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* and Thomas Lambert†
*Inria and University of Bordeaux, Talence, France
†University of Manchester, Manchester, United Kingdom
lionel.eyraud-dubois.fr, thomas.lambert@inria.fr

***Abstract*—We consider how to allocate data 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.**

I. 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 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 on a 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 improvements) of execution times.

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

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

II. 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 has 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 explicitely specified [3]. At runtime, the scheduler takes the scheduling and allocation decisions based on the set of ready tasks (tasks whose all data and control dependences have been solved), on the availability of the resources (estimated using expecting processing and communication times), and on the actual location of input data.

III. Models, Algorithms and Adaptation

A. Models

1) 2D-Partitioning: The standard matrix multiplication $C = A \times B$ can be seen as a set of N^3 tasks (for square matrices)

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

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.* submatrices of the original A, B and C. Our goal here is to avoid as much as possible these replications, under the assumption that we aim for an optimal makespan (that is theoretically achieved simply by giving to each processor a number of $T_{i,j,k}$s proportional to its relative computation speed).

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, limiting the problem to only two dimensions (the $A_{i,k}$'s and the $B_{k,j}$'s) to simplify it. In addition, note that each $C_{i,j}$ is allocated to a single processor, avoiding the problem of concurrent writing. This case can be modelled by the square partitioning problem PERI-SUM. We define the following notations. Let Z be a zone included into a unary 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 $\pi_2(Z) = |\Pi_2(Z)|$). Then we define the half-perimeter of Z as $p(Z) = \pi_1(Z) + \pi_2(Z)$, see Figure 1.

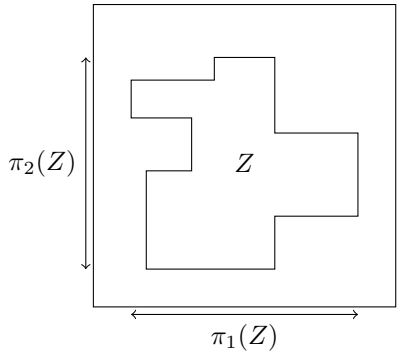
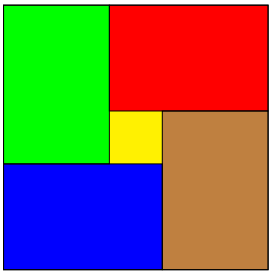


Fig. 1. Illustration of the definition of $\pi_1(Z)$ and $\pi_2(Z)$.

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.

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].

In PERI-SUM the square to be partitioned represents the result matrix C (or more precisely one step of its computation)



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

Fig. 2. Illustration of PERI-SUM.

with its different elements, the $C_{i,j}$s. The set $\{s_1, \ldots, s_m\}$ is the relative speed of the processors ($\frac{w_i}{\sum w_j}$) used to perform the parallel matrix multiplication: 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 the sum of the projections as defined above.

2) 3D-Partitioning: An alternative way to model this situation is to consider the whole set 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, in practice, read-write data and not read-only data. The problem becomes then a cube partitioning problem MSCubeP, 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. Then 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 $\sum Hs(Z_k)$.

Beaumont et al. introduce this problem and propose a proof of NP-completeness [15].

B. Algorithms

In this paper we use two algorithms to schedule parallel matrix multiplication, these algorithms being the algorithms with the best proven approximation ratio for respectively 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). Basically NRRP is a divide and conquer algorithm where, at each step, the relative speeds are split into two (sometimes more) subsets. Meanwhile 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 for each subset. Not that terminal zones are not always shaped as rectangles.

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.

C. Adaptation

In the case of NRRP, the solution is a set of zones. Zones are defined as the union of rectangles, themselves defined by their coordinates. In practice all zones produced by NRRP are composed of at most two rectangles. Meanwhile, in the case of 3D-NRRP, the solution is a set of polyhedra, where a polyhedron is defined as the union of cuboids (at most three for 3D-NRRP).

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 to be 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 allows to keep the projections on the different axes 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 transform the input speeds $\{s_1, \ldots, s_m\}$ into number of tasks to process for each processor, $\{n_1, \ldots, n_m\}$, 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 $\text{Round}(x) = \lceil x \rceil$),

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

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

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

- First, each rectangle is reduced to its *inner rectangle*, *i.e.* the larger 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 memory node and the n_is are updated ($n_i \leftarrow n_i - s(R_{in,i})$).

- Second, for each unassigned element t of the matrix (*i.e.* element that are not in an inner rectangle), one after the other, we check the assignment of the (up to) 8 neighbouring elements (we consider diagonals and there are fewer neighbours on the border). Then we look for a memory node with $n_i > 0$ that has been assigned at least one of these neighbours. If at least one exists, element t is assigned to the one with smallest n_i. Otherwise the task is attributed to the memory node with the smallest 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 ;
```

IV. Implementation

A. 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.

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 is basically the function that is executed when the task is processed. Since StarPU targets heterogeneous environments, in particular platforms with CPUs and GPUs, a codelet is in fact composed of several functions, 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 can also have a *performance model*. Indeed, for each execution of a given codelet on a processing node, StarPU keeps in memory timing data in order to produce this performance model. With such information, StarPU can evaluate, when needed, the performance of this codelet on a given processor, notably to produce an estimation of the computation time.

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 V, 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.

3) Scheduler: As stated above, StarPU provides some pre-implemented schedulers, in particular DMDA that, with the help of performance models, evaluates the computation and communication time 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 in 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, with the following constraints. A StarPU scheduler consists of several functions, the two most important are:

- `push_task`: Called when a submitted task is ready (*i.e.* all the tasks that precede it 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.

B. 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).

1) Work-Stealing: Work-Stealing strategies aim to fix possible load imbalance encountered during the execution. Basically, when a worker is idle with no remaining attributed tasks on its memory node, work-stealing strategies determine a task to be stolen (implying possible data transfers) from another memory node.

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.

2) Purely Dynamic Strategies: The strategies presented here do not rely on an initial static allocation, and are used for comparison purposes. All of them rely on a global list hich 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 smallest cost one.
- 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.

C. Pre-fetching and Reduction

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 behaviour, the communications would not be overlapped with computations, what would create an important difference with our assumptions and would also degrade the overall makespan. 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 we decide to forbid the stealing of already prefetched tasks in order to avoid additional communications.

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 StarPU default 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 V, all 20 CPUs are on a 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 on a same task. 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 handles 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 handles, $C_{i,j}$ and one $C_{i,j}^{aux}$. The reduction tasks are then executed after the GEMM tasks, ensuring the coherency. However, these additional tasks also imply additional computation time and data transfers whose costs have to be evaluated.

V. Experimental Results

In this section we present the first 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].

A. 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,
- 128Go of RAM,
- 4 Nvidia GK110BGL [Tesla K40m] (rev a1).

StarPU allows to plug several BLAS library. 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 CPUs. We use the 1.3 version of StarPU.

As experiment set, we use matrices of double with size 7680, 15360, 23040 and 30720 (the size of the matrix 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 should be to split and distribute (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 efficiency for the GEMM operation on GPUs.

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 dynamic strategies. 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.

B. Makespan

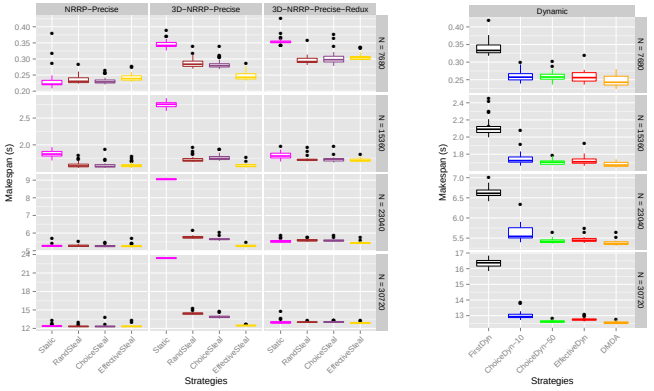
Let us first consider the makespan as metric, mainly to eliminate strategies that produce unsatisfying makespan. Note that in this section (and in Section V-C) the y scale on every figure does not start at 0.

On Figure 3(a) we propose a comparison between all three partitioning algorithms, NRRP-Precise, 3D-NRRP-Precise and 3D-NRRP-Precise-Redux. Let us first note that, as expected, Static strategies with 3D-NRRP as partitioning algorithm and without reduction operation have a catastrophic makespan that gets worse when N (and the number of tasks) increases, reaching up to twice the best makespan achieved by the other allocations. The ratio 2 can be explained by the fact that two memory nodes share all their $C_{i,j}$s and thus work one after the other. For the two other strategies, NRRP-Precise and 3D-NRRP-Precise-Redux, the performance of Static strategy is much closer to the other strategies, a bit worse in some cases ($N = 15360$ for example). Note that for NRRP-Precise, Static is the best strategy for $N = 7680$, probably because of the shorter execution time of its `pop_task` function. Anyway, the good results of Static on Precise partitions point the good quality of the performance model for the GEMM operations.

If we now look at work-stealing strategies, let us first note that EffectiveSteal is effective enough to make 3D-NRRP-Precise partitioning able to challenge the other partitioning algorithms whereas the two others, ChoiceSteal and RandSteal, improve the makespan but not enough. In other cases, there is very little difference between all three strategies. EffectiveSteal is a bit less effective in the case $N = 7680$, a bit more in the case $N = 15360$ and there is no significant winner in the other cases. To finish, note that in general NRRP-Precise partitioning appears to be the most effective for all the values of N (around $0.225s$ against $0.25s$ and $0.3s$ for $N = 7680$, around $1.6s$ against $1.6s$ and $1.75s$ for $N = 15360$, around $5.25s$ against $5.25s$ and $5.5s$ for $N = 23040$ and around $12s$ against $12s$ and $13s$ for $N = 30720$). The better parallelism (in comparison to 3D-NRRP-Precise with no EffectiveSteal) and the absence of reduction tasks (in comparison to 3D-NRRP-Precise-Redux) generate a real advantage.

On Figure 3(b), we present the results for dynamic strategies. This mainly emphasizes the unsatisfying results of First-Dyn that is removed on the following figure, Figure 4. In this one, we propose a comparison between strategies using NRRP as partitioning algorithm (the most effective according to Figure 3(a)) and compare both versions, NRRP-Rounded and NRRP-Precise, with purely dynamic strategies (without FirstDyn).

If we focus on purely dynamic strategies, we can mainly do three observations. First the criterion X in ChoiceDyn-X has a major influence (ChoiceDyn-10 is always worse than ChoiceDyn-50). Secondly EffectiveDyn is often less effective than ChoiceDyn-50. A possible interpretation is that stolen tasks in ChoiceDyn-50 are more or less of the same quality than the ones in EffectiveDyn but are found faster. Last, DMDA appears to be more effective than purely



(a) Makespan of strategies using NRRP-Precise, 3D-NRRP-Precise and 3D-NRRP-Precise-Redux

(b) Makespan of purely dynamic strategies

Fig. 3. Makespan of strategies using NRRP-Precise, 3D-NRRP-Precise and 3D-NRRP-Precise-Redux or purely dynamic strategies, on `sirocco`.

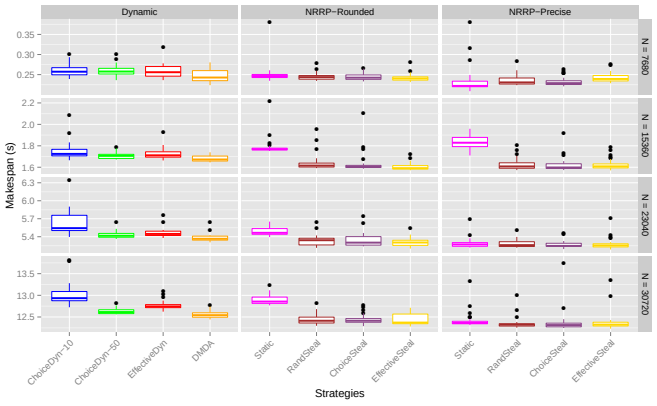


Fig. 4. Makespan of strategies using NRRP-Precise, NRRP-Rounded and purely dynamic strategies, on `sirocco`.

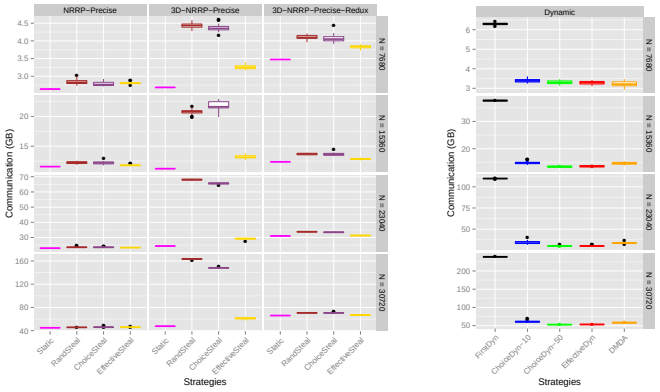
communication-based dynamic strategies (ChoiceDyn-50 is close but slightly less effective).

The comparison between Rounded and Precise is at the advantage of the second. In the case of Static strategies, Rounded, having a worse load-balancing, has a larger makespan except for the case $N = 15360$ (for reasons that are still to explain). For the other strategies, results are close, but Precise-based ones is slightly more stable.

To finish, static-based strategies, augmented with work-stealing, are slightly more makespan-effective than purely dynamic ones. One possible explanation is that DMDA, despite its data-aware property, induces too many data transfers, especially of $C_{i,j}$ tiles, and is thus less effective than the combination of NRRP and work-stealing strategies.

C. Communication

Let us now focus on the amount of communications induced by each strategy. We have two goals: first, determine if there is a correlation between makespan results and communication results; second determine if, with comparable makespan, some strategies are more communication-effective than others.



(a) Communication amount of strategies using NRRP-Precise, 3D-NRRP-Precise and 3D-NRRP-Precise-Redux

(b) Communication amount of purely dynamic strategies

Fig. 5. Communication amount of strategies using NRRP-Precise, 3D-NRRP-Precise and 3D-NRRP-Precise-Redux or purely dynamic strategies, on `sirocco`.

First, we consider in Figure 5(a) the comparison between all three partitioning strategies, NRRP-Precise, 3D-NRRP-Precise and NRRP-Precise-Redux. If we put Static with 3D-NRRP-Precise aside, NRRP-based strategies are significantly more effective (Static with 3D-NRRP-Precise can be slightly more communication-effective than NRRP-Precise, in the case $N = 15360$ for example, but at the cost of a bad makespan as shown in the previous section). If we compare (coarsely) the results of EffectiveSteal (that seems to be a good trade-off between makespan effectiveness and communication effectiveness) for NRRP-Precise and the second most effective, 3D-NRRP-Precise, we note that the second one can transfer around 33% more data, see Table I.

For the other notable observations, the poor makespan results of ChoiceSteal and RandSteal, in comparison to the ones of EffectiveSteal, with 3D-NRRP-Precise algorithm likely come from a bad choice of stolen tasks (at least one of the memory nodes is only functioning with work-stealing) that are here catastrophic (more than 2.5 as much data transfer as EffectiveSteal in the case $N = 30720$). Second, the reduction tasks have a clear impact on the NRRP-Precise-Redux results. A good way to evaluate the cost of reduction is to compare the cost of Static with 3D-NRRP-Precise and 3D-NRRP-Precise-Redux, see Table II. With high costs (around 44% more communications for $N = 30720$), the reduction tasks have a huge impact that make the reduction option not satisfying for both metrics.

Note that FirstDyn is also disappointing for both metrics, see Figure 5(b). FirstDyn can transfer more than 4 times as much data than the other strategies. For the rest of the dynamic strategies (more precise view on Figure 6), ChoiceDyn-50 and EffectiveDyn produce more or less the same amount of communications, what seems to confirm that the difference between them from the makespan point of view may be

	NRRP-Precise	3D-NRRP-Precise	
$N = 7680$	$\simeq 2.75\ GB$	$\simeq 3.25\ GB$	$\simeq +18\%$
$N = 15360$	$\simeq 17\ GB$	$\simeq 18\ GB$	$\simeq +6\%$
$N = 23040$	$\simeq 24\ GB$	$\simeq 30\ GB$	$\simeq +25\%$
$N = 30720$	$\simeq 45\ GB$	$\simeq 60\ GB$	$\simeq +33\%$

TABLE I
COMPARISON OF THE TWO PARTITIONING ALGORITHM NRRP-PRECISE AND 3D-NRRP-PRECISE WITH USE OF EFFECTIVESTEAL, ON SIROCCO.

	3D-NRRP-Precise	3D-NRRP-Precise-Redux	Reduction cost
$N = 7680$	$\simeq 2.75\ GB$	$\simeq 3.5\ GB$	$\simeq 0.75\ GB$ $(+27\%)$
$N = 15360$	$\simeq 16\ GB$	$\simeq 17.5\ GB$	$\simeq 1.5\ GB$ $(+9\%)$
$N = 23040$	$\simeq 25\ GB$	$\simeq 30\ GB$	$\simeq 5\ GB$ $(+20\%)$
$N = 30720$	$\simeq 45\ GB$	$\simeq 65\ GB$	$\simeq 20\ GB$ $(+44\%)$

TABLE II
COMPARISON OF THE TWO PARTITIONING ALGORITHMS NRRP-PRECISE AND 3D-NRRP-PRECISE-REDUX WITH USE OF STATIC, ON SIROCCO. THE DIFFERENCE REPRESENTS THE REDUCTION COST.

a consequence of ChoiceDyn being a bit faster to compute. Both these strategies are more communication-effective than DMDA, whose results are comparable to the ones of ChoiceDyn-10.

The comparison between Rounded and Precise is more even. Precise is expected to achieve a better load-balancing, inducing less work-stealing (that can be seen on Figure 6 with few differences between EffectiveSteal, ChoiceSteal, RandSteal and Static when NRRP-Precise is used), but with an increase of the communication volume, notably in the initial allocation. On Figure 6 we can observe that this increase can be significant. For example, in the case $N = 30720$, EffectiveSteal produces around $41GB$ of transferred data (with reasonable makespan, unlike Static) when NRRP-Rounded is used, against $45GB$ for Static with NRRP-Precise $(+9.75\%)$. However in some other cases, $N = 7980$ or $N = 23040$, NRRP-Precise performs better than NRRP-Rounded. Although the difference in these cases is less important than in the case $N = 30720$, this shows there is no clear winner between Rounded and Precise approaches (for makespan and communication metrics), the difference is probably highly correlated to the initial allocation, depending on the error coming from the rounding.

Finally, we compare dynamic and work-stealing strategies, always on Figure 6. We already showed that, on the makespan metric, the results of both NRRP-Rounded and NRRP-Precise are very satisfying, with a slight improvement over the native DMDA. In the case of communication metric we can highlight that the EffectiveSteal strategies provide a significant improvement with sometimes up to a 25% of reduction in amount of transferred data. Therefore the use of static allocation appears to bring a strong gain in communication with no loss of makespan-efficiency. The slightly better results for the makespan also comes from this important reduction of communications.

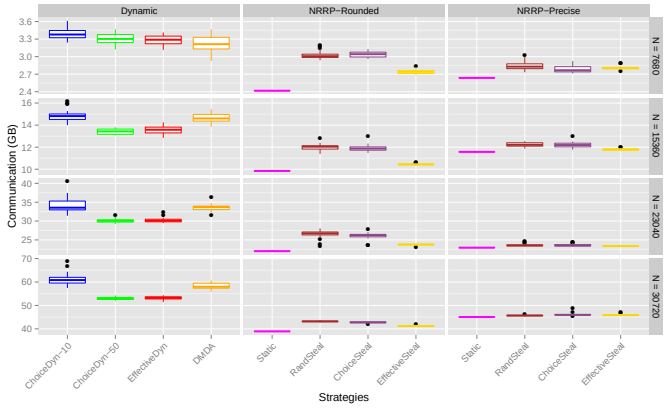


Fig. 6. Communication amount of strategies using NRRP-Precise, NRRP-Rounded and or purely dynamic strategies, on sirocco.

	DMDA	NRRP-Rounded	NRRP-Precise
$N = 7680$	$\simeq 3.2\ GB$	$\simeq 2.7\ GB$ (-15.6%)	$\simeq 2.8\ GB$ (-12.5%)
$N = 15360$	$\simeq 14.5\ GB$	$\simeq 10.5\ GB$ (-27.6%)	$\simeq 12\ GB$ (-17.2%)
$N = 23040$	$\simeq 33\ GB$	$\simeq 24\ GB$ (-27.2%)	$\simeq 23.5\ GB$ (-28.8%)
$N = 30720$	$\simeq 58\ GB$	$\simeq 41\ GB$ (-29.3%)	$\simeq 45\ GB$ (-24.1%)

TABLE III
COMPARISON OF THE TWO PARTITIONING ALGORITHMS NRRP-ROUNDED AND NRRP-PRECISE WITH USE OF EFFECTIVESTEAL AND DMDA, ON SIROCCO (BETWEEN PARENTHESIS THE GAIN FROM DMDA IS GIVEN).

D. Results on an other platform

In addition to the sirocco node, we also test our implementation on a mirage node of PlaFRIM2. A mirage node is composed of 3 GPUs and 12 CPUS (3 of which are dedicated to GPU management). More precisely the platform contains:

- 2 Hexa-core Westmere Intel® Xeon® X5650 @ 2,67 GHz
- 36 Go of RAM,
- 3 Nvidia GF100GL [Tesla M2070] (rev a3).

In general, the results are very similar on the mirage node than the ones on the sirocco node. For example, the gain on makespan and communication with a NRRP-based strategy compared to DMDA are very close. Similarly the use of 3D-NRRP without reduction leads to unacceptable makespan or a too large amount of communication because of necessary work-stealing. Hovewer some differences appear, as shown on Figures 7 and 8. Note that for space reasons, we do not show results for all strategies: we only consider the dynamic ones (except FirstDyn), Static and EffectiveSteal (with NRRP-Precise and 3D-NRRP-Precise as partitioning algorithm).

The first thing to note is that, on this platform, communication-oriented dynamic strategies (ChoiceDyn and EffectiveDyn) are faster than DMDA and even, for $N = 30720$, slightly faster than NRRP-Precise-based strategies. Besides, we can also note than 3D-NRRP-Precise-based strategies are this time the fastest ones, possibly because of a smaller number of reduction tasks (there are fewer memory nodes on mirage).

From the communication point of view, we can also see that 3D-NRRP-Precise-based strategies have a cost close to the dynamic ones (except DMDA that is more expensive).

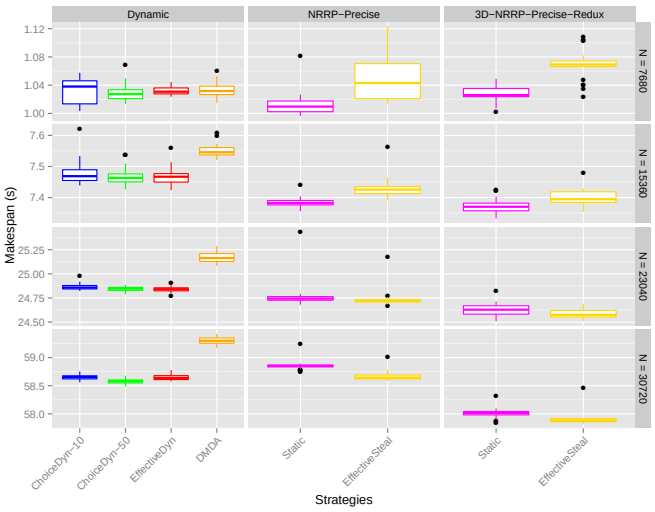


Fig. 7. Makespan of dynamic strategies and strategies using NRRP-Precise, 3D-NRRP-Precise-Reduxs, on `mirage`.

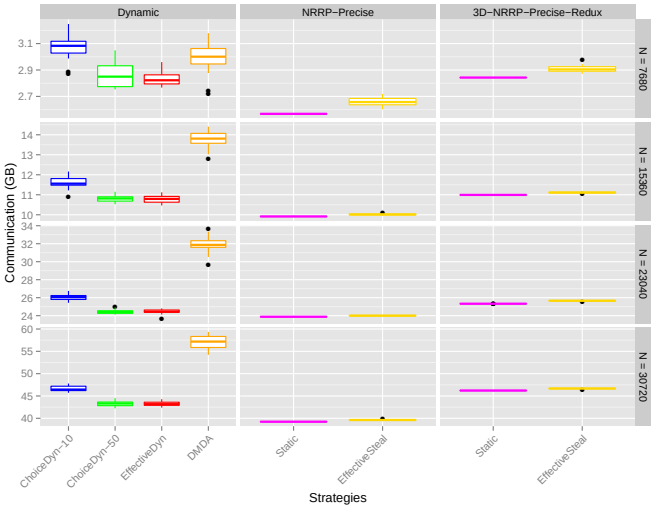


Fig. 8. Communication amount of dynamic strategies and strategies using NRRP-Precise, 3D-NRRP-Precise-Redux, on `mirage`.

However NRRP-Precise-based strategies are still largely the most effective strategies to avoid data transfers.

From this additional set of experiments, we can add see that the performance improvement from static strategies can be generalized to different architectures. Furthermore, this also shows that communication-oriented dynamic strategies can be competitive in certain environments, but lack the guarantees of static approaches. In addition, the good results of 3D-partitioning justify looking at this approach in deeper details, for example by refining the cost model to take reduction tasks into account.

VI. 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. For this implementation, we use dynamic and static strategies, mixing 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.

The adaptation of 3D-based approaches is promising, but requires a finer communication model, taking into account the increased cost of sharing parts of the result matrix. In addition, we propose fully dynamic and communication-based strategies whose performance is in some cases close to the static approaches. 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. Such techniques allow to keep the good theoretical efficiency of static algorithms, correcting its lack of reliability for practical concerns, and this opens many perspectives for the future design of hybrid strategies.

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/