Revisiting Matrix Product on Master-Worker Platforms

Jean-François Pineau Yves Robert Frédéric Vivien Zhiao Shi
Jack Dongarra

November 2006

LIP Research Report RR-2006-39

Abstract

This paper is aimed at designing efficient parallel matrix-product algorithms for heterogeneous master-worker platforms. While matrix-product is well-understood for *homogeneous 2D-arrays of processors* (e.g., Cannon algorithm and ScaLAPACK outer product algorithm), there are three key hypotheses that render our work original and innovative:
- *Centralized data.* We assume that all matrix files originate from, and must be returned to, the master. The master distributes both data and computations to the workers (while in ScaLAPACK, input and output matrices are initially distributed among participating resources). Typically, our approach is useful in the context of speeding up MATLAB or SCILAB clients running on a server (which acts as the master and initial repository of files).
- *Heterogeneous star-shaped platforms.* We target fully heterogeneous platforms, where computational resources have different computing powers. Also, the workers are connected to the master by links of different capacities. This framework is realistic when deploying the application from the server, which is responsible for enrolling authorized resources.
- *Limited memory.* Because we investigate the parallelization of large problems, we cannot assume that full matrix panels can be stored in the worker memories and re-used for subsequent updates (as in ScaLAPACK). The amount of memory available in each worker is expressed as a given number m_i of buffers, where a buffer can store a square block of matrix elements. The size q of these square blocks is chosen so as to harness the power of Level 3 BLAS routines: $q = 80$ or 100 on most platforms.

We have devised efficient algorithms for resource selection (deciding which workers to enroll) and communication ordering (both for input and result messages), and we report a set of numerical experiments on various platforms at École Normale Supérieure de Lyon and the University of Tennessee. However, we point out that in this first version of the report, experiments are limited to homogeneous platforms.

1 Introduction

Matrix product is a key computational kernel in many scientific applications, and it has been extensively studied on parallel architectures. Two well-known parallel versions are Cannon's algorithm [14] and the ScaLAPACK outer product algorithm [13]. Typically, parallel implementations work well on 2D processor grids, because the input matrices are sliced horizontally and vertically into square blocks that are mapped one-to-one onto the physical resources; several communications can take place in parallel, both horizontally and vertically. Even better, most of these communications can be overlapped with (independent) computations. All these characteristics render the matrix product kernel quite amenable to an efficient parallel implementation on 2D processor grids.

However, current architectures typically take the form of heterogeneous clusters, which are composed of heterogeneous computing resources, interconnected by a *sparse* network: there are no direct links between any pair of processors. Instead, messages from one processor to another are routed via several links, likely to have different capacities. Worse, congestion will occur when two messages, involving two different sender/receiver pairs, collide because the same physical link happens to belong to both their routing paths. Therefore, an accurate estimation of the communication cost requires a precise knowledge of the underlying target platform. In addition, it becomes necessary to include the cost of both the initial distribution of the matrices to the processors and of collecting back the results. These input/output operations have always been neglected in the analysis of the conventional algorithms. This is because only $O(n^2)$ coefficients need to be distributed in the beginning, and gathered at the end, as opposed to the $O(n^3)$ computations to be performed (where n is the problem size). The assumption that these communications can be ignored could have made sense on dedicated processor grids like, say, the Intel Paragon, but it is no longer reasonable on heterogeneous platforms.

There are two possible approaches to tackle the parallelization of matrix product on heterogeneous clusters when aiming at reusing the 2D processor grid strategy. The first (drastic) approach is to *ignore* communications. The objective is then to load-balance computations as evenly as possible on a heterogeneous 2D processor grid. This corresponds to arranging the n available resources as a (virtual) 2D grid of size $p \times q$ (where $p.q \leq n$) so that each processor receives a share of the work, i.e., a rectangle, whose area is proportional to its relative computing speed. There are many processor arrangements to consider, and determining the optimal one is a highly combinatorial problem, which has been proven NP-complete in [5]. In fact, because of the geometric constraints imposed by the 2D processor grid, a perfect load-balancing can only be achieved in some very particular cases.

The second approach is to relax the geometric constraints imposed by a 2D processor grid. The idea is then to search for a 2D partitioning of the input matrices into rectangles that will be mapped one-to-one onto the processors. Because the 2D partitioning now is irregular (it is no longer constrained to a 2D grid), some processors may well have more than four neighbors. The advantage of this approach is that a perfect load-balancing is always possible; for instance partitioning the matrices into horizontal slices whose vertical dimension is proportional to the computing speed of the processors always leads to a perfectly balanced distribution of the computations. The objective is then to minimize the total cost of the communications. However, it is very hard to accurately predict this cost. Indeed, the processor arrangement is virtual, not physical: as explained above, the underlying interconnection network is not expected to be a complete graph, and communications between neighbor processors in the arrangement are likely to be realized via several physical links constituting the communication path. The actual repartition of the physical links across all paths is hard to predict, but contention is almost certain to occur. This is why a natural, although pessimistic assumption, to estimate the communication cost, is to assume that all communications in the execution of the algorithm will be implemented sequentially. With this hypothesis, minimizing the total communication cost amounts to minimizing the total communication volume. Unfortunately, this problem has been shown NP-complete as well [6]. Note that even under the optimistic assumption that all communications at a given step of the algorithm can take place in parallel, the problem remains NP-complete [7].

In this paper, we do not try to adapt the 2D processor grid strategy to heterogeneous clusters. Instead, we adopt a realistic application scenario, where input files are read from a fixed repository (disk on a data server). Computations will be delegated to available resources in the target architecture, and results will be returned to the repository. This calls for a master-worker paradigm, or more precisely for a computational scheme where the master (the processor holding the input data) assigns computations to other resources, the workers. In this centralized approach, all matrix files originate from, and must be returned to, the master. The master distributes both data and computations to the workers (while in ScaLAPACK, input and output matrices are supposed to be equally distributed among participating resources beforehand). Typically, our approach is useful in the context of speeding up MATLAB or SCILAB clients running on a server (which acts as the master and initial repository of files).

We target fully heterogeneous master-worker platforms, where computational resources have different computing powers. Also, the workers are connected to the master by links of different capacities. This framework is realistic when deploying the application from the server, which is responsible for enrolling authorized resources.

Finally, because we investigate the parallelization of large problems, we cannot assume that full matrix panels can be stored in worker memories and re-used for subsequent updates (as in ScaLAPACK). The amount of memory available in each worker is expressed as a given number m_i of buffers, where a buffer can store a square block of matrix elements. The size q of these square blocks is chosen so as to harness the power of Level 3 BLAS routines: $q = 80$ or 100 on most platforms.

To summarize, the target platform is composed of several workers with different computing powers, different bandwidth links to/from the master, and different, limited, memory capacities. The first problem is *resource selection*. Which workers should be enrolled in the execution? All of them, or maybe only the faster computing ones, or else only the faster-communicating ones? Once participating resources have been selected, there remain several scheduling decisions to take: how to minimize the number of communications? which order workers should receive input data and return results? what amount of communications can be overlapped with (independent) computations? The goal of this paper is to design efficient algorithms for resource selection and communication ordering. In addition, we report numerical experiments on various heterogeneous platforms at the École Normale Supérieure de Lyon and at the University of Tennessee.

The rest of the paper is organized as follows. In Section 2, we state the scheduling problem precisely, and we introduce some notations. In Section 3, we start with a theoretical study of the simplest version of the problem, without memory limitation, which is intended to show the intrinsic difficulty of the scheduling problem. Next, in Section 4, we proceed with the analysis of the total communication volume that is needed in the presence of memory constraints, and we improve a well-known bound by Toledo [38, 27]. We deal with homogeneous platforms in Section 5, and we propose a scheduling algorithm that includes resource selection. Section 6 is the counterpart for heterogeneous platforms, but the algorithms are much more complicated. In Section 7, we briefly discuss how to extend previous approaches to LU factorization. We report several MPI experiments in Section 8. Section 9 is devoted to an overview of related work. Finally, we state some concluding remarks in Section 10.

2 Framework

In this section we formally state our hypotheses on the application (Section 2.1) and on the target platform (Section 2.2).

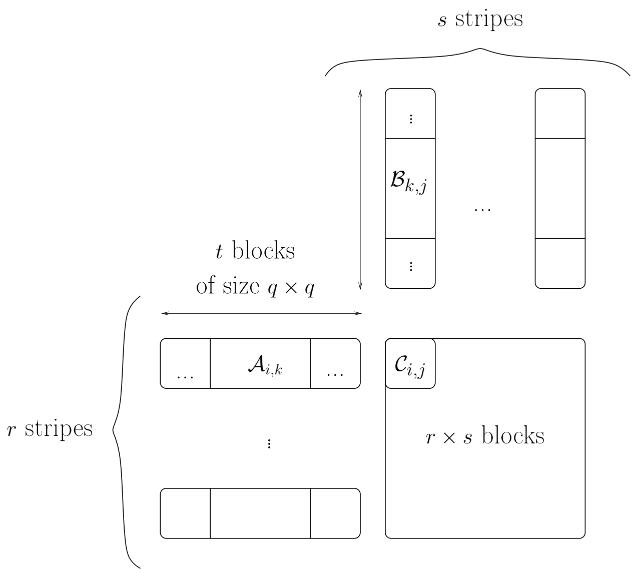


Figure 1: Partition of the three matrices $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$.

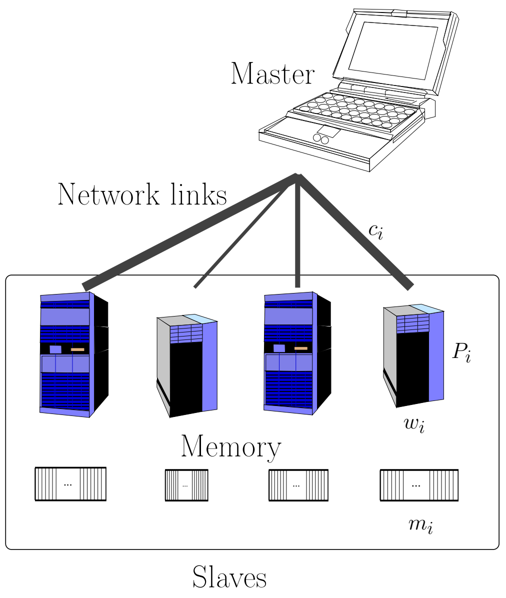


Figure 2: A fully heterogeneous master-worker platform.

2.1 Application

We deal with the computational kernel $\mathcal{C} \leftarrow \mathcal{C} + \mathcal{A} \times \mathcal{B}$. We partition the three matrices $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ as illustrated in Figure 1. More precisely:

- We use a block-oriented approach. The atomic elements that we manipulate are not matrix coefficients but instead square *blocks* of size $q \times q$ (hence with q^2 coefficients). This is to harness the power of Level 3 BLAS routines [12]. Typically, $q = 80$ or 100 when using ATLAS-generated routines [40].
- The input matrix $\mathcal{A}$ is of size $n_{\mathcal{A}} \times n_{\mathcal{AB}}$:
  - we split $\mathcal{A}$ into r horizontal stripes $\mathcal{A}_i$, $1 \leq i \leq r$, where $r = n_{\mathcal{A}}/q$;
  - we split each stripe $\mathcal{A}_i$ into t square $q \times q$ blocks $\mathcal{A}_{i,k}$, $1 \leq k \leq t$, where $t = n_{\mathcal{AB}}/q$.
- The input matrix $\mathcal{B}$ is of size $n_{\mathcal{AB}} \times n_{\mathcal{B}}$:
  - we split $\mathcal{B}$ into s vertical stripes $\mathcal{B}_j$, $1 \leq j \leq s$, where $s = n_{\mathcal{B}}/q$;
  - we split stripe $\mathcal{B}_j$ into t square $q \times q$ blocks $\mathcal{B}_{k,j}$, $1 \leq k \leq t$.

- We compute $\mathcal{C} = \mathcal{C} + \mathcal{A} \times \mathcal{B}$. Matrix $\mathcal{C}$ is accessed (both for input and output) by square $q \times q$ blocks $\mathcal{C}_{i,j}$, $1 \leq i \leq r$, $1 \leq j \leq s$. There are $r \times s$ such blocks.

We point out that with such a decomposition all stripes and blocks have same size. This will greatly simplify the analysis of communication costs.

2.2 Platform

We target a *star network* $\mathcal{S} = \{P_0, P_1, P_2, \ldots, P_p\}$, composed of a master P_0 and of p workers P_i, $1 \leq i \leq p$ (see Figure 2). Because we manipulate large data blocks, we adopt a linear cost model, both for computations and communications (i.e., we neglect start-up overheads). We have the following notations:

- It takes $X.w_i$ time-units to execute a task of size X on P_i;
- It takes $X.c_i$ time units for the master P_0 to send a message of size X to P_i or to receive a message of size X from P_i.

Our star platforms are thus fully heterogeneous, both in terms of computations and of communications. A fully homogeneous star platform would be a star platform with identical workers and identical communication links: $w_i = w$ and $c_i = c$ for each worker P_i, $1 \leq i \leq p$. Without loss of generality, we assume that the master has no processing capability (otherwise, add a fictitious extra worker paying no communication cost to simulate computation at the master).

Next, we need to define the communication model. We adopt the *one-port* model [10, 11], which is defined as follows:

- the master can only send data to, and receive data from, a single worker at a given time-step,
- a given worker cannot start execution before it has terminated the reception of the message from the master; similarly, it cannot start sending the results back to the master before finishing the computation.

In fact, this *one-port* model naturally comes in two flavors with return messages, depending upon whether we allow the master to simultaneously send and receive messages or not. If we do allow for simultaneous sends and receives, we have the *two-port* model. Here we concentrate on the true *one-port* model, where the master cannot be enrolled in more than one communication at any time-step.

The *one-port* model is *realistic*. Bhat, Raghavendra, and Prasanna [10, 11] advocate its use because "current hardware and software do not easily enable multiple messages to be transmitted simultaneously." Even if non-blocking multi-threaded communication libraries allow for initiating multiple send and receive operations, they claim that all these operations "are eventually serialized by the single hardware port to the network." Experimental evidence of this fact has recently been reported by Saif and Parashar [35], who report that asynchronous MPI sends get serialized as soon as message sizes exceed a hundred kilobytes. Their result hold for two popular MPI implementations, MPICH on Linux clusters and IBM MPI on the SP2. Note that all the MPI experiments in Section 8 obey the one-port model.

The one-port model fully accounts for the heterogeneity of the platform, as each link has a different bandwidth. It generalizes a simpler model studied by Banikazemi, Moorthy, and Panda [1], Liu [32], and Khuller and Kim [30]. In this simpler model, the communication time only depends on the sender, not on the receiver. In other words, the communication speed from a processor to all its neighbors is the same. This would restrict the study to bus platforms instead of general star platforms.

Our final assumption is related to memory capacity; we assume that a worker P_i can only store m_i blocks (either from $\mathcal{A}$, $\mathcal{B}$, or $\mathcal{C}$). For large problems, this memory limitation will considerably impact the design of the algorithms, as data re-use will be greatly dependent on the amount of available buffers.

3 Combinatorial complexity of a simple version of the problem

This section is almost a digression; it is devoted to the study of the simplest variant of the problem. It is intended to show the intrinsic combinatorial difficulty of the problem. We make the following simplifications:

- We target a fully homogeneous platform (identical workers and communication links).
- We consider only rank-one block updates; in other words, and with previous notations, we focus on the case where $t = 1$.
- Results need not be returned to the master.
- Workers have *no* memory limitation; they receive each stripe only once and can re-use them for other computations.

There are five parameters in the problem; three platform parameters (c, w, and the number of workers p) and two application parameters (r and s). The scheduling problem amounts to deciding which files should be sent to which workers and in which order. A given file may well be sent several times, to further distribute computations. For instance, a simple strategy is to partition $\mathcal{A}$ and to duplicate $\mathcal{B}$, i.e., send each block $\mathcal{A}_i$ only once and each block $\mathcal{B}_j$ p times; all workers would then be able to work fully in parallel.

$$\begin{tikzcd}
& \mathcal{B}_1 \arrow[d] \arrow[dd, bend right] \arrow[ddd, bend right] & \mathcal{B}_2 \arrow[d] \arrow[dd, bend right] \arrow[ddd, bend right] \\
\mathcal{A}_1 \arrow[r] \arrow[rr, bend left] & \mathcal{C}_{11} & \mathcal{C}_{12} \\
\mathcal{A}_2 \arrow[r] \arrow[rr, bend left] & \mathcal{C}_{21} & \mathcal{C}_{22} \\
\mathcal{A}_3 \arrow[r] \arrow[rr, bend left] & \mathcal{C}_{31} & \mathcal{C}_{32}
\end{tikzcd}$$

Figure 3: Dependence graph of the problem (with $r = 3$ and $s = 2$).

The dependence graph of the problem is depicted in Figure 3. It suggests a natural strategy for enabling workers to start computing as soon as possible. Indeed, the master should alternate sending $\mathcal{A}$-blocks and $\mathcal{B}$-blocks. Of course it must be decided how many workers to enroll and in

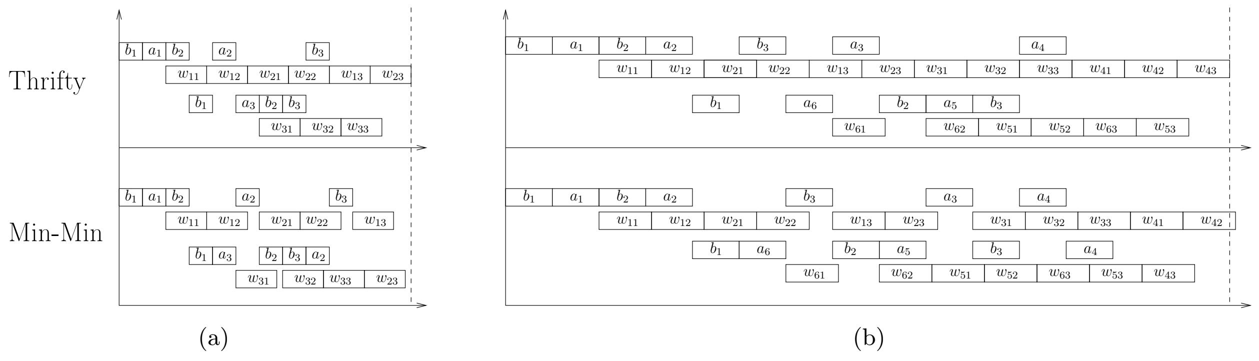


Figure 4: Neither Thrifty nor Min-min is optimal: (a) with $p = 2$, $c = 4$, $w = 7$, and $r = s = 3$, Min-min wins; (b) with $p = 2$, $c = 8$, $w = 9$, $r = 6$, and $s = 3$, Thrifty wins.

which order to send the blocks to the enrolled workers. But with a single worker, we can show that the *alternating greedy* algorithm is optimal:

Proposition 1. *With a single worker, the* alternating greedy *algorithm is optimal.*

Proof. In this algorithm, the master sends blocks as soon as possible, alternating a block of type $\mathcal{A}$ and a block of type $\mathcal{B}$ (and proceeds with the remaining blocks when one type is exhausted). This strategy maximizes at each step the total number of tasks that can be processed by the worker. To see this, after x communication steps, with y files of type $\mathcal{A}$ sent, and z files of type $\mathcal{B}$ sent, where $y + z = x$, the worker can process at most $y \times z$ tasks. The greedy algorithm enforces $y = \lceil \frac{x}{2} \rceil$ and $z = \lfloor \frac{x}{2} \rfloor$ (as long as $\max(x, y) \leq \min(r, s)$, and then sends the remaining files), hence its optimality. $\square$

Unfortunately, for more than one worker, we did not succeed in determining an optimal algorithm. There are (at least) two greedy algorithms that can be devised for p workers:

Thrifty: This algorithm "spares" resources as it aims at keeping each enrolled worker fully active. It works as follows:

- Send enough blocks to the first worker so that it is never idle,
- Send blocks to a second worker during spare communication slots, and
- Enroll a new worker (and send blocks to it) only if this does not delay previously enrolled workers.

Min-min: This algorithm is based on the well-known min-min heuristic [33]. At each step, all tasks are considered. For each of them, we compute their possible starting date on each worker, given the files that have already been sent to this worker and all decisions taken previously; we select the best worker, hence the first *min* in the heuristic. We take the minimum of starting dates over all tasks, hence the second *min*.

It turns out that neither greedy algorithm is optimal. See Figure 4(a) for an example where Min-min is better than Thrifty, and Figure 4(b) for an example of the opposite situation.

We now go back to our original model.

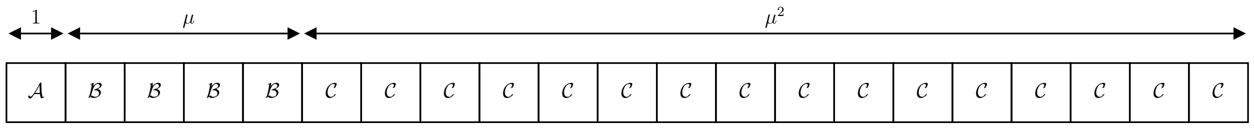
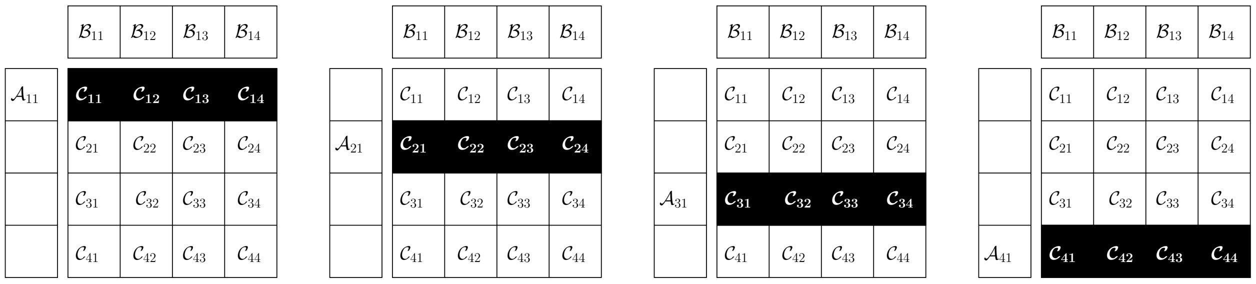


Figure 5: Memory usage for the *maximum re-use* algorithm when $m = 21$: $\mu = 4$; 1 block is used for $\mathcal{A}$, μ for $\mathcal{B}$, and μ^2 for $\mathcal{C}$.



$\mathcal{B}_{11}$ $\mathcal{B}_{12}$ $\mathcal{B}_{13}$ $\mathcal{B}_{14}$
$\mathcal{A}_{11}$
$\mathcal{A}_{21}$
$\mathcal{A}_{31}$
$\mathcal{A}_{41}$
$\mathcal{C}_{11}$ $\mathcal{C}_{12}$ $\mathcal{C}_{13}$ $\mathcal{C}_{14}$
$\mathcal{C}_{21}$ $\mathcal{C}_{22}$ $\mathcal{C}_{23}$ $\mathcal{C}_{24}$
$\mathcal{C}_{31}$ $\mathcal{C}_{32}$ $\mathcal{C}_{33}$ $\mathcal{C}_{34}$
$\mathcal{C}_{41}$ $\mathcal{C}_{42}$ $\mathcal{C}_{43}$ $\mathcal{C}_{44}$

Figure 6: Four steps of the *maximum re-use* algorithm, with $m = 21$ and $\mu = 4$. The elements of $\mathcal{C}$ updated are displayed on white on black.

4 Minimization of the communication volume

In this section, we derive a lower bound on the total number of communications (sent from, or received by, the master) that are needed to execute any matrix multiplication algorithm. We point out that, since we are not interested in optimizing the execution time (a difficult problem, according to Section 3) but only in minimizing the total communication volume, we can simulate any parallel algorithm on a single worker. Therefore, we only need to consider the one-worker case.

We deal with the original, and realistic, formulation of the problem as follows:

- The master sends blocks $\mathcal{A}_{ik}$, $\mathcal{B}_{kj}$, and $\mathcal{C}_{ij}$,
- The master retrieves final values of blocks $\mathcal{C}_{ij}$, and
- We enforce limited memory on the worker; only m buffers are available, which means that at most m blocks of $\mathcal{A}$, $\mathcal{B}$, and/or $\mathcal{C}$ can simultaneously be stored on the worker.

First, we describe an algorithm that aims at re-using $\mathcal{C}$ blocks as much as possible after they have been loaded. Next, we assess the performance of this algorithm. Finally, we improve a lower bound previously established by Toledo [38, 27].

4.1 The *maximum re-use* algorithm

Below we introduce and analyze the performance of the *maximum re-use* algorithm, whose memory management is illustrated in Figure 5. Four consecutive execution steps are shown in Figure 6. Assume that there are m available buffers. First we find μ as the largest integer such that $1+\mu+\mu^2 \leq m$. The idea is to use one buffer to store $\mathcal{A}$ blocks, μ buffers to store $\mathcal{B}$ blocks, and μ^2 buffers to store $\mathcal{C}$ blocks. In the outer loop of the algorithm, a $\mu \times \mu$ square of $\mathcal{C}$ blocks is loaded. Once these μ^2 blocks have been loaded, they are repeatedly updated in the inner loop of the algorithm until their final value is computed. Then the blocks are returned to the master, and μ^2 new $\mathcal{C}$ blocks are sent by the master and stored by the worker. As illustrated in Figure 5, we need μ buffers to store a row of $\mathcal{B}$ blocks, but only one buffer for $\mathcal{A}$ blocks: $\mathcal{A}$ blocks are sent in sequence, each of them is used in combination with a row of μ $\mathcal{B}$ blocks to update the corresponding row of $\mathcal{C}$ blocks. This leads to the following sketch of the algorithm:

Outer loop: while there remain $\mathcal{C}$ blocks to be computed

- Store μ^2 blocks of $\mathcal{C}$ in worker's memory:
  send a $\mu \times \mu$ square $\{\mathcal{C}_{i,j} \ / \ i_0 \leq i < i_0 + \mu, \ j_0 \leq j < j_0 + \mu\}$
- **Inner loop**: For each k from 1 to t:
  1. Send a row of μ elements $\{\mathcal{B}_{k,j} \ / \ j_0 \leq j < j_0 + \mu\}$;
  2. Sequentially send μ elements of column $\{\mathcal{A}_{i,k} \ / \ i_0 \leq i < i_0 + \mu\}$. For each $A_{i,k}$, update μ elements of $\mathcal{C}$
- Return results to master.

4.2 Performance and lower bound

The performance of one iteration of the outer loop of the *maximum re-use* algorithm can readily be determined:

- We need $2\mu^2$ communications to send and retrieve $\mathcal{C}$ blocks.
- For each value of t:
  - we need μ elements of $\mathcal{A}$ and μ elements of $\mathcal{B}$;
  - we update μ^2 blocks.

In terms of block operations, the communication-to-computation ratio achieved by the algorithm is thus

$$\text{CCR} = \frac{2\mu^2 + 2\mu t}{\mu^2 t} = \frac{2}{t} + \frac{2}{\mu}.$$

For large problems, i.e., large values of t, we see that CCR is asymptotically close to the value $\text{CCR}_\infty = \frac{2}{\sqrt{m}}$. We point out that, in terms of data elements, the communication-to-computation ratio is divided by a factor q. Indeed, a block consists of q^2 coefficients but an update requires q^3 floating-point operations.

How can we assess the performance of the *maximum re-use* algorithm? How good is the value of CCR? To see this, we refine an analysis due to Toledo [38]. The idea is to estimate the number of computations made thanks to m consecutive communication steps (again, the unit is a matrix block here). We need some notations:

- We let α_{old}, β_{old}, and γ_{old} be the number of buffers dedicated to $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ at the beginning of the m communication steps;
- We let α_{recv}, β_{recv}, and γ_{recv} be the number of $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ blocks sent by the master during the m communication steps;
- Finally, we let γ_{send} be the number of $\mathcal{C}$ blocks returned to the master during these m steps.

Obviously, the following equations must hold true:

$$\begin{cases} \alpha_{old} + \beta_{old} + \gamma_{old} \leq m \\ \alpha_{recv} + \beta_{recv} + \gamma_{recv} + \gamma_{send} = m \end{cases}$$

The following lemma is given in [38]: consider any algorithm that uses the standard way of multiplying matrices (this excludes Strassen's or Winograd's algorithm [19], for instance). If N_A elements of $\mathcal{A}$, N_B elements of $\mathcal{B}$ and N_C elements of $\mathcal{C}$ are accessed, then no more than K computations can be done, where

$$K = \min\left\{(N_A + N_B)\sqrt{N_C}, (N_A + N_C)\sqrt{N_B}, (N_B + N_C)\sqrt{N_A}\right\}.$$

To use this result here, we see that no more than $\alpha_{old} + \alpha_{recv}$ blocks of $\mathcal{A}$ are accessed, hence $N_A = (\alpha_{old} + \alpha_{recv})q^2$. Similarly, $N_B = (\beta_{old} + \beta_{recv})q^2$ and $N_C = (\gamma_{old} + \gamma_{recv})q^2$ (the $\mathcal{C}$ blocks returned are already counted). We simplify notations by writing:

$$\begin{cases} \alpha_{old} + \alpha_{recv} = \alpha m \\ \beta_{old} + \beta_{recv} = \beta m \\ \gamma_{old} + \gamma_{recv} = \gamma m \end{cases}$$

Then we obtain

$$K = \min\left\{(\alpha + \beta)\sqrt{\gamma}, (\beta + \gamma)\sqrt{\alpha}, (\gamma + \alpha)\sqrt{\beta}\right\} \times m\sqrt{m}q^3.$$

Writing $K = km\sqrt{m}q^3$, we obtain the following system of equations

$$\begin{cases} \text{Maximize } k \text{ s.t.} \\ k \leq (\alpha + \beta)\sqrt{\gamma} \\ k \leq (\beta + \gamma)\sqrt{\alpha} \\ k \leq (\gamma + \alpha)\sqrt{\beta} \\ \alpha + \beta + \gamma \leq 2 \end{cases}$$

whose solution is easily found to be

$$\alpha = \beta = \gamma = \frac{2}{3}, \text{ and } k = \sqrt{\frac{32}{27}}.$$

This gives a lower bound for the communication-to-computation ratio (in terms of blocks) of any algorithm:

$$\text{CCR}_{\text{opt}} = \frac{m}{km\sqrt{m}} = \sqrt{\frac{27}{32m}}.$$

In fact, it is possible to refine this bound. Instead of using the lemma given in [38], we use Loomis-Whitney inequality [27]: if N_A elements of $\mathcal{A}$, N_B elements of $\mathcal{B}$, and N_C elements of $\mathcal{C}$ are accessed, then no more than K computations can be done, where

$$K = \sqrt{N_A N_B N_C}.$$

Here

$$K = \sqrt{\alpha + \beta + \gamma} \times m\sqrt{m}q^3$$

We obtain

$$\alpha = \beta = \gamma = \frac{2}{3}, \text{ and } k = \sqrt{\frac{8}{27}},$$

so that the lower bound for the communication-to-computation ratio becomes:

$$\text{CCR}_{\text{opt}} = \sqrt{\frac{27}{8m}}.$$

The *maximum re-use* algorithm does not achieve the lower bound:

$$\text{CCR}_{\infty} = \frac{2}{\sqrt{m}} = \sqrt{\frac{32}{8m}}$$

but it is quite close!

Finally, we point out that the bound CCR_{opt} improves upon the best-known value $\sqrt{\frac{1}{8m}}$ derived in [27]. Also, the ratio CCR_{∞} achieved by the *maximum re-use* algorithm is lower by a factor $\sqrt{3}$ than the ratio achieved by the *blocked matrix-multiply* algorithm of [38].

5 Algorithms for homogeneous platforms

In this section, we adapt the *maximum re-use* algorithm to fully homogeneous platforms. In this framework, contrary to the simplest version, we have a limitation of the memory capacity. So we must first decide which part of the memory will be used to stock which part of the original matrices, in order to maximize the total number of computations per time unit. Cannon's algorithm [14] and the ScaLAPACK outer product algorithm [13] both distribute square blocks of $\mathcal{C}$ to the processors. Intuitively, squares are better than elongated rectangles because their perimeter (which is proportional to the communication volume) is smaller for the same area. We use the same approach here, but we have not been able to assess any optimal result.

Principle of the algorithm

We load into the memory of each worker μ $q \times q$ blocks of $\mathcal{A}$ and μ $q \times q$ blocks of $\mathcal{B}$ to compute μ^2 $q \times q$ blocks of $\mathcal{C}$. In addition, we need 2μ extra buffers, split into μ buffers for $\mathcal{A}$ and μ for $\mathcal{B}$, in order to overlap computation and communication steps. In fact, μ buffers for $\mathcal{A}$ and μ for $\mathcal{B}$ would suffice for each update, but we need to prepare for the next update while computing. Overall, the number of $\mathcal{C}$ blocks that we can simultaneously load into memory is the largest integer μ such that

$$\mu^2 + 4\mu \leq m.$$

We have to determine the number of participating workers $\mathfrak{P}$. For that purpose, we proceed as follows. On the communication side, we know that in a round (computing a $\mathcal{C}$ block entirely), the master exchanges with each worker $2\mu^2$ blocks of $\mathcal{C}$ (μ^2 sent and μ^2 received), and sends μt blocks of $\mathcal{A}$ and μt blocks of $\mathcal{B}$. Also during this round, on the computation side, each worker computes $\mu^2 t$ block updates.

If we enroll too many processors, the communication capacity of the master will be exceeded. There is a limit on the number of blocks sent per time unit, hence on the maximal processor number $\mathfrak{P}$, which we compute as follows: $\mathfrak{P}$ is the smallest integer such that

$$2\mu tc \times \mathfrak{P} \geq \mu^2 tw.$$

Algorithm 1: Homogeneous version, master program.

```
$\mu \leftarrow \lfloor \sqrt{4+m} - 2 \rfloor$;
$\mathfrak{P} \leftarrow \min\left\{p, \lceil \frac{\mu w}{2c} \rceil\right\}$;
Split the matrix into squares $\mathbf{C}_{i',j'}$ of $\mu^2$ blocks (of size $q \times q$):
$\mathbf{C}_{i',j'} = \{\mathcal{C}_{i,j} \setminus (i'-1)\mu + 1 \leq i \leq i'\mu, (j'-1)\mu + 1 \leq j \leq j'\mu\}$;
for $j'' \leftarrow 0$ to $\frac{s}{\mathfrak{P}\mu}$ by Step $\mathfrak{P}$ do
    for $i' \leftarrow 1$ to $\frac{r}{\mu}$ do
        for $id_{worker} \leftarrow 1$ to $\mathfrak{P}$ do
            $j' \leftarrow j'' + id_{worker}$;
            Send block $\mathbf{C}_{i',j'}$ to worker $id_{worker}$;
        for $k \leftarrow 1$ to $t$ do
            for $id_{worker} \leftarrow 1$ to $\mathfrak{P}$ do
                $j' \leftarrow j'' + id_{worker}$;
                for $j \leftarrow (j'-1)\mu + 1$ to $j'\mu$ do
                    Send $\mathcal{B}_{k,j}$;
                for $i \leftarrow (i'-1)\mu + 1$ to $i'\mu$ do
                    Send $\mathcal{A}_{i,k}$;
        for $id_{worker} \leftarrow 1$ to $\mathfrak{P}$ do
            $j' \leftarrow j'' + id_{worker}$;
            Receive $\mathbf{C}_{i',j'}$ from worker $id_{worker}$;
```

Indeed, this is the smallest value to saturate the communication capacity of the master required to sustain the corresponding computations. We derive that

$$\mathfrak{P} = \left\lceil \frac{\mu^2 tw}{2\mu tc} \right\rceil = \left\lceil \frac{\mu w}{2c} \right\rceil .$$

In the context of matrix multiplication, we have $c = q^2\tau_c$ and $w = q^3\tau_a$, hence $\mathfrak{P} = \left\lceil \frac{\mu q}{2} \frac{\tau_a}{\tau_c} \right\rceil$. Moreover, we need to enforce that $\mathfrak{P} \leq p$, hence we finally obtain the formula

$$\mathfrak{P} = \min\left\{p, \left\lceil \frac{\mu q}{2} \frac{\tau_a}{\tau_c} \right\rceil\right\} .$$

For the sake of simplicity, we suppose that r is divisible by μ, and that s is divisible by $\mathfrak{P}\mu$. We allocate μ block columns (i.e., $q\mu$ consecutive columns of the original matrix) of $\mathcal{C}$ to each processor. The algorithm is decomposed into two parts. Algorithm 1 outlines the program of the master, while Algorithm 2 is the program of each worker.

Impact of the start-up overhead

If we follow the execution of the homogeneous algorithm, we may wonder whether we can really neglect the input/output of $\mathcal{C}$ blocks. Contrary to the greedy algorithms for the simplest instance described in Section 3, we sequentialize here the sending, computing, and receiving of the $\mathcal{C}$ blocks,

Algorithm 2: Homogeneous version, worker program.

```
for all blocks do
    Receive $\mathbf{C}_{i',j'}$ from master;
    for $k \leftarrow 1$ to $t$ do
        for $j \leftarrow (j'-1)\mu+1$ to $j'\mu$ do Receive $\mathcal{B}_{k,j}$;
        for $i \leftarrow (i'-1)\mu+1$ to $i'\mu$ do
            Receive $\mathcal{A}_{i,k}$;
            for $j \leftarrow (j'-1)\mu+1$ to $j'\mu$ do
                $\mathcal{C}_{i,j} \leftarrow \mathcal{C}_{i,j} + \mathcal{A}_{i,k}.\mathcal{B}_{k,j}$;
    Return $\mathbf{C}_{i',j'}$ to master;
```

so that each worker loses $2c$ time-units per block, i.e., per tw time-units. As there are $\mathfrak{P} \leq \frac{\mu w}{2c} + 1$ workers, the total loss would be of $2c\mathfrak{P}$ time-units every tw time-units, which is less than $\frac{\mu}{t} + \frac{2c}{tw}$. For example, with $c = 2$, $w = 4.5$, $\mu = 4$ and $t = 100$, we enroll $\mathfrak{P} = 5$ workers, and the total lost is at most 4%, which is small enough to be neglected. Note that it would be technically possible to design an algorithm where the sending of the next block is overlapped with the last computations of the current block, but the whole procedure gets much more complicated.

Dealing with "small" matrices or platforms

We have shown that our algorithm should use $\mathfrak{P} = \min\left\{p, \left\lceil \frac{\mu q}{2} \frac{\tau_a}{\tau_c} \right\rceil\right\}$ processors, each of them holding μ^2 blocks of matrix $\mathcal{C}$. For this solution to be feasible, $\mathcal{C}$ must be large enough. In other words, this solution can be implemented if and only if $r \times s \geq \min\left\{p, \left\lceil \frac{\mu q}{2} \frac{\tau_a}{\tau_c} \right\rceil\right\} \mu^2$. If $\mathcal{C}$ is not large enough, we will only use $\mathfrak{Q} < \mathfrak{P}$ processors, each of them holding ν^2 blocks of $\mathcal{C}$, such that:

$$\left\{ \begin{array}{l} \mathfrak{Q}\nu^2 \leq r \times s \\ \nu^2 w \leq 2\nu\mathfrak{Q}c \end{array} \right. \quad \Leftrightarrow \quad \left\{ \begin{array}{l} \mathfrak{Q}\nu^2 \leq r \times s \\ \frac{\nu w}{2c} \leq \mathfrak{Q} \end{array} \right. ,$$

following the same line of reasoning as previously. We obviously want ν to be the largest possible in order for the communications to be most beneficial. For a given value of ν we want $\mathfrak{Q}$ to be the smallest to spare resources. Therefore, the best solution is given by the largest value of ν such that:

$$\left\lceil \frac{\nu w}{2c} \right\rceil \nu^2 \leq r \times s,$$

and then $\mathfrak{Q} = \left\lceil \frac{\nu w}{2c} \right\rceil$.

If the platform does not contain the desired number of processors, i.e., if $\mathfrak{P} > p$ in the case of a "large" matrix $\mathcal{C}$ or if $\mathfrak{Q} > p$ otherwise, then we enroll all the p processors and we give them ν^2 blocks of $\mathcal{C}$ with $\nu = \min\left\{\frac{r \times s}{p}, \frac{2c}{w} p\right\}$, following the same line of reasoning as previously.

6 Algorithms for heterogeneous platforms

In this section, all processors are heterogeneous, in term of memory size as well as computation or communication time. As in the previous section, m_i is the number of $q \times q$ blocks that fit in the memory of worker P_i, and we need to load into the memory of P_i $2\mu_i$ blocks of $\mathcal{A}$, $2\mu_i$ blocks of $\mathcal{B}$, and μ_i^2 blocks of $\mathcal{C}$. This number of blocks loaded into the memory changes from worker to worker, because it depends upon their memory capacities. We first compute all the different values of μ_i so that

$$\mu_i^2 + 4\mu_i \leq m_i.$$

To adapt our *maximum re-use* algorithm to heterogeneous platforms, we first design a greedy algorithm for resource selection (Section 6.1), and we discuss its limitations. We introduce our final algorithm for heterogeneous platforms in Section 6.2.

6.1 Bandwidth-centric resource selection

Each worker P_i has parameters c_i, w_i, and μ_i, and each participating P_i needs to receive $\delta_i = 2\mu_i t c_i$ blocks to perform $\phi_i = t\mu_i^2 w_i$ computations. Once again, we neglect I/O for $\mathcal{C}$ blocks. Consider the steady-state of a schedule. During one time-unit, P_i receives a certain amount y_i of blocks, both of $\mathcal{A}$ and $\mathcal{B}$, and computes x_i $\mathcal{C}$ blocks. We express the constraints, in terms of communication —the master has limited bandwidth— and of computation —a worker cannot perform more work than it receives. The objective is to maximize the amount of work performed per time-unit. Altogether, we gather the following linear program:

$$\left\{\begin{array}{r} \textsc{Maximize } \sum_i x_i \\ \textsc{subject to} \\ \displaystyle\sum_i y_i c_i \leq 1 \\ \forall i,\quad x_i w_i \leq 1 \\ \forall i,\quad \dfrac{x_i}{\mu_i^2} \leq \dfrac{y_i}{2\mu_i} \end{array}\right.$$

Obviously, the best solution for y_i is $y_i = \frac{2x_i}{\mu_i}$, so the problem can be reduced to :

$$\left\{\begin{array}{l} \textsc{Maximize } \sum_i x_i \\ \textsc{subject to} \\ \forall i,\quad x_i \leq \frac{1}{w_i} \\ \quad \sum_i \frac{2c_i}{\mu_i} x_i \leq 1 \end{array}\right.$$

The optimal solution for this system is a bandwidth-centric strategy [8, 3]; we sort workers by non-decreasing values of $\frac{2c_i}{\mu_i}$ and we enroll them as long as $\sum \frac{2c_i}{\mu_i w_i} \leq 1$. In this way, we can achieve the throughput $\rho \approx \sum_{i \text{ enrolled}} \frac{1}{w_i}$.

This solution seems to be close to the optimal. However, the problem is that workers may not have enough memory to execute it! Consider the example described by Table 1.

Using the bandwidth-centric strategy, every 160 seconds:

- P_1 receives 80 blocks (20 $\mu_1 \times \mu_1$ chunks) in 80 seconds, and computes 80 blocks in 160 seconds;
- P_2 receives 4 blocks (1 $\mu_2 \times \mu_2$ chunk) in 80 seconds, and computes 4 blocks in 160 seconds.

	P_1	P_2
c_i	1	20
w_i	2	40
μ_i	2	2
$\frac{2c_i}{\mu_i w_i}$	$\frac{1}{2}$	$\frac{1}{2}$

Table 1: Platform for which the bandwidth centric solution is not feasible.

	P_1	P_2	P_3
c_i	2	3	5
w_i	2	3	1
μ_i	6	18	10
μ_i^2	36	324	100
$2\mu_i c_i$	24	108	100

Table 2: Platform used to demonstrate the processor selection algorithms.

But P_1 computes two quickly, and it needs buffers to store as many as 20 blocks to stay busy while one block is sent to P_2:

$$\begin{array}{cccccc} \textit{Communications} & 11111111111111111111 & 20 & 11111111111111111111 & 20 & 111111111\ldots \\ \textit{Processor} & P_1 & P_2 & P_1 & P_2 & P_1\ldots \end{array}$$

Therefore, the bandwidth-centric solution cannot always be realized in practice, and we turn to another algorithm described below. To avoid the previous buffer problems, resource selection will be performed through a step-by-step simulation. However, we point out that the steady-state solution can be seen as an upper bound of the performance that can be achieved.

6.2 Incremental resource selection

The different memory capacities of the workers imply that we assign them chunks of different sizes. This requirement complicates the global partitioning of the $\mathcal{C}$ matrix among the workers. To take this into account and simplify the implementation, we decide to assign only full matrix column blocks in the algorithm. This is done in a two-phase approach.

In the first phase we pre-compute the allocation of blocks to processors, using a processor selection algorithm we will describe later. We start as if we had a huge matrix of size $\infty \times \sum_{i=1}^{n} \mu_i$. Each time a processor P_i is chosen by the processor selection algorithm it is assigned a square chunk of μ_i^2 $\mathcal{C}$ blocks. As soon as some processor P_i has enough blocks to fill up μ_i block columns of the initial matrix, we decide that P_i will indeed execute these columns during the parallel execution. Therefore we maintain a panel of $\sum_{i=1}^{p} \mu_i$ block columns and fill them out by assigning blocks to processors. We stop this phase as soon as all the $r \times s$ blocks of the initial matrix have been allocated columnwise by this process. Note that worker P_i will be assigned a block column after it has been selected $\lceil \frac{r}{\mu_i} \rceil$ times by the algorithm.

In the second phase we perform the actual execution. Messages will be sent to workers according to the previous selection process. The first time a processor P_i is selected, it receives a square chunk of μ_i^2 $\mathcal{C}$ blocks, which initializes its repeated pattern of operation: the following t times, P_i receives μ_i $\mathcal{A}$ and μ_i $\mathcal{B}$ blocks, which requires $2\mu_i c_i$ time-units.

There remains to decide which processor to select at each step. We have no closed-form formula for the allocation of blocks to processors. Instead, we use an incremental algorithm to compute which worker the next blocks will be assigned to. We have two variants of the incremental algorithm, a *global* one that aims at optimizing the overall communication-to-computation ratio, and a *local* one that selects the best processor for the next stage. Both variants are described below.

Algorithm 3: Global selection algorithm.

```
Data:
completion-time: the completion time of the last communication
$\mathsf{ready}_i$: the completion time of the work assigned to processor $P_i$
$\mathsf{nb\text{-}block}_i$: the number of $A$ and $B$ blocks sent to processor $P_i$
total-work: the total work assigned so far (in terms of block updates)
nb-column: the number of fully processed $\mathcal{C}$ block columns
INITIALIZATION
completion-time $\leftarrow 0$;
total-work $\leftarrow 0$;
for $i \leftarrow 1$ to $p$ do
    $\mathsf{ready}_i \leftarrow 0$;
    $\mathsf{nb\text{-}block}_i \leftarrow 0$;
SIMULATION
repeat
    next $\leftarrow$ worker that realizes $\max_{i=1}^{p} \frac{\mathsf{total\text{-}work}+\mu_i^2}{\max(\mathsf{completion\text{-}time}+2\mu_i c_i, \mathsf{ready}_i)}$;
    total-work $\leftarrow$ total-work $+ \mu_{\mathsf{next}}^2$;
    completion-time $\leftarrow \max(\mathsf{completion\text{-}time} + 2\mu_{\mathsf{next}} c_{\mathsf{next}}, \mathsf{ready}_{\mathsf{next}})$;
    $\mathsf{ready}_{\mathsf{next}} \leftarrow \mathsf{completion\text{-}time} + \mu_{\mathsf{next}}^2 w_{\mathsf{next}}$;
    $\mathsf{nb\text{-}block}_{\mathsf{next}} \leftarrow \mathsf{nb\text{-}block}_{\mathsf{next}} + 2\mu_{\mathsf{next}}$;
    nb-column $\leftarrow \sum_{i=1}^{p} \left\lfloor \frac{\mathsf{nb\text{-}block}_i}{2\mu_i t \lceil \frac{r}{\mu_i} \rceil} \right\rfloor \mu_i$;
until nb-column $\geq s$ ;
```

6.2.1 Global selection algorithm

The intuitive idea for this algorithm is to select the processor that maximizes the ratio of the total work achieved so far (in terms of block updates) over the completion time of the last communication. The latter represents the time spent by the master so far, either sending data to workers or staying idle, waiting for the workers to finish their current computations. We have:

$$\mathsf{ratio} \leftarrow \frac{\text{total work achieved}}{\text{completion time of last communication}}$$

Estimating computations is easy: P_i executes μ_i^2 block updates per assignment. Communications are slightly more complicated to deal with; we cannot just use the communication time $2\mu_i c_i$ of P_i for the $\mathcal{A}$ and $\mathcal{B}$ blocks because we need to take its ready time into account. Indeed, if P_i is currently busy executing work, it cannot receive additional data too much in advance because its memory is limited. Algorithm 3 presents this selection process, which we iterate until all blocks of the initial matrix are assigned and computed.

Running the global selection algorithm on an example. Consider the example described in Table 2 with three workers P_1, P_2 and P_3. For the first step, we have $\mathsf{ratio}_i \leftarrow \frac{\mu_i^2}{2\mu_i c_i}$ for all i. We compute $\mathsf{ratio}_1 = 1.5$, $\mathsf{ratio}_2 = 3$, and $\mathsf{ratio}_3 = 1$ and select P_2: $\mathsf{next} \leftarrow 2$. We update variables as

$\mathsf{total\text{-}work} \leftarrow 0 + 324 = 324$, $\mathsf{completion\text{-}time} \leftarrow \max(0 + 108, 0) = 108$, $\mathsf{ready}_2 \leftarrow 108 + 972 = 1080$ and $\mathsf{nb\text{-}block}_2 \leftarrow 36$.

At the second step we compute $\mathsf{ratio}_1 \leftarrow \frac{324+36}{108+24} = 2.71$, $\mathsf{ratio}_2 \leftarrow \frac{324+324}{1080} = 0.6$ and $\mathsf{ratio}_3 \leftarrow \frac{324+100}{108+100} = 2.04$ and we select P_1. We point out that P_2 is busy until time $t = 1080$ because of the first assignment, which we correctly took into account when computing ready_2. For P_1 and P_3 the communication could take place immediately after the first one. There remains to update variables: $\mathsf{total\text{-}work} \leftarrow 324 + 36 = 360$, $\mathsf{completion\text{-}time} \leftarrow \max(108 + 24, 0) = 132$, $\mathsf{ready}_1 \leftarrow 132 + 72 = 204$ and $\mathsf{nb\text{-}block}_1 \leftarrow 12$.

At the third step the algorithm selects P_3. Going forward, we have a cyclic pattern repeating, with 13 consecutive communications, one to P_2 followed by 12 ones alternating between P_1 and P_3, and then some idle time before the next pattern (see Figure 7). The asymptotic value of ratio is 1.17 while the steady-state approach of Section 6.1 would achieve a ratio of 1.39 without memory limitations. Finally, we point out that it is easy to further refine the algorithm to get closer to the performance of the steady-state. For instance, instead of selecting the best processor greedily, we could look two-steps ahead and search for the best pair of workers to select for the next two communications (the only price to pay is an increase in the cost of the selection algorithm). From the example, the two-step ahead strategy achieves a ratio 1.30.

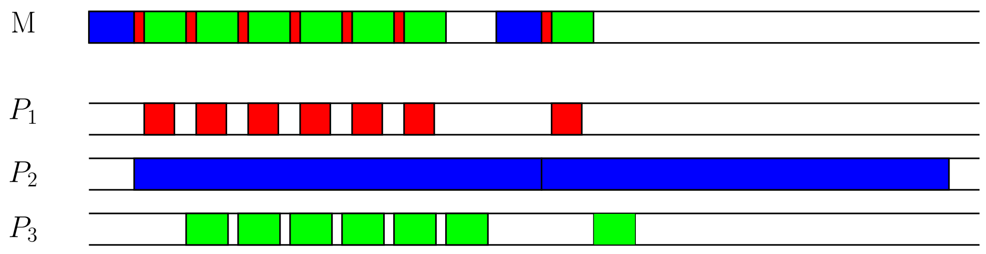


Figure 7: Global selection algorithm on the example of Table 2.

6.2.2 Local selection algorithm

The global selection algorithm picks, as the next processor, the one that maximizes the ratio of the total amount of work assigned over the time needed to send all the required data. Instead, the local selection algorithm chooses, as destination of the i-th communication, the processor that maximizes the ratio of the amount of work assigned by this communication over the time during which the communication link is used to performed this communication (i.e., the elapsed time between the end of $(i-1)$-th communication and the end of the i-th communication). As previously, if processor P_j is the target of the i-th communication, the i-th communication is the sending of μ_j blocks of $\mathcal{A}$ and μ_j blocks of $\mathcal{B}$ to processor P_j, which enables it to perform μ_j^2 updates.

More formally, the local selection algorithm picks the worker P_i that maximizes:

$$\frac{{\mu_i}^2}{\max\{2\mu_i c_i,\ \mathsf{ready}_i - \mathsf{completion\text{-}time}\}}$$

Once again we consider the example described in Table 2. For the first three steps, the global and selection algorithms make the same decision. In fact, they take the same first 13 decisions. However, for the 14-th selection, the global algorithm picks processor P_2 when the local selection selects processor P_1 and then processor P_2 for the 15-th decision, as illustrated in Figure 8. Under both selection processes, the second chunk of work is sent to processor P_2 at the same time but the local algorithm inserts an extra communication. For this example, the local selection algorithm achieves an asymptotic ratio of computation per communication of 1.21. This is better than what is achieved by the global selection algorithm but, obviously, there are examples where the global selection will beat the local one.

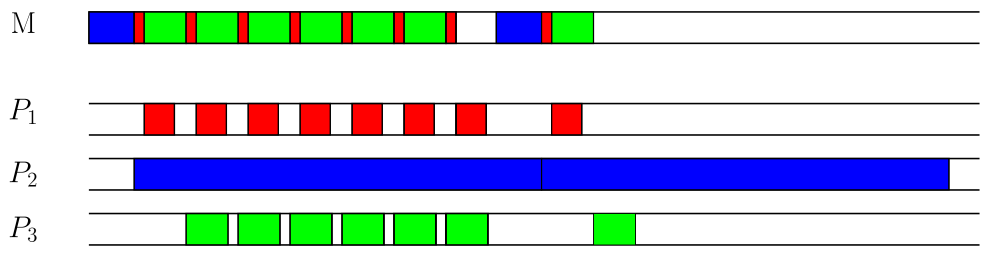


Figure 8: Local selection algorithm on the example of Table 2.

7 Extension to LU factorization

In this section, we show how our techniques can be extended to LU factorization. We first consider (Section 7.1) the case of a single worker, in order to study how we can minimize the communication volume. Then we present algorithms for homogeneous clusters (Section 7.2) and for heterogeneous platforms (Section 7.3).

We consider the right-looking version of the LU factorization as it is more amenable to parallelism. As previously, we use a block-oriented approach. The atomic elements that we manipulate are not matrix coefficients but instead square *blocks* of size $q \times q$ (hence with q^2 coefficients). The size of the matrix is then $r \times r$ blocks. Furthermore, we consider a second level of blocking of size μ. As previously, μ is the largest integer such that $\mu^2 + 4\mu \leq m$. The main kernel is then a rank-μ update $C \leftarrow C + A.B$ of blocks. Hence the similarity between matrix multiplication and LU decomposition.

7.1 Single processor case

The different steps of LU factorization are presented in Figure 9. Step k of the factorization consists of the following:

1. Factor pivot matrix (Figure 9(a)). We compute at each step a pivot matrix of size μ^2 (which thus contains $\mu^2 \times q^2$ coefficients). This factorization has a communication cost of $2\mu^2 c$ (to bring the matrix and send it back after the update) and a computation cost of $\mu^3 w$.

2. Update the μ columns below the pivot matrix (vertical panel) (Figure 9(b)). Each row x of this vertical panel is of size μ and must be replaced by xU^{-1} for a computation cost of $\frac{1}{2}\mu^2 w$.

   The most communication-efficient policy to implement this update is to keep the pivot matrix in place and to move around the rows of the vertical panel. Each row must be brought and sent back after update, for a total communication cost of $2\mu c$.

At the k-th step, this update has then an overall communication cost of $2\mu(r - k\mu)c$ and an overall computation cost of $\frac{1}{2}\mu^2(r - k\mu)w$.

3. Update the μ rows at the right of the pivot matrix (horizontal panel) (Figure 9(c)). Each column y of this horizontal panel is of size μ and must be replaced by $L^{-1}y$ for a computation cost of $\frac{1}{2}\mu^2 w$.

   This case is symmetrical to the previous one. Therefore, we follow the same policy and at the k-th step, this update has an overall communication cost of $2\mu(r - k\mu)c$ and an overall computation cost of $\frac{1}{2}\mu^2(r - k\mu)w$.

4. Update the core matrix (square matrix of the last $(r - k\mu)$ rows and columns) (Figure 9(d)). This is a rank-μ update. Contrary to matrix multiplication, the most communication-efficient policy is to not keep the result matrix in memory, but either a $\mu \times \mu$ square block of the vertical panel or of the horizontal panel (both solutions are symmetrical). Arbitrarily, we then decide to keep in memory a chunk of the horizontal panel. Then to update a row vector x of the core matrix, we need to bring to that vector the corresponding row of the vertical panel, and then to send back the updated value of x. This has a communication cost of $3\mu c$ and a computation cost of μ^2.

   At the k-th step, this update for μ columns of the core matrix has an overall communication cost of $(\mu^2 + 3(r - k\mu)\mu)c$ (counting the communications necessary to initially bring the μ^2 elements of the horizontal panel) and an overall computation cost of $(r - k\mu)\mu^2 w$.

   Therefore, at the k-th step, this update has an overall communication cost of $(\frac{r}{\mu} - k)(\mu^2 + 3(r - k\mu)\mu)c$ and an overall computation cost of $(\frac{r}{\mu} - k)(r - k\mu)\mu^2 w$.

Using the above scheme, the overall communication cost of the LU factorization is

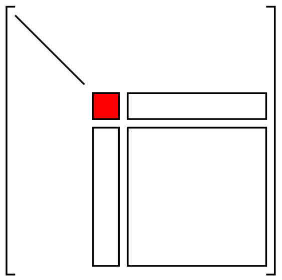
$$\sum_{k=1}^{\frac{r}{\mu}} \left(2\mu^2 + 4\mu(r - k\mu) + \left(\frac{r}{\mu} - k\right)(\mu^2 + 3(r - k\mu)\mu)\right) c = \left(\frac{r^3}{\mu} - r^2 + 2\mu r\right) c,$$

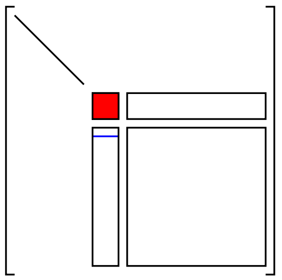
while the overall computation cost is

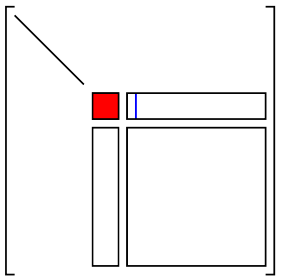
$$\sum_{k=1}^{\frac{r}{\mu}} \left(\mu^3 + \mu^2(r - k\mu) + \left(\frac{r}{\mu} - k\right)(r - k\mu)\mu^2\right) w = \frac{1}{3}\left(r^3 + 2\mu^2 r\right) w.$$

7.2 Algorithm for homogeneous clusters

The most time-consuming part of the factorization is the update of the core matrix (it has an overall cost of $\left(\frac{1}{3}r^3 - \frac{1}{2}\mu r^2 + \frac{1}{6}\mu^2 r\right) w$). Therefore, we want to parallelize this update by allocating blocks of μ columns of the core matrix to different processors. Just as for matrix multiplication, we would like to determine the optimal number of participating workers $\mathfrak{P}$. For that purpose, we proceed as previously. On the communication side, we know that in a round (each worker updating μ columns entirely), the master sends to each worker μ^2 blocks of the horizontal panel, then sends to each worker the $\mu(r - k\mu)$ blocks of the vertical panel, and exchanges with each of them $2\mu(r - k\mu)$ blocks of the core matrix ($\mu(r - k\mu)$ received and later sent back after update). Also during this


(a) The pivot matrix is factored.


(b) Update of vertical panel. A row x is replaced by xU^{-1}.


(c) Update of horizontal panel. A column y is replaced by $L^{-1}y$.

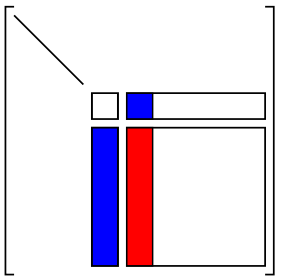

(d) μ columns of the core matrix are updated using the vertical panel and μ columns of the horizontal panel.

Figure 9: Scheme for LU factorization at step k.

round, on the computation side, each worker computes $\mu^2(r - k\mu)$ block updates. If we enroll too many processors, the communication capacity of the master will be exceeded. There is a limit on the number of blocks sent per time unit, hence on the maximal processor number $\mathfrak{P}$, which we compute as follows: $\mathfrak{P}$ is the smallest integer such that

$$(\mu^2 + 3\mu(r - k\mu))c\mathfrak{P} \geq \mu^2(r - k\mu)w.$$

We obtain that

$$\mathfrak{P} = \left\lceil \frac{\mu w}{3c} \right\rceil,$$

while neglecting the term μ^2 in the communication cost, as we assume $\frac{r}{\mu}$ to be large.

Once the resource selection is performed, we propose a straightforward algorithm: a single processor is responsible for the factorization of the pivot matrix and of the update of the vertical and horizontal panels, and then $\mathfrak{P}$ processors work in parallel at the update of the core matrix.

7.3 Algorithm for heterogeneous platforms

In this section, we simply sketch the algorithm for heterogeneous platforms. When targeting heterogeneous platforms, there is a big difference between LU factorization and matrix multiplication. Indeed, for LU once the size μ of the pivot matrix is fixed, all processors have to deal with it, whatever their memory capacities. There was no such fixed common constant for matrix multiplication. Therefore, a crucial step for heterogeneous platforms is to determine the size μ of the pivot matrix. Note that two pivot matrices at two different steps of the factorization may have different sizes, the constraint is that all workers must use the same size at any given step of the elimination.

In theory, the memory size of the workers can be arbitrary. In practice however, memory size usually is an integral number of Gigabytes, and at most a few tens of Gigabytes. So it is feasible to exhaustively study all the possible values of μ, estimate the processing time for each value, and then pick the best one. Therefore, in the following we assume the value of μ has been chosen, i.e., the pivot matrix is of a known size $\mu \times \mu$.

The memory layout used by each slave P_i follows the same policy than as for the homogeneous case:
- a chunk of the horizontal panel is kept in memory,
- rows of the horizontal panel are sent to P_i,
- and rows of the core matrix are sent to P_i and are returned to the master after update.

If $\mu_i = \mu$, processor P_i operates exactly as for the homogeneous case. But if the memory capacity of P_i does not perfectly correspond to the size chosen for the pivot matrix, we still have to decide the shape of the chunk of the horizontal panel that processor P_i is going to keep in its memory. We have two cases to consider:

1. $\mu_i < \mu$. In other words, P_i has not enough memory. Then we can imagine two different shapes for the horizontal panel chunk:

   (a) Square chunk, i.e., the chunk is of size $\mu_i \times \mu_i$. Then, for each update the master must send to P_i a row of size μ_i of the horizontal panel and a row of size μ_i of the core matrix, and P_i sends back after update the row of the core matrix. Hence a communication cost of $3\mu_i c$ for μ_i^2 computations. The computation-to-communication cost induced by this chunk shape is then:
   $$\frac{\mu_i^2 w}{3\mu_i c} = \frac{\mu_i w}{3c}.$$

   (b) Set of whole columns of the horizontal panel, i.e., the chunk is of size $\mu \times \left(\frac{\mu_i^2}{\mu}\right)$. Then, for each update the master must send to P_i a row of size μ of the horizontal panel and a row of size $\frac{\mu_i^2}{\mu}$ of the core matrix, and P_i sends back after update the row of the core matrix. Hence a communication cost of $\left(\mu + 2\frac{\mu_i^2}{\mu}\right) c$ for μ_i^2 computations. The computation to communication cost induced by this chunk shape is then:
   $$\frac{\mu_i^2 w}{\left(\mu + 2\frac{\mu_i^2}{\mu}\right) c}.$$

   The choice of the policy depends on the ratio $\frac{\mu_i}{\mu}$. Indeed,
   $$\frac{\mu_i^2 w}{3\mu_i c} < \frac{\mu_i^2 w}{\left(\mu + 2\frac{\mu_i^2}{\mu}\right) c} \quad \Leftrightarrow \quad \left(\mu + 2\frac{\mu_i^2}{\mu}\right) c < 3\mu_i c \quad \Leftrightarrow \quad \left(2\frac{\mu_i}{\mu} - 1\right)\left(\frac{\mu_i}{\mu} - 1\right) < 0.$$

   Therefore, the square chunk approach is more efficient if and only if $\mu_i \leq \frac{1}{2}\mu$.

2. $\mu_i > \mu$. In other words, P_i has more memory than necessary to hold a square matrix like the pivot matrix, that is a matrix of size $\mu \times \mu$. In that case, we propose to divide the memory of P_i into $\left\lfloor \frac{\mu_i^2}{\mu^2} \right\rfloor$ square chunks of size μ, and to use this processor as if there were in fact $\left\lfloor \frac{\mu_i^2}{\mu^2} \right\rfloor$ processors with a memory of size μ^2.

So far, we have assumed we knew the value of μ and we have proposed memory layout for the workers. We still have to decide which processor to enroll in the computation. We perform the resource selection as for matrix multiplication: we decide to assign only full matrix column blocks of the core matrix and of the horizontal panel to workers, and we actually perform resource selection using the same selection algorithms than for matrix-multiplication.

The overall process to define a solution is then:

1. For each possible value of μ do
    (a) Find the processor which will be the fastest to factor the pivot matrix, and to update the horizontal and vertical panels.
    (b) Perform resource selection and then estimate the running time of the update of the core-matrix.
2. Retain the solution leading to the best (estimated) overall running time.

8 MPI experiments

In this section, we aim at validating the previous theoretical results and algorithms. We conduct a variety of MPI experiments to compare our new schemes with several other algorithms from the literature. In the final version of this paper, we will report results obtained for heterogeneous platforms, assessing the impact of the degree of heterogeneity (in processor speed, link bandwidth and memory capacity) on the performance of the various algorithms. For this current version, we restrict to homogeneous platforms. Even in this simpler framework, using a sophisticated memory management turns out to be very important.

We start with a description of the platform, and of all the different algorithms that we compare. Then we describe the experiments that we have conducted and justify their purpose. Finally, we discuss the results.

8.1 Platform

For our experiments we are using a platform at the University of Tennessee. All experiments are performed on a cluster of 64 Xeon 3.2GHz dual-processor nodes. Each node of the cluster has four Gigabytes of memory and runs the Linux operating system. The nodes are connected with a switched 100Mbps Fast Ethernet network. In order to build a master-worker platform, we arbitrarily choose one processor as the master, and the other processors become the workers. Finally we used *MPI_ WTime* as timer in all experiments.

8.2 Algorithms

We choose six different algorithms from the general literature to compare our algorithm to. We partition these algorithms into two sets. The first set is composed of algorithms which use the same memory allocation than ours. The only difference between the algorithms is the order in which the master sends blocks to workers.

Homogeneous algorithm (**HoLM**) is our homogeneous algorithm. It makes resource selection, and sends blocks to the selected workers in a round-robin fashion.

Overlapped Round-Robin, Optimized Memory Layout (**ORROML**) is very similar to our homogeneous algorithm. The only difference between them is that it does not make any resource selection, and so sends tasks to all available workers in a round-robin fashion.

Overlapped Min-Min, Optimized Memory Layout (**OMMOML**) is a static scheduling heuristic, which sends the next block to the first worker that will be available to compute it. As it is looking for potential workers in a given order, this algorithm performs some resource selection too. Theoretically, as our homogeneous resource selection ensures that the first worker is free to compute when we finish to send blocks to the others, they should have similar behavior.

Overlapped Demand-Driven, Optimized Memory Layout (**ODDOML**) is a demand-driven algorithm. In order to use the extra buffers available in the worker memories, it will send the next block to the first worker which can receive it. This would be a dynamic version of our algorithm, if it took worker selection into account.

Demand-Driven, Optimized Memory Layout (**DDOML**) is a very simple dynamic demand-driven algorithm, close to **ODDOML**. It sends the next block to the first worker which is free for computation. As workers never have to receive and compute at the same time, the algorithm has no extra buffer, so the memory available to store $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ is greater. This may change the value of μ and so the behavior of the algorithm.

In the second set we have algorithms which do not use our memory allocation:

Block Matrix Multiply (**BMM**) is Toledo's algorithm [38]. It splits each worker memory equally into three parts, and allocate one slot for a square block of $\mathcal{A}$, another for a square block of $\mathcal{B}$, and the last one for a square block of $\mathcal{C}$, each square block having the same size. Then it sends blocks to the workers in a demand-driven fashion, when a worker is free for computation. First a worker receives a block of $\mathcal{C}$, then it receives corresponding blocks of $\mathcal{A}$ and $\mathcal{B}$ in order to update $\mathcal{C}$, until $\mathcal{C}$ is fully computed. In this version, a worker do not overlap computation with the receiving of the next blocks.

Overlapped Block Matrix Multiply (**OBMM**) is our attempt to improve the previous algorithm. We try to overlap the communications and the computations of the workers. To that purpose, we split each worker memory into five parts, so as to receive one block of $\mathcal{A}$ and one block of $\mathcal{B}$ while previous ones are used to update $\mathcal{C}$.

8.3 Experiments

We have built several experimental protocols in order to assess the performance of the various algorithms. In the following experiments we use nine processors, one master and eight workers. In all experiments we compare the execution time needed by the algorithms which use our memory allocation to the execution time of the other algorithms. We also point out the number of processors used by each algorithm, which is an important parameter when comparing execution times.

In the first set of experiments, we test the different algorithms on matrices of different sizes and shapes. The matrices we are multiplying are of actual size
- 8000×8000 for $\mathcal{A}$ and 8000×64000 for $\mathcal{B}$,
- 16000×16000 for $\mathcal{A}$ and 16000×128000 for $\mathcal{B}$, and
- 8000×64000 for $\mathcal{A}$ and 64000×64000 for $\mathcal{B}$.

All the algorithms using our optimized memory layout consider these matrices as composed of square blocks of size $q \times q = 80 \times 80$. For instance in the first case we have $r = t = 100$ and $s = 800$.

In the second set of experiments we check whether the choice of q was wise. For that purpose, we launch the algorithms on matrices of size 8000×8000 and 8000×64000, changing from one experiment to another the size of the elementary square blocks. Then q will be respectively equal to 40, 80, and 160. As the global matrix size is the same in all three experiments, we expect all three results to be the same.

In the third set of experiments we investigate the impact of the worker memory size onto the performance of the algorithms. In order to have reasonable execution times, we use matrices of size 16000×16000 and 16000×64000, and the memory size will vary from 132Mo to 2000Mo. We choose these values to reduce side effects due to the partition of the matrices into blocks of size $\mu q \times \mu q$.

In the fourth and last set of experiments we check the stability of the previous results. To that purpose we launch the same execution five times, in order to determine the maximum gap between two runs.

8.4 Results and discussion

We see in Figure 10 the results of the first set of experiments, where algorithms are computing different matrices. The first remark is that the shape of the three experiments is the same for all matrix sizes. We also underline the superiority of most of the algorithms which use our memory allocation against **BMM**: **HoLM**, **ORROML**, **ODDOML**, and **DDOML** are the best algorithms and have similar performance. Only **OMMOML** needs more time to complete its execution. This delay comes from its resource selection: it uses only two workers. For instance, **HoLM** uses four workers, and is as competitive as the other algorithms which all use the eight available workers.

In Figure 12, we see the impact of q on the performance of our algorithms. **BMM** and **OBMM** have same execution times in the three experiments as these algorithms do not split matrices into elementary square blocks of size $q \times q$ but, instead, call the Level 3 BLAS routines directly on the whole $\sqrt{\frac{m}{3}} \times \sqrt{\frac{m}{3}}$ matrices. In the two cases we see that the time of the algorithms are similar. We point out that this experiment shows that the choice of q has little impact on the algorithms performance.

In Figure 13 we have the impact of the worker memory size on the performance of the algorithms. As expected, the performance increases with the amount of memory available. It is interesting to underline that our resource selection always performs in the best possible way. **HoLM** will use respectively two and four workers when the memory available increases, compared to the other algorithms which will use all eight available workers on each test. **OMMOML** also makes some resource selection, but it performs worse.

Finally, Figure 11 shows the difference that we can have between two runs. This difference is around 6%. Thus if two algorithms have less than 6% of difference in execution time, they should be considered as similar.

To conclude, these experiments stress the superiority of our memory allocation. Furthermore, our homogeneous algorithm is as competitive as the others but uses fewer resources.

9 Related work

In this section, we provide a brief overview of related papers, which we classify along the following five main lines:

Load balancing on heterogeneous platforms – Load balancing strategies for heterogeneous platforms have been widely studied. Distributing the computations (together with the associated data) can be performed either dynamically or statically, or a mixture of both. Some

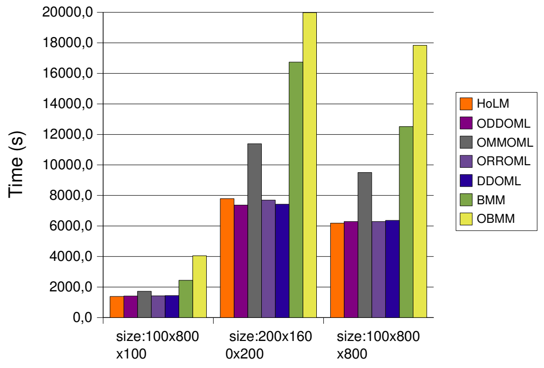


Figure 10: Performance of the algorithms on different matrices.

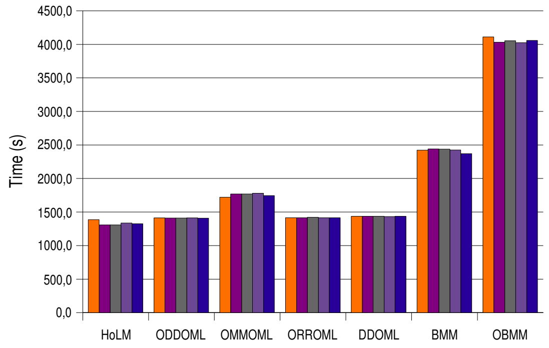


Figure 11: Variation of algorithm execution times.

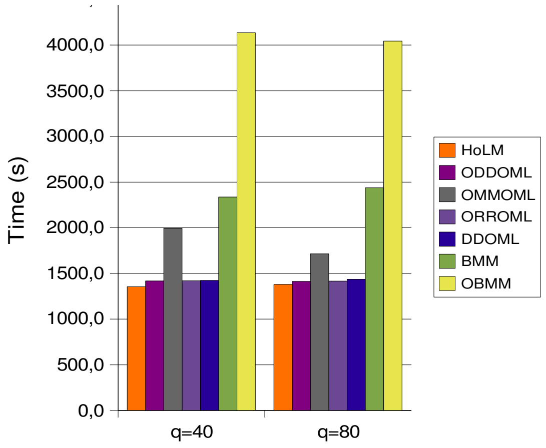


Figure 12: Impact of q on algorithm performance.

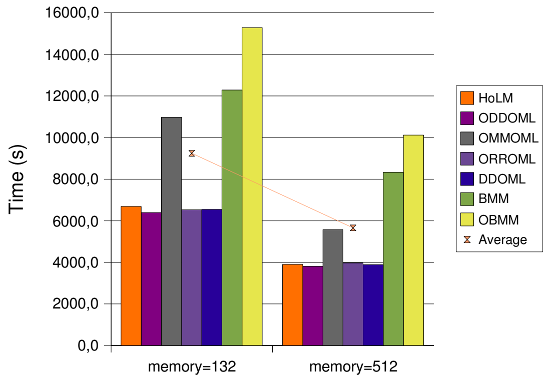


Figure 13: Impact of memory size on algorithm performance.

simple schedulers are available, but they use naive mapping strategies such as master-worker techniques or paradigms based upon the idea *“use the past to predict the future”*, i.e. use the currently observed speed of computation of each machine to decide for the next distribution of work [17, 18, 9]. Dynamic strategies such as *self-guided scheduling* [34] could be useful too. There is a challenge in determining a trade-off between the data distribution parameters and the process spawning and possible migration policies. Redundant computations might also be necessary to use a heterogeneous cluster at its best capabilities. However, dynamic strategies are outside the scope of this paper (but mentioned here for the sake of completeness). Because we have a library designer’s perspective, we concentrate on static allocation schemes that are less general and more difficult to design than dynamic approaches, but which are better suited for the implementation of fixed algorithms such as linear algebra kernels from the ScaLAPACK library [13].

Out-of-core linear algebra routines – As already mentioned, the design of parallel algorithms for limited memory processors is very similar to the design of out-of-core routines for classical parallel machines. On the theoretical side, Hong and Kung [26] investigate the I/O complexity of several computational kernels in their pioneering paper. Toledo [38] proposes a nice survey on the design of out-of-core algorithms for linear algebra, including dense and sparse computations. We refer to [38] for a complete list of implementations. The design principles followed by most implementations are introduced and analyzed by Dongarra et al. [22].

Linear algebra algorithms on heterogeneous clusters – Several authors have dealt with the *static* implementation of matrix-multiplication algorithms on heterogeneous platforms. One simple approach is given by Kalinov and Lastovetsky [29]. Their idea is to achieve a perfect load-balance as follows: first they take a fixed layout of processors arranged as a collection of processor columns; then the load is evenly balanced *within* each processor column independently; next the load is balanced *between* columns; this is the "heterogeneous block cyclic distribution" of [29]. Another approach is proposed by Crandall and Quinn [20], who propose a recursive partitioning algorithm, and by Kaddoura, Ranka and Wang [28], who refine the latter algorithm and provide several variations. They report several numerical simulations. As pointed out in the introduction, theoretical results for matrix multiplication and LU decomposition on 2D-grids of heterogeneous processors are reported in [5], while extensions to general 2D partitioning are considered in [6]. See also Lastovetsky and Reddy [31] for another partitioning approach.

Recent papers aim at making easier the process of tuning linear algebra kernels on heterogeneous systems. Self-optimization methodologies are described by Cuenca et al [21] and by Chen et al [16]. Along the same line, Chakravarti et al. [15] describe an implementation of Cannon's algorithm using self-organizing agents on a peer-to-peer network.

Models for heterogeneous platforms – In the literature, one-port models come in two variants. In the unidirectional variant, a processor cannot be involved in more than one communication at a given time-step, either a send or a receive. This is the model that we have used throughout the paper. In the bidirectional model, a processor can send and receive in parallel, but at most to a given neighbor in each direction. In both variants, if P_u sends a message to P_v, both P_u and P_v are blocked throughout the communication.

The bidirectional one-port model is used by Bhat et al. [10, 11] for fixed-size messages. They advocate its use because "current hardware and software do not easily enable multiple messages to be transmitted simultaneously." Even if non-blocking, multi-threaded communication libraries allow for initiating multiple send and receive operations, they claim that all these operations "are eventually serialized by the single hardware port to the network." Experimental evidence of this fact has recently been reported by Saif and Parashar [35], who report that asynchronous MPI sends get serialized as soon as message sizes exceed a few megabytes. Their results hold for two popular MPI implementations, MPICH on Linux clusters and IBM MPI on the SP2.

The one-port model fully accounts for the heterogeneity of the platform, as each link has a different bandwidth. It generalizes a simpler model studied by Banikazemi et al. [1] Liu [32] and Khuller and Kim [30]. In this simpler model, the communication time only depends on the sender, not on the receiver. In other words, the communication speed from a processor to all its neighbors is the same.

Finally, we note that some papers [2, 4] depart form the one-port model as they allow a sending processor to initiate another communication while a previous one is still on-going on the network. However, such models insist that there is an overhead time to pay before being engaged in another operation, so they are not allowing for fully simultaneous communications.

Master-worker on the computational grid – Master-worker scheduling on the grid can be based on a network-flow approach [37, 36] or on an adaptive strategy [24]. Note that the network-flow approach of [37, 36] is possible only when using a full multiple-port model, where the number of simultaneous communications for a given node is not bounded. This approach has also been studied in [25]. Enabling frameworks to facilitate the implementation of master-worker tasking are described in [23, 39].

10 Conclusion

The main contributions of this paper are the following:

1. On the theoretical side, we have derived a new, tighter, bound on the minimal volume of communications needed to multiply two matrices. From this lower bound, we have defined an efficient memory layout, i.e., an algorithm to share the memory available on the workers among the three matrices.

2. On the practical side, starting from our memory layout, we have designed an algorithm for homogeneous platforms whose performance is quite close to the communication volume lower bound. We have extended this algorithm to deal with heterogeneous platforms, and discussed how to adapt the approach for LU factorization.

3. Through MPI experiments, we have shown that our algorithm for homogeneous platforms has far better performance than solutions using the memory layout proposed in [38]. Furthermore, this static homogeneous algorithm has similar performance as dynamic algorithms using the same memory layout, but uses fewer processors. It is therefore a very good candidate for deploying applications on regular, homogeneous platforms.

We are currently conducting experiments to assess the performance of the extension of the algorithm for heterogeneous clusters.

References

[1] M. Banikazemi, V. Moorthy, and D. K. Panda. Efficient collective communication on heterogeneous networks of workstations. In *Proceedings of the 27th International Conference on Parallel Processing (ICPP'98)*. IEEE Computer Society Press, 1998.

[2] M. Banikazemi, J. Sampathkumar, S. Prabhu, D. Panda, and P. Sadayappan. Communication modeling of heterogeneous networks of workstations for performance characterization of collective operations. In *HCW'99, the 8th Heterogeneous Computing Workshop*, pages 125–133. IEEE Computer Society Press, 1999.

[3] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Scheduling strategies for master-slave tasking on heterogeneous processor platforms. *IEEE Trans. Parallel Distributed Systems*, 15(4):319–330, 2004.

[4] A. Bar-Noy, S. Guha, J. S. Naor, and B. Schieber. Message multicasting in heterogeneous networks. *SIAM Journal on Computing*, 30(2):347–358, 2000.

[5] O. Beaumont, V. Boudet, A. Petitet, F. Rastello, and Y. Robert. A proposal for a heterogeneous cluster ScaLAPACK (dense linear solvers). *IEEE Trans. Computers*, 50(10):1052–1070, 2001.

[6] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert. Matrix multiplication on heterogeneous platforms. *IEEE Trans. Parallel Distributed Systems*, 12(10):1033–1051, 2001.

[7] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert. Partitioning a square into rectangles: NP-completeness and approximation algorithms. *Algorithmica*, 34:217–239, 2002.

[8] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, L. Marchal, and Y. Robert. Centralized versus distributed schedulers for multiple bag-of-task applications. In *International Parallel and Distributed Processing Symposium IPDPS'2006*. IEEE Computer Society Press, 2006.

[9] F. Berman. High-performance schedulers. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, pages 279–309. Morgan-Kaufmann, 1999.

[10] P. Bhat, C. Raghavendra, and V. Prasanna. Efficient collective communication in distributed heterogeneous systems. In *ICDCS'99 19th International Conference on Distributed Computing Systems*, pages 15–24. IEEE Computer Society Press, 1999.

[11] P. Bhat, C. Raghavendra, and V. Prasanna. Efficient collective communication in distributed heterogeneous systems. *Journal of Parallel and Distributed Computing*, 63:251–263, 2003.

[12] L. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, 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 *Supercomputing '96*. IEEE Computer Society Press, 1996.

[13] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.

[14] L. E. Cannon. *A cellular computer to implement the Kalman filter algorithm*. PhD thesis, Montana State University, 1969.

[15] A. Chakravarti, G. Baumgartner, and M. Lauria. Self-organizing scheduling on the organic grid. *Int. Journal of High Performance Computing Applications*, 20(1):115–130, 2006.

[16] Z. Chen, J. Dongarra, P. Luszczek, and K. Roche. Self adapting software for numerical linear algebra and lapack for clusters. *Parallel Computing*, 29(11-12):1723–1743, 2003.

[17] M. Cierniak, M. Zaki, and W. Li. Compile-time scheduling algorithms for heterogeneous network of workstations. *The Computer Journal*, 40(6):356–372, 1997.

[18] M. Cierniak, M. Zaki, and W. Li. Customized dynamic load balancing for a network of workstations. *Journal of Parallel and Distributed Computing*, 43:156–162, 1997.

[19] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[20] P. E. Crandall and M. J. Quinn. Block data decomposition for data-parallel programming on a heterogeneous workstation network. In *2nd International Symposium on High Performance Distributed Computing*, pages 42–49. IEEE Computer Society Press, 1993.

[21] J. Cuenca, L. P. Garcia, D. Gimenez, and J. Dongarra. Processes distribution of homogeneous parallel linear algebra routines on heterogeneous clusters. In *HeteroPar'2005: International Conference on Heterogeneous Computing.* IEEE Computer Society Press, 2005.

[22] J. Dongarra, S. Hammarling, and D. Walker. Key concepts for parallel out-of-core LU factorization. *Parallel Computing*, 23(1-2):49–70, 1997.

[23] J. P. Goux, S. Kulkarni, J. Linderoth, and M. Yoder. An enabling framework for master-worker applications on the computational grid. In *Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00).* IEEE Computer Society Press, 2000.

[24] E. Heymann, M. A. Senar, E. Luque, and M. Livny. Adaptive scheduling for master-worker applications on the computational grid. In R. Buyya and M. Baker, editors, *Grid Computing - GRID 2000*, pages 214–227. Springer-Verlag LNCS 1971, 2000.

[25] B. Hong and V. Prasanna. Bandwidth-aware resource allocation for heterogeneous computing systems to maximize throughput. In *Proceedings of the 32th International Conference on Parallel Processing (ICPP'2003).* IEEE Computer Society Press, 2003.

[26] J.-W. Hong and H. Kung. I/O complexity: the red-blue pebble game. In *STOC '81: Proceedings of the 13th ACM symposium on Theory of Computing*, pages 326–333. ACM Press, 1981.

[27] D. Ironya, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distributed Computing*, 64(9):1017–1026, 2004.

[28] M. Kaddoura, S. Ranka, and A. Wang. Array decomposition for nonuniform computational environments. *Journal of Parallel and Distributed Computing*, 36:91–105, 1996.

[29] A. Kalinov and A. Lastovetsky. Heterogeneous distribution of computations while solving linear algebra problems on networks of heterogeneous computers. In P. Sloot, M. Bubak, A. Hoekstra, and B. Hertzberger, editors, *HPCN Europe 1999*, LNCS 1593, pages 191–200. Springer Verlag, 1999.

[30] S. Khuller and Y. Kim. On broadcasting in heterogenous networks. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1011–1020. Society for Industrial and Applied Mathematics, 2004.

[31] A. Lastovetsky and R. Reddy. Data partitioning with a realistic performance model of networks of heterogeneous computers. In *International Parallel and Distributed Processing Symposium IPDPS'2004.* IEEE Computer Society Press, 2004.

[32] P. Liu. Broadcast scheduling optimization for heterogeneous cluster systems. *Journal of Algorithms*, 42(1):135–152, 2002.

[33] M. Maheswaran, S. Ali, H. Siegel, D. Hensgen, and R. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Eight Heterogeneous Computing Workshop*, pages 30–44. IEEE Computer Society Press, 1999.

[34] C. D. Polychronopoulos. Compiler optimization for enhancing parallelism and their impact on architecture design. *IEEE Transactions on Computers*, 37(8):991–1004, Aug. 1988.

[35] T. Saif and M. Parashar. Understanding the behavior and performance of non-blocking communications in MPI. In *Proceedings of Euro-Par 2004: Parallel Processing*, LNCS 3149, pages 173–182. Springer, 2004.

[36] G. Shao. *Adaptive scheduling of master/worker applications on distributed computational resources*. PhD thesis, Dept. of Computer Science, University Of California at San Diego, 2001.

[37] G. Shao, F. Berman, and R. Wolski. Master/slave computing on the grid. In *Heterogeneous Computing Workshop HCW'00*. IEEE Computer Society Press, 2000.

[38] S. Toledo. A survey of out-of-core algorithms in numerical linear algebra. In *External Memory Algorithms and Visualization*, pages 161–180. American Mathematical Society Press, 1999.

[39] J. B. Weissman. Scheduling multi-component applications in heterogeneous wide-area networks. In *Heterogeneous Computing Workshop HCW'00*. IEEE Computer Society Press, 2000.

[40] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the ACM/IEEE Symposium on Supercomputing (SC'98)*. IEEE Computer Society Press, 1998.