2022 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)

An OpenMP-only Linear Algebra Library for Distributed Architectures

Carla Cardoso*, Hervé Yviquel*, Guilherme Valarini*, Gustavo Leite*, Rodrigo Ceccato*, Marcio Pereira*, Alan Souza†, and Guido Araujo*
*Institute of Computing, UNICAMP, Brazil
Email: {c212066,g168891,g264964,r176848}@dac.unicamp.br, {hyviquel,marciomp,guido}@unicamp.br
†Petrobras, Brazil
Email: alan.souza@petrobras.com.br

***Abstract*—This paper presents a dense linear algebra library for distributed memory systems called OMPC PLASMA. It leverages the OpenMP Cluster (OMPC) programming model to enable the execution of the PLASMA library using task parallelism on a distributed cluster architecture. OpenMP Cluster model is used to define the task regions that are then distribute across the cluster nodes by the OMPC runtime that automatically manages task scheduling, communications between nodes, and fault tolerance. The OMPC PLASMA library modifies various PLASMA functions to distribute the matrix across the nodes and perform the calculation using threads of the node. Experimental results show that OMPC PLASMA achieves 4.00x with 4 worker nodes, 7.00x with 8 worker nodes, and 12.00x with 16 worker nodes acceleration over its original implementation for a single node. A 3.00x speedup is achieved when comparing OMPC PLASMA execution to ScaLAPACK, for 4 worker nodes, and a matrix size of 90k×90k.**

***Index Terms*—Dense Linear Algebra, Distributed Memory Systems, OMPC Cluster, PLASMA.**

I. Introduction

Solving systems of linear equations is a central tool to address problems in many areas of knowledge like engineering, physics, chemistry, informatics and economics. Most of these problems rely on computing complex linear algebra operations over huge sets of data, resulting in costly matrices operations.

Many current linear algebra libraries use heterogeneous cluster architectures to leverage the performance of multiple CPU-GPU nodes. However, programming such architectures requires the combination of programming models such as message-based programming (e.g., MPI) [1], multi-threaded techniques, and GPGPU programming languages (e.g., CUDA) [2]. In this context, many linear algebra problems use task-based parallelism to improve readability and has shown better speedups [3], [4]. PLASMA is an example of a multi-core linear algebra library that uses tasks to execute its algorithms using the OpenMP syntax and BLAS routines [5]. Some works extended PLASMA for distributed memory architectures [6], [7], but they rely on a mix of programming languages or propose entirely new frameworks, resulting in complex and hard to maintain code.

To address this, we propose an extension to the PLASMA library using the OpenMP Cluster (OMPC) programming model [8]. OMPC is a distributed memory model that uses OpenMP syntax to offload computation to a computer cluster, providing transparent and automatic MPI-based data distribution, fault tolerance and load balancing. This approach allows us to propose a distributed PLASMA extension that relies on OpenMP-only syntax. The goal of this paper is to tailor two relevant PLASMA calls to execute on a distributed memory system using the OMPC runtime. We show that OMPC PLASMA leverage computing power from distributed worker nodes without proposing new frameworks or mixing programming models.

The rest of the paper is organized as follows. Section II details background information about task-based algorithms for linear algebra. Section III presents the OMPC PLASMA library. Section IV describes some related works. Section V shows the experimental results, and finally, Section VI concludes the work and proposes some future extensions.

II. Background

This section presents the solutions that laid the foundations for this work, including the development of tiled algorithms, the task-based programming paradigm, and the OMPC runtime, that offloads annotated tasks to CPU clusters.

A. Tiled Algorithms

Tiled algorithms process matrices using square sub-matrices smaller than the original ones, called tiles. These tiled computations are more eficient than whole-matrix operations since they maximize data reuse by favoring cache usage [4], [9]. This makes these algorithms efficient methods for implementing linear algebra operations on multi-core processors. The tiled algorithms for matrix decomposition such as LU, QR, or Cholesky factorization factor the diagonal tiles first and then operate the tiles below the diagonal block. Several libraries like PLASMA [5], FLAME [10], DPLASMA [11], and SLATE [6] use the task-based programming model to process such tiles. This allows the runtime to try to maximize the parallelization between the blocks and provide enough work to keep the processors busy.

B. Task-based programming models

In recent years, task-based programming has shown to be a scalable and flexible approach to extract regular and irregular parallelism from applications [12], [13]. In this model, users

978-1-6654-5157-4/22/$31.00 ©2022 IEEE
DOI 10.1109/SBAC-PADW56527.2022.00013

use specific constructs (e.g. OpenMP directives) to describe distinct units of code, called tasks, that can execute as soon as their dependencies are met. This model allows the programmer to concentrate on the logical dependencies between tasks and leave the computation scheduling to the runtime [14]. This programming model represents algorithms through Directed Acyclic Graphs (DAG), where the nodes are tasks, and the edges are dependencies between them.

C. Distributed Runtime based on Tasks

OMPC[1] is a task-based distributed runtime that transparently and automatically combines the best features of OpenMP with MPI [8]. With the OpenMP directive `target nowait`, it is possible to execute pieces of code on distributed systems. We thus chose OMPC for our proposal, as it abstracts the user from managing data and communications between tasks running on distinct worker nodes. OMPC supports HEFT [15] and Round-Robin algorithms as strategies for scheduling the tasks between the threads of the nodes, but other algorithms will be added in the future. OMPC automatically keeps data coherency across remote worker nodes using the tasks data dependencies. This data management system works for both OpenMP memory mapping methods for target devices - in our case, the remote CPU nodes. The first method is the `map` clause, that associates data to a given `target` region. This may lead to unnecessary data movement, since after each task defined in a target region the data is moved to or from the host. Alternatively, we can use the `enter/exit data map` constructs to specify data that is used by a set of tasks. With this method, as the tasks are scheduled for execution, the OMPC runtime automatically transfers data on-demand to the node that requires it. This control is handled by a dedicated head node, meaning that OMPC PLASMA will require a head node in addition to the number of worker nodes. In the rest of the paper we refer to the head node as the node that manages resources and the worker nodes as the nodes in which the computation kernels execute.

III. Proposal

This section introduces the linear algebra library, called OMPC PLASMA. OMPC allows us to use OpenMP 4.X semantics to describes tasks that are automatically offloaded across the cluster nodes. Our work was based on PLASMA tiled algorithms, written as tasks for the OMPC runtime.

Figure 1 describes the hierarchy of the OMPC PLASMA library. OMPC PLASMA has a set of parallel linear algebra routines that execute in parallel on distributed systems. It consists of OMPC runtime and PLASMA components. The OMPC runtime is responsible for managing resources and distributing work. The PLASMA component contains the routines for processing matrices. It uses the BLAS and LAPACK libraries to perform low-level procedures like basic linear algebra operations, such as matrix-matrix multiplication, matrix addition or row swapping. This component uses OpenMP to parallelize these procedures and leverage performance from multi-core architectures. Sections II and IV describe most of the third-party components. In this section, we will further explain the OMPC PLASMA component.

[1]OMPC is an open-source project freely available at https://ompcluster.gitlab.io/

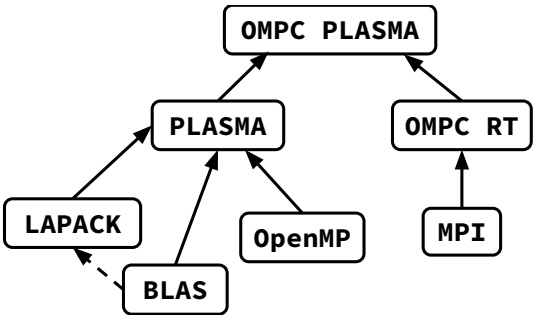


Fig. 1: OMPC PLASMA Software Hierarchy.

A. OMPC PLASMA Description

The original PLASMA library uses tiled algorithms with OpenMP directives to create tasks that correspond to BLAS kernels and task dependencies that correspond to data dependencies between the kernel computations [4], [5]. As PLASMA executes, the OpenMP runtime generates and assigns tasks to available CPU cores. To exploit the available cores and maximize performance, it is suggested to make square blocks sizes as small as possible. This will generate a DAG with more parallelism available, although incurred overhead from block sizes that are too small may degrade performance.

Alternatively, OMPC PLASMA targets distributed systems. To achieve this, we modified the PLASMA kernels to offload the computation across the remote worker nodes. As the execution is offloaded, OMPC PLASMA also leverages the multi-core resources from each node. To accomplish this, we replaced the OpenMP `#pragma omp task` directives with `#pragma omp target nowait` constructs to utilize the OMPC distributed runtime [8].

The OMPC PLASMA algorithm starts by dividing the matrix into square tiles. Then, the OMPC runtime schedules the routines that operate on these tiles to execute on remote nodes. Listing 1 shows an example of a routine that we modified to achieve distributed execution. As the listing shows, we modify the *pragmas* of the PLASMA functions labeled as `plasma_core_omp_routine`, which directly call the BLAS kernels.

```
1 void plasma_core_omp_routine( float *A, float *B, ...)
2 {
3   #pragma omp task depend(in:A[0:nb*nb])\
4                    depend(inout:B[0:nb*nb])
5     cblas_routine(CblasColMajor, A, nb, B, ldb, ...);
6 }
```

Listing 1: Annotated code with OpenMP in the original PLASMA.

Listing 1 shows how PLASMA implements the execution of a BLAS routine as an OpenMP task. Lines 3 to 5 create a task that executes a BLAS kernel, taking a tile A as input and a tile B as output. To change this snippet into a *target* region and thus utilize OMPC offloading for distributed execution, we changed the directives `#pragma omp task` to `#pragma omp target nowait`. Listing 2 exemplifies this modification on line 3. Since the two listings represent the same BLAS kernel, the `depend` clauses are the same. We also introduce *data* directives and the `firstprivate` clause to enable OMPC automatic data management. Data directives transmit the values of the matrices, and the `firstprivate` clause sends information such as the size of the blocks (nb) or the state of the process (Line 4 in Listing 2).

```
1  void plasma_core_ompc_routine( ...)
2  {
3    #pragma omp target nowait depend(in:*A,inout:*B)\
4        firstprivate(n,lda,...)
5      cblas_routine(CblasColMajor, ...., A, nb, B, ldb, ...);
6  }
7  void plasma_ompc_routine( ..., float *pA,float *pB, ...)
8  {
9    // Send data
10   for (...)
11     #pragma omp target enter data map(to: A_send[:nb*nb])\
12           depend(inout: *A_send) nowait
13     #pragma omp target enter data map(to: B_send[:nb*nb])\
14           depend(inout: *B_send) nowait
15   // OMPC PLASMA algorithm
16   for (...)
17     plasma_core_ompc_routine( ... , pA[i,j], pB[j,k] , ...)
18   // Exit data
19   for (...)
20     #pragma omp target exit data map(release: A_out[:nb*nb
         ])\
21             depend(inout: *A_out) nowait
22     #pragma omp target exit data map(from: B_out[:nb*nb])\
23             depend(inout: *B_out) nowait
24 }
```

Listing 2: Annotated code with OpenMP in OMPC PLASMA.

To correctly map the execution to remote nodes, we must also use the *data* directives for the sub-matrices. Lines 10 to 14 in Listing 2 show how sub-matrices are sent to worker nodes with the `enter data` directive. In the Listing 2, we use the `nowait` clause to delegate the asynchronous execution to the OMPC scheduler. We denote data that is read by remote workers with the `map(to: ...)` clause alongside `depend(in: ...)` to inform the OMPC runtime that it the transfer must occur before the kernel execution. In lines 19 to 23 the directive `exit data` tells the runtime that it can remove the corresponding data from the workers after the kernels execution. We use the `map(from: ...)` directive to list buffers that should be sent back to the head node, and `map(release: ...)` for buffers that can be deallocated. Finally, we use the `depend(out: ...)` to indicate that the data can be deleted as soon as all tasks have executed.When this happens, the tiles with the results are collected and stored in the matrices on the head node.

The OMPC runtime automatically handles the data movements, schedules and distributes the tasks among the cluster nodes and keep data coherency. Therefore, OMPC PLASMA must only describe the tasks and their dependencies. Next section explains how OMPC PLASMA performs the Cholesky factorization to further explain its workings.

B. A Test Case: Cholesky Factorization

This section illustrates our proposal with the tiled Cholesky factorization algorithm [16], [17]. The Cholesky factorization factors a square symmetric positive definite matrix A into the product of a lower triangular matrix L and its transpose ($A = LL^t$). The Cholesky factorization tiled algorithm consists of four BLAS kernel operations:

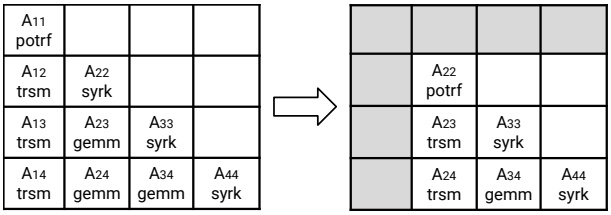


Fig. 2: A snapshot of block Cholesky factorization.

- **potrf:** Cholesky factorization.
- **trsm:** Solving triangular matrix with multiple right hand sides.
- **syrk:** Symmetric rank-k update to a matrix.
- **gemm:** Matrix-matrix multiplication.

Figure 2 shows how PLASMA executes the Cholesky algorithm on a 4×4 tiled matrix. The matrix on the left shows the first iteration of the algorithm. First, the algorithm runs the `potrf` kernel in tile A_{11}. This run fulfills the dependencies and thus unlocks the `trsm` tasks for all tiles below tile A_{11}. Each execution of `trsm` unlocks some `syrk` and `gemm` tasks from their corresponding row. After the first iteration, this process is repeated in the lower 3×3 matrix with the tile A_{22}, as shown in the matrix on the right side of Figure 2. This process is repeated for all the diagonal tiles.

Algorithm 1 PLASMA tiled Cholesky algorithm

```
1: procedure PLASMA_POTRF(A)
2:     #pragma omp parallel
3:     #pragma omp master
4:     for (k = 0; k < NB; k++) do
5:         plasma_core_omp_potrf(A_kk)
6:         for (m = k+1; m < NB; m++) do
7:             plasma_core_omp_trsm(A_kk, A_mk)
8:         end for
9:         for (n = k+1; n < NB; n++) do
10:            plasma_core_omp_syrk(A_nk, A_mn)
11:            for (m = n+1; m < NB; m++) do
12:                plasma_core_omp_gemm(A_mk, A_nk, A_mn)
13:            end for
14:        end for
15:    end for
16: end procedure
```

Algorithm 1, called `plasma_potrf`, shows the pseudocode of the Cholesky algorithm in PLASMA and Algorithm 2 the pseudocode for the same operation in OMPC PLASMA. The algorithm follows the steps explained above, starting with the `plasma_core_omp_potrf` function that

calls the BLAS kernel potrf and continuing with the following routines. PLASMA uses the OpenMP directives in the `plasma_core_omp_*` functions to create the tasks with the respective BLAS kernels (Listing 1). As Section III-A explains, we modified the `plasma_omp_core_*` functions to allow Algorithm 1 to execute across distributed nodes, by changing the `#pragma omp task` directive to `#pragma omp target nowait`. In Cholesky's case, the algorithm only uses a single matrix and, depending on the user specification, the `enter data` directive sends the upper or lower part of the matrix, as shown in line 5 of Algorithm 2. On Line 14, the directive `exit data` indicates that the final result must be brought back to matrix A before the execution finishes.

Algorithm 2 OMPC PLASMA tiled Cholesky algorithm

```
1: procedure PLASMA_OMPC_POTRF(A)
2:     #pragma omp parallel
3:     #pragma omp single
4:     ▷ Send the tiles of matrix A to the worker nodes
5:     for (k = 0; k < NB; k++) do
6:         #pragma omp target data enter map(to:A_kk) depend(inout:*A_kk) nowait
7:     end for
8:     ▷ Perform the Cholesky factorization
9:     for (k = 0; k < NB; k++) do
10:        plasma_core_ompc_potrf(A_kk, nb_inner)
11:        ...
12:    end for
13:    ▷ Copy the tiles from matrix A to the head node
14:    for (k = 0; k < NB; k++) do
15:        #pragma omp target data exit map(from:A_kk) depend(inout:*A_kk) nowait
16:    end for
17: end procedure
```

Figure 3 shows the task graph created by OMPC when running the modified Cholesky algorithm on 3 worker nodes with 2 threads on each node, using a matrix of 1000×1000, with a block size of 250×250. The matrix is divided into sixteen tiles, from which ten are used, corresponding to the lower part of the matrix.

In Figure 3, the tasks identified as `slipt` and `join` are local OpenMP tasks executed by the head node. The `split` tasks are the operations in charge of copying parts of the matrix in the blocks, and the `join` tasks copy the results of the blocks to the original matrix. The octagonal shapes represent the tasks in charge of sending the blocks to the respective nodes (Line 5 of Algorithm 2). The OMPC scheduler executes these tasks on the nodes that will use this data to minimize overall communication. The tasks identified in hexagonal shapes represent the removal of blocks at nodes (Line 14 of Algorithm 2). The tasks identified in circle-shaped represent the different functions of the algorithms. For example, the task identified as `A41 trsm` represents the **`plasma_core_ompc_trsm`** function executed in block `A41` of the matrix. The dependencies of this task are the tasks identified as `A11 potrf` and `A41 enter data` that represent other tasks in the DAG. When the dependencies are satisfied and the thread of `node 2` assigned to the task is free, task `A41 trsm` starts to run. When OMPC finishes the task, the result is sent to `node 3` so that task `A43 gemm` starts executing. Also, OMPC can start executing the tasks `A42 gemm` and `A44 syrk` in different threads of `node 2`.

OMPC is in charge of assigning each task to a corresponding node. The HEFT algorithm used by the scheduler is a heuristic that tries to minimize the application execution time. HEFT uses the task graph, the execution time of each task, and the amount of data transferred to schedule the tasks on the nodes. The first phase of HEFT consists of classifying the tasks on a ranking. This ranking defines the order that the HEFT will assign the task to resources. In our example, `A11 potrf` will rank higher than `A31 trsm` since `A31 trsm` task depends on `A11 potrf`. In the second phase, HEFT distributes every task on the available resources so that its execution finishes first on the assigned resource. In the task graph figure, when HEFT maps two related tasks to two different nodes, the edge between the two tasks is colored red, otherwise it is colored blue to represent intra-node communication. In Figure 3, we can observe some red edges between the tasks to allow load balancing between different nodes. For example, the tasks identified as `A21 trsm` and `A41 trsm` depend on task `A11 potrf`, but only `A21 trsm` is mapped to the same node as `A11 potrf`. In fact, if HEFT schedules both tasks on `node 1`, the execution time will take longer than sending the data to `node 2` and running `A41 trsm` on this node. No communication will be necessary, but the only one node will perform the computation, leading to worse execution times. Therefore, HEFT scheduling determines that it is preferable to pay the cost of moving data to utilize the two computing nodes.

IV. Related Work

Several libraries have focused on solving dense linear algebra problems with different programming models. In recent years, libraries have chosen to use the task-based programming model that allows linear algebra operations to be parallelized using tiled algorithms [9] and consider heterogeneous architectures for best results.

A well-known and established library is the Basic Linear Algebra Subprograms (BLAS), which groups together a set of low-level routines to perform linear algebra operations [18]. Linear Algebra Package (LAPACK) uses multi-threaded implementation of BLAS libraries [19]. LAPACK exploits the caches on modern cache-based architectures and the instruction-level parallelism of modern processors. It takes into account the cache hierarchies by maximizing data reuse, moving from vector to block operations. A multi-node implementation of LAPACK is ScaLAPACK. ScaLAPACK considers a 2D block-cyclic matrix distribution and block-partitioned algorithms to ensure high levels of data reuse [3]. In ScaLAPACK, the assignment of the blocks to the processors is static, which minimizes the time to allocate and deallocate the blocks from the processors' memory. ScaLAPACK does not use the task paradigm, which makes writing some algorithms, such as Cholesky factorization, difficult.

Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) is a library for multi-core architectures that

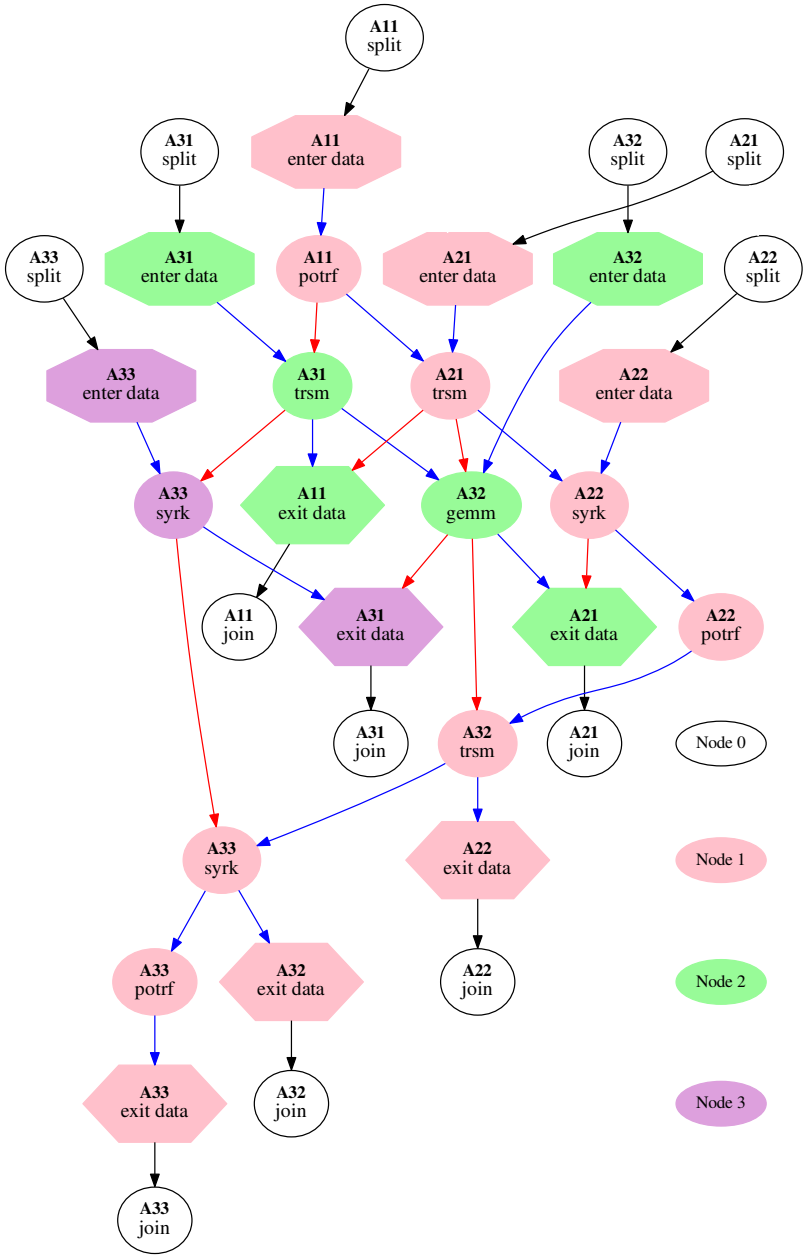


Fig. 3: Task graph of Cholesky factorization for a 4×4 tile matrix in 3 worker nodes with HEFT scheduler.

uses BLAS [4], [5]. PLASMA relies on tiled algorithms, which yields fine granularity parallelism using OpenMP. PLASMA uses OpenMP dynamic scheduling to keep kernels always busy. The OpenMP runtime starts executing the tasks that are already available without the need to have the DAG complete [20]. Similar to PLASMA, Formal Linear Algebra Methods Environment (FLAME) is a linear algebra library that obtains parallelism by data dependencies extracted at runtime [10]. The FLAME syntax is a mathematical notation to reduce implementation errors and increase the readability of algorithms.

Although the described libraries are efficient, very few uses all the computational resources that a cluster can provide; this is the case of the Software for Linear Algebra Targeting Exascale (SLATE) library. SLATE is a library that supports large-scale distributed nodes with accelerators [6]. Its algorithms minimize communication between tasks to improve their parallelization using lookahead panels to overlap communication

and computation. It mixes OpenMP tasks, MPI and CUDA, which makes it hard to control the scheduling since everything will depend on the work-stealing scheduler of the OpenMP library.

DPLASMA is a distributed memory implementation of PLASMA. DPLASMA uses a DAGuE engine to decode sequential nested loops into a format that can be analyzed and executed in a distributed environment [11]. DAGue is an engine for high-performance computing on distributed architectures, written on top of the PaRSEC runtime. The DAGuE engine uses eDAG and working-stealing to schedule tasks. The new versions of DPLASMA offer the availability to work with nodes with multiple sockets of multi-core processors and accelerators such as GPU or Intel Xeon Phi [21]. Programmers have to learn the language of the DAGuE engine since it has its own language. Chameleon is another library based on the PLASMA but relies on the StarPU generic runtime system to manage the resources [22]. The advantage of using StarPU is that Chameleon can run on heterogeneous architectures, but the disadvantage is that developers have to know various languages and APIs to write, read or maintain the algorithms. Also, Chameleon can handle a buffer that the user has to think about its size to have efficient results.

V. Results

This section compares the OMPC PLASMA library with ScaLAPACK and with the original PLASMA implementation. We compare the libraries performance executing matrix-matrix multiplication and the Cholesky factorization (see Section III-B). The results shows that OMPC PLASMA has competitive performance.

A. Experimental setup

For these experiments, we used the Santos Dumont (SDumont) supercomputer from the National Laboratory for Scientific Computing (LNCC) of Brazil [23]. The SDumont system is a Bull Sequana X cluster composed of 376 computational nodes interconnected with Infiniband EDR (100Gb/s). Each compute node has two 24-core Intel Xeon Cascade Lake Gold 6252 clocked at 2.4 GHz with 384GB of RAM. Each node runs the Bull ClusterStor 9000 v3.3 OS and the cluster uses SLURM to manage jobs execution. SLURM also guarantees that our benchmarks executed with exclusivity.

We adopted PLASMA `v19.8.1` as the basis for the OMPC PLASMA implementation. PLASMA uses OpenBLAS `v0.3.20` to execute its low-level routines using a single OpenBLAS thread. We used the OMPC runtime `v14.10.0` with Clang compiler `v14.0.0`, MPICH `v4.0.2`, and UCX `v1.12` as a communication framework between nodes. Section V-C compares ScaLAPACK `v2.2.0` with OMPC PLASMA. The version of OpenBLAS and MPI to compile ScaLAPACK are the same as described above. Finally, we ran each experiment 5 times using Singularity containers to maintain a reproducible environment.

B. Comparison with PLASMA

For this comparison, we take a PLASMA execution on a single node with 12 threads as a baseline. We then execute OMPC PLASMA using 4, 8 and 16 workers nodes with 12 threads each. For both measurments, the user can vary the blocks size. We experiment with sizes `500`, `1000` and `2000`. For PLASMA, the best block size is `1000`, and for OMPC PLASMA, it is `2000`.

Figure 4a and Figure 4b show the speedup of matrix-matrix multiplication and Cholesky factorization in OMPC PLASMA against PLASMA results. Figure 4 indicates that the performance SpeedUp obtained with OMPC PLASMA is almost linear. With 4 worker nodes, OMPC PLASMA achieves 4.00x speedup; with eight worker nodes, 7.00x; and with 16 worker nodes, 12.00x. The best results in both cases are with the largest matrix size, which is 150k×150k in matrix-matrix multiplication and 220k×220k in Cholesky factorization.

Figure 5 shows the matrix-matrix multiplication and Cholesky factorization performance of OMPC PLASMA. The first point of the figures represents the result obtained in a single node (12 cores), the second point with four nodes (48), the third point with eight nodes (96), and the last point with sixteen nodes (192). As the size of the core numbers increases, so does the number of operations performed per second.

C. Comparison with ScaLAPACK

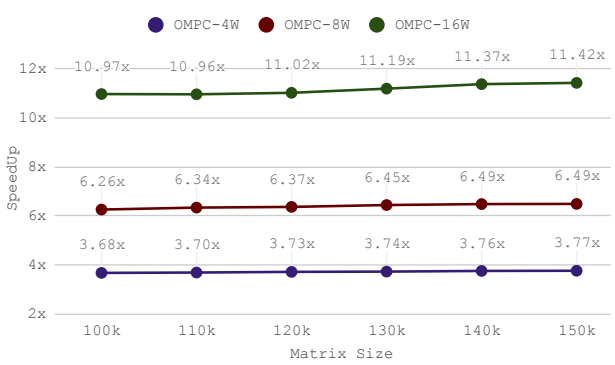
This section shows the results of the comparison between OMPC PLASMA and ScaLAPACK. The results obtained used 4, 8, and 16 worker nodes with 12 threads in each nodes. The OMPC PLASMA configuration is similar to the previous section. For ScaLAPACK, we modify the OpenBLAS thread number to 12. For ScaLAPACK, the best block size is 1000.

Figure 6a and Figure 6b show the results for matrix-matrix multiplication and Cholesky factorization, respectively. OMPC PLASMA always has better performance than ScaLAPACK, for 4, 8 and 16 worker nodes. In both experiments, OMPC PLASMA with 16 worker nodes obtains the best results in all matrix sizes.

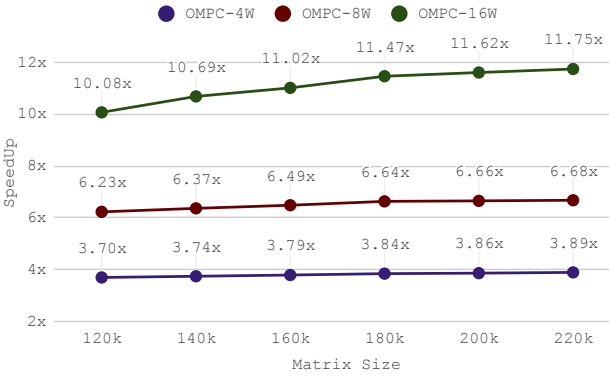
For 4, 8, and 16 worker nodes, OMPC PLASMA is close to 3x faster than ScaLAPACK in matrix-matrix multiplication (Figure 6a). For example, with a matrix size of 90k×90x with 4 worker nodes, OMPC PLASMA has a speedup of 3.52x. With eight worker nodes, it obtains 2.81x with 130k×130k matrix size. With 16 worker nodes, speedup is 2.85x with 150k×150k.

In Cholesky factorization (Figure 6b), OMPC PLASMA obtains a speedup close to 1.50x with 4, 8, and 16 worker nodes. With 4 worker nodes, OMPC PLASMA gets a speedup of 1.48x with a 90k×90k matrix size. With a matrix size of 130k×130x with 8 nodes, speedup is 1.35x. With 16 worker nodes, speedup is 1.38x with 170k×170k.

Missing values in Figure 6 correspond to ScaLAPACK experiments with 4 and 8 nodes that gave out of segmentation faults errors. In Figure 6a we also do not show results for 150k×150k matrix size with 4 and 8 nodes for the same reason. This problem also happens with other matrix sizes

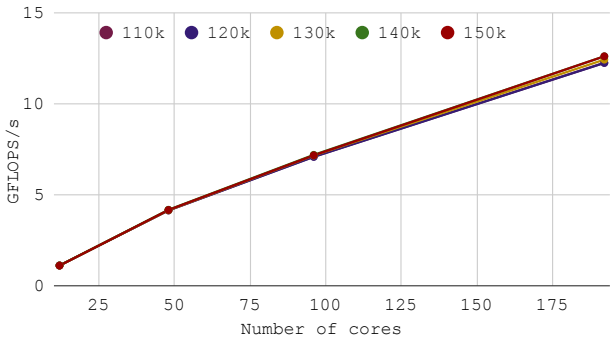


(a) Matrix-Matrix Multiplication

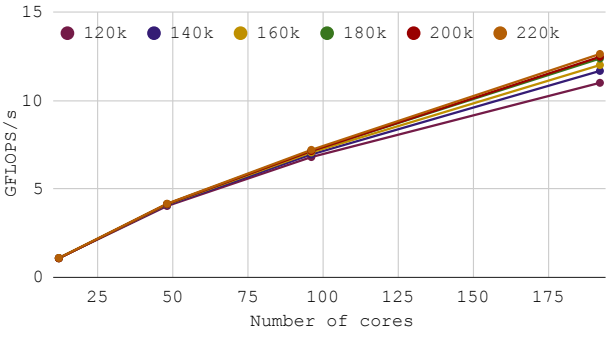


(b) Cholesky Factorization

Fig. 4: Speedup of PLASMA kernels using OMPC PLASMA distributed implementation vs original PLASMA single node implementation.

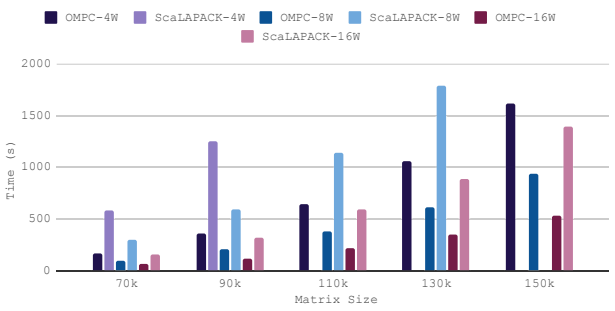


(a) Matrix-Matrix Multiplication

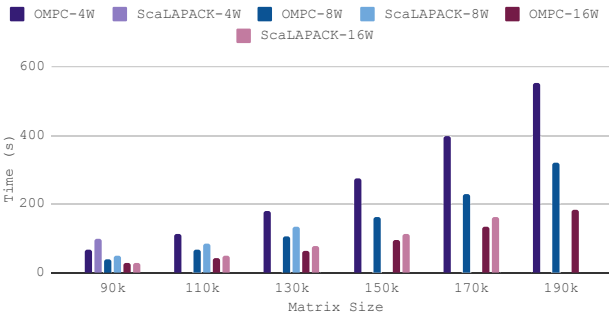


(b) Cholesky Factorization

Fig. 5: OMPC PLASMA performance with different matrix sizes.



(a) Matrix-Matrix Multiplication



(b) Cholesky Factorization

Fig. 6: Execution times of ScaLAPACK vs OMPC PLASMA.

in matrix-matrix multiplication and Cholesky factorization in ScaLAPACK. Nevertheless, the same experiments ran fine using OMPC PLASMA.

Although OMPC PLASMA shows a better memory efficiency than ScaLAPACK, OMPC PLASMA also has problems with matrix size limited by the amount of RAM. In Sections IV and V, the limit of OMPC PLASMA on matrix-matrix multiplication is 150k×150k, and Cholesky factorization is 220k×220k. The matrix size in matrix-matrix multiplication is smaller than in Cholesky decomposition since the first applica-

tion needs three matrices to execute, whereas the factorization only needs the lower part of one matrix.

VI. Conclusions and future work

This paper presented a library extension to run dense linear algebra kernels on distributed systems called OMPC PLASMA. OMPC PLASMA is based on the original PLASMA tiled algorithms, that represents computational kernels as a set of tasks. OMPC PLASMA uses the OpenMP `target` directives to define the code regions that will be distributed across the cluster nodes and uses the OMPC runtime to schedule these tasks and to automatically manage communication between nodes. Compared to the original PLASMA library, the OMPC PLASMA routines are superior in functionality and performance. Compared to the ScaLAPACK library, the OMPC PLASMA routines are superior in performance and accuracy. Unlike ScaLAPACK, OMPC PLASMA parallelization is easy to understand, as it uses OpenMP code annotation instead of MPI primitives. Based on these results, we conclude that OMPC PLASMA is a promising library for linear algebra in distributed systems that provides easy-to-understand programming. We also conclude that OMPC is a runtime that facilitates programming in distributed systems, as it abstracts the complexity of managing communications between tasks in the distributed system by using only OpenMP pragmas.

New linear algebra libraries use computing resources such as GPU accelerators or FPGAs to improve performance, functionality, and accuracy. These libraries use languages like CUDA or OpenCL to be able to interact with the clusters, bringing more complexity to the code. By adding support for the CUDA language in OMPC, the OMPC PLASMA library could interact with accelerators to further minimize application time. With the adding support for accelerators, we will compare OMPC PLASMA with libraries like DPLASMA or SLATE. Currently, OMPC PLASMA has five functions to run in distributed systems. We plan to implement more aplications, such as LU or QR factorization.

Acknowledgments

This work is supported by Petrobras under grant 2018/00347-4, and by the National Council for Scientific and Technological Development (CNPq) under grant 402467/2021-3. We would also like to thank the LNCC for granting the computational resources at the Santos Dumont supercomputer that were crucial to perform the experiments for this project.

References

[1] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.

[2] NVIDIA, "Cuda C Programming Guide," Tech. Rep. September, 2015.

[3] L. S. Blackford, A. Cleary, A. Petitet, R. Whaley, J. Dongarra, J. Choi, E. DAzevedo, J. Demmel, I. Dhillon, K. Stanley *et al.*, "ScaLAPACK: a linear algebra library for message-passing computers," Lockheed Martin Energy Research, Inc., Oak Ridge, TN (United States), Tech. Rep., 1997.

[4] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, P. Wu, I. Yamazaki, A. YarKhan, M. Abalenkovs, N. Bagherpour *et al.*, "PLASMA: Parallel linear algebra software for multicore using OpenMP," *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 2, pp. 1–35, 2019.

[5] "Plasma," Apr 2022. [Online]. Available: https://icl.utk.edu/plasma/overview/index.html

[6] A. Abdelfattah, H. Anzt, and A. Bouteiller, "Roadmap for the development of a linear algebra library," 2017.

[7] S. Abdulah, H. Ltaief, Y. Sun, M. G. Genton, and D. E. Keyes, "Exageostat: A high performance unified software for geostatistics on manycore systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 12, pp. 2771–2784, 2018.

[8] H. Yviquel, M. Pereira, E. Francesquini, G. Valarini, P. R. Gustavo Leite, R. Ceccato, C. Cusihualpa, V. Dias, S. Rigo, A. Souza, and G. Araujo, "The OpenMP Cluster Programming Model," *51st International Conference on Parallel Processing Workshop Proceedings (ICPP Workshops 22)*, 2022.

[9] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief *et al.*, "Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE, 2011, pp. 1432–1441.

[10] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. Van De Geijn, "FLAME: Formal linear algebra methods environment," *ACM Transactions on Mathematical Software (TOMS)*, vol. 27, no. 4, pp. 422–455, 2001.

[11] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemariner, H. Ltaeif, P. Luszczek, A. YarKhan, and J. Dongarra, "Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA." Anchorage, Alaska, USA: IEEE, 2011-05 2011, pp. 1432–1441.

[12] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin, "XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, may 2013, pp. 1299–1308.

[13] X. Tan, J. Bosch, C. Alvarez, D. Jiménez-González, E. Ayguadé, and M. Valero, "A Hardware Runtime for Task-based Programming Models," *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[14] Intel, "Intel(R) Threading Building Blocks Documentation," Tech. Rep., 2019.

[15] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE transactions on parallel and distributed systems*, vol. 13, no. 3, pp. 260–274, 2002.

[16] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, no. 1, pp. 38–53, 2009.

[17] J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley, "Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines," *Scientific Programming*, vol. 5, no. 3, pp. 173–184, 1996.

[18] [Online]. Available: http://www.netlib.org/blas/

[19] K. Kim, T. B. Costa, M. Deveci, A. M. Bradley, S. D. Hammond, M. E. Guney, S. Knepper, S. Story, and S. Rajamanickam, "Designing vector-friendly compact BLAS and LAPACK kernels," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.

[20] OpenMP, "OpenMP Application Programming Interface," Tech. Rep., 2013.

[21] "DPLASMA," accessed: 8-6-2022. [Online]. Available: https://github.com/ICLDisco/dplasma

[22] G. Themes, "Chameleon: A dense linear algebra software for heterogeneous architectures," accessed: 1-6-2022. [Online]. Available: https://project.inria.fr/chameleon/

[23] LNCC, "Santos Dumont (SDumont)," 2021, Bull Sequana X1000, Xeon Gold 6252 24C 2.1GHz, Mellanox InfiniBand EDR, NVIDIA Tesla V100 SXM2. [Online]. Available: https://sdumont.lncc.br/