Conference on Computational Science, 2017, pp. 12–14.

[28] Q. Cao, Y. Pei, K. Akbudak, A. Mikhalev, G. Bosilca, H. Ltaief, D. Keyes, J. Dongarra, Extreme-scale task-based cholesky factorization toward climate and weather prediction applications.
URL `http://hdl.handle.net/10754/656453`

[29] K. Sala, X. Teruel, J. M. Perez, A. J. Peña, V. Beltran, J. Labarta, Integrating blocking and non-blocking mpi primitives with task-based programming models, Parallel Computing 85 (2019) 153 – 166.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

Experiments were performed on the Bridge system at Pittsburgh Supercomputing Center (PSC). We compiled the entire software stack using Intel Parallel Studio XE 2019 suite and linked with the corresponding MPI and OpenMP library. We used PaRSEC library in the master branch as of June 2019 PaRSEC bitbucket Link with DTD dtd_update_count_of_flow in a development branch, and the release version 6.9.0 of Charm++ Link. The implementations of the algorithm is in a private bitbucket repository at the moment.

Experiments were run using all 28 cores per node starting from one node through up to 12 nodes (336 cores). We repeated several sample runs and did not observe significant performance variation. So in the result plot only one is shown.

ARTIFACT AVAILABILITY

Software Artifact Availability: —Some author-created software artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

Hardware Artifact Availability: —There are no author-created hardware artifacts.

Data Artifact Availability: – There are no author-created data artifacts.

Proprietary Artifacts: (One of these options remains.)
— None of the associated artifacts, author-created or otherwise, are proprietary.

List of URLs and/or DOIs where artifacts are available:

(1) https://bitbucket.org/icldistcomp/parsec
(2) http://charm.cs.uiuc.edu/software

CONSIDERATION FOR SCC:

(Authors did not reply.)

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Operating systems and versions: GNU Linux

Compilers and versions: Intel Parallel Studio XE 2019 suite

Applications and versions: HACApK for Block low rank generation

Libraries and versions: Charm++ 6.9.0

Key algorithms: LU factorization

Input datasets and versions:

Paper Modifications:

Output from scripts that gathers execution environment information.

ARTIFACT EVALUATION

Verification and validation studies: We calculate the backward error of the solution computed

Accuracy and precision of timings: Timing are done with MPI_Time or through the PaRSEC profiling system. Timing is more precise than microsecond.

Quantified the sensitivity of results to initial conditions and/or parameters of the computational environment: No significant system variations were observed

Controls, statistics, or other steps taken to make the measurements and analyses robust to variability and unknowns in the system.