Matrix Product on Heterogeneous Master-Worker Platforms

Jack Dongarra[1] Jean-François Pineau[2,4,5] Yves Robert[2,4,5] Frédéric Vivien[3,4,5]

[1]University of Tennessee and Oak Ridge National Lab, USA and University of Manchester UK [2]École Normale Supérieure de Lyon [3]INRIA [4]Université de Lyon [5]LIP, UMR 5668 CNRS - ENS Lyon - INRIA - UCBL, Lyon, France

dongarra@cs.utk.edu {Jean-Francois.Pineau, Yves.Robert, Frederic.Vivien}@ens-lyon.fr

Abstract

This paper is focused on 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 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).
- *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.* As we investigate the parallelization of large problems, we cannot assume that full matrix column blocks can be stored in the worker memories and be re-used for subsequent updates (as in ScaLAPACK).

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 a platform at our site. The experiments show that our matrix-product algorithm has smaller execution times than existing ones, while it also uses fewer resources.

Categories and Subject Descriptors F.2.1 [Analysis of algorithms and problem complexity]: Numerical Algorithms and Problems – Computations on matrices; F.2.2 [Analysis of algorithms and problem complexity]: Nonnumerical Algorithms and Problems – Scheduling

General Terms Algorithms, Experimentation, Theory

Keywords Matrix product, limited memory, communication.

PPoPP'08, February 20–23, 2008, Salt Lake City, Utah, USA.

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 [6] and the ScaLAPACK outer product algorithm [5]. 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 workers. Therefore, an accurate estimation of the communication cost requires 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 $\Theta(n^2)$ coefficients need to be distributed in the beginning, and gathered at the end, as opposed to the $\Theta(n^3)$ computations[1] 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. Furthermore, when processors cannot store all the matrices in their memory, the total volume of communication required can be larger than $\Theta(n^2)$, as a same matrix element may have to be sent several times to a same processor.

In this paper, we are not interested in adapting 2D processor grid strategies to heterogeneous clusters, as proposed in [11, 2, 3]. Instead, we adopt a realistic application scenario, where input files are read from a fixed repository (such as a disk on a data server). Computations are delegated to available computational resources, 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

[1] Of course, there are $\Theta(n^3)$ computations if we only consider algorithms that uses the standard way of multiplying matrices; e.g., this excludes Strassen's and Winograd's algorithms.

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, and workers are connected to the master by links of different capacities. This framework is realistic when deploying the application from a server, which is responsible for enrolling authorized resources.

Finally, because we investigate the parallelization of large problems, we cannot assume that full matrix panels, or column blocks, 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 [5]: $q = 80$ 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? only the fastest computing ones? only the fastest-communicating ones? Once participating resources have been selected, there remain several scheduling problems: how to minimize the number of communications? in which order workers should receive input data and return results? what amount of communications can be overlapped with (independent) computations? The main contributions of this paper are twofold:

- *On the theoretical side*, we have refined the existing bounds on the volume of communications needed to perform a matrix-product on a platform with insufficient memory to simultaneously store the whole three matrices.
- *On the practical side*, we have designed an algorithm for heterogeneous platforms which is quicker than the existing ones, and which uses less computational resources, according to our MPI experiments.

The rest of the paper is organized as follows. In Section 2, we state the scheduling problem precisely, and we introduce notations. Next, in Section 3, 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 [17, 10]. In order to help the reader apprehend the solution for heterogeneous platforms, we first deal with homogeneous platforms in Section 4, and we propose a scheduling algorithm that includes resource selection. Section 5 addresses the design of algorithms for heterogeneous platforms, which turns out to be a truly challenging algorithmic task. We report several MPI experiments in Section 6. Section 7 is devoted to an overview of related work. Finally, we state some concluding remarks in Section 8.

2. Framework

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. Formally:

- 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 [5]. Typically, $q = 80$ or 100 when using ATLAS-generated routines [7].
- 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 the same size. This will greatly simplify the analysis of communication costs.

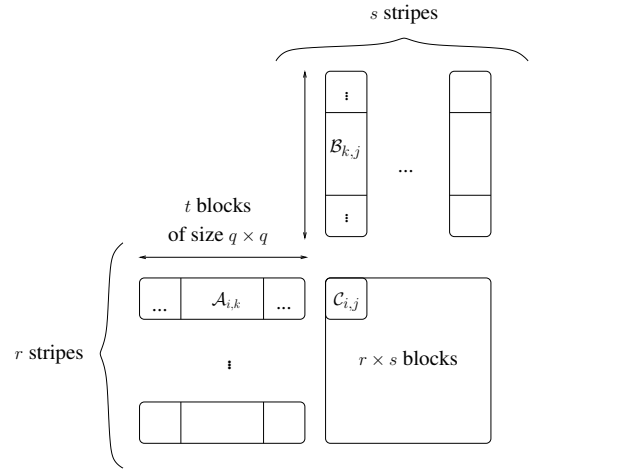


Figure 1. Partitioning of the three matrices $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$.

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, $1 \leq i \leq p$.In practice, this star network can be a logical overlay built upon a different physical interconnection network. 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).

For the communication model, we adopt the *one-port* model [4], which is defined as follows: 1) the master can only send data to, and receive data from, a single worker at a given time-step; 2) 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; 3) a given worker can overlap computation and communication of independent tasks.

The *one-port* model is *realistic*. Bhat, Raghavendra, and Prasanna [4] 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 [16], who

report that asynchronous MPI sends get serialized as soon as message sizes exceed a hundred kilobytes. Their results hold for two popular MPI implementations, MPICH on Linux clusters and IBM MPI on the SP2. Note that all the MPI experiments in Section 6 obey the one-port model.

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. 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. Any parallel algorithm can always be simulated on a single worker. Therefore, since we are not interested in optimizing the execution time, but only in minimizing the total communication volume, we only need to consider the one-worker case. We then deal with the following formulation of the problem:

- 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 improve the lower bound on the communication volume established by Toledo et al. [17, 10]. Then, we describe an algorithm that aims at re-using $\mathcal{C}$ blocks as much as possible after they have been loaded, and we assess its performance.

Lower bound on the communication volume

To derive the lower bound, we refine an analysis due to Toledo [17]. The idea is to estimate the number of computations made thanks to m consecutive communication steps (once again, the unit here is a matrix block). We need some notations:

- We let α_{old}, β_{old}, and γ_{old} be the number of buffers used by blocks of $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ right before 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.

Initially, the memory contains at most m blocks, and we consider a sequence of m communications. Therefore:

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

We then use Loomis-Whitney inequality [10]: in any algorithm that uses the standard way of multiplying matrices, if N_A elements of $\mathcal{A}$, N_B elements of $\mathcal{B}$, and N_C elements of $\mathcal{C}$ are accessed, no more than K computations can be done, where $K = \sqrt{N_A N_B N_C}$. Here $K = \sqrt{(\alpha_{old} + \alpha_{recv})(\beta_{old} + \beta_{recv})(\gamma_{old} + \gamma_{recv})}$. K is maximized when $\alpha_{old} + \alpha_{recv} = \beta_{old} + \beta_{recv} = \gamma_{old} + \gamma_{recv} = \frac{2}{3}m$, and the lower bound for the communication-to-computation ratio is: $\text{CCR}_{\text{opt}} \geq \sqrt{\frac{27}{8m}}$. This bound improves upon the best-known value $\sqrt{\frac{1}{8m}}$ derived by Ironya, Toledo, and Tiskin [10].

The *maximum re-use* algorithm

In the above study, the lower-bound on the communication volume is obtained when the three matrices $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ are equally accessed during a sequence of computations. This may suggest allocating one third of the memory to each of these matrices. In fact, Toledo [17] uses this memory layout. He even proves that, in the context of multiplication of square matrices of size r, his algorithm is "asymptotically optimal" as soon as the processor cannot store in its memory more than one sixth of one of the matrices: such an algorithm must have a communication-per-computation ratio which is $\Omega\left(\frac{r^3}{\sqrt{m}}\right)$ when his algorithm has a communication to computation ratio of $O\left(\frac{r^3}{\sqrt{m}}\right)$. We can, however, still significantly improve the performance of matrix multiplication in our context by reducing the constant hidden in the order of complexity of this ratio, as our experiments will show in Section 6.

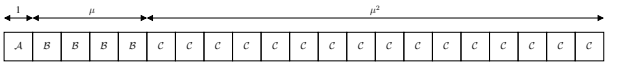


Figure 2. Memory layout for the *maximum re-use* algorithm when $m = 21$: $\mu = 4$; 1 block is used for $\mathcal{A}$, μ for $\mathcal{B}$, and μ^2 for $\mathcal{C}$.

A closer look at our problem shows that the multiplied matrices $\mathcal{A}$ and $\mathcal{B}$ have the same behavior, which differs from the behavior of the result matrix $\mathcal{C}$. Indeed, if an element of $\mathcal{C}$ is no longer used, it cannot be simply discarded from the memory as the elements of $\mathcal{A}$ and $\mathcal{B}$ are, but it must be sent back to the master. Intuitively, sending an element of $\mathcal{C}$ to a worker also costs the communication needed to retrieve it from the worker, and is thus twice as expensive as sending an element of $\mathcal{A}$ or $\mathcal{B}$. Therefore, we designed an algorithm that reuses as much as possible the elements of $\mathcal{C}$.

Cannon's algorithm [6] and the ScaLAPACK outer product algorithm [5] 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 data needed by a worker to compute the area) is smaller for the same area. We use the same approach here.

The *maximum re-use* algorithm uses the memory layout illustrated in Figure 2. Four consecutive execution steps are shown in Figure 3. 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 2, 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: $\{\mathcal{C}_{i,j} \mid i_0 \leq i < i_0 + \mu,\ j_0 \leq j < j_0 + \mu\}$.
- **Inner loop**: For each k from 1 to t:
  1. Send a row of μ elements $\{\mathcal{B}_{k,j} \mid j_0 \leq j < j_0 + \mu\}$;
  2. Sequentially send μ elements of column $\{\mathcal{A}_{i,k} \mid i_0 \leq i < i_0 + \mu\}$. For each $\mathcal{A}_{i,k}$, update μ elements of $\mathcal{C}$
- Return results to master.

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.

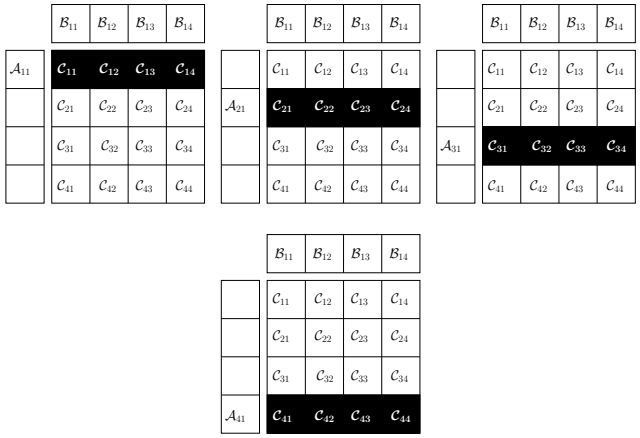


Figure 3. Four steps of the *maximum re-use* algorithm, with $m = 21$ and $\mu = 4$. Updated elements of $\mathcal{C}$ are written white on black.

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. 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 [17]. Finally, we remark that the performance of the *maximum re-use* algorithm is quite close to the lower bound derived earlier: $\text{CCR}_\infty = \frac{2}{\sqrt{m}} = \sqrt{\frac{32}{8m}}$.

4. Algorithms for homogeneous platforms

We now adapt the *maximum re-use* algorithm to fully homogeneous platforms. We have a limitation on 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.

Principle of the algorithm

We load into the memory of each worker μ blocks of $\mathcal{A}$ and μ blocks of $\mathcal{B}$ to compute μ^2 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 μ^2 of $\mathcal{C}$ blocks that we can simultaneously load into memory is defined by the largest integer μ such that $\mu^2 + 4\mu \leq m$.

We have to determine the number of participating workers $\mathfrak{P}$. 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 $\mathcal{B}$. 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 that it can send per time unit. On the contrary, if we enroll too few processors, they may be overloaded. We can compute the number of processors $\mathfrak{P}$ so that the time needed to send blocks to $\mathfrak{P}$ processors will be roughly equal to (or slightly greater than) the time spent by one processor for its computations. $\mathfrak{P}$ is the smallest integer such that $2\mu tc \times \mathfrak{P} \geq \mu^2 tw$. Indeed, this is the smallest value to saturate the communication capacity of the master while sustaining the corresponding computations. Finally, we cannot use more processors than available. In the context of matrix multiplication, c and w are of the form $c = q^2\tau_c$ and $w = q^3\tau_a$, where τ_c and τ_a respectively represent the elementary communication and computation times. Hence:

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

For the sake of simplicity, we suppose that r is divisible by μ, and s 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 made of two parts: Algorithm 1 outlines the program of the master, while Algorithm 2 is the program of each worker.

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 matrix $\mathcal{C}$ into squares $\mathbf{C}_{i',j'}$ of μ^2 $q \times q$ blocks :
$\mathbf{C}_{i',j'} = \{\mathcal{C}_{i,j} \mid (i'-1)\mu+1 \leq i \leq i'\mu, (j'-1)\mu+1 \leq j \leq j'\mu\}$
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}

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

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. We decide to sequentialize the sending, computing, and receiving of the $\mathcal{C}$ blocks, 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 would become much more complicated.

5. Algorithms for heterogeneous platforms

We now consider the general problem, i.e., when 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 memory changes from worker to worker, as it depends on their memory capacities: μ_i is the largest integer such that $\mu_i^2 + 4\mu_i \leq m_i$. To adapt our *maximum re-use* algorithm to heterogeneous platforms, we first present a steady-state-like approach and discuss its limitations. We then introduce our algorithm for heterogeneous platforms.

Bandwidth-centric resource selection

Each worker P_i has parameters c_i, w_i, and μ_i, and each participating P_i needs a time $2\mu_i t c_i$ to receive its blocks and a time $t\mu_i^2 w_i$ to perform its 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 has limited computing power and cannot perform more work than it receives. The objective is to maximize the amount of work performed per time-unit. Altogether, we gather the linear program presented in Figure 1. The optimal solution for this system is a bandwidth-centric strategy [1]: 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 2. Using the bandwidth-centric strategy, every $8x$ seconds:

- P_1 receives $4x$ blocks (x $\mu_1 \times \mu_1$ chunks) in $4x$ seconds, and computes $4x$ blocks in $8x$ seconds;
- P_2 receives 4 blocks (1 $\mu_2 \times \mu_2$ chunk) in $4x$ seconds, and computes 4 blocks in $8x$ seconds.

But P_1 computes too quickly: during the time x needed to send a block to P_2, P_1 updates $\frac{x}{2}$ blocks, which requires at least $\sqrt{2x}$ blocks and as many buffers. As x can be arbitrary large, 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, the steady-state solution is an upper bound on the performance that can be achieved.

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, while simplifying 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 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 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. 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 of the first phase. 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.

Global selection algorithm. The intuitive idea, here, 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:

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

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 ready times into account: if P_i is currently busy executing work, it cannot receive additional data too much in advance as its memory is limited.

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. On the other hand, 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.

$$\begin{cases} \text{MAXIMIZE } \sum_i x_i \text{ SUBJECT TO} \\ \sum_i y_i c_i \leq 1 \\ \forall i \quad x_i w_i \leq 1 \\ \forall i \quad \frac{x_i}{\mu_i^2} \leq \frac{y_i}{2\mu_i} \end{cases}$$

Table 1. Linear program maximizing the amount of work performed per time-unit.

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

Table 2. Platform for which the bandwidth centric solution is not feasible.

Variants. At this point, we can have a global or a local selection process. Each process can either make a decision only looking at the next communication, or in a look-ahead approach. Another variant would be to take into account the cost of initially sending μ_i^2 blocks of matrix $\mathcal{C}$ to processor P_i the first time its receives blocks. Overall, we thus define eight different selection algorithms (global or local, look-ahead or not, μ_i^2 $\mathcal{C}$ costs or not). There is no reason for one of these heuristics to always dominate the others. We will thus consider the eight of them in our experiments.

6. 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 algorithms from the literature. We target heterogeneous platforms, and we assess the impact of the degree of heterogeneity (in processor speed, link bandwidth, and memory capacity) on the performance of the various algorithms.

The code and the experimental results can be downloaded from: http://graal.ens-lyon.fr/~jfpineau/Downloads/ppopp.tgz.

6.1 Platforms

We used a heterogeneous cluster composed of twenty-seven processors located in Lyon. It is composed of four different homogeneous sets of machines. The different sets are composed of: 1) 8 SuperMicro servers 5013-GM, with processors P4 2.4 GHz; 2) 5 SuperMicro servers 6013PI, with processors P4 Xeon 2.4 GHz; 3) 7 SuperMicro servers 5013SI, with processors P4 Xeon 2.6 GHz; 4) 7 SuperMicro servers IDE250W, with processors P4 2.8 GHz.

All nodes have 1 GB of memory and are running the Linux operating system. The nodes are connected with a switched 10 Mbps Fast Ethernet network. As this platform may not be as heterogeneous as we would like, we sometimes artificially modify its heterogeneity. In order to artificially slow down a communication link, we send the same message several times to one worker. The same idea works for processor speeds: we ask a worker to compute a given matrix-product several times in order to slow down its computation capability. In all experiments, except the last batch, we used nine processors: one master and eight workers.

6.2 Algorithms

We choose four different algorithms from the literature which we compare our algorithms to. The closest work addressing our problem is Toledo's out-of-core algorithm [17]. Hence, this work will serve as the baseline reference. Then we will study hybrid algorithms, i.e., algorithms which use our memory layout and are based on classical principles such as round-robin, min-min [13], or a dynamic demand-driven approach. The first six algorithms below use our memory allocation, the only difference between them is the order in which the master sends blocks to workers.

Homogeneous algorithm (**Hom**) is our homogeneous algorithm. It makes resource selection and sends blocks to the selected workers in a round-robin fashion. When run on a heterogeneous platform, it tries to build a very simple homogeneous platform. As the algorithm's only constraint is to send same size blocks to all participating workers, for a given memory size, we consider the homogeneous virtual platform composed of those workers having at least that amount of memory, and we estimate the total execution time of our homogeneous algorithm, for the targeted matrix-product, on that virtual platform (the apparent processor speed is the minimum of the processor speeds, the apparent communication bandwidth is the minimum of the communication bandwidths). We do this process for all the different memory sizes present in the actual platform, and we pick the virtual platform that minimizes the total estimated execution time.

Homogeneous algorithm improved (**HomI**) is our homogeneous algorithm running on a more carefully chosen homogeneous platform. For each memory size, communication speed, and computation speed present in an heterogeneous platform, we consider the homogeneous virtual platform composed of those workers having at least that performance. Then, we compute the total execution time of our homogeneous algorithm, for the targeted matrix-product, on that virtual platform (the apparent processor speed is the considered processor speed, and so on). We do this process for all the existing values, and we pick the virtual platform that minimizes the total execution time.

Heterogeneous algorithm (**Het**) is our heterogeneous algorithm. As we can have eight different versions of the resource selection, in a first step we simulate the eight versions, and then we pick and run the best one.

Overlapped Round-Robin, Optimized Memory Layout (**ORROML**) sends tasks to all available workers in a round-robin fashion. It does not make any resource selection.

Overlapped Min-Min, Optimized Memory Layout (**OMMOML**) is a static scheduling heuristic, which sends the next block to the first worker that will finish 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 sending blocks to the others, **OMMOML** and **Hom** should have a similar behavior on homogeneous platforms.

Overlapped Demand-Driven, Optimized Memory Layout (**ODDOML**) is a demand-driven algorithm. In our memory layout, two buffers of size μ_i are reserved for matrix $\mathcal{A}$, and two for matrix $\mathcal{B}$. In order to use the two available extra buffers (the second for $\mathcal{A}$ and the second for $\mathcal{B}$), one sends 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.

Block Matrix Multiply (**BMM**) is Toledo's algorithm [17]. It splits each worker memory equally into three parts and allocates 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 $\mathcal{C}$, with the square blocks having the same size. It sends blocks to the workers in a demand-driven fashion.

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.

Note that the six algorithms using our optimized memory layout are considering matrices as composed of square blocks of size $q \times q = 80 \times 80$, while **BMM** loads three panels, each of size one third of the available memory, for $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$.

When launching an algorithm on the platform, the very first step we do is to determine the platform's parameters. For that purpose, we launch a benchmark on it, in order to get the memory size, the communication speed, and the computation speed. The different speeds are determined by sending and computing a square block of size $q \times q$ ten times on each worker, and computing the median of the times obtained. This step takes between 20 and 80 seconds, depending on the speed of the workers, and is made before each algorithm, even **ORROML**, **ODDOML**, and **BMM**, which only need the memory size. This step represents at most 2% of the total time of execution.

In the following section, the times given takes into account the decision process of the algorithms, i.e., the simulation of the eight different versions of the resource selection for **Het**, the construction of an homogeneous platform for **Hom** and **HomI**, etc.

6.3 Experimental results

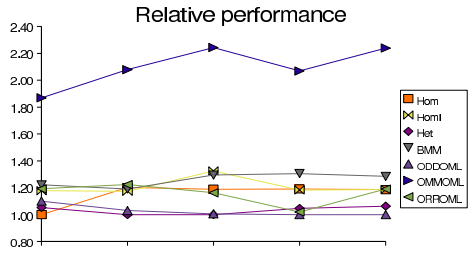
In the first three sets of experiments, we only have one parameter of heterogeneity, either the amount of memory, the communication speed, or the computation speed. We test the algorithms on such platforms with five matrices of increasing sizes. As we do not want to change several parameters at a time, we only change the value of parameter s (rather than, for instance, always consider square matrices). Matrix $\mathcal{A}$ is of size 8000×8000 whereas $\mathcal{B}$ is of increasing sizes 8000×64000, 8000×80000, 8000×96000, 8000×112000, and 8000×128000. For all other experiments, $\mathcal{A}$ is of size 8000×8000 and $\mathcal{B}$ is of size 8000×80000. The heterogeneous workers have different memory capacities, which implies that each algorithm, even **BMM**, assigns them chunks of different sizes. In order to simplify the global partitioning of matrix $\mathcal{C}$, we decide to only assign workers full matrix column blocks.

As we want to assess whether the performance of any studied algorithm depends on the matrix size, we look at the *relative cost* of the algorithms rather than at their absolute execution times. The relative cost of a given algorithm on a particular instance is equal to the execution time, or makespan, achieved on that instance by the algorithm, divided by the minimum makespan achieved on that instance. Using relative cost also enables us to build statistics on the performance of algorithms.

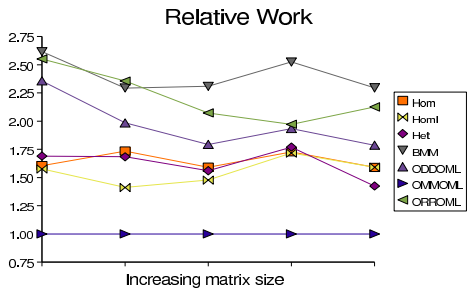
Besides relative cost, we take into account the number of processors used. To assess the efficiency of a given algorithm, we look at its *relative work*, which is equal, for a given instance, to its makespan times the number of enrolled processors, divided by the minimum of this value over all studied algorithms.

Heterogeneous memory size

Here we assess the performance of our algorithms with respect to memory heterogeneity. We launch the algorithms on a homogeneous platform in terms of communication and computation capabilities, but where workers have different memory capacities. We suppose that two workers only have 256 MB of memory, four of them 512 MB, and the last two 1024 MB.



(a) Cost of algorithms.



(b) Efficiency of algorithms.

Figure 4. Heterogeneous memory.

Figure 4(a) presents the relative cost of the algorithms, whose general shape is very similar for all five matrix sizes. **ODDOML** and our heterogeneous algorithm **Het** have the best *makespans*. At the other end of the spectrum, **OMMOML** is twice as bad. In between, **Hom**, **HomI**, **ORROML**, and **BMM** are roughly twenty percent slower. To give an idea of execution times, **Het** needs about 2000 seconds to compute the product of the smallest matrices, and about 3500 seconds for the largest.

The variations in the cost of **BMM** can easily be justified. The memory layout used in **BMM** is different from the other algorithms. Therefore the size of the matrix chunks used by **BMM** are different. The matrices are rather small (to be able to evaluate a significant number of algorithms on a significant number of platforms). Hence we observe some non negligible side effects (matrix size divided by chunk size not being a multiple of the number of processors used). Therefore, for a given platform, some memory sizes are more favorable for **BMM** than for the algorithms using our memory layout. We will see throughout our experiments on heterogeneous platforms that even if sometimes these side effects help **BMM** achieve reasonable cost, it may also have a very bad cost in some other situations..

The ranking of the algorithms is quite different when we look at the relative work (Figure 4(b)). First we have **OMMOML**, which only uses two workers, and is thus very thrifty, at the expense of its absolute cost. Then we have **HomI**, **Het**, **Hom**, and **ODDOML**. **Hom** relative work is always better than that of **ODDOML**: **Hom** is performing some resource selection contrarily to **ODDOML**, which always uses all the processors. And **HomI** is even better than **Hom**, thanks to a better platform selection. This is also the reason why the gap between the relative work of **Het** and that of **ODDOML** is significantly larger than the gap between their relative costs. Finally, we have **ORROML** and **BMM**, which do not achieve an efficient makespan and make no resource selection, and thus achieve very bad relative work.

Heterogeneous communication links

We now assess the performance of our algorithms when communication links have heterogeneous capabilities. The target platform is composed of two workers with a 10Mbps communication link, four workers with a 5Mbps communication link, and the last two have a 1Mbps communication link.

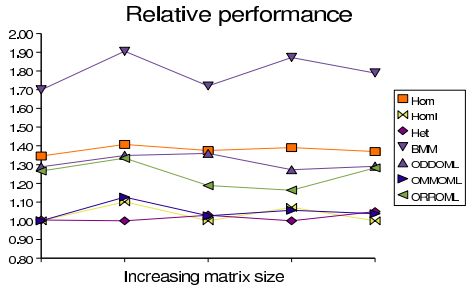
Figure 5(a) shows the relative cost. The superiority of our heterogeneous algorithm over **BMM** is clear.

Het, **HomI**, and **OMMOML** have excellent makespans, and make a good resource selection, as seen on Figure 5(b)). The first figure also shows the gap between **HomI** and **Hom**: **Hom** performs close to **ODDOML**, while **HomI** achieves a close to best makespan. This figure underlines the importance of carefully choosing the processors on which launching the algorithm: **Hom** only uses two processors because of the platform parameters and the way it extracts a homogeneous platform. **BMM** has the worst makespan and makes no resource selection, which explains its worse relative work. **BMM** achieves a makespan that is 70 to 90 percent worse than the best one. Concerning execution times, **Het** needs about 2500 seconds to compute the product of the smallest matrices, and about 5000 seconds for the largest.

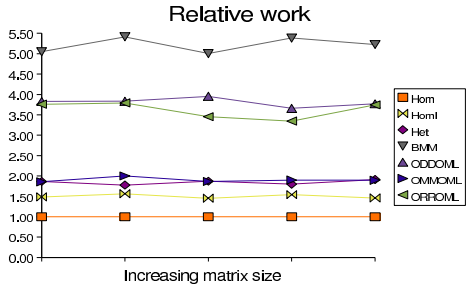
Heterogeneous computations

Here we assess the performance of our algorithms when computation capabilities are heterogeneous. Workers have homogeneous communications and memory capacities, but different computation speeds. The platform is composed of two fast workers of speed S, four workers of speed $S/2$ and two workers of speed $S/4$.

In Figure 6(a), we see the relative cost obtained during this set of experiments. **BMM** performs rather well, but its makespan is

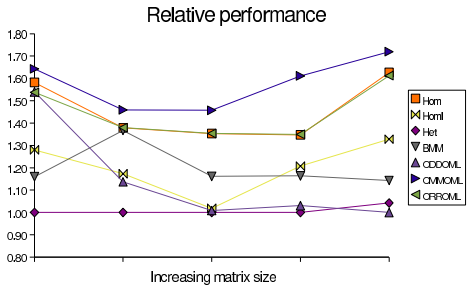


(a) Cost of algorithms.

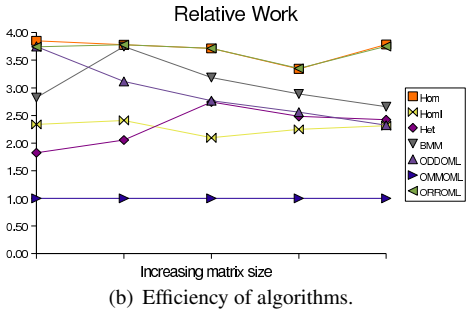


(b) Efficiency of algorithms.

Figure 5. Heterogeneous communication links.



(a) Cost of algorithms.



(b) Efficiency of algorithms.

Figure 6. Heterogeneous computations.

larger than that of **Het**. Moreover, looking at the relative work in Figure 6(b), we see that the gap between the algorithms becomes larger, as our algorithms enroll fewer resources during execution. Among the other algorithms, we see that **ODDOML** performs well. If we look at the relative work, we also see that **Het** uses more and more processors as matrix size increases. During these experiments, **Het** needs about 2000 seconds to compute the product of the smallest matrices, and about 4000 seconds for the largest.

Fully heterogeneous platforms

We now consider fully heterogeneous platforms. Communication links, computation capabilities, and memory capacities can take two different values, which leads to eight possibilities, one per worker. We build that way two different platforms by fixing the ratio between the small and large values for each characteristics to either 2 in the first setting or 4 in the second one (first two columns in Figures 7(a) and 7(b)). In order to show that our heterogeneous algorithm works on any heterogeneous platform, we also randomly create ten different platforms (last ten columns on the same figures). The ratio between minimum and maximum values of communication links, computation capacities, and memory size is up to four. Matrix $\mathcal{A}$ is of size 8000×8000 and $\mathcal{B}$ of size 8000×80000.

The results of these experiments are summarized on Figures 7(a) and 7(b). We see that **Het** achieves the best makespan for all but two of the 12 platforms, and in the remaining cases is no more than 9% and 2% away from the best studied algorithm. All the other algorithms are, at least once, more than 41% away from the best solution. For example, **ORROML** can be up to 88% worse than the best achieved makespan. Only **ODDOML** achieves reasonable makespans on average but, as it does not select resources, its relative work is far worse. The relative work of **Het** is the best among all algorithms except **HomI**, and the unusable **OMMOML**, whose makespan can be 215% away from the best solution. But even if our improved homogeneous algorithm performs a good resource selection, the makespan it achieves can be up to 80% larger than the best makespan, and is 34% larger on average. According to the experiments, **Het** needs between 2700 seconds and 6000 seconds.

Real platform

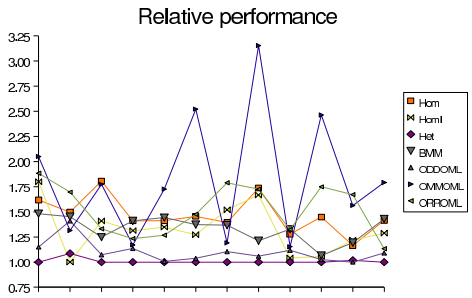
In this set of experiments, we use almost all the processors of our platform. We do not modify the communication speed nor the computation speed of the workers. We take five processors of each of the four sets of machines, which gives us a rather homogeneous platform. We either use this platform "as is" (*August 2007 configuration* of Figure 8(a)) or we limit the amount of memory available on each processor to its value before the last memory upgrade (*November 2006 configuration* of Figure 8(b)). The actual platform was then:

- 5 SuperMicro servers 5013-GM, with processors P4 2.4 GHz with 256 MB of memory;
- 5 SuperMicro servers 6013PI, with processors P4 Xeon 2.4 GHz with 1 GB of memory;
- 5 SuperMicro servers 5013SI, with processors P4 Xeon 2.6 GHz with 1 GB of memory;
- 5 SuperMicro servers IDE250W, with processors P4 2.8 GHz with 256 MB of memory.

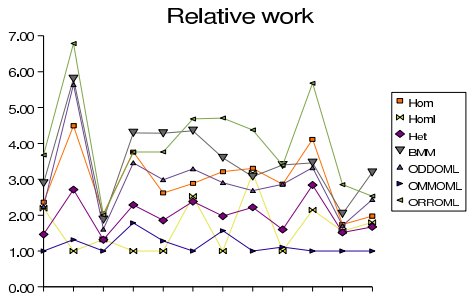
We use an extra processor as the master. The matrix is of size 8000×8000 for $\mathcal{A}$ and 8000×320000 for $\mathcal{B}$.

The results of these experiments are summarized in Figure 8. The results on the actual platform are similar to those obtained on homogeneous platforms [14]. All the algorithms but **BMM** have similar makespan. All algorithms making resource selection use eleven workers among the twenty available, which explains why they achieve similar relative work.

The results of the experiments on the older version of the platform are very similar to the ones previously obtained on memory heterogeneous platforms. **ODDOML** and our heterogeneous algorithm **Het** achieve the best *makespans*. Then we have **ORROML**, our homogeneous algorithms, **BMM**, and finally **OMMOML** which is 60% worse than **Het**. The execution time is around 7800 seconds for **Het**. If we look at resource selection,

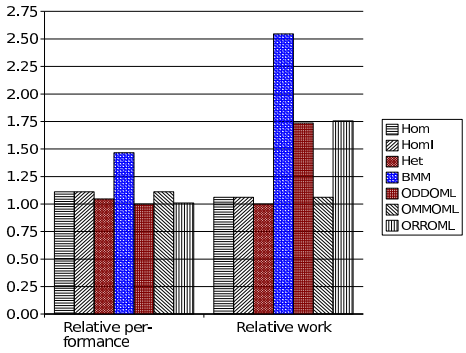


(a) Cost of algorithms.

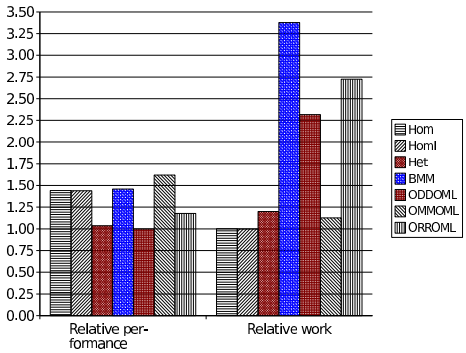


(b) Efficiency of algorithms.

Figure 7. Fully heterogeneous platforms.



(a) August 2007 configuration



(b) November 2006 configuration

Figure 8. Real platform.

Het uses only the ten workers which have 1 GB of memory, and achieves a makespan close to **ODDOML**'s, which uses the whole platform. On another side, **Hom**, **HomI**, **OMMOML** use six workers with small memories. All other algorithms use the whole platform. We can thus see the impact of the resource selection on the relative work of the algorithms.

Summary

Figure 9 summarizes all our MPI experiments. Figures 9(a) and 9(b) respectively present the relative cost and the relative work obtained for each experiment by our heterogeneous algorithm (**Het**), by Toledo's algorithm (**BMM**), and by the best of the dynamic heuristics using our memory layout (**ODDOML**). The results show the superiority of our memory allocation. Furthermore, if we add the resource selection of **Het**, not only do we achieve, most of the time, the best makespan, but also the best relative work as we also spare resources. Using our memory layout (**ODDOML**) rather than Toledo's (**BMM**) enables us to gain 19% of execution time on average. When this is combined with resource selection, this enables us to gain additionally 10%, which is 27% against Toledo's running time. We achieve this significant gain while sparing resources. Our **Het** algorithm is on average 1% away from the best achieved makespan. At worst **Het** is 14% away from the best makespan, **ODDOML** 61%, and **BMM** 128%. Moreover, we have seen that 80% of the time, the performance of **Het** was in fact obtained thanks to a global resource selection.

The steady-state approach described in Section 5 gives us an upper-bound on the best achievable throughput. This upper-bound is very optimistic as it assumes unbounded memories and does not take into account the communication costs due to the elements of matrix $\mathcal{C}$. This upper bound is nevertheless on average only 2.29 times greater than the throughput achieved by **Het** (and at worst is 3.42 times greater). Therefore, considering this upper-bound tells us that our **Het** algorithm not only has good relative cost when compared to the other algorithms, but also has very good absolute cost.

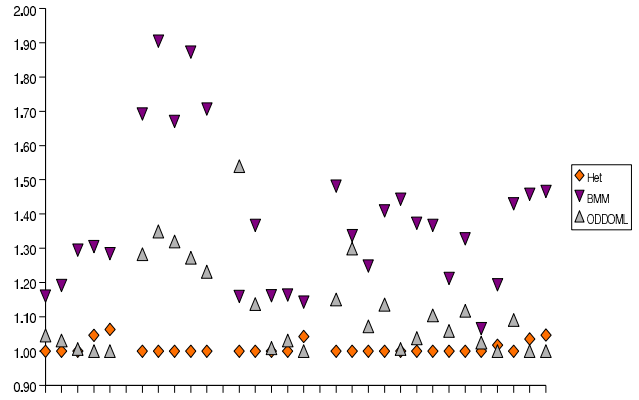
Altogether, we have thus been able to design an efficient, thrifty, and reliable algorithm.

7. Related work

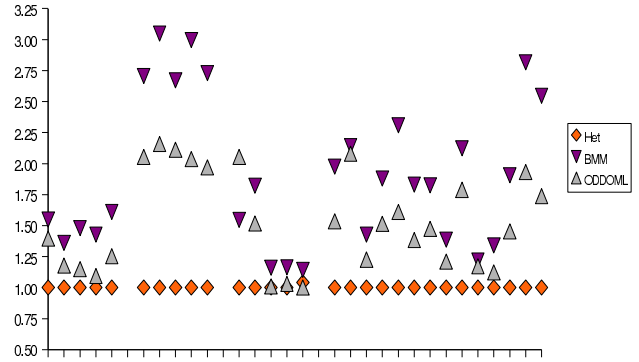
As already mentioned, the design of parallel algorithms for limited memory processors is very similar to the design of out-of-core routines for classical parallel machines. On the theoretical side, Hong and Kung [9] investigate the I/O complexity of several computational kernels in their pioneering paper. Toledo [17] proposes a nice survey on the design of out-of-core algorithms for linear algebra, including dense and sparse computations. We refer to [17] for a complete list of implementations. The design principles followed by most implementations are introduced and analyzed by Dongarra et al. [8].

A similar thread of work, although in a different context, deals with reconfigurable architectures, either pipelined bus systems [12], or FPGAs [18]. In the latter approach, tradeoffs must be found to optimize the size of the on-chip memory and the available memory bandwidth, leading to partitioned algorithms that re-use data intensively.

Please see the companion research report of this paper [14] for an overview or relevant literature dealing with (i) load balancing techniques on heterogeneous platforms; (ii) linear algebra algorithms on heterogeneous clusters; (iii) models for heterogeneous platforms; and (iv) master-worker scheduling on grid platforms. This research report also subsumes our previous work focusing on homogeneous platforms [15].



(a) Relative Cost.



(b) Relative work.

Figure 9. Summary of experiments.

8. 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. Please refer to the companion research report [14] for a discussion on how to adapt the approach for LU factorization.
3. Through MPI experiments, we have shown that our algorithm for heterogeneous platforms has far better performance than solutions using the memory layout proposed in [17]. Furthermore, this static heterogeneous algorithm has slightly better performance than dynamic algorithms using the same memory layout, but uses fewer processors, and has a far better worst case. It is therefore a very good candidate for deploying applications on heterogeneous platforms.

It would be interesting to assess whether our memory layout could prove useful in the context of out-of-core algorithms for heterogeneous platforms.

References

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

[2] 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.

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

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

[5] 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.

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

[7] R. Clint Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1-2):3–35, Jan. 2001.

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

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

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

[11] A. Kalinov and A. Lastovetsky. Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers. *J. Par. Distr. Computing*, 61(4):520–535, 2001.

[12] K. Li and V. Y. Pan. Parallel matrix multiplication on a linear array with a reconfigurable pipelined bus system. *IEEE Trans. Computers*, 50(5):519–525, 2001.

[13] 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 *Eighth Heterogeneous Computing Workshop*, pages 30–44. IEEE CS Press, 1999.

[14] J.-F. Pineau, Y. Robert, F. Vivien, Z. Shi, and J. Dongarra. Revisiting matrix product on master-worker platforms. Research Report 2006-39, LIP, ENS Lyon, France, Nov. 2006.

[15] J.-F. Pineau, Y. Robert, F. Vivien, Z. Shi, and J. Dongarra. Revisiting matrix product on master-worker platforms. *IEEE Advances in Parallel and Distributed Computational Models*, 2007.

[16] 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, 2004.

[17] 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.

[18] L. Zhuo and V. K. Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems. *IEEE TPDS*, 18(4):433–448, 2007.