International Journal of Foundations of Computer Science

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
{*jean-francois.pineau,yves.robert,frederic.vivien*}*@ens-lyon.fr*

[2] *Innovative Computing Laboratory*
Department of Computer Science
University of Tennessee, Knoxville, USA
{*dongarra, shi*}*@cs.utk.edu*

Received
Revised
Communicated by

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*, 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. 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. 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 [10] and the ScaLAPACK outer product algorithm [9]. 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: we want to achieve the smallest possible execution time while using as few processors as possible. 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. In Section 3, we start with a theoretical study of the simplest version of the problem, without memory limitation, which is intended to show the intrinsic difficulty of the scheduling problem. Next, in Section 4, we proceed with the analysis of the total communication volume that is needed in the presence of memory constraints, and we improve a well-known bound by Toledo [34, 22]. In Section 5, we propose a scheduling algorithm that includes resource selection. We report several MPI experiments in Section 6. Section 7 is dedicated to related work. Finally, we state some concluding remarks in Section 8.

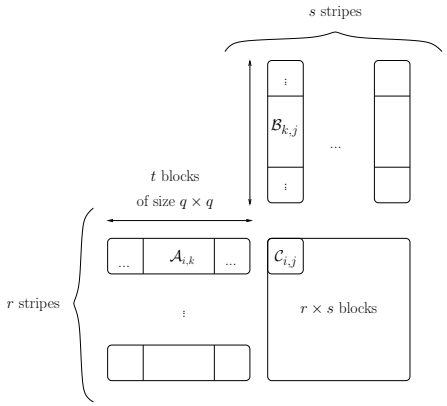

Figure 1: Partition of the three matrices $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$.

2. Framework

2.1. Application

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

- We use a block-oriented approach. The atomic elements that we manipulate are not matrix coefficients but instead square *blocks* of size $q \times q$ (hence with q^2 coefficients). This is to harness the power of Level 3 BLAS routines [9]. Typically, $q = 80$ or 100 when using ATLAS-generated routines [36].
- The input matrix $\mathcal{A}$ is of size $n_{\mathcal{A}} \times n_{\mathcal{AB}}$:
  - we split $\mathcal{A}$ into r horizontal stripes $\mathcal{A}_i$, $1 \leq i \leq r$, where $r = n_{\mathcal{A}}/q$;
  - we split each stripe $\mathcal{A}_i$ into t square $q \times q$ blocks $\mathcal{A}_{i,k}$, $1 \leq k \leq t$, where $t = n_{\mathcal{AB}}/q$.

- The input matrix $\mathcal{B}$ is of size $n_{\mathcal{AB}} \times n_{\mathcal{B}}$:
  - we split $\mathcal{B}$ into s vertical stripes $\mathcal{B}_j$, $1 \leq j \leq s$, where $s = n_{\mathcal{B}}/q$;
  - we split stripe $\mathcal{B}_j$ into t square $q \times q$ blocks $\mathcal{B}_{k,j}$, $1 \leq k \leq t$.
- We compute $\mathcal{C} = \mathcal{C} + \mathcal{A} \times \mathcal{B}$. Matrix $\mathcal{C}$ is accessed (both for input and output) by square $q \times q$ blocks $\mathcal{C}_{i,j}$, $1 \leq i \leq r$, $1 \leq j \leq s$. There are $r \times s$ such blocks.

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

2.2. Platform

We target a *star network* $\mathcal{S} = \{P_0, P_1, P_2, \ldots, P_p\}$, composed of a master P_0 and of p identical workers P_i, $1 \leq i \leq p$. 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 $X.w$ 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 [7, 8], 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 [8] 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 [31], who report that asynchronous MPI sends get serialized as soon as message sizes exceed a hundred kilobytes. Their result holds 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 blocks (either from $\mathcal{A}$, $\mathcal{B}$, or $\mathcal{C}$). For large problems, this memory limitation will considerably impact the design of the algorithms, as data re-use will be greatly dependent on the amount of available buffers.

3. Combinatorial complexity of a simple version of the problem

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

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

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

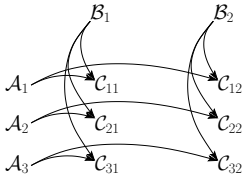

Figure 2: Dependence graph of the problem (with $r = 3$ and $s = 2$).

The dependence graph of the problem is depicted in Figure 2. It suggests a natural strategy for enabling workers to start computing as soon as possible. Indeed, the master should alternate sending $\mathcal{A}$-blocks and $\mathcal{B}$-blocks. Of course it must be decided how many workers to enroll and in which order to send the blocks to the enrolled workers. But with a single worker, we can show that the *alternating greedy* algorithm is optimal:

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

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

Unfortunately, for more than one worker, we did not succeed in determining an optimal algorithm. The difficulty lies in the selection of the next worker to

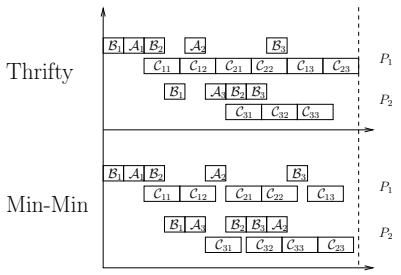

Figure 3: Example showing that Thrifty is not optimal: with $p = 2$, $c = 4$, $w = 7$, and $r = s = 3$, Min-min has a lower makespan. For each algorithm, the first line presents the communications for P_1, the second the computations of P_1, the third the communications for P_2, and the fourth the computations of P_2.

enroll. One can see this on the following example. There are (at least) two greedy algorithms that can be devised for p workers:

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

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

Min-min: This algorithm is based on the well-known min-min heuristic [29]. At each step, all tasks are considered. For each of them, we compute their possible starting date on each worker, given the files that have already been sent to this worker and all decisions taken previously; we select the best worker, hence the first *min* in the heuristic. We take the minimum of starting dates over all tasks, hence the second *min*.

It turns out that neither greedy algorithm is optimal. See Figure 3 for an example where Min-min is better than Thrifty, and Figure 4 for an example of the opposite situation.

We now go back to our original model.

4. Minimization of the communication volume

In this section, we derive a lower bound on the total number of communications (sent from, or received by, the master) that are needed to execute any matrix multiplication algorithm satisfying our hypotheses (centralized data and limited memory). 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:

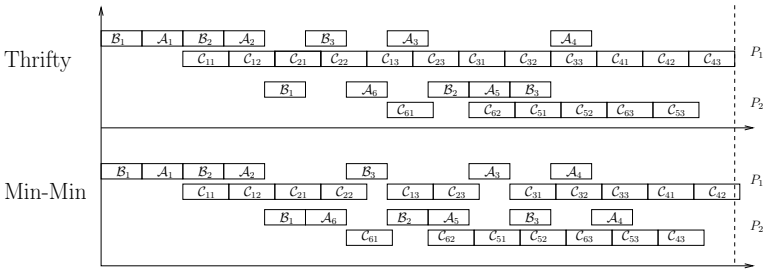

Figure 4: Example showing that Min-min is not optimal: with $p = 2$, $c = 8$, $w = 9$, $r = 6$, and $s = 3$, Thrifty has a lower makespan.

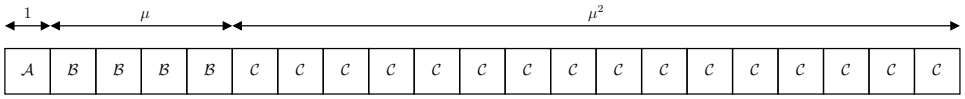

Figure 5: Memory usage for the *maximum re-use* algorithm when $m = 21$: $\mu = 4$; 1 block is used for $\mathcal{A}$, μ for $\mathcal{B}$, and μ^2 for $\mathcal{C}$.

- 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 [34, 22].

4.1. The maximum re-use *algorithm*

Below we introduce and analyze the performance of the *maximum re-use* algorithm, whose memory management is illustrated in Figure 5. Four consecutive execution steps are shown in Figure 6. Assume that there are m available buffers. First we find μ as the largest integer such that $1 + \mu + \mu^2 \leq m$. The idea is to use one buffer to store $\mathcal{A}$ blocks, μ buffers to store $\mathcal{B}$ blocks, and μ^2 buffers to store $\mathcal{C}$

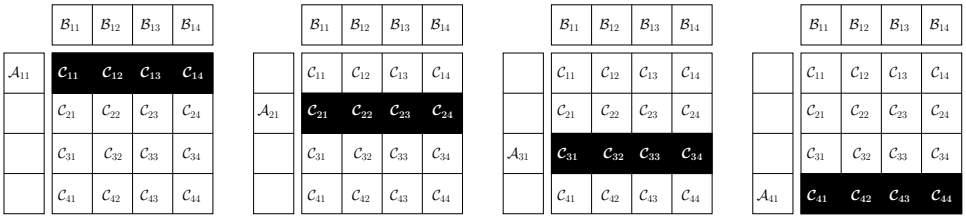

Figure 6: Four steps of the *maximum re-use* algorithm, with $m = 21$ and $\mu = 4$. The elements of $\mathcal{C}$ updated are displayed in white on black.

blocks. In the outer loop of the algorithm, a $\mu \times \mu$ square of $\mathcal{C}$ blocks is loaded. Once these μ^2 blocks have been loaded, they are repeatedly updated in the inner loop of the algorithm until their final value is computed. Then the blocks are returned to the master, and μ^2 new $\mathcal{C}$ blocks are sent by the master and stored by the worker. As illustrated in Figure 5, we need μ buffers to store a row of $\mathcal{B}$ blocks, but only one buffer for $\mathcal{A}$ blocks: $\mathcal{A}$ blocks are sent in sequence, each of them is used in combination with a row of μ $\mathcal{B}$ blocks to update the corresponding row of $\mathcal{C}$ blocks. This leads to the following sketch of the algorithm:

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

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

4.2. Performance and lower bound

The performance of one iteration of the outer loop of the *maximum re-use* algorithm can readily be determined. We need $2\mu^2$ communications to send and retrieve $\mathcal{C}$ blocks. For each value of t, we need μ elements of $\mathcal{A}$ and μ elements of $\mathcal{B}$, 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 [34]. 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 [34]: consider any algorithm that uses the standard way of multiplying matrices (this excludes Strassen's and Winograd's algorithms, for instance). If N_A elements of $\mathcal{A}$, N_B elements of $\mathcal{B}$ and N_C elements of $\mathcal{C}$ are accessed, then no more than K computations can be done, where

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

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

$$\text{MAXIMIZE } k \text{ S.T.}$$
$$\begin{cases} k \leq (\alpha + \beta)\sqrt{\gamma} \\ k \leq (\beta + \gamma)\sqrt{\alpha} \\ k \leq (\gamma + \alpha)\sqrt{\beta} \\ \alpha + \beta + \gamma \leq 2 \end{cases}$$

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 [34], we use Loomis-Whitney inequality [22]: 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 [22]. 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 [34].

5. 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 [10] and the ScaLAPACK outer product algorithm [9] 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.

5.1. 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 $w = q^3\tau_a$, where τ_c and τ_a respectively represent the speed of the communication link and the speed of the processor. 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.

5.2. Impact of the start-up overhead

Algorithm 1: 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 $q \times q$):
$\mathbf{C}_{i',j'} = \{\mathcal{C}_{i,j} \mid (i'-1)\mu + 1 \leq i \leq i'\mu, (j'-1)\mu + 1 \leq j \leq j'\mu\}$;
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};

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 technically be 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 get much more complicated.

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 other algorithms from the literature.

6.1. Platform

For our experiments we are using a platform at the University of Tennessee. All experiments are performed on a cluster of 64 Xeon 3.2GHz dual-processor nodes running the Linux operating system. Each node has four Gigabytes of memory, but

Algorithm 2: Worker program.

```
for all blocks do
    Receive C_{i',j'} from master;
    for k ← 1 to t do
        for j ← (j' − 1)μ + 1 to j'μ do Receive B_{k,j};
        for i ← (i' − 1)μ + 1 to i'μ do
            Receive A_{i,k};
            for j ← (j' − 1)μ + 1 to j'μ do
                C_{i,j} ← C_{i,j} + A_{i,k}.B_{k,j};
    Return C_{i',j'} to master;
```

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. In order to build a master-worker platform, we arbitrarily choose one processor as the master, and the other processors will serve as workers. Finally we used *MPI_WTime* as timer in all experiments.

6.2. Algorithms

We choose four different algorithms from the general literature, and adapt them into six algorithms designed for matrix multiplication. The objective is to use these six algorithms as the basis for comparison with our new algorithm. The four original algorithms are *Round-Robin*, *Minimum Completion Time* (or *Min-Min*) and *Demand-Driven*, which are well known algorithms in the scheduling literature, and Toledo's algorithm [34], which is a dedicated algorithm for matrix multiplication.

As the first three chosen algorithms are designed for job distribution, and not for matrix multiplication, we adapt them in the following way: they all use our memory layout to divide matrices into chunks and to determine in which order chunks have to be sent to participating workers. The only difference between these algorithms and ours is the order in which the master serves workers.

All the algorithms used during the MPI experiments are now described.

First, our algorithm:

- *Homogeneous algorithm*: **HoLM** is our homogeneous algorithm, described in section 5. It makes resource selection, and sends blocks to the selected workers in a round-robin fashion.

The other four algorithms using our memory layout:

- *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 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 to send blocks to the others, **HoLM** and **OMMOML** should have similar behavior on homogeneous platforms.

- *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 compare to **ODDOML**.

Finally, the two adapted algorithms which do not use our memory allocation:

- *Block Matrix Multiply*: **BMM** is Toledo's algorithm. 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 $\mathcal{C}$, 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 does 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 $\mathcal{C}$.

6.3. Experiments

We have built several experimental protocols in order to assess the performance of the various algorithms. In the following experiments we use nine processors, one master and eight workers. We restricted the number of workers to eight after an initial experiment using the whole platform. This experiment showed that, because of platform parameters and of memory limitations, if the master serves five processors or more, on average one processor is idle at any time. 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 $\mathcal{B}$,

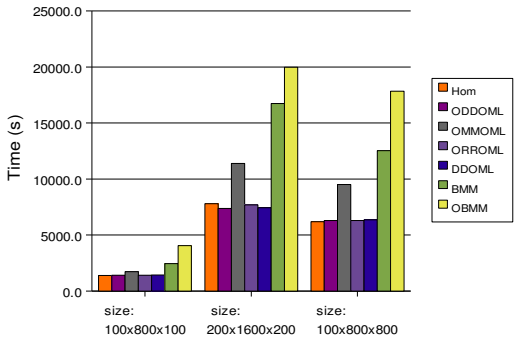

Figure 7: Performance of the algorithms on different matrices.

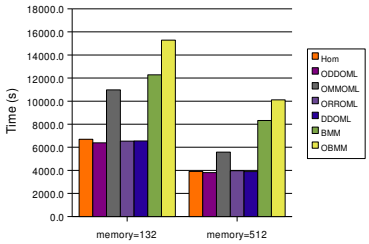

Figure 8: Impact of memory size on algorithm performance.

- 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.

6.4. Results and discussion

We see in Figure 7 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 only uses three workers. For instance, **HoLM** uses four workers, and is as competitive as the other algorithms which all

use the eight available workers. This difference can be explained by the resource selection process. Before computing the schedules, we measure the communication and computation times by sending the same task to all slaves. The different measures obtained are not absolutely identical, for instance because of experimental measurement errors. **OMMOML** uses all the exact measures. As **HoLM** has to consider an absolutely homogeneous platform, it works on a more pessimistic platform using the maximum of the computation times, and the maximum of the communication times.

In Figure 8 we have the impact of the worker memory size on the performance of the algorithms. As expected, the performance increases (as the execution time needed to perform the multiplication decreases) 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.

In Figure 9, we see the impact of q on the performance of our algorithms. **BMM** and **OBMM** have constant execution times in the 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 execution 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.

Finally, figure 10 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.

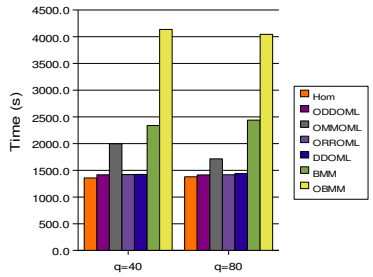

Figure 9: Impact of block size q on algorithm performance.

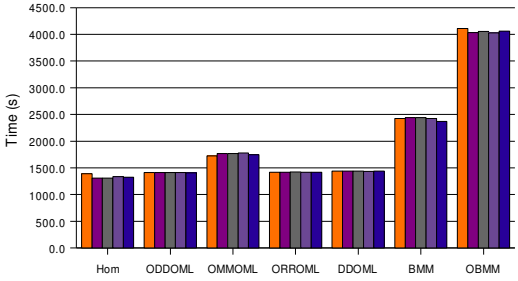

Figure 10: Variation of algorithm execution times.

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.

7. Related work

In this section, we provide a brief overview of the related work. Even if, in this article, we only focused on homogeneous platforms, we consider the related work in the more general scope of heterogeneous platforms. We classify the related papers along the following six main lines:

Load balancing on heterogeneous platforms – Load balancing strategies for heterogeneous platforms have been widely studied. Distributing the computations (together with the associated data) can be performed either dynamically or statically, or a mixture of both. Some simple schedulers are available, but they use naive mapping strategies such as master-worker techniques or paradigms based upon the idea *"use the past to predict the future"*, i.e. use the currently observed speed of computation of each machine to decide for the next distribution of work [13, 14, 6]. Dynamic strategies such as *self-guided scheduling* [30] could be useful too. There is a challenge in determining a trade-off between the data distribution parameters and the process spawning and possible migration policies. Redundant computations might also be necessary to use a heterogeneous cluster at its best capabilities. However, dynamic strategies are outside the scope of this paper (but mentioned here for the sake of completeness). Because we have a library designer's perspective, we concentrate on static allocation schemes that are less general and more difficult to design than dynamic approaches, but which are better suited for the implementation of fixed algorithms such as linear algebra kernels from the ScaLAPACK library [9].

Out-of-core linear algebra routines – As already mentioned, the design of parallel algorithms for limited memory processors is very similar to the design of out-of-core routines for classical parallel machines. On the theoretical side, Hong and Kung [21] investigate the I/O complexity of several computational kernels in their pioneering paper. Toledo [34] proposes a nice survey on the design of out-of-core algorithms for linear algebra, including dense and sparse computations. We refer to [34] for a complete list of implementations. The design principles followed by most implementations are introduced and analyzed by Dongarra et al. [17].

Matrix product on reconfigurable architectures– A similar thread of work, although in a different context, deals with reconfigurable architectures, either pipelined bus systems [27], or FPGAs [37]. 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.

Linear algebra algorithms on heterogeneous clusters – Several authors have dealt with the *static* implementation of matrix-multiplication algorithms on heterogeneous platforms. One simple approach is given by Kalinov and Lastovetsky [24]. Their idea is to achieve a perfect load-balance as follows: first

they take a fixed layout of processors arranged as a collection of processor columns; then the load is evenly balanced *within* each processor column independently; next the load is balanced *between* columns; this is the "heterogeneous block cyclic distribution" of [24]. Another approach is proposed by Crandall and Quinn [15], who propose a recursive partitioning algorithm, and by Kaddoura, Ranka, and Wang [23], who refine the latter algorithm and provide several variations. They report several numerical simulations. As pointed out in the introduction, theoretical results for matrix multiplication and LU decomposition on 2D-grids of heterogeneous processors are reported in [4], while extensions to general 2D partitioning are considered in [5]. See also Lastovetsky and Reddy [26] for another partitioning approach.

Recent papers aim at making easier the process of tuning linear algebra kernels on heterogeneous systems. Self-optimization methodologies are described by Cuenca et al. [16] and by Chen et al. [12]. Along the same line, Chakravarti et al. [11] describe an implementation of Cannon's algorithm using self-organizing agents on a peer-to-peer network.

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

The bidirectional one-port model is used by Bhat et al. [7, 8] for fixed-size messages. They advocate its use because "current hardware and software do not easily enable multiple messages to be transmitted simultaneously." Even if non-blocking, multi-threaded communication libraries allow for initiating multiple send and receive operations, they claim that all these operations "are eventually serialized by the single hardware port to the network". Experimental evidence of this fact has recently been reported by Saif and Parashar [31], who report that asynchronous MPI sends get serialized as soon as message sizes exceed a few megabytes. Their results hold for two popular MPI implementations, MPICH on Linux clusters and IBM MPI on the SP2.

The one-port model fully accounts for the heterogeneity of the platform, as each link has a different bandwidth. It generalizes a simpler model studied by Banikazemi et al. [1], Liu [28], and Khuller and Kim [25]. In this simpler model, the communication time only depends on the sender, not on the receiver. In other words, the communication speed from a processor to all its neighbors is the same.

Finally, we note that some papers [2, 3] depart form the one-port model as they allow a sending processor to initiate another communication while a previous one is still on-going on the network. However, such models insist that there is

an overhead time to pay before being engaged in another operation, so they are not allowing for fully simultaneous communications.

Master-worker on the computational grid – Master-worker scheduling on the grid can be based on a network-flow approach [33, 32] or on an adaptive strategy [19]. Note that the network-flow approach of [33, 32] is possible only when using a full multiple-port model, where the number of simultaneous communications for a given node is not bounded. This approach has also been studied in [20]. Enabling frameworks to facilitate the implementation of master-worker tasking are described in [18, 35].

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., a way 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 theoretical 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 [34]. 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.

Acknowledgments. We thank the reviewers for their numerous comments and suggestions, which greatly improved the final version of the paper.

References

1. M. Banikazemi, V. Moorthy, and D. K. Panda. Efficient collective communication on heterogeneous networks of workstations. In *Proceedings of the 27th International Conference on Parallel Processing (ICPP'98)*. IEEE Computer Society Press, 1998.
2. M. Banikazemi, J. Sampathkumar, S. Prabhu, D.K. Panda, and P. Sadayappan. Communication modeling of heterogeneous networks of workstations for performance characterization of collective operations. In *HCW'99, the 8th Heterogeneous Computing Workshop*, pages 125–133. IEEE Computer Society Press, 1999.
3. Amotz Bar-Noy, Sudipto Guha, Joseph (Seffi) Naor, and Baruch Schieber. Message multicasting in heterogeneous networks. *SIAM Journal on Computing*, 30(2):347–358, 2000.

4. O. Beaumont, V. Boudet, A. Petitet, F. Rastello, and Y. Robert. A proposal for a heterogeneous cluster ScaLAPACK (dense linear solvers). *IEEE Transactions on Computers*, 50(10):1052–1070, 2001.
5. O. Beaumont, V. Boudet, F. Rastello, and Y. Robert. Matrix multiplication on heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1033–1051, 2001.
6. F. Berman. High-performance schedulers. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, pages 279–309. Morgan-Kaufmann, 1999.
7. P.B. Bhat, C.S. Raghavendra, and V.K. 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.
8. P.B. Bhat, C.S. Raghavendra, and V.K. Prasanna. Efficient collective communication in distributed heterogeneous systems. *Journal of Parallel and Distributed Computing*, 63:251–263, 2003.
9. 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.
10. L. E. Cannon. *A cellular computer to implement the Kalman filter algorithm*. PhD thesis, Montana State University, 1969.
11. A.J. Chakravarti, G. Baumgartner, and M. Lauria. Self-organizing scheduling on the organic grid. *International Journal of High Performance Computing Applications*, 20(1):115–130, 2006.
12. Z. Chen, J. Dongarra, P. Luszczek, and K. Roche. Self adapting software for numerical linear algebra and lapack for clusters. *Parallel Computing*, 29(11-12):1723–1743, 2003.
13. M. Cierniak, M.J. Zaki, and W. Li. Compile-time scheduling algorithms for heterogeneous network of workstations. *The Computer Journal*, 40(6):356–372, 1997.
14. M. Cierniak, M.J. Zaki, and W. Li. Customized dynamic load balancing for a network of workstations. *Journal of Parallel and Distributed Computing*, 43:156–162, 1997.
15. P. E. Crandall and M. J. Quinn. Block data decomposition for data-parallel programming on a heterogeneous workstation network. In *2nd International Symposium on High Performance Distributed Computing*, pages 42–49. IEEE Computer Society Press, 1993.
16. Javier Cuenca, Luis Pedro Garcia, Domingo Gimenez, and Jack Dongarra. Processes distribution of homogeneous parallel linear algebra routines on heterogeneous clusters. In *HeteroPar'2005: International Conference on Heterogeneous Computing*. IEEE Computer Society Press, 2005.
17. Jack Dongarra, Swen Hammarling, and David Walker. Key concepts for parallel out-of-core LU factorization. *Parallel Computing*, 23(1-2):49–70, 1997.
18. J. P Goux, S. Kulkarni, J. Linderoth, and M. Yoder. An enabling framework for master-worker applications on the computational grid. In *Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*. IEEE Computer Society Press, 2000.
19. E. Heymann, M. A. Senar, E. Luque, and M. Livny. Adaptive scheduling for master-worker applications on the computational grid. In R. Buyya and M. Baker, editors,

Grid Computing - GRID 2000, pages 214–227. Springer-Verlag LNCS 1971, 2000.

20. B. Hong and V.K. Prasanna. Bandwidth-aware resource allocation for heterogeneous computing systems to maximize throughput. In *Proceedings of the 32th International Conference on Parallel Processing (ICPP'2003)*. IEEE Computer Society Press, 2003.
21. J.-W. Hong and H.T. Kung. I/O complexity: the red-blue pebble game. In *Proceedings of STOC'81*, pages 326–333. ACM Press, 1981.
22. Dror Ironya, Sivan Toledo, and Alexander Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026, 2004.
23. M. Kaddoura, S. Ranka, and A. Wang. Array decomposition for nonuniform computational environments. *Journal of Parallel and Distributed Computing*, 36:91–105, 1996.
24. A. Kalinov and A. Lastovetsky. Heterogeneous distribution of computations while solving linear algebra problems on networks of heterogeneous computers. In P. Sloot, M. Bubak, A. Hoekstra, and B. Hertzberger, editors, *HPCN Europe 1999*, LNCS 1593, pages 191–200. Springer Verlag, 1999.
25. S. Khuller and Y.A. Kim. On broadcasting in heterogenous networks. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1011–1020. Society for Industrial and Applied Mathematics, 2004.
26. A. Lastovetsky and R. Reddy. Data partitioning with a realistic performance model of networks of heterogeneous computers. In *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society Press, 2004.
27. Keqin Li and Victor Y. Pan. Parallel matrix multiplication on a linear array with a reconfigurable pipelined bus system. *IEEE Transactions on Computers*, 50(5):519–525, 2001.
28. P. Liu. Broadcast scheduling optimization for heterogeneous cluster systems. *Journal of Algorithms*, 42(1):135–152, 2002.
29. M. Maheswaran, S. Ali, H.J. Siegel, D. Hensgen, and R.F. 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.
30. C. D. Polychronopoulos. Compiler optimization for enhancing parallelism and their impact on architecture design. *IEEE Transactions on Computers*, 37(8):991–1004, August 1988.
31. 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.
32. G. Shao. *Adaptive scheduling of master/worker applications on distributed computational resources*. PhD thesis, Dept. of Computer Science, University Of California at San Diego, 2001.
33. G. Shao, F. Berman, and R. Wolski. Master/slave computing on the grid. In *Heterogeneous Computing Workshop HCW'00*. IEEE Computer Society Press, 2000.
34. Sivan 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.
35. J. B. Weissman. Scheduling multi-component applications in heterogeneous wide-area networks. In *Heterogeneous Computing Workshop HCW'00*. IEEE Computer

Society Press, 2000.

36. R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *Proceedings of the ACM/IEEE Symposium on Supercomputing (SC'98)*. IEEE Computer Society Press, 1998.
37. Ling Zhuo and Viktor K. Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 18(4):433–448, 2007.