2D Static Resource Allocation for Compressed Linear Algebra and Communication Constraints

Olivier Beaumont, Lionel Eyraud-Dubois, Mathieu Verite

HAL Id: hal-02900244
https://inria.hal.science/hal-02900244v2

Submitted on 24 Jul 2020

2D Static Resource Allocation for Compressed Linear Algebra and Communication Constraints

Olivier Beaumont
Inria Bordeaux Sud-Ouest
University of Bordeaux
Talence, France
Olivier.Beaumont@inria.fr

Lionel Eyraud-Dubois
Inria Bordeaux Sud-Ouest
University of Bordeaux
Talence, France
Lionel.Eyraud-Dubois@inria.fr

Mathieu Verite
Inria Bordeaux Sud-Ouest
University of Bordeaux
Talence, France
Mathieu.Verite@inria.fr

Abstract

This paper adresses static resource allocation problems for irregular distributed parallel applications. More precisely, we focus on two classical tiled linear algebra kernels: the Matrix Multiplication and the LU decomposition algorithms on large linear systems. In the context of parallel distributed platforms, data exchanges can dramatically degrade the performance of linear algebra kernels and in this context, compression techniques such as *Block Low Rank (BLR)* are good candidates both for limiting data storage on each node and data exchanges between nodes. On the other hand, the use of BLR representation makes the static allocation problem of tiles to nodes more complex. Indeed, the workload associated to each tile depends on its compression factor, which induces an heterogeneous load balancing problem. In turn, solving this load balancing problem optimally might lead to complex allocation schemes, where the tiles allocated to a given node are scattered over the whole matrix. This in turn causes communication complexity problems, since matrix multiplication and LU decompositions heavily rely on broadcasting operations along rows and columns of processors, so that the communication volume is minimized when the number of different nodes on each row and column is minimized. In the fully homogeneous case, 2D Block cyclic allocation solves both load balancing and communication minimization issues simultaneously, but it might lead to bad load balancing in the heterogeneous case. Our goal in this paper is to propose data allocation schemes dedicated to BLR format and to prove that it is possible to obtain good performance on makespan when simultaneously balancing the load and minimizing the maximal number of different resources in any row or column.

1 Introduction

In this paper, we consider the implementation of several linear algebra operations on CPU and GPU platforms where the matrix is described in a compressed form in BLR (Block Low Rank) format. Indeed, many of the time-consuming tasks performed on supercomputers are linear algebra operations. With the advent of multi-core architectures and massive parallelism, it is therefore particularly interesting to optimize and understand their parallel behavior. In this paper, we consider the problems of Cholesky factorization, LU factorization and the matrix product.

Recently, these problems have received particular attention, especially because they are used to validate the behaviour of runtime schedulers [3, 5, 12, 24, 41, 47]. For instance, tiled Cholesky factorization

has been considered in StarPU [11, 27], SMPSs [43], SuperMatrix [45] and DAGuE [19]. In practice, the kernels that we consider in this paper have been implemented in Dense Linear Algebra state of the art libraries, such as Chameleon [2], DPLASMA [18], FLAME [33], and PLASMA [20]. Recently, the practical design of good static schedule for heterogeneous resources has been considered in [4] and extensions to sparse matrices [37] and incomplete Cholesky factorization [40], have also been proposed.

Today, it is acknowledged that storing and moving data across a distributed platform are the factors that limit the performance of linear algebra algorithms. In this framework alternative approaches have been proposed to store data, in particular using the opportunity of representing the matrix in a compressed form. For example, the $\mathcal{H}$-matrix representation introduced in [34] and is now widely used to reduce storage costs and also execution time significantly, when the matrix can actually be compressed. For instance, it is known [31] that in the context of LU-factorization, the number of floating point operations can be reduced from $\Theta(2/3n^3)$ to $\Theta(nk^2 \log^2 n)$, where k is a parameter that represents the compression that can be achieved. Similar results have been proved in [6] for semi-separable matrices. Recently, these techniques have also been extended to sparse matrices [29, 44].

Complementary approaches have been proposed to avoid issues related to complex and irregular data accesses induced by $\mathcal{H}$-matrix representation. Block Low Rank (BLR) decompositions have been advocated for instance for multi-frontal methods [7, 42] or finite-element matrices [8]. In BLR decompositions, matrices are represented as regular tiled matrices, except that each block is represented using a low rank decomposition (when compression is possible). Another approach is lattice $\mathcal{H}$-matrices [35, 48]. The work in [22] follows the same line of research. In all cases, the idea is increase simplicity and to keep a high compression ratio, even at the price of a slight increase in the flop complexity. Indeed, keeping the regular tiled structure makes it possible to use the runtime schedulers, that have proven their efficiency in making use of heterogeneous resources, at least at the level of a single computing node consisting of one multicore CPU and several accelerators [2].

When considering more distributed platforms, typically consisting in several such nodes with multicores and GPUs, the problem of reducing communications and therefore to compress involved matrices, becomes even more crucial. Indeed, it has been proved [3, 4] that, at the level of a single heterogeneous node, basic dynamic runtime schedulers are such as those proposed in [11] are in general enough to produce a schedule where communications can be overlapped by computations, provided that the tile size is large enough. This property does not hold true anymore when considering distributed nodes, since moving matrix tiles across the network is a costly operation, both in terms of time and memory. In this context, it has been proved in [2] that in the case of distributed memory platforms, a careful static distribution of the tiles of the matrices is crucial.

In this context, our general goal in the present paper is to propose static allocation schemes that are adapted to BLR matrices for Matrix Multiplication, Cholesky Factorization and LU Factorization kernels in the context of a platform consisting of a set of distributed memory nodes. These static distributions should be able to

- balance the load between the different nodes, knowing the relative weight of each tile, related to the rank of its low rank decomposition in the BLR format. Note that in this context, balancing the load is closely related to balancing the memory needs on each node.

- minimize the volume of communications between nodes and minimize the number of communications between nodes. Since we consider applications where most of the communications are based on broadcasts among the nodes that share the same row or the same column of tiles, we will bound the maximal number of participating nodes in each row/column.

In all cases, the achieved makespan, *i.e.* the time to completion, will be used as a metric to evaluate the quality of data distribution schemes.

The rest of paper is organized as follows. In Section 2, we review the related work related to packing problems with additional constraints, heterogeneous partitioning problems arising in linear algebra and practical solutions used in the case of compressed matrices. Then, we formalize the platform model and the linear algebra kernels that are used throughout the paper in Section 3, and derive an optimization problem for obtaining efficient tile allocations. In Section 4, we present heuristics to solve this problem: an extension of the standard block-cyclic approach and a randomized strategy. An experimental framework and a comparison of their performance is described in Section 5. A more in-depth analysis of the behavior

of our heuristics is provided in Section 6. A last, concluding remarks and perspectives are proposed in Section 7.

2 Related Work

2.1 Heterogeneous Allocation Problems arising in Linear Algebra

The dual problem of the one we address in this paper, i.e. how to allocate dense homogeneous matrices to heterogeneous resources while minimizing communications, has been considered in the literature. For example, in the context of matrix multiplication, the problem of partitioning a matrix between heterogeneous resources has been modeled by Kalinov et al. [39] as the problem of partitioning a square into fixed area rectangles while minimizing the sum of the perimeters of these rectangles. Since then, numerous works have successively improved the approximation ratio of this NP-Complete problem, which are summarized in a recent survey paper [13]. In a more formal way, but in a homogeneous framework, the problem of communication minimization associated with the matrix product, considered as a cube instead of a square, has been considered by [36] and in the development of communication avoiding algorithms [46].

A closely related problem is the one of allocating a sparse matrix to a set of homogeneous processors in order to balance the load between processors while keeping a regular allocation structure has been introduced by F. Manne et al. under the name of rectilinear partitioning or generalized block distribution in [32] and is still the object of an active literature [49]. In rectilinear partitioning, the problem consist in partitioning the rows and the columns into groups, the intersection between a group of columns and a group of rows being in turn assigned to a given processor. In this context, the main goal is to build groups such as the load (*i.e.* the number of non-zero elements allocated to a processor) is well balanced.

2.2 Applied Perspective

Numerical linear algebra techniques appear at the core of several computation intensive scientific applications requiring large linear system resolution. In this context, state-of-the-art methods are based on using efficient linear algebra kernels under task-based runtime systems for scheduling.

In [10] the authors detail techniques to perform the Cholesky factorization of matrix which off-diagonal tiles are compressed; the undelying linear system arises from Boundary Elements Methods (BEM) applied to wave propagation modeling. The proposed implementation for a distributed memory platform makes use of plain 2D block cyclic method to allocate tiles to nodes, which is supposed to balance the workload associated to each node while reducing the number of nodes involved in each data transfer, thus allowing to expect a larger overlap of communications by computations.

2D block cyclic method has been proven to achieve optimal load balancing between resources in the dense case [17] and is the only allocation rule implemented in historical ScaLAPACK library [25]. Nevertheless, better suited allocation techniques can be developped for distributed applications in specific contexts.

In the field of weather forecast, maximum-likelyhood estimation method requires the resolution of large scale linear systems. In [1] the authors tackle the problem using Cholesky decomposition of a matrix which tiles are compressed using BLR technique. Since full rank diagonal tiles are associated to longer work time than compressed off-diagonal ones, they are treated specifically to ensure better load balancing between nodes, in distributed platforms implementation. The proposed hybrid method, detailed in [21], allocates diagonal tiles in a round robin fashion while a 2D block cyclic allocation scheme is applied for off diagonal ones.

2.3 Bin Packing Problems

The optimization problem considered in this paper can be considered as a bin packing problem, where tiles of the matrix are items to be packed into the different processors, considered as bins. Considerations on limiting communications lead use to include additional constraints on the feasible packings, with a limit on the number of processors alloted to a given row or column. This can be seen as a generalization of some related literature on bin packing: bin packing with class constraints [28, 38], in which there is a limit to the number of classes allowed in a bin, and bin packing with minimum color fragmentation [15],

in which the number of bins containing a given class is limited. Since we consider both rows and columns, our case is a two-dimensional version of these problems, however the techniques developed in these papers can not be generalized to our context.

3   Model and Notations

3.1   Notations

Let us consider a matrix $\mathbf{A}$ of size $n \times n$, divided in N^2 tiles of size $n_b \times n_b$ where $N = \frac{n}{n_b}$. In the paper, we will consider that n_b is fixed, typically large enough so that the granularity of resulting tasks is large enough for a computing node consisting of several GPUs, and small enough so as to be able to balance the load among resources. The choice of n_b will be discussed in Section 5.

For any $(i,j) \in [\![1;N]\!]^2$ the tile in position (i,j) will be denoted by $\mathbf{A}_{i,j}$. Following BLR format, we assume that each tile is represented as a (low) rank matrix, *i.e.* that $\forall i,j, \quad \exists n_{i,j} \in [\![1;n_b]\!]$ and two rectangular matrices $\mathbf{U_{i,j}}$, $\mathbf{V_{i,j}}$ of respective size $n_b \times n_{i,j}$ and $n_{i,j} \times n_b$ such that

$$\mathbf{A}_{i,j} \approx \mathbf{U}_{i,j}\mathbf{V}_{i,j}$$

The ratio $d_{i,j} = \frac{n_{i,j}}{n_b}$ can be seen as the density of tile in position (i,j).

Concerning the computing platform, we will assume that it consists into P identical (homogeneous) nodes, that can in turn consist in several resources such as accelerators.

Our goal is to allocate the set of tiles $\mathbf{A}_{i,j}$ onto these P distributed memory nodes while minimizing data exchanges between nodes. On each given node, we assume that a runtime scheduler such as StarPU [11,27] is used to actually map elementary tasks to resources (CPU, GPU). Note that once tile $\mathbf{A}_{i,j}$ is allocated to node P_k, then P_k will be responsible for all the tasks that modify the value of tile $\mathbf{A}_{i,j}$ and that P_k will receive the other input tiles if it does not hold them. This assumption concerning the organization of computations is classic and all libraries such as Chameleon [2] or DPLASMA [18] rely on the same model.

The total makespan associated to such allocation for the considered algorithm, LU factorization or matrix multiplication, is the objective function which we seek to minimize.

3.2   Matrix Multiplication

Given a similar square matrix $\mathbf{A}$ of $N \times N$ tiles, where the size of tile $\mathbf{A}_{i,j}$ is $n_b \times n_b$ and let us assume that we no want to perform the matrix product $\mathbf{A}\mathbf{A}^T = \mathbf{C}$. In order to compute $\mathbf{C}$, we rely of a variant of Canon's algorithm [30] adapted to low rank representation. More specifically, the algorithm consists in N stages. During stage $k \in \{1,....,N\}$, all block matrix multiplications of type $\mathbf{C}_{i,j} + = \mathbf{A}_{i,k}\mathbf{A}_{k,j}$ are performed. In a distributed implementation, the data distribution for $\mathbf{A}$ and $\mathbf{C}$ are the same and in terms of computations, at stage k, the processor in charge of $\mathbf{A}_{i,k}$ tile for any value i broadcasts $\mathbf{A}_{i,k}$ to the processors that are in charge of $\mathbf{C}_{i,j}$ (we will denote them as the processors of its row of processors) and the processor in charge of $\mathbf{A}_{k,j}$ tile for any value j broadcasts $\mathbf{A}_{k,j}$ to the processors that are in charge of $\mathbf{C}_{i,j}$ (we will denote them as the processors of its column of processors). Therefore, at each stage, exactly one broadcast takes place in each row and each column of processors

3.3   LU Factorization

Still considering a square matrix $\mathbf{A}$ of $N \times N$ tiles, where the size of tile $\mathbf{A}_{i,j}$ is $n_b \times n_b$, we consider the textbook right looking variant of LU decomposition described in [30]. The algorithm consist in N stages and each step can itself be decomposed in 4 steps. All operations are performed in-place and each operation therefore replaces one of its entries by its output. For the sake of simplicity, we will denote the block in position (i,j) of the resulting matrix by $\mathbf{A}_{i,j}$, even if it is not a block of the initial input matrix $\mathbf{A}$. During stage $k \in \{1,...,N\}$, the tiled LU decomposition algorithm (i) computes the LU decomposition of the tile in position (k,k) using GETRF LAPACK operation, (ii) then updates the tiles in column k (with $\mathbf{L}_k$) and the tiles in row k (with $\mathbf{U}_k$), using the factors $\mathbf{L}_k$ and $\mathbf{U}_k$ (produced by the previous GETRF operation) and TRSM LAPACK operation and then (iii) updates the remaining blocks of $\mathbf{A}$ in the lower right corner with respect to the block row and column panels (just computed
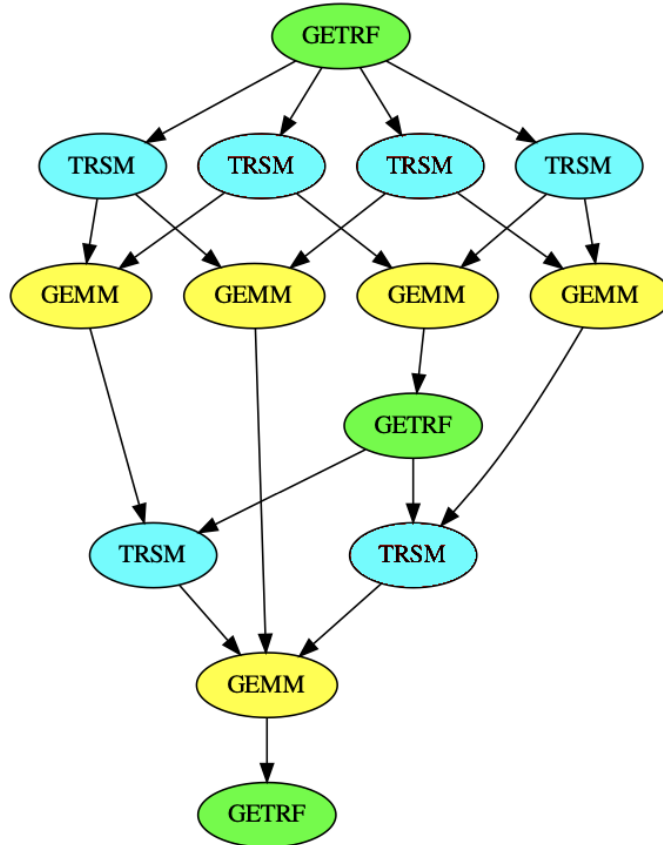
Figure 1: Task graph of the LU decomposition for a 3×3 matrix

with TRSM operations) using GEMM LAPACK operation. The set of operations, together with their dependences, are depicted in Figure 1. Let us first remark that only the edges depicted in Figure 1 correspond to actual data dependencies, and that above presentation, based on stages, in only used for the sake of simplicity. Indeed, one of the level 1 GEMM operation (the one that correspond to the lower right corner) does not have to be performed before the GETRF and and the TRSMs of stage 2, even on this small example. If we now consider a distributed setting, in terms of communications, all operations correspond to broadcast operations. Indeed, at each stage, the block produced by the GETRF operation is broadcasted along the kth row of processors and the the kth column of processors. Then, the blocks produced by the TRSM operations are then broadcasted, either in their respective row or column of processors.

3.4 Constraints

Regarding the communication model, we assume that the bandwidth of the links between nodes is not large enough for purely dynamic allocation strategies to be able to overlap communications and computations. Therefore, it is crucial to minimize the overall volume of communications.

In the applications described in Section 3.2 and Section 3.3, communications consist in broadcast operations performed on rows and columns of processors. In this context, a crucial metric for performance is to minimize the number of different nodes in the same row or column. Indeed, let us consider a broadcast operation of a message of size S performed on a row of size N, but with only n_p different processors in the same row. Then, the broadcast requires the transmission of exactly $(n_p - 1)S$ bits (S bits per unique processor) and the duration can even be reduced to order $\log n_p$ when using a diffusion tree. It is therefore crucial to minimize n_p in order to minimize the volume of data transmitted over the

network.

In the 2D block-cycling case with P processors, the solution consists in creating a virtual grid of $\sqrt{P} \times \sqrt{P} = P$ processors, which define $\sqrt{P}$ row and column types. Then, the different row (resp. column) types are allocated in a round robin fashion to the rows and columns of the matrix. Considering the total load $\mathcal{L}$, such allocation scheme leads to perfect load balancing between processors if the processing costs of the different tasks are homogeneous since:

- if $\sqrt{P}$ and $N/\sqrt{P}$ are integers, then each processor receives a load equal to $\mathcal{L}/P$;

- a solution in which each row and each column would have strictly less than $\sqrt{P}$ processors cannot achieve perfect load balancing.

In the case where the workloads associated to the different tiles are heterogeneous, as in the context of this paper where compression is used, the solution is more complex. Indeed, if the workload of each tile is identified to an item weight, trying to balance the loads among processors (which is a hard NP-difficult problem), is likely to yield an allocation such that some rows or columns feature all processors. This would in turn induce very important communication costs. Conversely, it is expected that a solution in which one would impose to have exactly $\sqrt{P}$ processors per row and column, especially if one imposes a cyclic round robin distribution of rows and columns, may lead to a significant load imbalance.

Hence, to limit the overhead due to communications while maintaining a good load balance throughout the computation, constraints are added to limit the number of different on each row and column. More specifically, a feasible allocation is a one to one tile to resource assignment, denoted as a *mapping* $\mathbf{M}$, such that no more than $\alpha\sqrt{P}$ different processors appear on any row and column of $\mathbf{M}$.

The $\alpha \in [1; +\infty[$ parameter controls the tightness of communication restriction; $\alpha = 1$ being the basic *2D block cyclic* solution used for homogeneous case. In the following, larger values of α ($\alpha = 1.1$, 1.5, 2 or 3) will be considered as an illustration that the trade off between heavier data transfer and better load balancing leads to shorter completion time.

3.5 Evaluation Metric

The objective is to find a feasible allocation $\mathbf{M}$ that minimizes the makespan of the target kernel (LU factorization or Matrix Multiplication). However, finding a closed form formula to evaluate the performance of an allocation, that would take into account the communication costs, the evolution of the ranks of the tiles, the contentions on the communication links and the capability of overlapping communications and computations is impossible in practice. Therefore, the evaluation of the makespan denoted by $\mathfrak{R}(\mathbf{M})$ in Section 5 is performed in this paper using either direct experiments (for computational costs) or fine grain simulations using SimGrid.

Nevertheless, in order to build the allocations, we will rely on surrogate metrics in order to evaluate the expected performance of an allocation and to guide the search of a feasible solution. To clearly explain the way they are calculated, some additional handy notations are introduced.

- Regardless of the applied algorithm, LU factorization or matrix multiplication, the task to be performed on tile $\mathbf{A}_{i,j}$, $(i,j) \in [\![1; N]\!]^2$ at iteration $k \in [\![1; N]\!]^2$ is denoted $\tau_{i,j}^k$.

- Its type, denoted $\mathtt{typ}(\tau_{i,j}^k)$, is among standard LAPACK linear algebra operation: `GETRF`, `TRSM`, `GEMM`.

- Its stimated execution time is: $W_{i,j}^k = d_{i,j} C_{\mathtt{typ}(\tau_{i,j}^k)}$ where $C_{\mathtt{OP}}$ represent the estimated execution time of type `OP` operation on a full rank tile.

- The matrix W of cumulated execution time is defined as: $\forall (i,j) \in [\![1; N]\!]^2 : W_{i,j} = \sum\limits_{k \in [\![1; \kappa(i,j)]\!]} W_{i,j}^k$.

The surrogate metrics are then detailed below.

1. $\mathfrak{B}(\mathbf{M})$ is associated to load balancing. For a given allocation $\mathbf{M}$, $\mathfrak{B}(\mathbf{M})$ denotes the overall load allocated to any processor assuming that the processors are never idle and that communications

can always be overlapped with computations.

$$\mathfrak{B}(\mathbf{M}) = \max_{p \in \{1,\ldots,P\}} \left(\sum_{\substack{(i,j) \in [\![1;N]\!]^2 \\ \mathbf{M}_{i,j}=p}} W_{i,j} \right)$$

2. $\mathfrak{D}(\mathbf{M})$ denotes the maximum (simulated) makespan, assuming that communications can always be overlapped with computations but taking into account the idle time that can be induced by task dependencies. $\mathfrak{D}(\mathbf{M})$ value is calculated using a simple discrete events based alogrithm where each processor is fed by a queue of ready tasks, ordered according to their priority. The priority of any task is the longest path in the DAG from this task to the end task, each edge being weighted according to the estimated execution time of the origin task. The chosen implementation of this algorithm allows preemption of incomplete tasks, so as to model a real task-based scheduler behavior. Indeed, since we are considering tasks at the scale of large tiles which actually consist in several elementary operations on the tile, stopping the execution at the end of one of these elementary operations can therefore be assimilated to preemption for large tile-scale tasks.

3. $\mathfrak{C}(\mathbf{M})$ denotes the maximum (simulated) makespan, using the realistic communication model provided by SimGRID [23] and taking into account the idle time that can be induced by task dependencies. This simulation is based on an actual implementation in the Chameleon library, using the SimGrid based simulation backend of StarPU, which allows to obtain very realistic results.

Since $\mathfrak{D}$ and $\mathfrak{C}$ metrics require some form of simulation, they can be highly computation intensive for large test cases and therefore may not be suitable as guiding criterion for resolution, though used for evaluation of solutions.

The methods described in the following are thus designed to search the best mapping according to the single load balancing metric $\mathfrak{B}$. The formal optimization problem we seek to solve can be stated as:

- given a weight matrix $W \in \mathcal{M}_{N \times N}(\mathbb{R}_+)$, a number of processors $P \in \mathbb{N}^*$ and a limit $n_P \in \mathbb{N}^*$ (in practice selected as $n_P = \lceil \alpha \sqrt{P} \rceil$ with $\alpha \in [1; +\infty[$),

- find a mapping $\mathbf{M} \in \mathcal{M}_{N \times N}(\{1, \ldots, P\})$ that maps each tile to one processor, such that:

$$\forall i \in [\![1;N]\!] \quad \mathrm{Card}(\{\mathbf{M}_{i,l} : l \in [\![1;N]\!]\}) \leqslant n_P$$
$$\forall j \in [\![1;N]\!] \quad \mathrm{Card}(\{\mathbf{M}_{k,j} : k \in [\![1;N]\!]\}) \leqslant n_P$$

- which minimizes the maximum load $\mathfrak{B}(\mathbf{M}) = \max_{p \in \{1,\ldots,P\}} W^{\mathbf{M}}(p)$, where $W^{\mathbf{M}}(p)$ is the total weight of tiles allocated to p.

The general strategy is therefore to search for the most balanced solution, assuming that it will also lead to the shortest execution time.

4   Data Distribution Schemes

In this section, we review some data distribution schemes, the objective of which is both to balance the computation load between the different resources throughout the computation and to minimize the number of processors participating in the same row or column of the matrix to minimize communications. In Section 4.1, we describe the 2D Block Cyclic algorithm that is classically used in the case of homogeneous cost tiles. In Section 4.2, we consider a natural extension of the 2D Block Cyclic algorithm, called Extended 2D Block Cyclic. Variants of this scheme have been proposed in [16] for example, but we propose here a formalization of the algorithm and an optimal algorithm to solve the associated optimization problem. Finally, we propose a randomized strategy in Section 4.3, that does not rely on a repeated pattern.

4.1　2D Block Cyclic

Plain *2D block cyclic* is a straightforward and extensively used method in numerical algebra kernels to perform the allocation in the more regular dense case. Tiles are allocated to processors following a rectangular pattern, subsequently called *grid* and denoted $\mathbf{G}$. This pattern is then repeated over the whole matrix. The grid of size $r \times c$ is almost square: typically $r = c - 1$ and c the largest integer such that $P \geqslant rc$. It is filled using all processors in a round-robin fashion. This shape choice ($r = c - 1$) is expected to ensure the variety of processors allocated to diagonal tiles. In the following, *2D block cyclic* method is denoted as BC. In the homogeneous case, BC is known to have all desirable properties, provided that c is close to $\sqrt{P}$. Indeed, in this case, (i) there are only (close to) optimal number $\sqrt{P}$ of different processors on each row and column and (ii) the load associated to each processor is expected to be almost well balanced, since each processor is allocated $\lfloor \frac{N^2}{P} \rfloor$ (or $\lceil \frac{N^2}{P} \rceil$) homogeneous tiles. In the heterogeneous case we consider in this paper, (i) still holds under the same conditions on P but (ii) might not hold true anymore, since tiles have different ranks and therefore different costs. Nevertheless, if $\frac{N^2}{P}$ is large enough, the load associated to each processor will be the summation of a large number of individual tiles, and the resulting overall load balancing between processors might be acceptable due to the law of large numbers. We will analyze these observations in more details in Section 6.

For any given value of P, the following procedure describe the BC method as used for our tests.

- select the largest $c \in \mathbb{N}^*$ such that $P \geqslant (c - 1)c$

- the grid $\mathbf{G}$ with dimensions $r \times c$, $r = c - 1$ is defined as:

$$\forall (i, j) \in [\![1; r]\!] \times [\![1; c]\!] : \quad \mathbf{G}_{i,j} = (i - 1)c + j$$

- the final mapping $\mathbf{M}$ is then:

$$\forall (i, j) \in [\![1; N]\!]^2 : \quad \mathbf{M}_{i,j} = G_{i \bmod (r), j \bmod (c)}$$

The associated aggregated grid of execution time has the same dimensions as $\mathbf{G}$ and each of its position is the total load of a processor:

$$\forall (i, j) \in [\![1; r]\!] \times [\![1; c]\!] : \quad \overline{W}_{i,j} = \sum_{\substack{(k,l) \in [\![1; N]\!]^2 \\ i \equiv k[c]; \, j \equiv l[r]}} W_{k,l}$$

Therefore, accessing the load balancing metric for such solution is straightforward:

$$\mathfrak{B}(\mathbf{M}) = \max_{(i,j) \in [\![1; r]\!] \times [\![1; c]\!]} \left(\overline{W}_{i,j} \right)$$

4.2　Extended 2D Block Cyclic Distribution

In order to balance the load between the processors, one possible idea is to relax condition (i) a bit so as to better control the load allocated to each processor. This strategy has already been proposed in [16] but with a non-optimal computation algorithm. To do this, we authorize ourselves to use $n_P = \lceil \alpha \sqrt{P} \rceil$ processors per row and per column. The resolution is then carried out following two steps:

1. compute the aggregated execution time $\overline{W}$ associated to each position of a grid of dimension $r \times c$ with r and c smaller than n_P;

2. produce a grid $\mathbf{G}$ by allocating each position to a processor; the final mapping $\mathbf{M}$ is then derived the same way as when using BC method.

The whole process can be seen as a resolution according to BC method using $\alpha^2 P$ virtual processors since $\mathbf{G}$ features approximately α^2 times more positions than P. The procedure is then completed by associating those virtual processors to the real ones.

Step 1 is not straightforward since relaxing the constraint on processors diversity on rows and columns allow many pairs of dimensions for the grid thus leading to as many different feasible allocation after
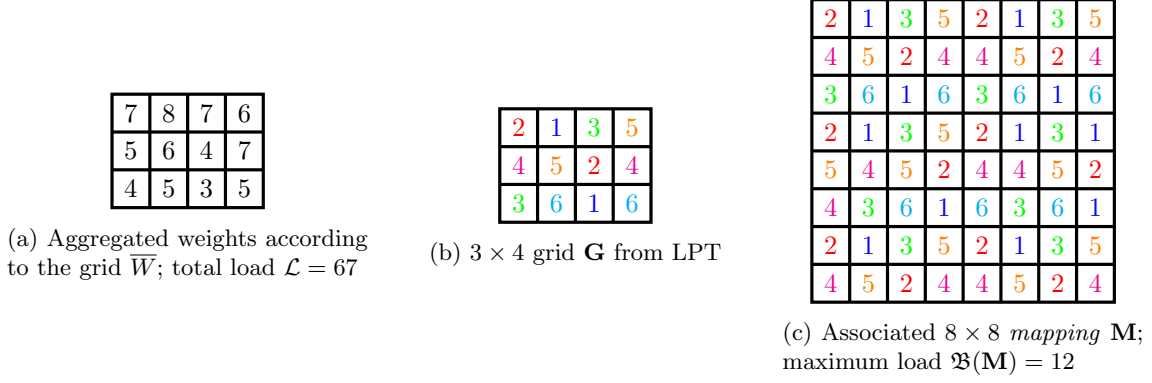
(a) Aggregated weights according to the grid $\overline{W}$; total load $\mathcal{L} = 67$

(b) 3×4 grid **G** from LPT

(c) Associated 8×8 *mapping* **M**; maximum load $\mathfrak{B}(\mathbf{M}) = 12$

Figure 2: Example of BCE allocation: $P = 6$, $n_P = 4$, **G** is 3×4

performing step 2. However the selection of such dimensions appears crucial for the quality of load balancing in the final allocation, depending on the method used to complete the allocation during step 2 as discussed in section 6.

Once the aggregated execution times $\overline{W}$ are calculated, building up the grid **G** in step 2 reduces to a well known bin packing problem for which several resolution method can be used: exact ones such as Integer Linear Programming, and approximated ones.

The linear program which has to be solved when using such exact method may be quite large, $\mathcal{O}(\alpha^2 P)$ variables and $\mathcal{O}(\alpha\sqrt{P})$ constraints, thus leading to long resolution time. On the other hand, some simple heuristics provide, in virtually no time, good quality approximated solutions to this problem, though this should be tampered according to the specific cases mentioned above. Hence we chose to perform step 2 of BCE method using *Largest Processing Time* (LPT) greedy heuristic well known for its good average performances. Its implementation is detailled in algorithm 1.

Algorithm 1: *Largest Processing Time* greedy algorithm

    input : P, $\overline{W}$

1 Total load for each processor: $L[p] = 0$, $\forall p \in \{1, ..., P\}$
2 **for** $(i,j) \in [\![1;r]\!] \times [\![1;c]\!]$ considered in decreasing order of $\overline{W}_{i,j}$ **do**
3   find least loaded processor: $p_{\text{least}} = \underset{p \in [\![1;P]\!]}{\operatorname{argmin}}(L[p])$
4   allocate processor p_{least} to position (i,j): $\mathbf{G}_{i,j} = p_{\text{least}}$
5   update the load of processor p_{least}: $L[p_{\text{least}}] \leftarrow L[p_{\text{least}}] + \overline{W}_{i,j}$

    output: **G**, L

4.3 Random Subsets Algorithm

Both BC and BCE impose a very regular pattern for allocating tiles to processors. This very regular pattern has the advantage of allowing direct control of the number of processors allocated to each row and each column and of easily implementing the algorithm for allocating tiles to the processors. On the other hand, this very regular pattern (i) imposes constraints on P because of the size of the pattern and (ii) imposes an imperfect solution to the load balancing problem because of the pattern's own constraints, although BCE is expected to limit this issue somewhat.

The idea behind Random Subset (RS) is to get away from the regular pattern constraint, and try to apply the non expensive and relatively efficient LPT heuristic directly on the whole matrix of cumulated execution times W. The problem arising when trying to do so is that the constraint on processors diversity may lead to *dead end* configurations where the algorithm cannot complete the allocation. Figure 3 illustrates this problem on a small example. The probability of the algorithm getting stuck in such configuration dramatically increases with the size of the problem N and the number of processors P, making such direct application of LPT unusable in practice for almost all use cases.

3	2	1	2	1	0	0	1
2	4	4	3	1	1	0	0
2	4	4	5	5	3	2	0
1	3	4	7	3	2	0	0
0	2	2	7	7	8	5	3
0	1	3	8	14	12	7	11
0	2	6	10	7	19	18	15
0	0	4	9	16	17	7	20

(a) 8×8 matrix of estimated execution time W

−	−	−	−	−	−	−	−
−	−	−	−	−	−	−	−
−	−	−	−	−	−	−	−
−	−	−	−	−	−	−	−
−	−	−	5	−	−	−	−
−	−	−	?	6	4	−	3
−	−	−	2	−	2	3	6
−	−	−	1	5	4	−	1

(b) Incomplete allocation: next position to allocate is $(6, 4)$ but no suitable processor can be found since the current allocation allows processors $\{3, 4, 6\}$ on row 6 and $\{1, 2, 5\}$ on column 4

Figure 3: Example of *dead end* configuration ($N = 8$, $P = 6$, $n_P = 3$)

To circumvent this problem RS method is based on predefined subsets of processors associated to either rows or columns. Processors allocated on each row (resp. column) all belong to the associated subset. Those subsets have the following properties:

1. each has no more than n_P elements;

2. each row (resp. column) subset is *compatible*, *i.e.* has a non empty intersection with each column (resp.row) subset.

Hence, using row and column subsets, a modified version of LPT can be applied to directly allocate processors to tiles over the whole matrix $\mathbf{A}$. This greedy heuristic, detailed in algorithm 2, necessarily provides a feasible mapping.

Having available row and column subsets of processors is a prerequisite to the above mentionned heuristic. Since defining subsets featuring the required properties 1 and 2 in a deterministic way seems at least tedious for small use cases and extremely computation intensive for large values of P, those subsets are generated using a randomized strategy.

More precisely, a number $Q \in \mathbb{N}^*$ of row and column subsets to create is selected, along with $K \in \mathbb{N}^*$, the minimum accepted intesection size between any two row and column subsets. Then the subsets are simply generated using random sampling according to the procedure described in algorithm 3.

This process produces the two necessary families of row and column subsets of processors which features the desired properties 1 and 2 previously detailed.

Although quite straightforward, the subsets generation procedure may require a long time to complete due to the fact that, given the $\mathcal{R}$ family of row subsets, each randomly sampled column subset C is tested against a compatibility criterion, line 6 of algorithm 3, that if not met imposes to resample C thus lengthening the execution.

The difficulty to sample a suitable C subset can be linked to the input parameters Q and K values. Assuming unbiased and independent samplings for each subsets, the probability of C being compatible with each subset in $\mathcal{R}$ is:

$$\mathbf{P}(\{R_1 \bigcap C \geqslant K\}; ...; \{R_Q \bigcap C \geqslant K\})$$

which reduces to

$$\mathbf{P}(\{R \bigcap C \geqslant K\})^Q \tag{1}$$

since all subsets are supposed identically and independently sampled (R being any of the row subsets in family $\mathcal{R}$).

This probability can be calculated from the following expressions:

$$\mathbf{P}(\{R \bigcap C \geqslant K\}) = 1 - \sum_{k \in [\![0; K-1]\!]} \mathbf{P}(\{R \bigcap C = k\}) \tag{2}$$

Algorithm 2: `random-subsets-direct` greedy algorithm

input : P, W, $\mathcal{R} = \{R_1, ..., R_Q\}$, $\mathcal{C} = \{C_1, ..., C_Q\}$

1 Initialisation
2 Total load for each processor: $L[p] = 0$, $\forall p \in \{1, ..., P\}$
3 Initially empty $N \times N$ mapping of tiles: $\mathbf{M}$
4 Initial set of usable subsets for each row: $\mathcal{R}_i = \mathcal{R}$, $\forall i \in [\![1; N]\!]$
5 Initial set of usable subsets for each column: $\mathcal{C}_j = \mathcal{C}$, $\forall j \in [\![1; N]\!]$
6 Initial set of usable processors for each position: $\mathcal{P}_{i,j} = \{1, ..., P\}$, $\forall(i,j) \in [\![1; N]\!]^2$
7 Initial list of tiles to allocate: $\mathcal{B} = \{(i,j) : \forall(i,j) \in [\![1; N]\!]^2\}$

8 Allocation
9 while $\mathcal{B} \neq \emptyset$ **do**
10 $\quad$ get the *heaviest* tile's position to allocate: $(i,j) = \underset{(i,j) \in \mathcal{B}}{\operatorname{argmax}}(W_{i,j})$, $w = W_{i,j}$
11 $\quad$ remove it from the list: $\mathcal{B} \leftarrow \mathcal{B} \setminus \{(i,j)\}$
12 $\quad$ find least loaded processor among usable ones: $p_{\text{least}} = \underset{p \in \mathcal{P}_{i,j}}{\operatorname{argmin}}(L[p])$
13 $\quad$ allocate processor p_{least} to position (i,j): $\mathbf{M}_{i,j} = p_{\text{least}}$
14 $\quad$ update the load of processor p_{least}: $L[p_{\text{least}}] \leftarrow L[p_{\text{least}}] + w$
15 $\quad$ update the set of still usable subsets on row i:
16 $\quad$ **for** $R \in \mathcal{R}_i$ **do**
17 $\quad\quad$ **if** $\exists p \in \{\mathbf{M}_{i,l} : l \in [\![1; N]\!]\} : p \notin R$ **then**
18 $\quad\quad\quad$ $\mathcal{R}_i \leftarrow \mathcal{R}_i \setminus R$
19 $\quad$ update the sets of still usable processors on row i: $\forall l \in [\![1; N]\!] : \mathcal{P}_{i,l} = \bigcup_{R \in \mathcal{R}_i} R$
20 $\quad$ update the set of still usable subsets on column j, $\mathcal{C}_j$, the same way as $\mathcal{R}_i$
21 $\quad$ update the sets of still usable processors on column j, $\mathcal{P}_{k,j}$, $\forall k \in [\![1; N]\!]$, the same way as $\mathcal{P}_{i,l}$, $\forall l \in [\![1; N]\!]$
22 $\quad$ **for** $(k,l) \in \mathcal{B}$ such that $\operatorname{Card}(\mathcal{P}_{k,l}) = 1$ **do**
23 $\quad\quad$ allocate the only possible processor $p \in \mathcal{P}_{k,l}$ to position (k,l): $\mathbf{M}_{k,l} = p$
24 $\quad\quad$ update its load: $L[p] \leftarrow L[p] + W_{k,l}$
25 $\quad\quad$ remove position (k,l) from $\mathcal{B}$: $\mathcal{B} \leftarrow \mathcal{B} \setminus \{(k,l)\}$

output: $\mathbf{M}$, L

$$\mathbf{P}(\{R \bigcap C = k\}) = \sum_{1 \leqslant l_1 < ... < l_k \leqslant m} [(\prod_{t=1}^{k} \frac{n_P}{P - l_t + 1}) \times (\prod_{\substack{l=1 \\ l \neq l_1, ..., l \neq l_k}}^{n_P} 1 - \frac{n_P}{P - l + 1})] \tag{3}$$

One can select parameter Q and K to ensure that the subsets generation process is fast, *i.e.* that this probability is large enough. Given a target lower bound $\eta \in]0; 1[$ for this probability, the number of subsets in each family must not exceed:

$$Q \leqslant \lfloor \frac{\ln(\eta)}{\ln(\mathbf{P}(\{R \bigcap C \geqslant K\}))} \rfloor \tag{4}$$

In practice, we choose the Q to ensure a sufficient expected number of occurences of each processor in the subsets so that the subsequent allocation using the greedy algorithm leads to a good load balancing.

For instance, requiring an average of $\beta \in \mathbb{N}^*$ occurences of each proc in the subsets imposes to select Q such that:

$$Q \geqslant \frac{\beta P}{n_P} \tag{5}$$

In turns, this leads to a probability of a successful column subset sampling η:

$$\eta \leqslant \mathbf{P}(\{R \bigcap C \geqslant K\})^{\frac{\beta P}{n_P}} \tag{6}$$

Algorithm 3: `subsets-generation` algorithm

---
    input : P, n_P, Q, K

---
1   randomly sample Q sets of n_P processors among $\{1, ..., P\}$ without replacement; they are the
      row subsets $\{R_1, ..., R_Q\}$ gathered in $\mathcal{R}$

2   initialize the family of column subsets: $\mathcal{C} = \emptyset$

3   set $t = 0$

4   **while** $t < Q$ **do**

5 | randomly sample a subset C of n_P processors among $\{1, ..., P\}$ without replacement

6 | **if** C is compatile: $\forall q \in [\![1; Q]\!]: \; \mathrm{Card}(C \bigcap R_q) \geqslant K$ **then**

7 | | add C to the family of *column* subsets: $\mathcal{C} \leftarrow \mathcal{C} \bigcup \{C\}$

8 | | update t: $t \leftarrow t + 1$

---
    output: $\mathcal{R}, \mathcal{C}$

There is therefore a trade off to handle between the number of subsets and the sampling difficulty and associated generation time. However, one must keep in mind that the generation process is only specific to a pair of (P, α) parameters but not to the matrix size N.

In the following experiments, for each (P, α) pair, subsets are generated using the value of parameter Q derived form equation 5 with $\beta = 10$, and simply $K = 1$. 10 families $(\mathcal{R}, \mathcal{C})$ were generated using those parameters and for each test case, resolutions have been performed using each of the 10 families of subsets. The best result according to metric $\mathfrak{B}$ has been kept as final result for the test case.

The RS allocation process, derived from LPT heuristic, is expected to have good performance regarding load balancing inasmuch as it allows choice between different ressources at each iteration. Hence, one may be tempted to enforce such behavior by generating subsets using a parameter $K \geqslant 2$. However, it makes the sampling process significantly longer, because of a tighter compatibility criterion, for only slight improvement: we indeed observed that the generation process using $K = 1$ leads to an average intersection size between any two row and column subsets R and C generally larger than 1 and, in practice, often close to 2.

Compared to BC and BCE, RS is expected to produce better load balancing. On the other hand, the distribution of tiles allocated to a processor is not a priori regular in the matrix, contrarily to pattern based allocations, which is likely to cause load balancing problems over time, for factorizations such as LU in which the set of available tasks is not available from the beginning.

5   Experiments

5.1   Test Cases

The purpose of this study is to investigate and to compare different data distribution strategies to balance the load and to minimize the makespan for different kernels of linear algebra when the matrices are compressed using the BLR format. It is difficult to get access to matrices corresponding to actual use cases. For our experiments we therefore relied on synthetic matrices, which nevertheless reproduce the main characteristics of actual matrices, as can be encountered in various large scale simulation problems ([1, 10]). Thoses synthetic matrices allowed us to validate the approaches over a larger range of parameters and to ensure the reproducibility of the experiments.

A matrix **A** is thus characterized, in what follows, by its size N (in number of blocks) and by the data of $d_{i,j}$, where $d_{i,j}$ represents the density of tile $\mathbf{A}_{i,j}$ (*i.e.* the rank of $\mathbf{A}_{i,j}$ is $d_{i,j} \times n_b$).

The generation of test cases therefore requires the definition of a density distribution of the tiles according to their position in the matrix and in particular on their distance from the diagonal. The following process is used.

- All diagonal tiles are associated with $d = 1$ which means that diagonal tiles are assumed to be non-compressible (full rank).

- $\forall(i,j),\ d_{i,j} = \max(\min(v(i,j) + \gamma, 1), 0)$ where $v(i,j)$ follows an exponential distribution as a function of the distance to the diagonal

$$v(i,j) = \exp^{-\frac{\delta}{2} * (\frac{i-j}{N-1})^2}$$

and γ is a noise that follows a normal distribution $\gamma \sim \mathcal{N}(0, \frac{1}{20})$.

- In addition to diagonal tiles, a number θ of tiles are assumed to be full rank. They are located at uniformly randomly in the matrix, and θ follows a normal distribution $\theta \sim \mathcal{N}(\sqrt{N}, \frac{\sqrt{N}}{2})$.

Finally the costs of standard LAPACK operations, representing a density to execution time coefficient, are chosen as follows: $C_{\text{GETRF}} = 1$, $C_{\text{TRSM}} = 3$ and $C_{\text{GEMM}} = 6$.

5.2 Preliminary Results

The performance of the three strategies described in Section 4 has been evaluated on numerous generated linear systems for several values of (N, P, α) parameters and different criteria: maximum load (dots) and simulated execution time without communications (triangles) and maximum execution time with communications (squares).

Results show that both *BCE* and *RS* lead to better solutions that plain *block cyclic* method in specific regions of the parameters space: (details). A meta-strategy based on selecting the solution with minimum maximum load among all three methods is therefore a significant improvement over *block cyclic* and over each method alone. Such a strategy provides a good quality *mapping* for almost any combination of parameters.

Results for $\delta = 8$ are depicted for Matrix Multiplication in Figure 4 and LU factorization in Figure 5. In both figures, we choose values for P such that $\exists c, c(c-1) = P$ so that we can concentrate on the load balancing itself and not on rounding errors. Note that this situation favors BC (red) and BCE (purple) with respect to RS (green). All displayed results are normalized with respect to ideal load balancing where all processors would receive exactly the same load. BC results are shown on $\alpha = 2$ and $\alpha = 3$ plots only to ease the comparison with the other methods but actually correspond to $\alpha = 1$ case.

From Figure 4, we can draw some interesting conclusions. Since the matrix product consists of a set of independent tasks, there is in general no difference between pure load balancing (the dots) and simulations without communications (the triangles). When $\alpha = 1$, i.e. when using a minimum number of processors per row and per column, the difference with simulations with communication (the squares) is not significant but it becomes so as soon as α becomes larger ($\alpha = 2$ or 3), especially in the case of BCE. For the matrix product, the best heuristic is clearly RS for all values of P and N, provided that α is large enough. This observation is true even though the chosen P values benefit BC and BCE. With $\alpha = 3$, for all the configurations tested here, the value of the makespan with communication is indeed always less than 1% of the value that would be obtained without communication and with optimal load balancing.

The conclusions that can be drawn from Figure 5 are quite similar, except that the dependencies between tasks generate a difference between dots and triangles. For the rest, we can notice that load balancing is less strong for LU than for MM when using BC (red). This situation is related to the fact that the load associated with the tiles is less homogeneous (the weight of a tile increases with i and j), which makes load balancing more difficult. In this context, both BCE and RS are strategies of choice for data distribution and both give excellent results. Nevertheless, the performance degradation between pure load balancing and makespan without communications is more important in the case of RS than in the case of BCE. This can be explained by the fact that the tiles assigned to a given processor are always distributed regularly by construction with BCE, whereas this property is not automatically met with RS, which can lead, at certain instants, to an overload for one of the processors, which in turn can slow down the whole process. For example, as soon as $\alpha = 3$, BCE returns allocations that are within 5% of the lower bound.

In all cases, we can observe that BCE's behavior is very robust, as soon as α is greater than two. BCE generally performs at least as well as RS, which is a more expensive and sophisticated heuristic. In Section 6, our goal is to understand the performance of BCE from a more specific study of load balancing.
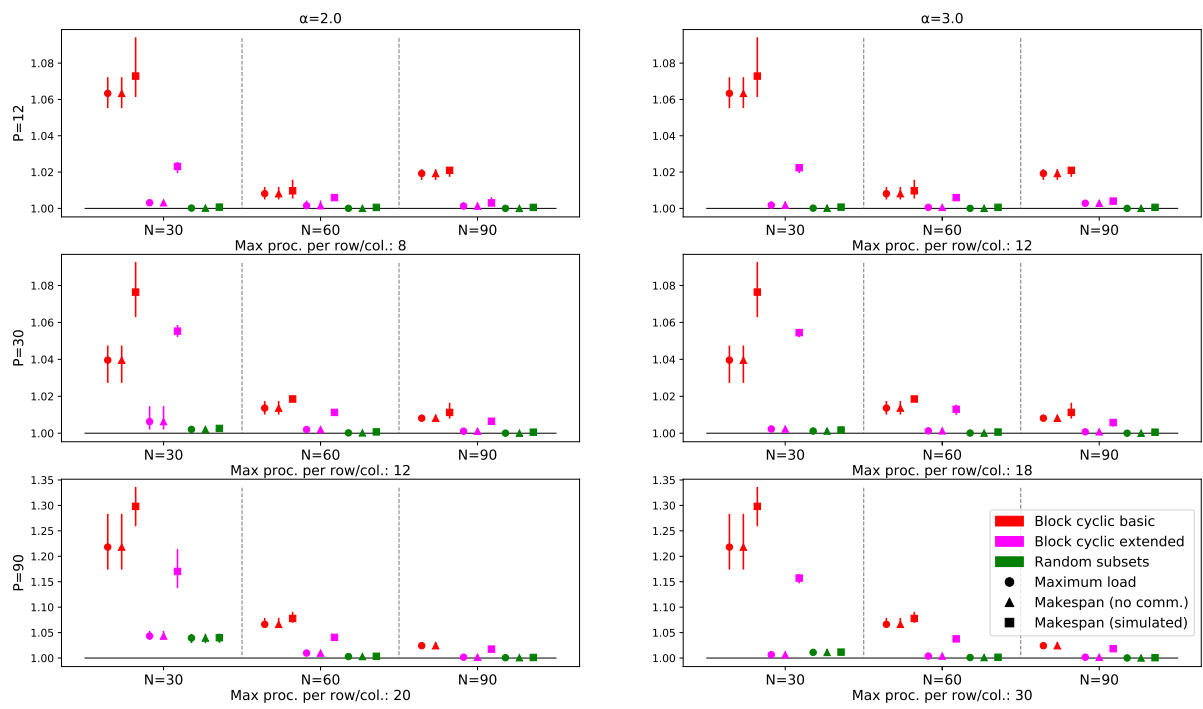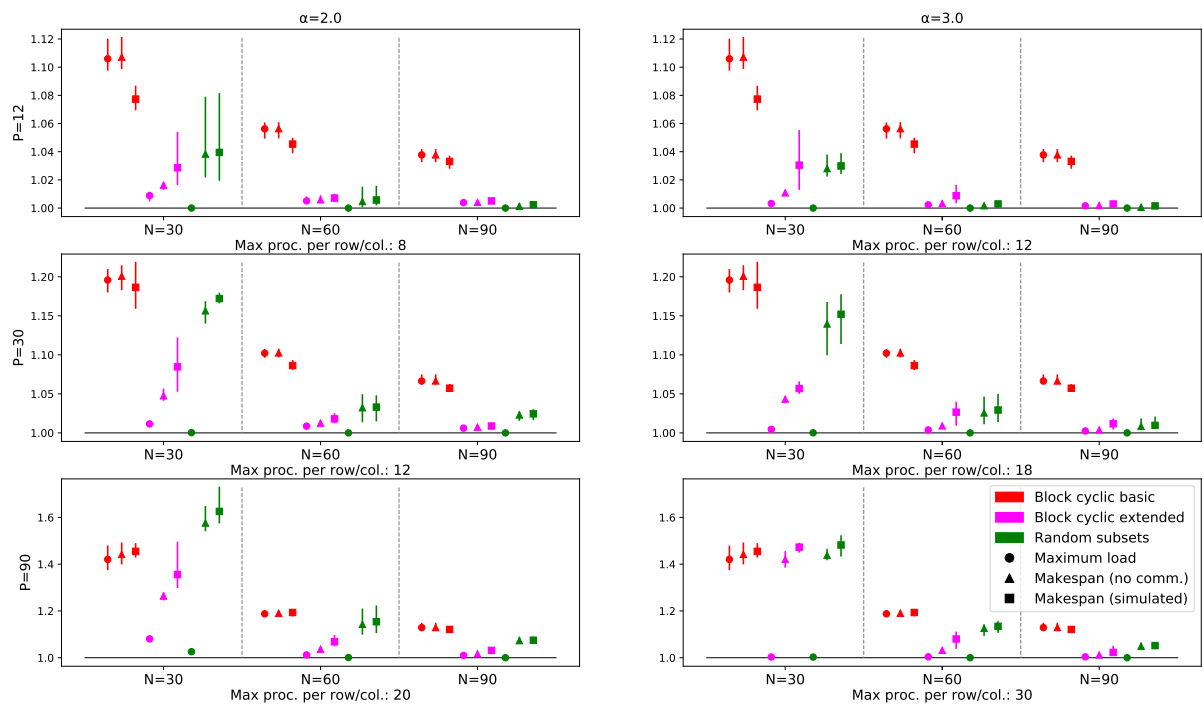
Figure 4: Results for Matrix Multiplication

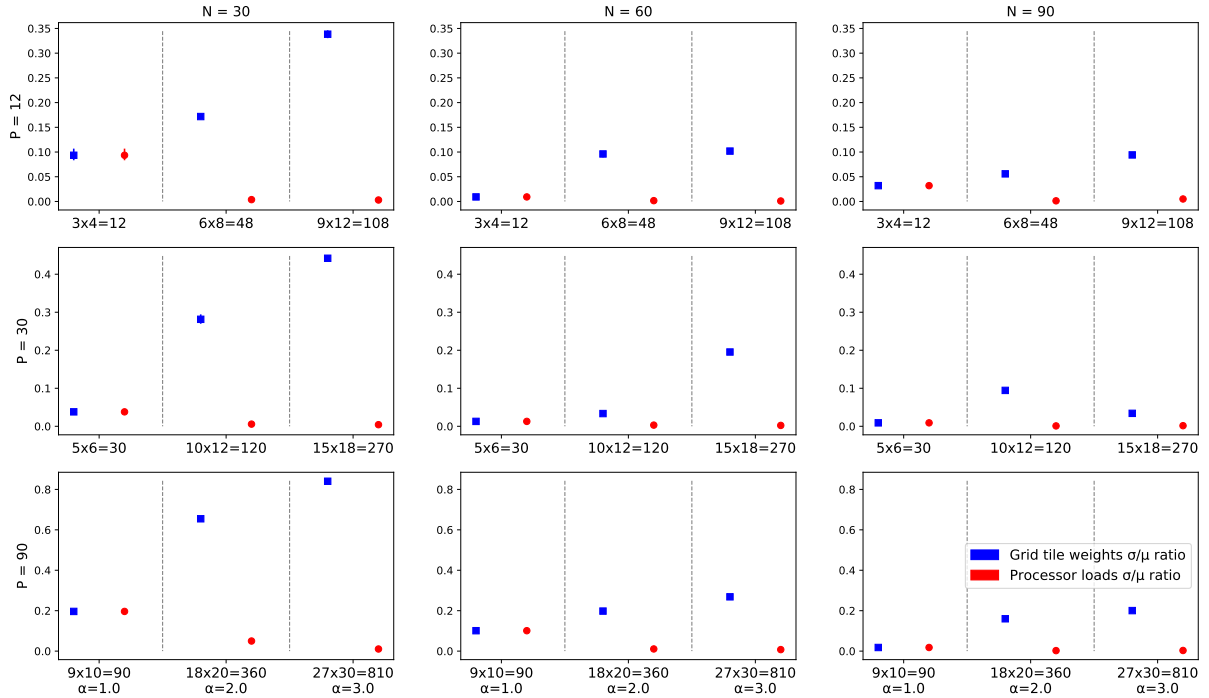Figure 5: Results for LU factorization

Figure 6: BCE load balancing with MM tiles

6 Analysis of Block Cyclic Extended Algorithm

6.1 Why does BCE balance the load well?

In this section, our objective is to understand and analyze the behavior of BC and BCE for two types of tile weight distribution. In the case of the matrix multiplication (Figure 6), the tile weights in the matrix depend only on their distance to the diagonal. In the case of LU factorization (Figure 7), the weights of the tiles in the matrix depend on both their distance to the diagonal and to the bottom right corner, since the bottom right tiles induce a naturally higher computation cost, since they are updated at all stages of factorization. The heterogeneity of tile weights is higher for LU factorization.

We consider here different sizes of matrices ($N = 30, 60, 90$), different values of the number of processors ($P = 12, 30, 90$) and different values of $\alpha = 1, 2, 3$, where $\alpha\sqrt{P}$ is the maximum number of different processors that can be present on the same row or column. $\alpha = 1$ corresponds to BC, whereas $\alpha = 2, 3$ correspond to two different variants of BCE.

Our goal is to measure the quality of the load balancing between the processors for pattern based distributions BC and BCE. Blue points are associated to grid tiles, *i.e.* show the balance between the loads of the virtual processors, whereas red points are associated to actual processors. When $\alpha = 1$ red and blue are the same since the numbers of virtual and real processors are the same whereas the number of virtual processors is α^2 times higher in general. Each point represents the dispersion of the load of virtual processors (blue points) or real processors (red points), where the dispersion of processor loads is defined as

$$\frac{\text{standard deviation of the distribution of loads}}{\text{mean value of the distribution of loads}}$$

so that a lower value represents a better load balancing (a ratio of 0 corresponds to perfect load balancing).

The first conclusion that can be drawn is related to the observation of the blue dots on any cell for $\alpha = 1, 2, 3$. We can notice that the first phase of tile aggregation following the regular pattern is efficient enough to limit the dispersion and that the dispersion is much greater when the number of virtual processors is increased (and therefore the number of tiles allocated to each virtual processor is decreased).
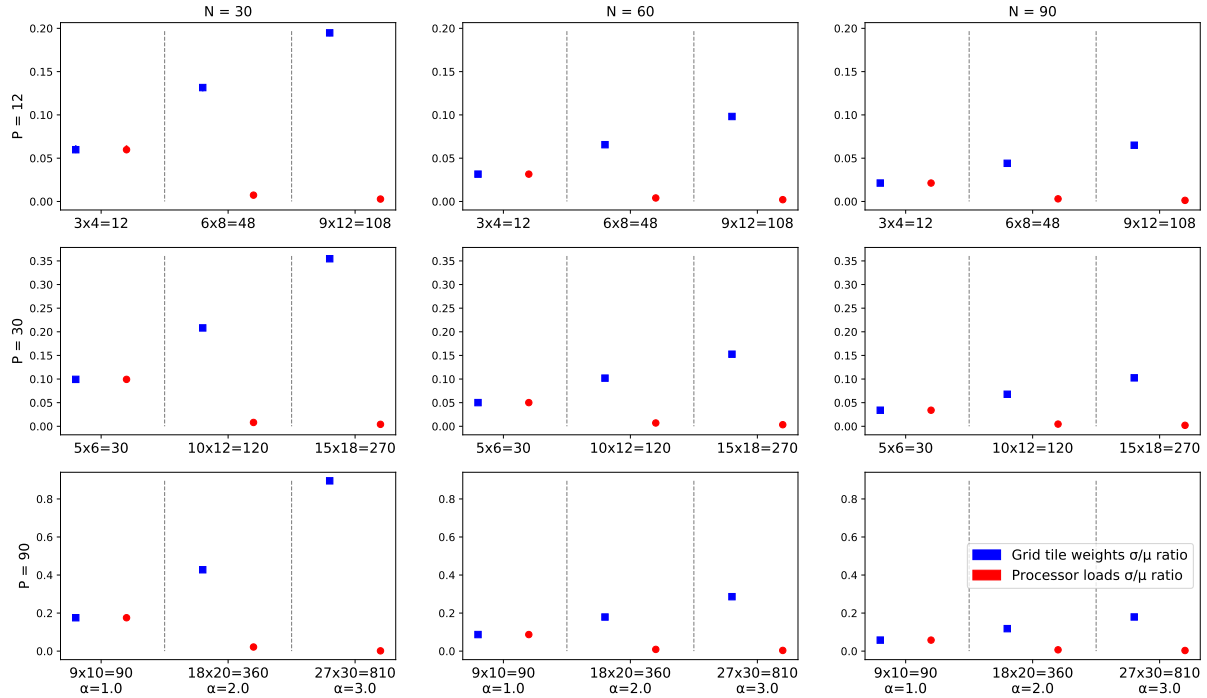
15

Figure 7: BCE load balancing with LU tiles

The second conclusion is related to the observation of the relative values of the blue and red dots when $\alpha = 2, 3$. It shows that the second phase of packing virtual processors into BCE is very efficient. Indeed, we can observe that the dispersion is almost null as soon as we group the virtual processors by groups of 4 ($\alpha = 2$) or 9 ($\alpha = 3$) virtual processors.

The intuition is as follows. If we use for example $\alpha = \sqrt{2}$, we will obtain $2P$ virtual processors, whose respective loads are already rather homogeneous since each virtual processor corresponds to the aggregation of a certain number of tiles. In this case, the packing algorithm results in the most loaded k-th virtual processor being matched with the least loaded k-th virtual processor inside the real k-th processor, resulting in a very homogeneous distribution of weights.

We have tried to establish this result theoretically but this result is in fact notoriously difficult. Indeed, even if we assume that the initial weight distribution is known and simple (uniform or normal for example), then the distribution of the mean value and standard deviation for the k-th element are only known by approximate and non-closed formulas [9, 26]. This is why we relied on simulations to establish these properties.

6.2 Limitations of BC and BCE solutions

For Figures 4 and 5, the number of processors P used the experiments have been carefully chosen to be written as $c(c-1)$ so that the rectangular pattern allows all processors to be used. This is not the case for arbitrary values of P and we are faced with the problem of finding a rectangular grid $r \times c$, with (i) r and c close together so that the grid is not too long and the number of different processors per row or column is not too large, and (ii) $rc \leqslant P$ but close to P so as not to waste resources. In practice, these two constraints are not easy to fulfill and their simultaneous implementation naturally leads to choosing a number of processors (a reservation of resources in practice) that is smaller than the number of processors actually available. In this case, the RS strategy proves to be very efficient, because it is agnostic of any property on P and its efficiency does not depend on the value of P.

As an example, we show in Figure 8 a particular case where $P = 34$ and $\alpha = 2$, so that the limit on the number of processors per row or column is 12. The basic block-cyclic strategy is limited to a 5×6 grid, and BCE can not use all 34 processors since $34 = 17 \times 2$ and $17 > 12$. Three possibilities are
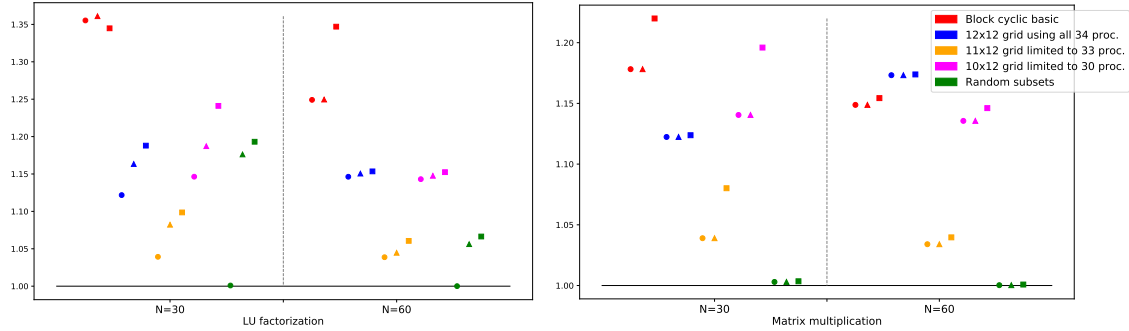
Figure 8: Results obtained for $P = 34$ with various grid sizes for BCE.

explored on Figure 8: a 10×12 grid which corresponds to doubling the block-cyclic grid size, a 12×11 grid which allows to use 33 processors and have all processors get the same number of tiles, or a 12×12 grid using all 34 processors with an uneven allocation. We observe that indeed, removing one processor to use the 12×11 grid allows to obtain better performance than with all 34 processors. The load balancing is not perfect though, because one processor has zero load. On the other hand, Random Subsets is able to obtain perfect load balancing even in that case. For Matrix Multiplication, this translates into a very good makespan as well. For LU factorization however, the dependencies incur a significant amount of idle time, and the resulting makespan of Random Subsets is at most equivalent (for $N = 60$) or worse (for $N = 30$) than the makespan of BCE. For these particular cases, we expect that a hybrid strategy can be designed to gain the best of both aspects: good load balancing and good compatibility with the dependencies of LU factorization.

For Figures 4 and 5, the number of processors P used the experiments have been carefully chosen to be written as $c(c-1)$ so that the rectangular pattern allows all processors to be used. This is not the case for arbitrary values of P and we are faced with the problem of finding a rectangular grid $r \times c$, with (i) r and c close together so that the grid is not too long and the number of different processors per row or column is not too large, and (ii) $rc \leqslant P$ but close to P so as not to waste resources. In practice, these two constraints are not easy to fulfill and their simultaneous implementation naturally leads to choosing a number of processors (a reservation of resources in practice) that is smaller than the number of processors actually available. In this case, the RS strategy proves to be very efficient, because it is agnostic of any property on P and its efficiency does not depend on the value of P.

7 Conclusion and Perspectives

In this paper, we consider the problem of data distribution in the context of compressed matrices in BLR format for linear algebra kernels such as the matrix product and LU factorization. In the compressed case, the load associated with each tile is no longer homogeneous and sophisticated strategies other than the usual 2D Block Cyclic (BC) have to be implemented in order to balance the load throughout the computation. We have proposed and analyzed two new strategies. The first BCE strategy is an extension of BC, which allows better load balancing at the cost of a slightly higher, but controlled, number of processors per row and per column. The second strategy is a randomized strategy that generates row and column types before performing the allocation. Both of these strategies significantly improve the results compared to BC. BCE proves to be particularly efficient for certain P values, which we were able to analyze and justify using order statistics. RS, on the other hand, has the advantage of giving very interesting results regardless of the value of P. This work opens several perspectives. First of all, one can try to improve the RS strategy so that it better takes into account the specific dependencies of the application's tasks, in order to balance the load throughout the computation. It may also be possible to hybridize the different strategies by adding a randomized component to BCE so that it can run on any number of processors.

References

[1] ABDULAH, S., LTAIEF, H., SUN, Y., GENTON, M. G., AND KEYES, D. E. Geostatistical modeling and prediction using mixed precision tile cholesky factorization. *HIPC* (2019), 152–162.

[2] AGULLO, E., AUMAGE, O., FAVERGE, M., FURMENTO, N., PRUVOST, F., SERGENT, M., AND THIBAULT, S. P. Achieving high performance on supercomputers with a sequential task-based programming model. *IEEE Transactions on Parallel and Distributed Systems* (2017).

[3] AGULLO, E., BEAUMONT, O., EYRAUD-DUBOIS, L., HERRMANN, J., KUMAR, S., MARCHAL, L., AND THIBAULT, S. Bridging the gap between performance and bounds of cholesky factorization on heterogeneous platforms. In *IPDPSW* (2015), IEEE, pp. 34–45.

[4] AGULLO, E., BEAUMONT, O., EYRAUD-DUBOIS, L., AND KUMAR, S. Are static schedules so bad? a case study on cholesky factorization. In *IPDPS* (2016), IEEE, pp. 1021–1030.

[5] AGULLO, E., HADRI, B., LTAIEF, H., AND DONGARRA, J. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In *SC'09. ACM/IEEE Conference on Supercomputing, Portland, OR, November* (2009).

[6] AMBIKASARAN, S., AND DARVE, E. An $o(n \log n)$ fast direct solver for partial hierarchically semi-separable matrices. *Journal of Scientific Computing 57*, 3 (2013), 477–501.

[7] AMESTOY, P., ASHCRAFT, C., BOITEAU, O., BUTTARI, A., L'EXCELLENT, J.-Y., AND WEISBECKER, C. Improving multifrontal methods by means of block low-rank representations. *SIAM Journal on Scientific Computing 37*, 3 (2015), A1451–A1474.

[8] AMINFAR, A., AMBIKASARAN, S., AND DARVE, E. A fast block low-rank dense solver with applications to finite-element matrices. *Journal of Computational Physics 304* (2016), 170–188.

[9] ARNOLD, B. C., BALAKRISHNAN, N., AND NAGARAJA, H. N. *A first course in order statistics.* SIAM, 2008.

[10] AUGONNET, C., GOUDIN, D., KUHN, M., LACOSTE, X., NAMYST, R., AND RAMET, P. A hierarchical fast direct solver for distributed memory machines with manycore nodes. Research report, CEA;Total; Université de Bordeaux, Oct. 2019.

[11] AUGONNET, C., THIBAULT, S., NAMYST, R., AND WACRENIER, P.-A. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009 23* (Feb. 2011), 187–198.

[12] BADIA, R. M., HERRERO, J. R., LABARTA, J., PÉREZ, J. M., QUINTANA-ORTÍ, E. S., AND QUINTANA-ORTÍ, G. Parallelizing dense and banded linear algebra libraries using SMPSs. *Concurrency and Computation: Practice and Experience 21*, 18 (2009), 2438–2456.

[13] BEAUMONT, O., BECKER, B. A., DEFLUMERE, A., EYRAUD-DUBOIS, L., LAMBERT, T., AND LASTOVETSKY, A. Recent advances in matrix partitioning for parallel computing on heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems 30*, 1 (2018), 218–229.

[14] BEAUMONT, O., LIONEL, E.-D., AND VERITE, M. 2D Static Resource Allocation strategies for load balancing in for Compressed Linear Algebra and Communication Constraints. preprint `https://hal.inria.fr/view/index/docid/2900244`, July 2020.

[15] BERGMAN, D., CARDONHA, C., AND MEHRANI, S. Binary decision diagrams for bin packing with minimum color fragmentation. In *CPAIOR* (2019), Springer, pp. 57–66.

[16] BEZIAU, P. Data distribution strategies for cholesky decomposition. In *Compas* (2019).

[17] BLACKFORD, L. S., CHOI, J., CLEARY, A., D'AZEVEDO, E., DEMMEL, J., DHILLON, I., DONGARRA, J., HAMMARLING, S., HENRY, G., PETITET, A., STANLEY, K., WALKER, D., AND WHALEY, R. C. *ScaLAPACK Users' Guide.* Society for Industrial and Applied Mathematics, 1997.

[18] Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, A., Herault, T., Kurzak, J., Langou, J., Lemarinier, P., Ltaief, H., Luszczek, P., YarKhan, A., and Dongarra, J. Flexible development of dense linear algebra algorithms on massively parallel architectures with dplasma. In *IPDPSW* (May 2011), pp. 1432–1441.

[19] Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., and Dongarra, J. DAGuE: A generic distributed dag engine for high performance computing. *Parallel Computing 38*, (1-2) (2012), 37–51.

[20] Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing 35*, 1 (2009), 38 – 53.

[21] Cao, Q., Pei, Y., Herault, T., Akbudak, K., Mikhalev, A., Bosilca, G., Ltaief, H., Keyes, D., and Dongarra, J. Performance analysis of tile low-rank Cholesky factorization using parsec instrumentation tools. In *ProTools* (2019), ACM.

[22] Carratalá-Sáez, R., Faverge, M., Pichon, G., Sylvand, G., and Quintana-Ortí, E. Tiled algorithms for efficient task-parallel h-matrix solvers. In *PDSEC* (2020).

[23] Casanova, H., Legrand, A., and Quinson, M. Simgrid: A generic framework for large-scale distributed experiments. In *Tenth International Conference on Computer Modeling and Simulation (uksim 2008)* (2008), IEEE, pp. 126–131.

[24] Chan, E., Van Zee, F. G., Bientinesi, P., Quintana-Ortí, E. S., Quintana-Ortí, G., and van de Geijn, R. Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks. In *PPoPP '08* (2008), ACM, pp. 123–132.

[25] Choi, J., Dongarra, J. J., Ostrouchov, L. S., Petitet, A. P., Walker, D. W., and Whaley, R. C. Design and implementation of the scalapack lu, qr, and cholesky factorization routines. *Sci. Program. 5*, 3 (Aug. 1996), 173–184.

[26] Coffman, E., Frederickson, G., and Lueker, G. Probabilistic analysis of the lpt processor scheduling heuristic. In *Deterministic and stochastic scheduling*. Springer, 1982, pp. 319–331.

[27] Cojean, T., Guermouche, A., Hugo, A., Namyst, R., and Wacrenier, P.-A. Resource aggregation for task-based cholesky factorization on top of modern architectures. *Parallel Computing 83* (2019), 73–92.

[28] Epstein, L., Imreh, C., and Levin, A. Class constrained bin packing revisited. *Theoretical Computer Science* (2010).

[29] Faverge, M., Pichon, G., Ramet, P., and Roman, J. On the use of h-matrix arithmetic in pastix: a preliminary study. In *Workshop on Fast Solvers, Toulouse, France* (2015).

[30] Golub, G. H., and Van Loan, C. F. *Matrix computations*, vol. 3. JHU press, 2012.

[31] Grasedyck, L., and Hackbusch, W. Construction and arithmetics of h-matrices. *Computing 70*, 4 (2003), 295–334.

[32] Grigni, M., and Manne, F. On the complexity of the generalized block distribution. In *International Workshop on Parallel Algorithms for Irregularly Structured Problems* (1996), Springer, pp. 319–326.

[33] Gunnels, J. A., Gustavson, F. G., Henry, G. M., and van de Geijn, R. A. Flame: Formal linear algebra methods environment. *ACM Trans. Math. Softw. 27*, 4 (Dec. 2001), 422–455.

[34] Hackbusch, W. A sparse matrix arithmetic based on h-matrices. *Computing 62*, 2 (1999), 89–108.

[35] Ida, A. Lattice h-matrices on distributed-memory systems. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2018), IEEE, pp. 389–398.

[36] Irony, D., Toledo, S., and Tiskin, A. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput. 64*, 9 (Sept. 2004), 1017–1026.

[37] JACQUELIN, M., ZHENG, Y., NG, E., AND YELICK, K. An asynchronous task-based fan-both sparse cholesky solver. *arXiv preprint arXiv:1608.00044* (2016).

[38] JANSEN, K., LASSOTA, A., AND MAACK, M. Approximation algorithms for scheduling with class constraints. *arXiv preprint arXiv:1909.11970* (2019).

[39] KALINOV, A., AND LASTOVETSKY, A. Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers. *Journal of Parallel and Distributed Computing 61*, 4 (2001), 520–535.

[40] KIM, K., RAJAMANICKAM, S., STELLE, G., EDWARDS, H. C., AND OLIVIER, S. L. Task parallel incomplete cholesky factorization using 2d partitioned-block layout. *arXiv preprint arXiv:1601.05871* (2016).

[41] KURZAK, J., LTAIEF, H., DONGARRA, J., AND BADIA, R. M. Scheduling dense linear algebra operations on multicore processors. *Concurrency and Computation: Practice and Experience 22*, 1 (2010), 15–44.

[42] MARY, T. *Block Low-Rank multifrontal solvers: complexity, performance, and scalability*. PhD thesis, 2017.

[43] PÉREZ, J. M., BADIA, R. M., AND LABARTA, J. A flexible and portable programming model for SMP and multi-cores. Tech. rep., Barcelona Supercomputing Center, 2007.

[44] PICHON, G., DARVE, E., FAVERGE, M., RAMET, P., AND ROMAN, J. Sparse supernodal solver using block low-rank compression: Design, performance and analysis. *Journal of computational science 27* (2018), 255–270.

[45] QUINTANA-ORTÍ, E. S., QUINTANA-ORTÍ, G., VAN DE GEIJN, R. A., VAN ZEE, F. G., AND CHAN, E. Programming matrix algorithms-by-blocks for thread-level parallelism. vol. 36.

[46] SOLOMONIK, E., AND DEMMEL, J. Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. In *EuroPar* (2011), Springer.

[47] SONG, F., YARKHAN, A., AND DONGARRA, J. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *SC'09* (2009).

[48] YAMAZAKI, I., IDA, A., YOKOTA, R., AND DONGARRA, J. Distributed-memory lattice h-matrix factorization. *The International Journal of High Performance Computing Applications 33*, 5 (2019), 1046–1063.

[49] YAŞAR, A., AND ÇATALYÜREK, Ü. V. Heuristics for symmetric rectilinear matrix partitioning. *arXiv preprint arXiv:1909.12209* (2019).