Revisiting matrix product on master-worker platforms

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

To cite this version:

Jack Dongarra, Jean-François Pineau, Yves Robert, Zhiao Shi, Frédéric Vivien. Revisiting matrix product on master-worker platforms. 9th Workshop on Advances in Parallel and Distributed Computational Models APDCM 2007, 2007, Unknown, United States. 10.1142/S0129054108006303 . hal-00803519

HAL Id: hal-00803519

https://inria.hal.science/hal-00803519v1

Submitted on 20 Nov 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Revisiting Matrix Product on Master-Worker Platforms

Jack Dongarra[2], Jean-François Pineau[1], Yves Robert[1], Zhiao Shi[2], and Frédéric Vivien[1]

[1]LIP, CNRS-ENS Lyon-INRIA-UCBL
Université de Lyon
École normale supérieure de Lyon, France

[2]Innovative Computing Laboratory
Department of Computer Science
University of Tennessee, Knoxville, USA

{jean-francois.pineau,yves.robert,frederic.vivien}@ens-lyon.fr {dongarra, shi}@cs.utk.edu

Abstract

This paper is aimed at designing efficient parallel matrix-product algorithms for homogeneous 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 two 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).*
- 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 of buffers, where a buffer can store a square block of matrix elements. These square blocks are chosen so as to harness the power of Level 3 BLAS routines; they are of size* 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 MPI experiments conducted on a platform at the University of Tennessee.

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 [4] and the ScaLAPACK outer product algorithm [3]. 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 clusters, which are composed of 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 a same physical link happens to belong to the two 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 networks of workstations.

In this paper, we do not try to adapt the 2D processor grid strategy to networks of workstations. 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).

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 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 limited memory capacities. The first problem is *resource selection*. How many workers should be enrolled in the execution? All of them, or maybe only a fraction? Once participating resources have been selected, there remain several scheduling decisions to take: 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 goal of this paper is to design efficient algorithms for resource selection and communication ordering. In addition, we report MPI experiments on platforms 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. 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 [8, 6]. In Section 4, we propose a scheduling algorithm that includes resource selection. We report several MPI experiments in Section 5. Finally, we state some concluding remarks in Section 6. Due to lack of space, related work is not discussed in this paper: please refer to the extended version [5] for an overview of relevant literature.

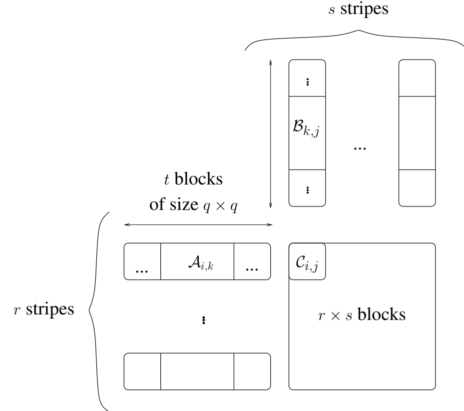

Figure 1. Partition of the three matrices $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$.

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. 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 [3]. Typically, $q = 80$ or 100 when using ATLAS-generated routines [9].
- 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.

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 identical workers P_i, $1 \leq i \leq p$. Because we manipulate large data blocks, we adopt a lin-

ear cost model, both for computations and communications (i.e., we neglect start-up overheads). We have the following notations:

- It takes $X.w$ time-units to execute a task of size X on P_i;
- It takes $X.c$ 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.

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 [1, 2], which is defined as follows: (i) the master can only send data to, and receive data from, a single worker at a given time-step; (ii) 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, 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 actually 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 [2] 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 [7], 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 5 obey the one-port model.

Our final assumption is related to memory capacity; we assume that a worker P_i can only store m 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. Since we aim at 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 [8, 6].

3.1 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 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: 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.

3.2 Performance and lower bound

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

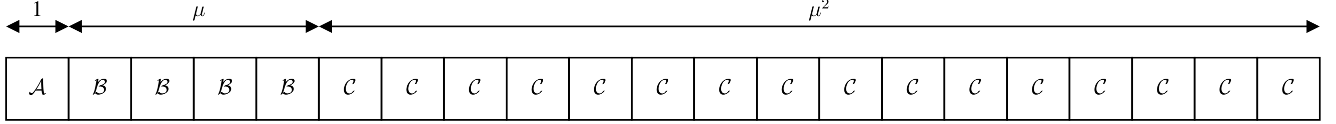

Figure 2. Memory usage for the *maximum re-use* **algorithm when $m = 21$: $\mu = 4$; 1 block is used for $\mathcal{A}$, μ for $\mathcal{B}$, and μ^2 for $\mathcal{C}$.**

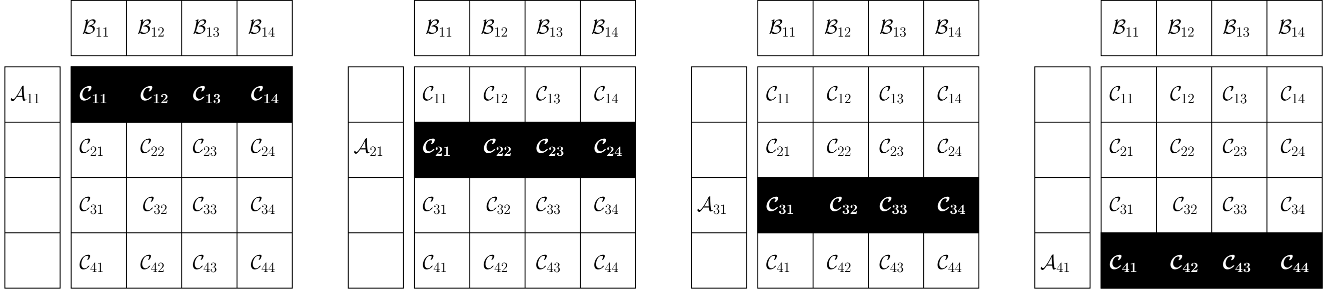

Figure 3. Four steps of the *maximum re-use* **algorithm, with $m = 21$ and $\mu = 4$. The elements of $\mathcal{C}$ updated are displayed on white on black.**

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 $\mathcal{B}$, and 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 [8]. 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 [8]: consider any algorithm that uses the standard way of multiplying matrices (this excludes Strassen's or Winograd's algorithm, 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

$$K = \min\left\{(N_A + N_B)\sqrt{N_C}, (N_A + N_C)\sqrt{N_B}, (N_B + N_C)\sqrt{N_A}\right\}. \quad (1)$$

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 $\alpha_{old} + \alpha_{recv} = \alpha m$, $\beta_{old} + \beta_{recv} = \beta m$, and $\gamma_{old} + \gamma_{recv} = \gamma m$. 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 $\alpha = \beta = \gamma = \frac{2}{3}$, 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 [8], we use Loomis-Whitney inequality [6]: 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 $\alpha = \beta = \gamma = \frac{2}{3}$, 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 [6]. 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 [8].

4 Algorithms for homogeneous platforms

In this section, we adapt the *maximum re-use* algorithm to fully homogeneous platforms. 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 [4] and the ScaLAPACK outer product algorithm [3] 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

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

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

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. Here we 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 gets much more complicated.

5 MPI experiments

In this section, we aim at validating the previous theoretical results and algorithms. We conduct a variety of MPI

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 μ^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};

experiments to compare our new schemes with several other algorithms from the literature.

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 running the Linux operating system. Each node has four Gigabytes of memory, but we only use 512 MB of memory to further stress the impact of limited memories. The nodes are connected with a switched 100Mbps Fast Ethernet network.

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;

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.

Algorithms

We choose and adapt four 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 [8]. 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

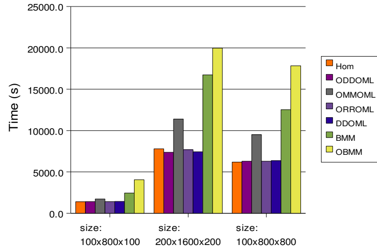

Figure 4. Performance of the algorithms on different matrices.

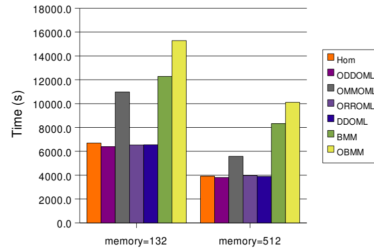

Figure 5. Impact of memory size on algorithm performance.

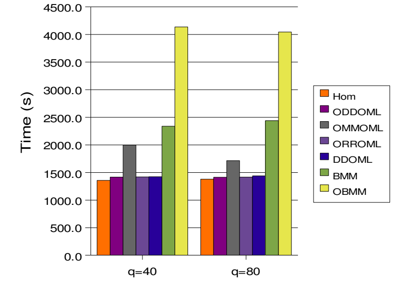

Figure 6. Impact of block size q on algorithm performance.

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 $\mathcal{C}$.

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, 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 and 80. As the global matrix size is the same in both experiments, we expect both 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 132MB to 512MB. 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.

Results and discussion

We see in Figure 4 the results of the first set of experiments, where algorithms are computing different matrices. The first remark is that the shape of the two 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 6, 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 5 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 7 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.

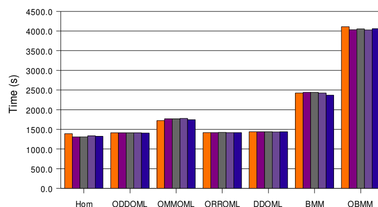

Figure 7. Variation of algorithm execution times.

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.

6 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.

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 [8]. 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.

Future work is devoted to extending the memory management strategy and the corresponding algorithms to heterogeneous platforms. Preliminary results are available in [5].

References

[1] 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.

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

[3] 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.

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

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

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

[7] 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.

[8] 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.

[9] 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.