Using Static Allocation Algorithms for Matrix Matrix Multiplication on Multicores and GPUs

Lionel Eyraud-Dubois, Thomas Lambert

Using Static Allocation Algorithms for Matrix Matrix Multiplication on Multicores and GPUs

Lionel Eyraud-Dubois
Inria
University of Bordeaux
lionel.eyraud-dubois@inria.fr

Thomas Lambert
University of Manchester
thomas.lambert@inria.fr

ABSTRACT

We consider the problem of data allocation when performing matrix multiplication on a heterogeneous node, with multicores and GPUs. Classical (cyclic) allocations designed for homogeneous settings are not appropriate, but the advent of task-based runtime systems makes it possible to use more general allocations. Previous theoretical work has proposed square and cube partitioning algorithms aimed at minimizing data movement for matrix multiplication. We propose techniques to adapt these continuous square partitionings to allocating discrete tiles of a matrix, and strategies to adapt the static allocation at runtime. We use these techniques in an implementation of Matrix Multiplication based on the StarPU runtime system, and we show through extensive experiments that this implementation allows to consistently obtain a lower communication volume while improving slightly the execution time, compared to standard state-of-the-art dynamic strategies.

CCS CONCEPTS

• **Mathematics of computing** → *Solvers*; • **Computing methodologies** → *Linear algebra algorithms*; *Parallel algorithms*; • **General and reference** → *Performance*;

ACM Reference Format:
Lionel Eyraud-Dubois and Thomas Lambert. 2018. Using Static Allocation Algorithms for Matrix Matrix Multiplication on Multicores and GPUs. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225066

1 INTRODUCTION

The current trend in the architecture of High Performance Computing platforms is to go towards larger scale and more heterogeneous systems. Heterogeneity comes mainly with the increased usage of specialized accelerators (like GPUs) together with the more generic CPU multicores. The increase in scale allows to achieve higher and higher flop counts, but such a growth places a high strain on the communication infrastructure of these machines, whose communication capabilities have durably grown slower than their computational power. In this context, it is crucial to be able to lower the amount of communications incurred by numerical computations, since this has an effect both on the performance of the computation (by reducing contention on communication links) and on the corresponding energy consumption. Indeed, data movements account for a significant part of the total energy consumption of such machines, which becomes a bottleneck as their scale continues to increase.

In this paper, we consider one of the most fundamental numerical computational routine, namely matrix multiplication. Parallel implementations of this routine have existed for a long time. These implementations have been designed for homogeneous architectures, usually using workload repartition among the nodes based on cyclic or block-cyclic distribution. Adapting these distributions to heterogeneous settings is actually a hard algorithmic and technical challenge. However, to cope with the increasing heterogeneity of the architectures, task-based runtime systems (such as StarPU [1], StarSs [2], ParSEC [3], and others) are currently proposed, with the goal of enabling better performance portability for applications across different architectures. To this end, such runtime systems provide an efficient separation of the numerical computation from the associated scheduling and resource allocation decisions. This makes it possible to consider less constrained workload repartition.

On the other hand, in a more theoretical perspective, the problem of workload distribution for Matrix Multiplication with communication concerns has been modeled as a square partitioning problem [4]. Several successive studies have proposed efficient partitioning algorithms for this problem, ranging from column-based partitioning [5] to non-rectangular partitionings [6]. In this paper, we propose an implementation of tiled Matrix Multiplication for one heterogeneous node made of several CPUs and GPUs, which bridges these two contributions. Thanks to the expressiveness offered by task-based runtime systems, we were able to efficiently implement the partitionings computed by the above mentioned algorithms, and obtain significant reductions of data movement with no degradation (and sometimes slight improvement) of execution time.

The main contributions of this article are the following:

- We propose two different techniques to convert theoretical square partitionings (which are based on a continuous model, and thus assume that data can be divided arbitrarily) into distribution of the (discrete) tiles of the matrix to the different processors.
- We propose and evaluate the performance of several dynamic strategies to help balancing the load over the different processors, allowing to retain the benefit of clever partitioning algorithms while providing quasi-optimal load balance.

ICPP 2018, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225066

- These building blocks allow us to provide an efficient implementation of tiled Matrix Multiplication on heterogeneous nodes.

The paper is organized as follows. Related works are discussed in Section 2. A review of the theoretical models and algorithms is given in Section 3, together with the presentation of techniques to convert continuous partitionings into discrete allocations. Section 4 contains all the details about our proposed implementation, which is evaluated in Section 5. Concluding remarks and perspectives are given in Section 6.

2 RELATED WORKS

In this paper, we consider dense Matrix Multiplication, and we more specifically focus on Cannon-like algorithms, which involve N^3 elementary operations of type $C_{i,j} \leftarrow C_{i,j} + A_{i,k}B_{k,j}$, *i.e.* we ignore variants such as Strassen [7] (for which communication-avoiding algorithms have also been proposed [8]) or Coppersmith-Winograd ([9]). Parallel implementations of Matrix Multiplication have been developed some time ago, notably with the well-known ScaLaPack distribution [10] which uses 2D-cyclic distributions. Recently 2.5D approaches have been considered [11], where several partially aggregated copies of the resulting matrix C are maintained to allow several processors to work in parallel on updating the same result tile.

The square partitioning optimization problem has first been introduced by Lastovetsky and Kalinov [4], and models the 2D case when the communication link is a bus. Beaumont *et al.* proved its NP-completeness [5] and proposed the first approximation algorithm, with a 1.75 ratio. This was improved on the one hand by works on optimal non-rectangular partitionings for 2 and 3 processors [12, 13], and on the other hand by recursive approximation algorithms, which provide rectangular partitionings with a 1.25-approximation ratio [14] and a 1.15 ratio in the case of weakly heterogeneous processors. The most recent work [6] is NRRP, an algorithm which recursively computes non-rectangular partitionings, and yields a 1.15-approximation ratio in the general case. A natural generalization is to study the cube partitioning problem, thus allowing to compute several contributions to the result matrix C by several different processors. A generalization of NRRP to the cube partitioning problem yields 3D-NRRP [15], which is a 1.51-approximation algorithm for this (harder) problem. A simulated comparison of these static algorithms to dynamic and hybrid strategies has been presented in [16], highlighting the benefit of introducing more static allocation decisions in runtime libraries. However, the present paper represents to the best of our knowledge the first attempt at using these static heterogeneous strategies in real implementations.

Recently, in order to cope with resource heterogeneity and enable performance portability, the use of dynamic runtime schedulers have been proposed, such as StarPU [1], StarSs [2], QUARK [17] or PaRSEC [3]. Applications are described as a set of tasks, whose dependencies can be automatically deduced from access to shared data with the STF model [1], or explicitly specified [3]. At runtime, the scheduler takes the scheduling and allocation decisions based on the set of ready tasks (tasks whose data and control dependences have all been resolved), on the availability of the resources (estimated using expected processing and communication times), and on the actual location of input data.

3 MODELS, ALGORITHMS AND ADAPTATION

3.1 Models

3.1.1 2D-Partitioning. The standard matrix multiplication $C = A \times B$ for square matrices can be seen as a set of N^3 tasks

$$T_{i,j,k} : C_{i,j} \leftarrow C_{i,j} + A_{i,k}B_{k,j}$$

for $\{i, j, k\} \in [1, N]^3$. Each task $T_{i,j,k}$ needs exactly three elements, $C_{i,j}$, $A_{i,k}$ and $B_{k,j}$ to be computed and each of these elements are also needed by other tasks. For example if two tasks $T_{i,j,k}$ and $T_{i,j,k'}$ are allocated to two different processors, respectively M_1 and M_2, then replicates of $C_{i,j}$ must reside in the local memory of M_1 and M_2. Note that what we call elements, the $A_{i,k}$s, $B_{k,j}$s and $C_{i,j}$s, are tiles, *i.e.* sub-matrices of the original A, B and C. Our goal here is to avoid such replications as much as possible, under the assumption that we aim for an optimal execution time. This assumption is theoretically achieved simply by assigning to each processor p_l a number of $T_{i,j,k}$s proportional to its computational power, so that all processors finish their computation at the same time.

In the first model we allocate tasks by bags. For a fixed k we allocate $T_{i,j,k}$ between the different processors and reproduce this allocation for each k, thus simplifying the problem to consider only two dimensions (the $A_{i,k}$'s and the $B_{k,j}$'s). In addition, this implies that each $C_{i,j}$ is allocated to a single processor, avoiding the problem of concurrent writing. This case can be modeled by the square partitioning problem PERI-SUM. To give a formal definition, we introduce the following notations. Let Z be a zone included into a unitary square $S = [0, 1] \times [0, 1]$. We define by $s(Z)$ its area (formally $\iint_Z dxdy$). Let $\Pi_1(Z) = \{x,\ \exists (x, y) \in Z\}$ and $\Pi_2(Z) = \{y,\ \exists (x, y) \in Z\}$ be the projections of Z on both dimensions and denote by $\pi_1(Z)$ and $\pi_2(Z)$ their sizes ($\pi_1(Z) = |\Pi_1(Z)|$ and $\pi_2(Z) = |\Pi_2(Z)|$). Finally, we define the half-perimeter of Z as $p(Z) = \pi_1(Z) + \pi_2(Z)$, see Figure 1.

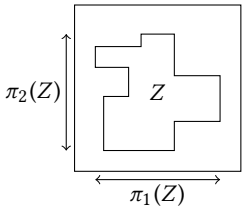
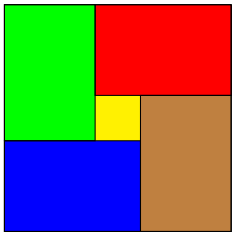


Figure 1: Illustration of the definition of $\pi_1(Z)$ and $\pi_2(Z)$.

Problem 1 (PERI-SUM)**.** Given a set of m strictly positive rational numbers $\{s_1, \ldots, s_m\}$ such that $\sum s_k = 1$, and the square $S = [0, 1] \times [0, 1]$, find for each s_k a zone $Z_k \in S$ such that the surface of Z_k is s_k, $\bigcup Z_k = S$, and such that $\sum p(Z_k)$ is minimized.



$$\begin{aligned} & 2/5 + 3/5 \\ + \; & 3/5 + 2/5 \\ + \; & 3/5 + 2/5 \\ + \; & 2/5 + 3/5 \\ + \; & 1/5 + 1/5 \\ \sum p(Z) \; = \; & 22/5 \end{aligned}$$

Figure 2: Illustration of PERI-SUM.

An illustration of PERI-SUM can be found on Figure 2. This problem has been firstly introduced by Kalinov and Lastovetsky [4] and proven NP-Complete by Beaumont et al. [5].

PERI-SUM models the allocation of a matrix product in the following way: the square to be partitioned represents the result matrix C (or more precisely one step of its computation) with its different elements, the $C_{i,j}$s. The values $\{s_1, \ldots, s_m\}$ represent the relative speeds of the processors ($\frac{w_i}{\sum w_j}$, where w_i is the computing power of processor i) used to perform the parallel matrix multiplication: indeed, a faster processor should compute a larger set of $C_{i,j}$s. To compute $C_{i,j}$, $A_{i,k}$ and $B_{k,j}$ are needed. Therefore, to compute a subset Z of elements of C, a processor has to load $\{A_{i,k} \mid C_{i,j} \in Z\} \sim \Pi_1(Z)$ and $\{B_{k,j} \mid C_{i,j} \in Z\} \sim \Pi_2(Z)$ and thus the amount of communication for this processor is represented by the sum of the projections, *i.e.* its half-perimeter $p(Z)$.

3.1.2 3D-Partitioning. An alternative way to model the matrix product is to consider the whole set of tasks at once, adding a dimension to the previous problem. To simplify the model, the $A_{i,k}$s, the $B_{k,j}$s and the $C_{i,j}$s are considered equally, even if these last ones are actually read-write data and not read-only data. The problem becomes a cube partitioning problem MSCubeP [15], where each element of the cube represents a task $T_{i,j,k}$. We define the following notations. Let Z be a zone included into a cube Cu. We define by $v(Z)$ its volume. Let $\Pi_1(Z) = \{(x,z),\ \exists y,\ (x,y,z) \in Z\}$, $\Pi_2(Z) = \{(y,z),\ \exists x,\ (x,y,z) \in Z\}$ and $\Pi_3(Z) = \{(x,y),\ \exists z,\ (x,y,z) \in Z\}$ be the projections of Z on the three faces and $\pi_1(Z) = |\Pi_1(Z)|$, $\pi_2(Z) = |\Pi_2(Z)|$ and $\pi_3(Z) = |\Pi_3(Z)|$ be their sizes. Finally, we define the half-covering surface of Z as $Hs(Z) = \pi_1(Z) + \pi_2(Z) + \pi_3(Z)$.

Problem 2 (Minimizing-Surface-Cube-Partition (MSCubeP))**.** Given a set of m rational numbers $\{v_1, \ldots, v_m\}$ such that $\sum v_k = 1$, and the cube $Cu = [0,1] \times [0,1] \times [0,1]$, find a set of zones Z_k of Cu such that $v(Z_k) = v_k$ and $\bigcup Z_k = Cu$, so as to minimize $\sum Hs(Z_k)$.

3.2 Algorithms

In this paper, we schedule parallel matrix multiplication using the algorithms with the best proven approximation ratio for PERI-SUM and MSCubeP (respectively $\frac{2}{\sqrt{3}} \simeq 1.15$ and $\frac{5}{6^{2/3}} \simeq 1.51$).

The first algorithm, NRRP, has been introduced by Beaumont et al. [6] and we refer the interested reader to this paper for a complete description of the algorithm (omitted here to save space). In summary, NRRP is a divide-and-conquer algorithm where, at each step, the current set of processors is split into two subsets (and sometimes more in special cases). To accomodate for this, the current rectangle (a square at the beginning of the algorithm) is partitioned into zones whose areas correspond to the the sum of relative speeds of processors attributed to each subset. Note that final zones may have non rectangular shapes.

The second algorithm, 3D-NRRP, has been introduced by Beaumont et al. [15]. 3D-NRRP is an adaptation of a simplified version of NRRP to MSCubeP, with the same basic principle: the current cuboid is recursively split into two 3-dimensional zones.

3.3 Adaptation

In the case of NRRP, the solution is a set of zones, where each zone is the union of at most two rectangles. In the case of 3D-NRRP, the solution is a set of polyhedra, where a polyhedron is the union of at most three cuboids.

However, both PERI-SUM and MSCubeP are continuous problems. This implies that the rectangles and cuboids from the outputs of these algorithms may have non-integer coordinates. Therefore the results of these algorithms have to be transformed before being used on an $N \times N$ matrix multiplication.

In this paper we propose two approaches:

- Rounded routine, where the coordinate of each rectangle is rounded to the closest integer. This ensures that the projections on the different axes remain as close as possible to the original solution, however this may significantly degrade the load-balancing between processors.
- Precise routine, in which the perfect load-balancing is preserved at the cost of deformations of the zones and additional communication cost.

The Precise routine is done as follows: we first compute the exact number of tasks each processor should perform, based on the input speeds $\{s_1, \ldots, s_m\}$, by rounding the partial sums. More precisely, with Round defined as the classical rounding operation (if $x - \lfloor x \rfloor < 0.5$ then $\text{Round}(x) = \lfloor x \rfloor$, else $\text{Round}(x) = \lceil x \rceil$),

$$n_1 = \text{Round}(s_1 \times N^2)$$
$$n_k = \text{Round}(N^2 \times \sum_{i=1}^{k} s_i) - \sum_{i=1}^{k-1} n_i$$

where N is the size of the matrices (in the case of solutions of MSCubeP, replace N^2 by N^3).

The Precise routine then goes in two waves (we describe here the 2D version for simplicity):

- First, each rectangle is reduced to its *inner rectangle*, *i.e.* the largest rectangle with integer coordinates fully contained in the initial rectangle. More precisely, if $R = [x_1, x_2] \times [y_1, y_2]$, its inner rectangle is the rectangle $R_{in} = [\lceil x_1 \rceil, \lfloor x_2 \rfloor] \times [\lceil y_1 \rceil, \lfloor y_2 \rfloor]$. Each element of an inner rectangle is assigned to the corresponding processor and the n_is are updated ($n_i \leftarrow n_i - s(R_{in,i})$).
- Second, all unassigned elements of the matrix (*i.e.* elements that are not in an inner rectangle) are assigned one after the other according to the following criterion. For each element t, if among its (up to) 8 neighboring elements some are already assigned to processors with strictly positive n_is, then t is assigned to the processor among them with the smallest n_i. Otherwise (if neighbors of t are unassigned or assigned to

processors p_is with $n_i = 0$), t is assigned to the processor with the smallest strictly positive n_i.

Algorithm 1 is a transcription of PRECISE in the two dimensional case.

Algorithm 1: PRECISE-2D($N, ((R_1, p_1), \ldots, (R_q, p_q)), (s_1, \ldots, s_m)$)

Input: An integer N, a set of q couples composed of a rectangle R_k and a processor ID p_k such that $\bigcup R_k = [0,1] \times [0,1]$ and the set of the m relative speeds ($\sum s_k = 1$).

Output: An $N \times N$ matrix M where $M_{i,j} = p_k$ if and only if the output element $C_{i,j}$ is computed by p_k.

```
M = −1 ;
n_1 = Round(s_1 × N^2) ;
for k = 2 to m do
    n_k = Round(N^2 × Σ_{i=1}^{k} s_i) − Σ_{i=1}^{k−1} n_i ;
for k = 1 to q do
    [x_1, x_2] × [y_1, y_2] = R_k ;
    for i = ⌈x_1⌉ to ⌊x_2⌋ and j = ⌈y_1⌉ to ⌊y_2⌋ do
        M_{i,j} = p_k ; n_{p_k} −− ;
foreach (i, j) such that M_{i,j} = −1 do
    p_k = processor which minimizes
      {n_{p_k'}, n_{p_k'} > 0, M_{i',j'} = p_k', |i − i'| ≤ 1, |j − j'| ≤ 1} ;
    if p_k exists then
        M_{i,j} = p_k ; n_{p_k} −− ;
    else
        p_k = processor which minimizes {p_k', n_{p_k'} > 0} ;
        M_{i,j} = p_k ; n_{p_k} −− ;
return M ;
```

4 IMPLEMENTATION

4.1 StarPU

StarPU [1] is a task programming library for hybrid architectures developed at Inria Bordeaux Sud-Ouest. For our needs we only use a fraction of the possibilities of StarPU. This section does not claim to provide a good overview of StarPU but focuses on the parts of StarPU we use in order to understand how we implement our scheduler and what difficulties we encountered.

4.1.1 Tasks. The central objects in StarPU are *tasks*. A program using StarPU usually consists in a set of tasks that are submitted to StarPU that handles the scheduling and dispatching of these tasks among available resources. We will discuss it later, but note that StarPU provides some schedulers while also allowing programmers to create their own.

The structure of a task is very simple. The main component is the *codelet* which contains the function that is executed when the task is processed. Since StarPU targets heterogeneous environments, in particular platforms with CPUs and GPUs, a codelet may contains several implementations of this function, one for each type of processor. For example, in our case, we need to provide a GEMM function that works on CPU and a GEMM function that works on GPU. In addition, codelets are also associated to a *performance model*. At each execution of a given codelet on a processing node, StarPU updates this performance model with the timing of this execution. This information allows StarPU to evaluate the performance of this codelet on a given processor, notably to produce an estimation of the computation time.

4.1.2 Workers. For StarPU, each processor is a *worker*. Each worker has a type (CPU or GPU for example) and is associated to a *memory node*. Memory nodes represent parts of the available memory, and a worker can only access data stored on its own memory node. StarPU generates explicit data transfers to copy data from one memory node to another. In the platform we consider in Section 5, all CPUs share the same memory node (that is also the main RAM) and each GPU has its own memory node. As our algorithms (NRRP and the others) aim at reducing the number of data transfers, we chose to not distinguish two workers having the same memory node and to consider them as a single worker during the static allocation (memory nodes in StarPU, with aggregated speeds of all associated workers, are exactly the processors in the algorithms of Sections 3.2 and 3.3).

4.1.3 Scheduler. As stated above, StarPU provides some pre-implemented schedulers. In particular, the DMDA scheduler uses the performance models to evaluate the computation and communication times of each ready task on all workers, and selects the worker on which the task will finish the earliest. This default scheduler provides good performance for most applications, allowing the programmer to focus on writing the codelets and defining the tasks with their input and output data.

StarPU allows to define custom schedulers: a StarPU scheduler consists of several functions, the two most important ones are:

- `push_task`: Called when a submitted task is ready (*i.e.* all its predecessor tasks in the dependency graph are done), mostly to put the task in a task list (that can be general or for a specific worker/memory node).
- `pop_task`: Called when a worker is idle. It is used to choose the next task to be processed by this worker.

The task cycle of StarPU is then simple. As soon as a task has been submitted with all its dependencies solved, this task is pushed (with `push_task`) into the system and ready to be executed. Later, a worker will receive this task (with `pop_task`) and process it.

4.2 Dynamic Strategies

We present here the dynamic scheduling strategies that we implemented within StarPU. We can distinguish two kinds: work-stealing strategies and purely dynamic strategies. The first ones complement a static allocation (turning it into a hybrid strategy), the second ones do not rely on an initial allocation. For both, we define the cost of a task for a memory node as the number of matrix elements to move or duplicate to execute this task on a worker that belongs to this memory node (therefore this cost can be between 0 and 3).

4.2.1 Work-Stealing. The goal of work-stealing strategies is to fix possible load imbalance encountered during the execution. Generally speaking, when a worker is idle with no remaining attributed tasks on its memory node, work-stealing strategies determine a

task to be stolen from another memory node, which may imply some data transfers.

We propose three strategies:

- RandSteal: Choose a memory node at random and steal the last ready task submitted on it. If there is no remaining task on this memory node, RandSteal tries the next one (in Round-Robin fashion) until a task is found or all memory nodes have been checked.
- ChoiceSteal: Check the last submitted tasks of each memory node and choose the one with the smallest cost.
- EffectiveSteal: Check all unprocessed tasks to find the minimum cost task. To limit the need for synchronization between workers, EffectiveSteal does not lock the whole task list to do so. Instead, only individual task lists of each memory nodes are locked, and during a first pass, EffectiveSteal stops if there is a task with cost 0. Otherwise, EffectiveSteal remembers the memory nodes that own a task of cost 1 or 2 and attempts to find these tasks during a second pass (they might have been processed in the meantime). If after these two passes there is still no task to steal, then RandSteal is called.

In addition to these three strategies, Static denotes a strategy without work-stealing where the predefined task allocation is enforced until the end.

4.2.2 Purely Dynamic Strategies. We now present strategies which do not rely on an initial static allocation, and which are used for comparison purposes. All of them rely on a global list which contains all ready unprocessed tasks. Each strategy is defined by a function that is called by idle workers. We implement three such strategies:

- FirstDyn: Idle workers begin the execution of the first submitted and unprocessed task.
- ChoiceDyn-X: An idle worker looks at the (at most) X first submitted and unprocessed tasks and chooses the one with smallest cost.
- EffectiveDyn: An idle worker looks at all the submitted and unprocessed tasks and chooses the one with the smallest cost.

In addition, we also use DMDA, an already implemented task-centric dynamic scheduler based on MCT (each task, at submission, is allocated to the worker that will complete it first). Note that DMDA is communication-aware, and evaluates the data transfer duration to choose among the workers.

4.3 Pre-fetching and Reduction

4.3.1 Prefetching. When prefetching is not enabled in StarPU, the data required by a task is loaded into a memory node just before the beginning of its execution. With such behavior, the communications would not be overlapped with computations, and this would create an important difference with our assumptions and would also degrade the overall execution time. This is why, in our implementation, we use the StarPU prefetch option that allows to start data transfers during the planning of a future task execution. More precisely, we chose to pre-allocate two tasks in advance to each worker. Hence, whenever a worker starts processing a task, this worker also begins loading the data for the next two tasks. In practice, we also use work-stealing in advance: a task is not stolen when a worker is really idle, but when it has not enough ready tasks (strictly less than 2). Note that in all algorithms, stealing already prefetched tasks is forbidden in order to avoid additional communications.

4.3.2 Reduction. One problem that has to be avoided during the computation of the $T_{i,j,k}$ tasks is the concurrent writing of a same element of the output matrix. More precisely, $T_{i,j,k}$ and $T_{i,j,k'}$ use the same output $C_{i,j}$. Therefore these two tasks cannot be scheduled on different workers at the same time. In the default StarPU mode, the scheduler assumes that there is a dependency between them. More precisely, if $T_{i,j,k}$ is submitted before $T_{i,j,k'}$, then `push_task` is called on $T_{i,j,k'}$ only once $T_{i,j,k}$ is terminated. So there can be no overlap of $T_{i,j,k}$ and $T_{i,j,k'}$.

In the case of 2D-strategies, as $T_{i,j,k}$ and $T_{i,j,k'}$ are scheduled on the same memory node, this is a problem only if there are many workers on the same memory node (the pool of tasks present on the task list of the memory node can be too small to satisfy all the workers, and they may begin to steal tasks from other memory nodes). In the platform we consider in Section 5, all 20 CPUs are on the same memory node and thus the problem occurs. To avoid that, we use the work of Cojean et al. [18] which allows to use several CPUs to work on a given task in parallel. More precisely, the set of CPUs is split into two subsets (one for each socket), each can be considered as a single worker. This trick reduces the number of workers on this memory node to 2 and avoids the problem of early stealing. Since GEMMs are highly parallel, this has very low overhead.

In the case of 3D-strategies, the non-overlapping of $T_{i,j,k}$ and $T_{i,j,k'}$ is a real problem. Indeed, it is very common that two memory nodes share $C_{i,j}$ tiles on many (or all) of their tasks. In such a case, one of the two memory nodes must wait for the completion of many tasks from the other memory node before beginning its own set, implying idle workers for a long time or early steals. To avoid this problem, we create reduction tasks, at the cost of additional data. For each $C_{i,j}$, if at least two memory nodes share this $C_{i,j}$, one $C_{i,j}^{aux}$ is created for each memory node that needs it. The use of different data means that StarPU does not consider the tasks as having a dependency and both can be pushed simultaneously. In addition, after the submission of all GEMM tasks, we create reduction tasks, each having two tiles as input data, $C_{i,j}$ and one $C_{i,j}^{aux}$. The reduction tasks are then executed after the GEMM tasks, ensuring data consistency. However, these additional tasks also imply additional computation time and data transfers, whose costs have to be evaluated.

5 EXPERIMENTAL RESULTS

In this Section we present the experimental results from our StarPU implementation. We first present results from a recent node, and also quickly show that the conclusions also apply to an older node with similar architecture. All the code that was necessary for these experiments can be found online [19].

5.1 Settings

The tests we run are made on a `sirocco` node of PlaFRIM2 [20]. A `sirocco` node is composed of 4 GPUs and 24 CPUS (4 of which are dedicated to GPU management). More precisely the platform contains:

- 2 Dodeca-core Haswell Intel® Xeon® E5-2680 v3 @ 2,50 GHz,
- 128 GB of RAM,
- 4 Nvidia GK110BGL [Tesla K40m] (rev a1).

StarPU allows to plug several BLAS libraries. For our experiments, we use the MKL library (version 11.2) for the GEMM operation on CPUs and cuBLAS library (version 7.5.18) for the GEMM operation on GPUs. We use the 1.3 version of StarPU.

We perform our experiments with double precision matrices of size 7680, 15360, 23040 and 30720 (the size of the matrices A, B and C) split into square tiles of size 960 (the size of one $A_{i,k}$, $B_{k,j}$ or $C_{i,j}$). Therefore the matrices of tiles, which are then split and distributed (with a PERI-SUM or MSCubeP solver) among the memory nodes, have size 8, 16, 24 and 32. The size of the tile is chosen to achieve the best compromise: larger tile sizes increase the efficiency of the execution on the GPUs, but incur more communications because of a lower granularity.

As stated earlier, we use NRRP and 3D-NRRP to compute static allocations. Each time two versions are proposed: NRRP-Rounded and NRRP-Precise (respectively 3D-NRRP-Rounded and 3D-NRRP-Precise) and each version is tested on every work-stealing strategy. In addition, each version of 3D-NRRP is used with and without reduction. Therefore, we have:

- 5 purely dynamic strategies (DMDA, FirstDyn, ChoiceDyn-10, ChoiceDyn-50, EffectiveDyn).
- 4 work-stealing strategies (Static, RandSteal, ChoiceSteal, EffectiveSteal) and 6 allocation strategies (NRRP-Rounded, NRRP-Precise, 3D-NRRP-Rounded, 3D-NRRP-Precise, 3D-NRRP-Rounded-Redux, 3D-NRRP-Precise-Redux), thus 24 hybrid strategies.

We compare these 29 different strategies on the 4 different sizes for matrices (8×8, 16×16, 24×24 and 32×32 tiles). For each configuration we perform 25 runs.

In the rest of this Section, we consider specific questions about the performance of some of the above strategies, so as to make the analysis easier to follow. An overall comparison of the best strategies is provided at the end of the section.

5.2 Dynamic strategies

We first focus on the dynamic strategies, which work by selecting an available task whenever a worker becomes idle. We have proposed several of them (see Section 4.2) to analyze the compromise between the quality of the selected task and the time spent looking for the task. We show the comparison between these strategies on Figure 3. In this Figure, and in all others in this Section, the results are displayed in a combined way: the horizontal axis represents the amount of communication, and the vertical axis represents execution time. For each strategy, a dot is drawn at the average of the values obtained with the 25 runs, and the coordinates of the rectangle are equal to the minimum and the maximum of the

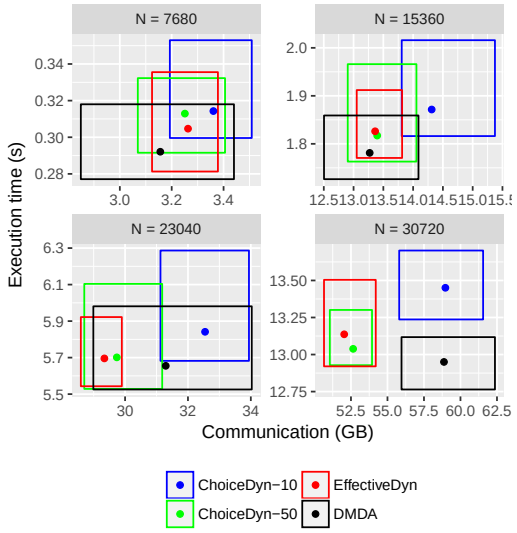


Figure 3: Execution time and communication results for all dynamic strategies

25 measured values, so that all measurements are enclosed within the rectangle. On such plots, the most efficient strategies are at the left and at the bottom. For space efficiency, we did not include the FirstDyn strategy: it obtains much worse performance than the other strategies, and including them on the plot makes it impossible to see the differences between the others.

The first observation to make is that there is a rather small variation between strategies as far as execution time is concerned (around 4% difference). However the differences on the communication scale are much larger, and we can see that ChoiceDyn-10 and DMDA behave significantly worse than ChoiceDyn-50 and EffectiveDyn as soon as the number of tiles is more than 8 ($N > 7680$). This is an indication that it is important to take enough time to look for efficient tasks to execute, and that this time can be recovered by making fewer communications. Since DMDA considers tasks in an arbitrary order, fewer choices are available (choosing between workers for a given task instead of choosing between many tasks for a given worker), which incurs higher communication costs. The performance of ChoiceDyn-50 and EffectiveDyn are rather indistinguishable with the variability involved. In the rest of the experiments, we will thus consider ChoiceDyn-50 as a representative of dynamic strategies, to be compared with strategies based on static allocations.

5.3 Work-stealing strategies

We now study the benefits of the different stealing strategies used to complement the static allocations. As in the previous Section, the question is about the trade-off between spending much time looking for a good task to steal and the quality of the found task in terms of communication reduction. Figure 4 shows the performance obtained by all four stealing strategies (Static, RandSteal, ChoiceSteal, EffectiveSteal) when using the NRRP allocation (other

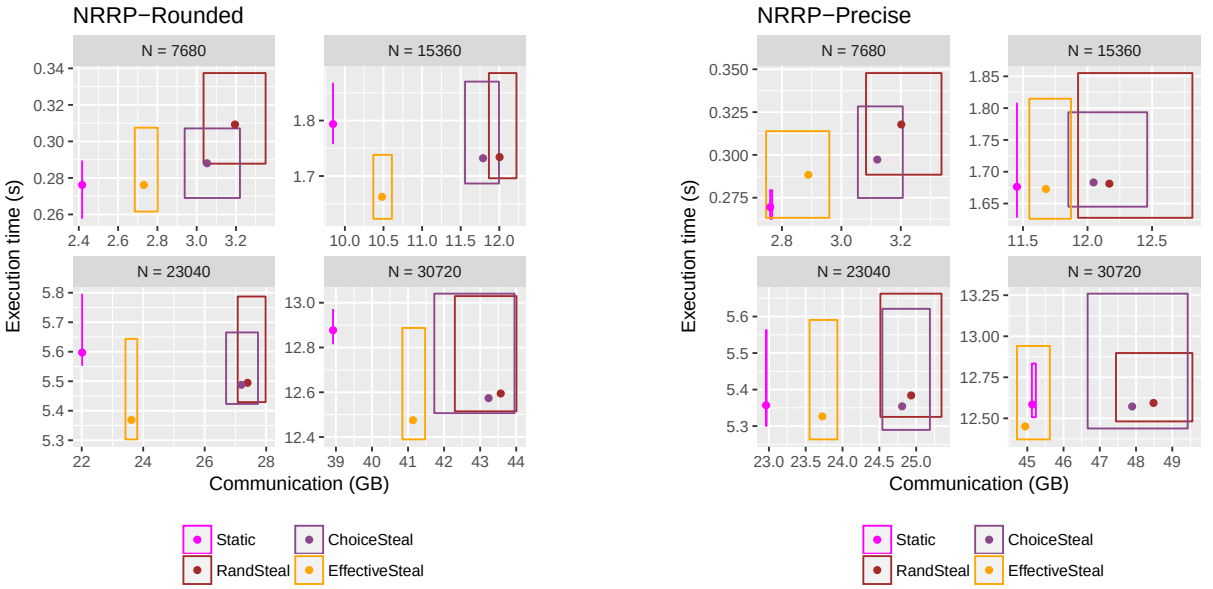


Figure 4: Execution time and communication results for different stealing strategies, with NRRP allocations.

allocations show the same kind of behavior). The results here are different if we consider NRRP-Rounded or NRRP-Precise. With NRRP-Rounded, as expected, the Static strategy obtains the smallest amount of communications, but incurs a rather significantly higher execution time. Rather unexpectedly, more involved stealing strategies obtain better execution times, which shows that the additional time incurred by more precise strategies is not significant with respect to the total running time. Actually, the lower amount of communications involved allows EffectiveSteal to obtain a better execution time, which indicates that searching for a more efficient task does provide a significant improvement.

With the Precise routine, the load is well balanced in the initial allocation, and thus the stealing strategies do not succeed in reducing the execution time. Some stealing still occurs however, and we can still observe a degradation on the amount of communication involved. The conclusion about the different stealing strategies remain however the same: EffectiveSteal performs a lower amount of communications than RandSteal and ChoiceSteal, without any degradation on the execution time. In the following, we will thus focus on two strategies: Static for lowest communications at the cost of higher execution time (especially when paired with Rounded), and EffectiveSteal for competitive execution time with slightly more communications.

5.4 Precise or Rounded

We now consider the differences between the Rounded and Precise routines used to transform the continuous solutions of NRRP into discrete allocations adapted to the decomposition of the matrix into tiles. There is another trade-off here: Precise solutions trade higher communication costs for better load balancing compared to the Rounded algorithm. Figure 5 shows the results obtained by using the allocations of both algorithms, with the Static and

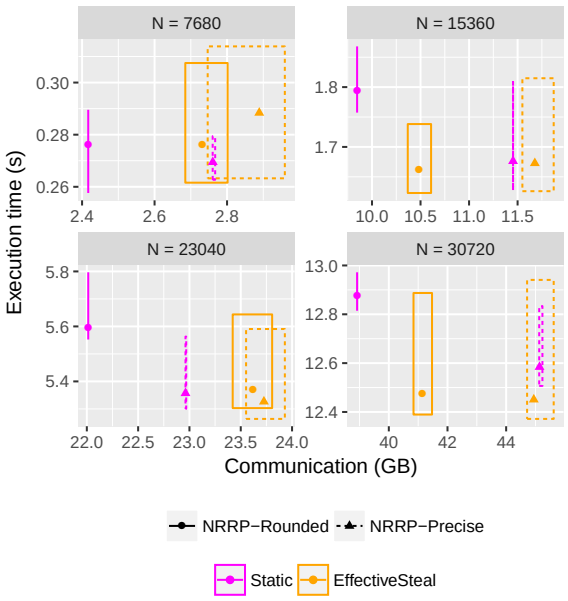


Figure 5: Execution time and communication results for different rounding algorithms.

EffectiveSteal strategies. The results obtained by the Static strategy are as expected: Precise obtains lower execution time at the cost of higher communications, and the difference in execution time is more pronounced when N is larger. However, using a stealing strategy allows to offset the load imbalance of Rounded and to

obtain a similar execution time, and the communication overhead is low enough that this combination yields lower communication costs than even Precise-Static. The exception for $N = 23040$ shows however that such a conclusion is valid for our particular platform, but actually highly depends on how well the continuous matrix partitioning is adapted to the number of tiles of the matrix. Drawing more general conclusions would require to compare a broader set of platforms, and is out of the scope of this paper. Nevertheless, as far as our test platform is concerned, the Rounded-EffectiveSteal combination obtains the best performance overall.

5.5 Benefits of the 3D approach

We now analyze the performance difference between the 2D and 3D approaches to the problem. For the 3D approach, we have two different ways to implement it: the normal one without any reduction task, and the Redux implementation in which we add reduction tasks for computations on the same $C_{i,j}$ tile assigned to different memory nodes. Figure 6 compares the results obtained with NRRP, 3D-NRRP-Rounded and 3D-NRRP-Rounded-Redux, with the Static and EffectiveSteal strategies. Indeed, the conclusions from the previous sections about the relative performance of Precise and of other stealing strategies also hold for the 3D-NRRP allocations.

In order to have a better view of the differences between the other strategies, we have not drawn the results of 3D-NRRP-Rounded with the Static strategy on Figure 6. Indeed, this combination obtains poor performance, that gets worse when N increases, with an execution time up to twice as large as other allocations for $N = 23040$ and $N = 30720$. This ratio can be explained by the fact that two memory nodes share all their $C_{i,j}$ tiles and thus work one after the other. The three other 3D strategies obtain reasonable execution time, but incur a really high communication cost (up to +25% for $N = 23040$ or $N = 30720$ between NRRP-Rounded and 3D-NRRP-Rounded-Redux, both with the Static strategy). The increase of communication for the Redux implementation come for the additional reduction tasks which incur additional data movement because some $C_{i,j}$ tiles have several copies over the platform. On the other hand, the increase of communication for the version without reduction and with EffectiveSteal has the same explanation as the high execution time for its Static counterpart: since two memory nodes share their $C_{i,j}$ tiles, in order to avoid working one after the other they need to exchange many tiles and thus obtain a significantly different allocation.

The 3D-NRRP approach is thus very disappointing on our platform, despite theoretical results that state that it is expected to outperform NRRP in the worst case [21]. This result however holds for large number of processors, and in the platform considered here, we have only 5 memory nodes. We thus expect the 3D approach to obtain better performance if we can run it on a larger platform with more nodes. Another important note is that since the reduction tasks are necessary to make the 3D approach effective, this implies a trade-off between communication and memory usage: reduction tasks imply to have several times the same $C_{i,j}$ tile in memory.

5.6 Overall comparison

In this section we present a comparison between the best strategies according to the previous sections, *i.e.* DMDA, ChoiceDyn-50

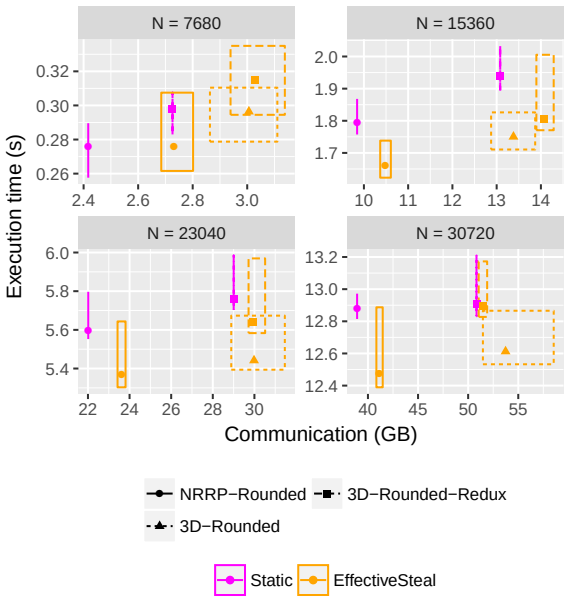


Figure 6: Execution time and communication results for NRRP and 3D-NRRP

and NRRP-Rounded (Static and EffectiveSteal versions). We also compare to results obtained with `Chameleon` [22], which is a state-of-the-art library for linear algebra on heterogeneous platforms. Results can be found in Figure 7, and direct comparison between NRRP-Rounded with EffectiveSteal and `Chameleon` is provided in Table 1.

As can be seen very clearly on this Figure, NRRP-Rounded associated with the EffectiveSteal strategy allows to consistently outperform the dynamic strategies. The differences in execution time between the different strategies can reach 15%, which is significant for a compute-intensive operation like matrix multiplication. For the smallest value of N, `Chameleon` obtains the best execution time. For all other values however, NRRP-Rounded with EffectiveSteal yields a significantly lower execution time than all other strategies. Furthermore, it also yields much lower communication costs, up to twice lower than `Chameleon`, and almost 30% reduction compared to DMDA (see Table 2).

N	Chameleon		NRRP-Rounded	
	Time (s)	Comm (GB)	Time (s)	Comm (GB)
7680	0.242	3.10	0.276 (+14%)	2.73 (-12%)
15360	1.94	17.8	1.66 (-15%)	10.5 (-41%)
23040	6.18	44.6	5.37 (-13%)	23.6 (-47%)
30720	13.4	78.7	12.5 (-7.2%)	41.1 (-48%)

Table 1: Comparison of the performance of NRRP-Rounded with EffectiveSteal strategy against the performance of `Chameleon`.

Figure 8 provides the same results on a different (older) platform called `mirage`. A node of this platform has 36 GB of RAM, and is

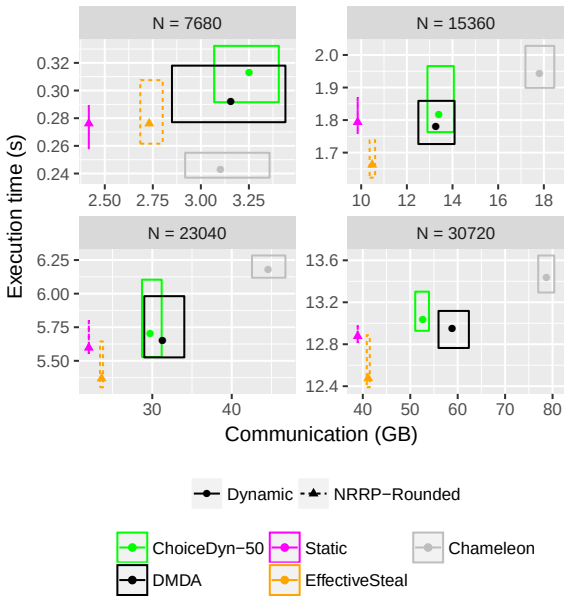


Figure 7: Execution time and communication results for DMDA, ChoiceDyn-50, NRRP-Rounded (Static and EffectiveSteal versions) and `Chameleon`

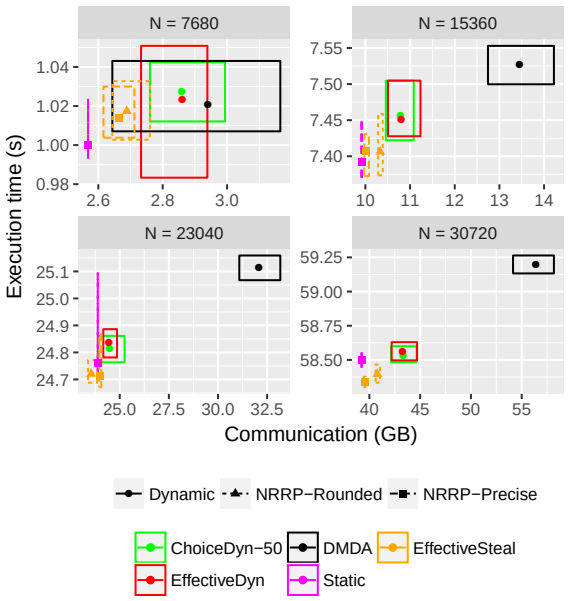


Figure 8: Execution time and communication results on the `mirage` platform.

N	DMDA		NRRP-Rounded	
	Time (s)	Comm (GB)	Time (s)	Comm (GB)
7680	0.292	3.16	0.276 (-5.5%)	2.73 (-14%)
15360	1.78	13.3	1.66 (-6.7%)	10.5 (-21%)
23040	5.66	31.3	5.37 (-5.1%)	23.6 (-25%)
30720	13.0	58.9	12.5 (-3.8%)	41.1 (-30%)

Table 2: Comparison of the performance of NRRP-Rounded with EffectiveSteal strategy against DMDA.

composed of 3 GPUs Nvidia GF100GL (Tesla M2070, rev a3) and 12 CPU cores (2 Hexa-core Westmere Intel® Xeon® X5650 @ 2.67 GHz, with 3 cores dedicated to GPU management)

On Figure 8 we also include NRRP-Precise and EffectiveDyn because, as mentioned earlier, we did not identify a strong difference between EffectiveDyn and ChoiceDyn-50 on `sirocco`, and we expect the choice between Rounded and Precise versions to be highly dependent on the platform. And indeed, the difference in execution time obtained by the Static strategies is much more pronounced than on the `sirocco` platform. Nevertheless, the results obtained by the EffectiveSteal strategies are very close, with a light advantage towards Precise, and both versions of the NRRP algorithm with the EffectiveSteal strategy outperforms the DMDA strategy, whose communication amounts are very consistent across both platforms. The ChoiceDyn-50 and EffectiveDyn strategies perform significantly better on `mirage` compared to `sirocco`, but the NRRP strategies are still the most effective to reduce communication.

Additionally, we present on Figure 9 the results of another experiment performed on the `sirocco` platform, where we analyze the performance gain incurred by including the CPUs into the computation. For this experiment, we have used NRRP, DMDA, ChoiceDyn-50 and EffectiveDyn in two different settings: first in the normal platform which uses all available resources, then on the same node but without using any CPU for the actual computations. The results show that using the CPUs always allow to perform fewer communications. This surprising result comes from the fact that CPUs have a direct access to the main memory, and thus communications for tasks allocated to CPUs are not counted, but allow to send less data to GPUs. Furthermore, except for $N = 7680$ where Static without CPUs performs well, the CPUs actually perform enough computations to incurr a significant reduction in execution time, increasingly so when the size of the matrix increases. We can also see that the strategies based on NRRP are able to make much better usage of the CPUs than dynamic ones: with more CPUs, NRRP obtains much stronger reductions on the execution time and on the communication amount. This experiment demonstrates the interest of making use of the whole computation platform, even for a very regular computation such as matrix multiplication. However, benefiting fully from the performance of such heterogeneous platforms requires to design specialized algorithms like NRRP.

6 CONCLUSION

In this paper we propose to use the theoretical work done during the last years on communication-avoiding algorithms for parallel matrix multiplication to provide a practical implementation of this problem on heterogeneous platforms. For this purpose, we use the StarPU library that allows to write applications in a portable and generic manner. In this implementation, we present dynamic and

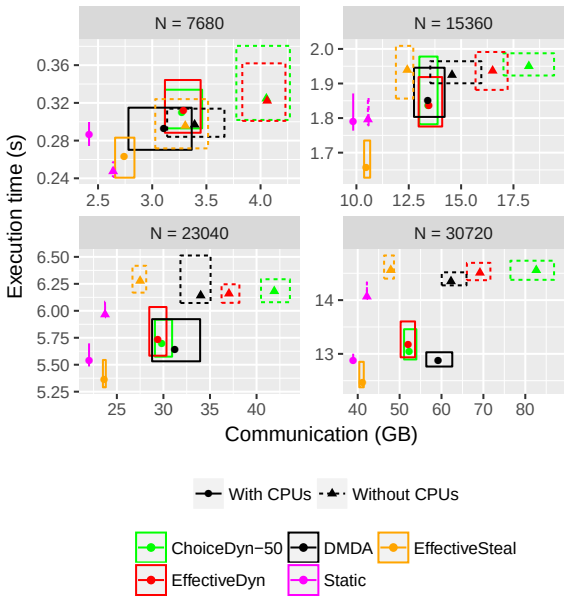


Figure 9: Execution time and communication results on the `sirocco` platform, with and without using the CPUs.

static strategies, as well as hybrid strategies that combine them to produce reliable schedulers. These schedulers use an initial allocation, computed with the effective algorithms from theoretical work, and correct it dynamically if necessary with a work-stealing mechanism. The experimental results are very positive, showing the ability of the hybrid strategies to simultaneously reduce the execution time and the amount of communications, compared to the state-of-the-art generic strategy DMDA and to `Chameleon`, a widely used linear algebra library. In addition to the previous experimental results, this study shows the efficiency of static schedulers with the help of work-stealing techniques, allowing to make the best use of highly heterogeneous platforms. With such techniques, it is possible to keep the good theoretical efficiency of static algorithms while correcting its lack of reliability for practical concerns.

This work opens many challenging research questions, both on the practical and theoretical sides of the problem. First, the low performance of 3D approaches calls for a refinement of the model to take reduction tasks into account (for example by increasing the cost on the axis corresponding to the C matrix); a deeper study about the best way to implement these reduction tasks might also be interesting, as well as testing these solutions on bigger platforms (since the theoretical results predict better performance for the 3D approach on large platforms). Second, in many practical scenarios the matrices involved in the computations are the results of previous computations, and thus may already partially reside in the memory of some GPUs. Studying a more general problem which takes into account an existing data allocation could be very impactful in practice. Finally, generalizations of the algorithms to higher dimensions (for tensor product) or to sparse matrix multiplication (which incurs heterogeneous workloads) are very challenging problems for which very few results exist in the literature.

REFERENCES

[1] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[2] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical Task-based Programming with StarSs," *International Journal of High Performance Computing Applications (IJHPCA)*, vol. 23, no. 3, pp. 284–299, 2009.

[3] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. Dongarra, "PaRSEC: A Programming Paradigm Exploiting Heterogeneity for Enhancing Scalability," *Computing in Science and Engineering*, vol. 15, no. 6, pp. 36–45, Nov. 2013.

[4] A. Kalinov and A. Lastovetsky, "Heterogeneous Distribution of Computations Solving Linear Algebra Problems on Networks of Heterogeneous Computers," *Journal of Parallel and Distributed Computing*, vol. 61, no. 4, pp. 520–535, 2001.

[5] O. Beaumont, V. Boudet, F. Rastello, Y. Robert *et al.*, "Partitioning a Square into Rectangles: NP-completeness and Approximation Algorithms," *Algorithmica*, vol. 34, no. 3, pp. 217–239, 2002.

[6] O. Beaumont, L. Eyraud-Dubois, and T. Lambert, "A New Approximation Algorithm for Matrix Partitioning in Presence of Strongly Heterogeneous Processors," in *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 474–483.

[7] V. Strassen, "Gaussian Elimination is Not Optimal," *Numerische mathematik*, vol. 13, no. 4, pp. 354–356, 1969.

[8] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz, "Communication-optimal Parallel Algorithm for Strassen's Matrix Multiplication," in *Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 2012, pp. 193–204.

[9] D. Coppersmith and S. Winograd, "Matrix Multiplication via Arithmetic Progressions," *Journal of symbolic Computation*, vol. 9, no. 3, pp. 251–280, 1990.

[10] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, "ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers: Design Issues and Performance," in *Computer Physics Communications*. Elsevier, 1996, vol. 97, no. 1, pp. 1 – 15.

[11] E. Solomonik and J. Demmel, "Communication-optimal Parallel 2.5 D Matrix Multiplication and LU factorization Algorithms," in *International European Conference on Parallel and Distributed Computing (Euro-Par)*. Springer, 2011, pp. 90–109.

[12] B. Becker and A. Lastovetsky, "Towards Data Partitioning for Parallel Computing on Three Interconnected Clusters," in *Parallel and Distributed Computing, 2007. ISPDC'07. Sixth International Symposium on*. IEEE, 2007, pp. 39–39.

[13] A. DeFlumere, A. Lastovetsky, and B. Becker, "Optimal Data Partitioning Shape for Matrix Multiplication on Three Fully Connected Heterogeneous Processors," in *International European Conference on Parallel and Distributed Computing (Euro-Par) Workshop*. Springer, 2014, pp. 201–214.

[14] H. Nagamochi and Y. Abe, "An Approximation Algorithm for Dissecting a Rectangle into Rectangles with Specified Areas," *Discrete Applied Mathematics*, vol. 155, no. 4, pp. 523 – 537, 2007.

[15] O. Beaumont, L. Eyraud-Dubois, and T. Lambert, "Cuboid Partitioning for Parallel Matrix Multiplication on Heterogeneous Platforms," in *International European Conference on Parallel and Distributed Computing (Euro-Par)*. Springer, 2016, pp. 171–182.

[16] O. Beaumont, L. Eyraud-Dubois, A. Guermouche, and T. Lambert, "Comparison of Static and Dynamic Resource Allocation Strategies for Matrix Multiplication," in *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2015*. IEEE, 2015, pp. 170–177.

[17] A. YarKhan, J. Kurzak, and J. Dongarra, *QUARK Users' Guide: QUeueing And Runtime for Kernels*, UTK ICL, 2011.

[18] T. Cojean, A. Guermouche, A. Hugo, R. Namyst, and P.-A. Wacrenier, "Resource Aggregation for Task-Based Cholesky Factorization on Top of Heterogeneous Machines," in *International European Conference on Parallel and Distributed Computing (Euro-Par) Workshop*. Springer, 2016.

[19] L. Eyraud-Dubois and T. Lambert, "Matrix Matrix Multiplication using Static Algorithms on Multicores and GPUs." [Online]. Available: https://gitlab.inria.fr/ordo-bdx/nrrp-with-starpu

[20] "Plateforme Fédérative pour la Recherche en Informatique et Mathématiques," 2009. [Online]. Available: https://www.plafrim.fr/fr/accueil/

[21] T. Lambert, "On the Effect of Replication of Input Files on the Efficiency and the Robustness of a Set of Computations," Ph.D. dissertation, Université de Bordeaux, Talence, 2017.

[22] "Chameleon a dense linear algebra software for heterogeneous architectures," 2014. [Online]. Available: https://project.inria.fr/chameleon