A Proposal for a Heterogeneous Cluster ScaLAPACK (Dense Linear Solvers)

Olivier Beaumont, Vincent Boudet, Antoine Petitet, Fabrice Rastello, and Yves Robert, *Member*, *IEEE*

Abstract—In this paper, we study the implementation of dense linear algebra kernels, such as matrix multiplication or linear system solvers, on heterogeneous networks of workstations. The uniform block-cyclic data distribution scheme commonly used for homogeneous collections of processors limits the performance of these linear algebra kernels on heterogeneous grids to the speed of the slowest processor. We present and study more sophisticated data allocation strategies that balance the load on heterogeneous platforms with respect to the performance of the processors. When targeting unidimensional grids, the load-balancing problem can be solved rather easily. When targeting two-dimensional grids, which are the key to scalability and efficiency for numerical kernels, the problem turns out to be surprisingly difficult. We formally state the 2D load-balancing problem and prove its NP-completeness. Next, we introduce a data allocation heuristic, which turns out to be very satisfactory: Its practical usefulness is demonstrated by MPI experiments conducted with a heterogeneous network of workstations.

Index Terms—Heterogeneous network, heterogeneous grid, different-speed processors, load-balancing, data distribution, data allocation, numerical libraries, numerical linear algebra, heterogeneous platforms, cluster computing.

1 Introduction

Heterogeneous networks of workstations (HNOWs) are ubiquitous in university departments and companies. They represent the typical poor man's parallel computer: Running a large PVM or MPI experiment (possibly all night long) is a cheap alternative to buying supercomputer hours. The idea is to make use of *all* available resources, namely slower machines *in addition to* more recent ones.

The major limitation to programming heterogeneous platforms arises from the additional difficulty of balancing the load when using processors running at different speeds. This paper is devoted to providing the required framework to build an extension of the ScaLAPACK library [8] capable of running on top of HNOWs or nondedicated parallel machines. More precisely, we concentrate on dense linear algebra kernels, such as matrix multiplication, or LU and QR decompositions. Our goal is to come up with an efficient implementation of such kernels within the framework of the library: The idea is not to rebuild ScaLAPACK kernels from scratch; instead, we take advantage of the deep modularity of the library and we modify only high-level routines related to data distribution. With processors running at different speeds, block-cyclic distribution is no longer enough; new data distribution schemes must be determined and analyzed. We show that deriving efficient distribution schemes is a rather simple task for linear networks, but turns out to be surprisingly difficult for two-dimensional grids. Technically, we prove that the underlying optimization problem is NP-hard for two-dimensional grids and we provide an efficient approximation of the optimal solution.

The rest of the paper is organized as follows: In Section 2, we discuss the framework for implementing our heterogeneous kernels and we briefly review the existing literature. The core of the paper is composed of Sections 3 and 4, where we outline our main results: Section 3 is devoted to data distribution schemes for unidimensional grids; Section 4 is its counterpart for two-dimensional grids. The practical usefulness of our tuned data distribution schemes is demonstrated in Section 5 through several MPI experiments run on two HNOWs configured as both unidimensional and two-dimensional grids. Finally, we give some remarks and conclusions in Section 6.

2 Framework

2.1 Static Distribution Schemes

Because we have a library designer's approach, we target static strategies to allocate data (and associated computations) to the heterogeneous processors. These processors will be configured either as a unidimensional or as a two-dimensional grids, according to the (virtual) hardware configurations currently supported by ScaLAPACK. For homogeneous platforms, ScaLAPACK uses a block-cyclic distribution approach. This means that subblocks (rather than single elements) of the matrices are distributed to processors in a wraparound fashion along the processor grid (this in both dimensions for a two-dimensional grid). Processors are then responsible for the computations to be performed on the data blocks that have been assigned to them.

• *The authors are with LIP, UMR CNRS-ENS Lyon-INRIA 5668, Ecole Normale Supérieure de Lyon, F-69364 Lyon Cedex 07, France. E-mail: {Olivier.Beaumont, Vincent.Boudet, Antoine.Petitet, Fabrice.Rastello, Yves.Robert}@ens-lyon.fr.*

Manuscript received 7 July 2000; accepted 23 May 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 112416.

The advantages of block-cyclic distribution for homogeneous platforms are easily understood. Blocked versions of the classical (systolic-like) parallel algorithms for matrix multiplication and linear system solvers [30] are used in ScaLAPACK to squeeze the most out of state-of-the-art processors with pipelined arithmetic units and multilevel memory hierarchy [19], [10]. Blocked algorithms naturally imply a distribution of matrix blocks to processors. Because matrix blocks are not accessed and operated upon evenly in dense system solvers (the matrix shrinks as the decomposition progresses), a cyclic distribution of blocks is used rather than a plain block distribution. Altogether, the block-cyclic distribution provides an efficient load-balancing of the work while enabling local (scalar) computations to be performed at the highest rate.

Blocked algorithms will be used for heterogeneous platforms, too, so matrix blocks will be distributed to processors as in the homogeneous case. However, a cyclic distribution of blocks is no longer likely to provide good load balancing. Intuitively, if a processor is, say, twice as fast as another one, it should be allocated twice as many blocks. Deriving efficient distribution schemes for heterogeneous machines is the main objective of this paper.

Our ScaLAPACK extension exposes a major difficulty when using heterogeneous platforms. Indeed, assume the platform is configured as a (virtual) two-dimensional grid: Such a configuration may well be used for a large cluster of workstations linked by a fast and dedicated network, such as Myrinet [17]. In that case, a two-dimensional grid would be preferred to a linear array for scalability reasons [10]: For kernels such as matrix multiplication or LU and QR decompositions, a 2D block-cyclic distribution is superior to a 1D block-cyclic distribution, both theoretically [18], [11] and experimentally [8]. It turns out that configuring n heterogeneous processors into a $p \times q$ grid, with $pq \leq n$, is very difficult: To build up the grid, we have to choose the best layout of the processors among an exponential number of possible processor arrangements. We formally state this optimization problem and prove its NP-completeness in Section 4.3.2; then, we provide an efficient heuristic in Section 4.3.4.

2.2 Static versus Dynamic Strategies

Consider an HNOW: Whereas programming a large application made up of several loosely coupled tasks can be performed rather easily (because these tasks can be dispatched dynamically on the available processors), implementing a tightly coupled algorithm, such as a dense linear algebra kernel, requires carefully tuned scheduling and mapping strategies.

Distributing the computations (together with the associated data) can be performed either dynamically or statically or a mixture of both. On one hand, we may think that dynamic strategies are likely to perform better because the machine loads will be self-regulated, hence self-balanced, if processors pick up new tasks just as they terminate their current computation. However, data dependences, in addition to communication costs and control overhead, may well lead to slowing the whole process down to the pace of the slowest processors [9]. On the other hand, static strategies will suppress (or at least minimize) data redistributions and control overhead during execution. Furthermore, in the context of a numerical library, static allocations seem to be necessary for a simple and efficient memory allocation. This paper is devoted to the design of an extension of the ScaLAPACK library and will consider only static strategies. We agree, however, that targeting larger platforms such as distributed collections of heterogeneous clusters, e.g., available from the metacomputing grid [20], may well enforce the use of dynamic schemes.

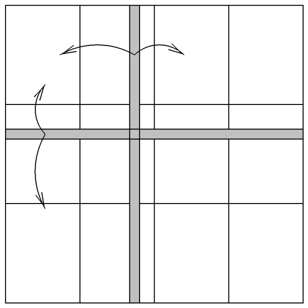


Fig. 1. The MM algorithm on a 3×4 homogeneous 2D-grid.

2.3 Matrix Multiplication on Homogeneous Grids

In this section, we survey the algorithm used in ScaLAPACK for matrix multiplication. Assume that we target a 2D homogeneous: The $p \times q$ processors are identical. In that case, ScaLAPACK uses a block version of the outer product algorithm[1] described in [1], [21], [30], which can be summarized as follows:

- Take a macroscopic view and concentrate on allocating (and operating on) matrix blocks to processors: Each element in A, B, and C is a square $r \times r$ block and the unit of computation is the updating of one block, i.e., a matrix multiplication of size r. In other words, we shrink the actual matrix size N by a factor r and we perform the multiplication of two $n \times n$ matrices whose elements are square $r \times r$ blocks, where $n = N/r$.
- At each step, a column of blocks (the pivot column) is communicated (broadcast) horizontally and a row of blocks (the pivot row) is communicated (broadcast) vertically.
- The A, B, and C matrices are identically partitioned into $p \times q$ rectangles. There is a one-to-one mapping between these rectangles and the processors. Each processor is responsible for updating its C rectangle: More precisely, it updates each block in its rectangle with one block from the pivot row and one block form the pivot column, as illustrated in Fig. 1. For square $p \times p$ homogeneous 2D-grids, and when the number of blocks in each dimension n is a multiple of p (the actual matrix size is thus $N = n.r$), it turns out that all rectangles are identical squares of $\frac{n}{p} \times \frac{n}{p}$ blocks.

This paper is devoted to an extension of this algorithm to a heterogeneous set of computing resources.

1. ScaLAPACK uses a two-dimensional grid rather than a linear array for scalability reasons [8].

2.4 Related Work

There are many papers in the literature dealing with dynamic schedulers to distribute the computations (together with the associated data) onto heterogeneous platforms. Most schedulers use simple mapping strategies such as master-slave 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: See the survey paper of Berman [6] and the more specialized references [2], [13] for further details. Several scheduling and mapping heuristics have been proposed to map task graphs onto HNOWs [34], [35], [32], [26]. Scheduling tools such as Prophet [36] or AppLeS [6] are available (see also the survey paper [33]).

The static mapping of numerical kernels has, however, received much less attention. To the best of our knowledge, there is a single paper by Kalinov and Lastovetsky [28] which has similar objectives as ours. They are interested in LU decomposition on heterogeneous 1D and 2D grids. They propose a "heterogeneous block cyclic distribution" to map matrix blocks onto the different-speed processors. They use the *mpC* programming tool [3] to program the heterogeneous 1D or 2D grids, which they consider as fixed (they do not discuss how to configure the grid). In fact, their "heterogeneous block cyclic distribution" does not lead to a "true" 2D-grid because each processor has more than four direct neighbors to communicate with. We come back to their work in Section 4.1.2.

For the sake of completeness, we quote the following papers, which are not directly related to our work but which share some of our objectives:

- The decomposition of general data-parallel programs for execution onto heterogeneous clusters has been studied by Crandall and Quinn [15], [14]: The idea is to split a data domain into different-size blocks for different-speed processors. Similarly, various array decomposition algorithms are proposed by Kaddoura et al. [27].
- The NP-complete optimization problem studied in Section 4.3.2 is related to some geometric problems such as partitioning a rectangle with interior points [31], [24] or array partitioning [25], [29]. Several NP-complete geometric optimization problems are listed in the NP Compendium [16], [4].

3 Unidimensional Grids

In this section, we target unidimensional grids, i.e., HNOWs configured as linear arrays. We investigate data allocation schemes to evenly balance the workload for ScaLAPACK kernels on such platforms.

We start with the simple problem of distributing independent chunks of computations to linear arrays of heterogeneous processors (Section 3.1). We use this result to tackle the implementation of linear solvers, for which we propose an optimal data distribution in Section 3.2.

3.1 Distributing Independent Chunks

Consider the following simple problem: Given M independent chunks of computations, each of equal size (i.e., each requiring the same amount of work), how can we assign these chunks to p physical processors $P_1, P_2, \ldots, P_p$ of respective cycle-times $t_1, t_2, \ldots, t_p$ so that the workload is best balanced? Here, the execution time is understood as the number of time units needed to perform one chunk of computation, i.e., each processor P_i executes each computation chunk within t_i time units. Then, how do we distribute chunks to processors? The intuition is that the load of P_i should be inversely proportional to t_i. Since the loads (i.e., number of chunks) on each processor must be integers, we use the following algorithm to solve the problem, where c_i denotes the number of chunks allocated to processor P_i. Thus, the overall execution obtained with an allocation $C = (c_1, c_2, \ldots, c_p)$ is given by $\max_i c_i t_i$.

Algorithm 3.1: Optimal distribution for M independent chunks over p processors of cycle-times $t_1, \ldots, t_p$:

```
# Initialization: Approximate the $c_i$ so that
    $c_i \times t_i \approx Constant$ and $c_1 + c_2 + \ldots + c_p \leq M$.
Let $c_i = \left\lfloor \frac{\frac{1}{t_i}}{\sum_{i=1}^{p} \frac{1}{t_i}} \times M \right\rfloor$ for $1 \leq i \leq p$.
# Iteratively increment some $c_i$ until
    $c_1 + c_2 + \ldots + c_p = M$
For $m = c_1 + c_2 + \ldots + c_p$ to M
  find $k \in \{1, \ldots, p\}$ such that
    $t_k \times (c_k + 1) = \min\{t_i \times (c_i + 1))\}$
  $c_k = c_k + 1$
```

Proposition 1. *Algorithm 3.1 gives the optimal allocation.*

Proof. Consider an optimal allocation denoted by $o_1, \ldots, o_p$. Let j be such that $\forall i \in \{1, \ldots, p\}$, $o_j t_j \geq o_i t_i$. To prove the correctness of the algorithm, we prove the invariant

$$(I) : \forall i \in \{1, \ldots, p\}, c_i t_i \leq o_j t_j.$$

After the initialization,

$$c_i \leq \frac{\frac{1}{t_i}}{\sum_{k=1}^{p} \frac{1}{t_k}} \times M.$$

We have

$$M = \sum_{k=1}^{p} o_k \leq o_j t_j \times \sum_{k=1}^{p} \frac{1}{t_k}.$$

Hence,

$$c_i t_i \leq \frac{M}{\sum_{k=1}^{p} \frac{1}{t_k}} \leq o_j t_j$$

and invariant (I) holds.

We use an induction to prove that invariant (I) holds after each incrementation. Suppose that, at a given step, some c_k will be incremented. Before that step, $\sum_{i=1}^{p} c_i < M$, hence, there exists $k' \in \{1, \ldots, p\}$ such that $c_{k'} < o_{k'}$. We have $t_{k'}(c_{k'} + 1) \leq t_{k'} o_{k'} \leq t_j o_j$ and the choice of k implies that $t_k(c_k + 1) \leq t_{k'}(c_{k'} + 1)$. Invariant (I) does hold after the incrementation.

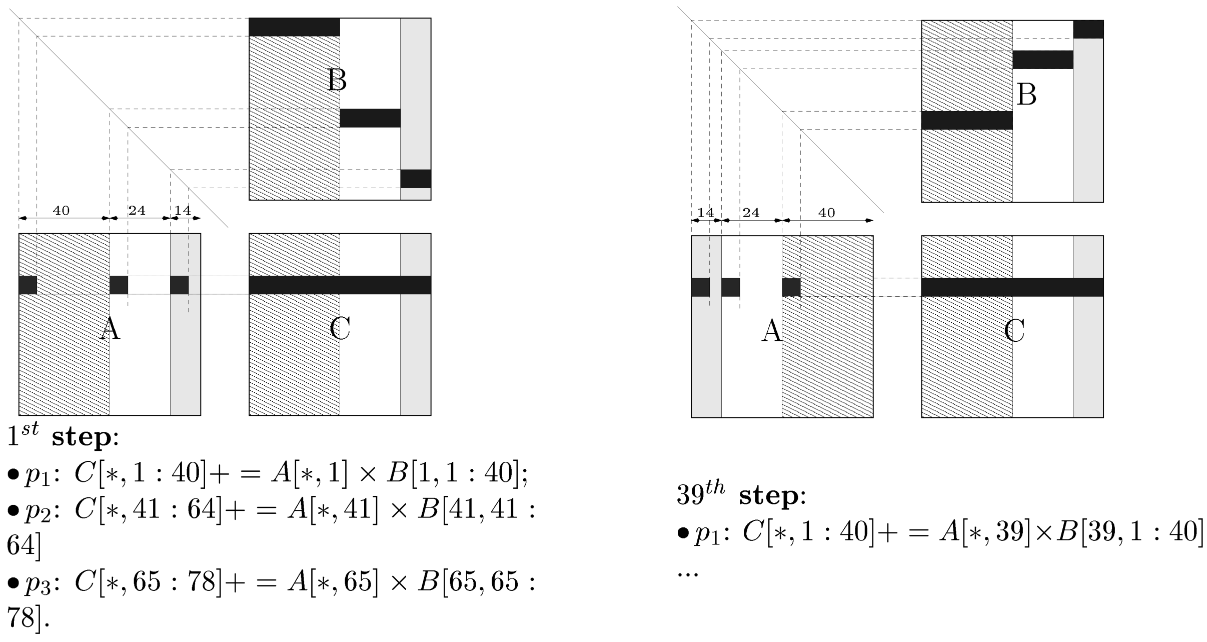

B
40
24
14
A
C
1st step:
• p_1: $C[*,1:40]+=A[*,1]\times B[1,1:40]$;
• p_2: $C[*,41:64]+=A[*,41]\times B[41,41:64]$
• p_3: $C[*,65:78]+=A[*,65]\times B[65,65:78]$.
B
14
24
40
A
C
39th step:
• p_1: $C[*,1:40]+=A[*,39]\times B[39,1:40]$
...

Fig. 2. Different steps of matrix multiplication on a platform made of three heterogeneous processors of respective cycle-times $t_1 = 3$, $t_2 = 5$, and $t_3 = 8$. All indices in the figure are block numbers.

Finally, the time needed to compute the M chunks with the allocation $c_1, \ldots, c_p$ is $\max\{t_i \times c_i\} \leq o_j t_j$ and our allocation is optimal. □

3.1.1 Complexity and Use

Since, after the initialization step, $c_1 + c_2 + \ldots + c_p \geq M - p$, there are at most p steps of incrementation, so that the complexity[2] of Algorithm 3.1 is $O(p^2)$. This algorithm can only be applied to simple load balancing problems such as matrix product on a processor ring (see Section 3.2). Indeed, such a program can be decomposed into successive communication-free steps. The communication between steps is reduced to a simple shift across the ring of processes. Each step consists of a bunch of independent chunks that can be distributed using Algorithm 3.1. Consider a toy example with three processors of respective cycle-times $t_1 = 3$, $t_2 = 5$, and $t_3 = 8$. We aim to compute the product $C = A \times B$, where A and B are of size $2,496 \times 2,496$. The matrices can be decomposed into 78×78 square blocks of size 32×32 (32 is a typical block size for cache-based workstations [8]. Hence, $M = 78$ blocks of columns have to be distributed among the processors and $M = 78$ blocks of columns will be computed at each step. Table 1 applies Algorithm 3.1 to this load balancing problem. A few different steps for matrix multiplication are represented in Fig. 2. Our simple allocation is quite sufficient for matrix multiplication because each step is optimally load-balanced.

3.2 Linear Solvers

Whereas the previous solution is well-suited to matrix multiplication, it is not adapted for LU and QR decompositions, as explained below. Roughly speaking, the LU decomposition algorithm works as follows on a homogeneous linear array [11]: As pointed out in Section 2.1, the preferred distribution is a `CYCLIC(b)` distribution of columns. At each step, the processor that owns the pivot block factors it and broadcasts it to all the processors, which update their remaining column blocks. At the next step, the next block of b columns becomes the pivot panel, and the computation progresses.

Because the largest fraction of the work takes place in the update, we would like to load-balance the work so that the update is best balanced. Consider the first step. After the factorization of the first block, all updates are independent chunks: Here, a chunk consists of the update of a single block of b columns. If the matrix size is $n = M \times b$, there remains $M - 1$ chunks to be updated. We can use Algorithm 3.1 to distribute these independent chunks. But, the size of the matrix shrinks as the computation goes on. At the second step, the number of blocks to update is only $M - 2$. If we want to distribute these chunks independently of the first step, redistribution of data will have to take place between the two steps and this will incur a lot of communications. Rather, in our library-oriented perspective, we search for a static allocation of column blocks to processors that will remain the same throughout the computations as the decomposition progresses. We aim at balancing the updates of all steps with the same

TABLE 1
Steps of Algorithm 3.1 for Three Processors with $t_1 = 3$, $t_2 = 5$, and $t_3 = 8$ and $M = 78$

Steps	c_1	c_2	c_3	$\max_i(c_i t_i)$
Init, m=76	39	23	14	117
m=77	40	23	14	120
m=M=78	40	24	14	120

2. Using a naive implementation. The complexity can be reduced down to $O(p\log(p))$ using ad hoc data structures.

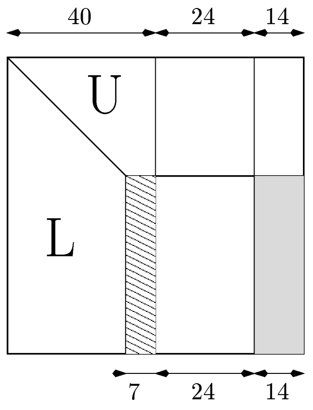


Fig. 3. Thirty-third step of LU decomposition (indices are block numbers): with the former distribution, the computation becomes less balanced. Here, after factoring block 33, processor 1 has seven updates and works for $7 \times 3 = 21$ units of time, while processor 2 works $24 \times 5 = 120$ units of time.

allocation. As illustrated in Fig. 3, we need a distribution that is somewhat repetitive (because the matrix shrinks), but not fully cyclic (because processors have different speeds).

Looking closer at the successive updates, we see that only column blocks of index $i+1$ to M are updated at step i. Hence, our objective is to find a distribution such that, for each $i \in \{2, \ldots, M\}$, the amount of blocks in $\{i, \ldots, M\}$ owned by a given processor is approximately inversely proportional to its cycle-time (proportional to its speed). To derive such a distribution, we use a dynamic programming algorithm [9], which is best explained using the former toy example.

3.3 A Dynamic Programming Algorithm

Consider an example with three processors of (relative) cycle-times $t_1 = 3$, $t_2 = 5$, and $t_3 = 8$. In Table 2, we report the allocations found by the algorithm up to $M = 10$. The entry "Selected processor" denotes the rank of the processor chosen to build the next allocation. At each step, "Selected processor" is computed so that the cost of the allocation is minimized. The cost of the allocation is computed as follows: Let us denote by c_i the number of chunks allocated to processor P_i, then the execution time for an allocation $\mathcal{C} = (c_1, c_2, \ldots, c_p)$ is $\max_{1 \le i \le p} c_i t_i$ (the maximum is taken over all processor execution times). So, the average cost to execute one chunk is

$$\widehat{\mathcal{C}} = \frac{\max_{1 \le i \le p} c_i t_i}{\sum_{i=1}^{p} c_i}.$$

For instance, at step 4, i.e., to allocate the fourth chunk, we start from the solution for three chunks, i.e., $(c_1, c_2, c_3) = (2, 1, 0)$. Which processor P_i should receive the fourth chunk, i.e., which c_i should be incremented? There are three possibilities, $(c_1 + 1, c_2, c_3) = (3, 1, 0)$, $(c_1, c_2 + 1, c_3) = (2, 2, 0)$, and $(c_1, c_2, c_3 + 1) = (2, 1, 1)$, of respective average costs $\frac{9}{4}$ (P_1 is the slowest), $\frac{10}{4}$ (P_2 is the slowest), and $\frac{8}{4}$ (P_3 is the slowest). Hence, we select $i = 3$ and we retain the solution $(c_1, c_2, c_3) = (2, 1, 1)$.

TABLE 2
Running the Dynamic Programming Algorithm with Three Processors: $t_1 = 3$, $t_2 = 5$, and $t_3 = 8$

Number of chunks	c_1	c_2	c_3	Cost	Selected processor
0	0	0	0		1
1	1	0	0	3	2
2	1	1	0	2.5	1
3	2	1	0	2	3
4	2	1	1	2	1
5	3	1	1	1.8	2
6	3	2	1	1.67	1
7	4	2	1	1.71	1
8	5	2	1	1.87	2
9	5	3	1	1.67	3
10	5	3	2	1.6	

Of course, if we are to allocate 10 chunks, we can use Algorithm 3.1 and find that five chunks should be given to processor P_1, three to P_2, and two to P_3. But, the dynamic programming algorithm returns the optimal solution for allocating any number of chunks, from one chunk up to M chunks:

Proposition 2. *The dynamic programming algorithm returns the optimal allocation for any subset of chunks* $[1, s]$, *where* $s \le M$.

See [9] for the proof. The complexity of the dynamic programming algorithm is $O(pM)$,[3] where p is the number of processors and M is the upper bound on the number of chunks. Note that the cost of the allocations is not a decreasing function of M.

3.4 Application to LU Decomposition

For LU decomposition, we allocate slices of B blocks of width r to processors, as illustrated in Fig. 4. B is a parameter to be discussed below. For a matrix of size $n = M \times b$, we can simply let $B = M$, i.e., define a single slice.

Within each slice, we use the dynamic programming algorithm in a "reverse" order. In other words, the kth chunk (for $1 \le k \le B$) is allocated to processor $\sigma(B - k + 1)$. Consider the toy example in Table 1 with three processors of relative speed $t_1 = 3$, $t_2 = 5$, and $t_3 = 8$. The dynamic programming algorithm allocates chunks to processors, as shown in Table 2. Hence, the obtained pattern is $(P_3 P_2 P_1 P_1 P_2 P_1 P_3 P_1 P_2 P_1)$ (see Fig. 5 for the detailed allocation within a slice). As illustrated in Fig. 4, at a given step, there are several slices of at most B chunks and the number of chunks in the first slice decreases as the computation progresses (the leftmost chunk in a slice is computed first and then there only remains $B - 1$ chunks in the slice, and so on). In the example, the reversed allocation best balances the updates in the first slice at each step: At the first step, when there are the initial 10 chunks and nine updates (the first chunk is not updated), but also at the second step, when only eight updates remain, and so on. The updating

3. As for Algorithm 3.1, the complexity can easily be improved in $O(\log(p)M)$.

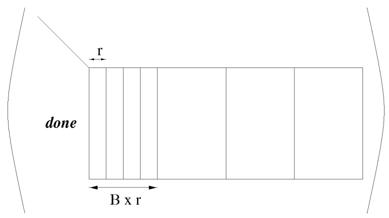


Fig. 4. Allocating slices of B chunks.

of the other slices remains well-balanced by construction since their size does not change and we keep the best allocation for $B = 10$. See Fig. 5 for the detailed allocation within a slice, together with the cost of the updates.

3.4.1 ScaLAPACK on a Heterogeneous Linear Array

We are ready to propose an extension of the ScaLAPACK library on a heterogeneous cluster configured as a unidimensional array. The ScaLAPACK library is devoted to dense linear solvers such as LU or QR factorizations. It turns out that all these solvers share the same computation unit, namely the processing of a block of b columns at a given step. They all exhibit the same control graph: The computation processes by steps; at each step, the pivot block is processed and then it is broadcast to update the remaining blocks.

The proposed data allocation is periodic: At the beginning of the computation, we distribute slices of the matrix to processors in a cyclic fashion. Each slice is composed of B chunks (blocks of b columns) and is allocated according to the previous discussion. The value of B is defined by the user and can be chosen as M if $n = M \times b$, i.e., we can define a single slice for the whole matrix. But, we can also choose a value independent of the matrix size: We may look for a fixed value, chosen from the relative processor speeds, to ensure good load-balancing.

A major advantage of a fully static distribution with a fixed parameter B is that we can use the current ScaLAPACK release with little programming effort. In the homogeneous case with p processors, we use a `CYCLIC(b)` distribution for the matrix data and we define p processes. In the heterogeneous case, we still use a `CYCLIC(b)` distribution for the data, but we define B processes which we allocate to the p physical processors according to our load-balancing strategy. The experiments reported in Section 5 fully demonstrate that this approach is quite satisfactory in practice.

4 Two-Dimensional Grids

In this section, we first summarize existing algorithms for matrix multiplication and dense linear solvers on 2D (homogeneous) grids and we discuss their extension on 2D heterogeneous grids. Section 4.3 contains our most involved results: We state the optimization problem to be solved and we prove its NP-completeness, we discuss how to find optimal solutions (with exponential cost), and we introduce an efficient heuristic.

4.1 Linear Algebra Kernels on 2D Grids

In this section, we briefly recall the algorithms implemented in the ScaLAPACK library [8] on 2D homogeneous grids, which exhibit better scalability properties than unidimensional grids because the communication operations are more uniformly distributed so that, when the physical network allows it, higher parallel efficiency can be achieved, e.g., there will be almost no difference between 1D and 2D on simple Ethernet. Then, we discuss how to modify these algorithms to cope with 2D heterogeneous grids.

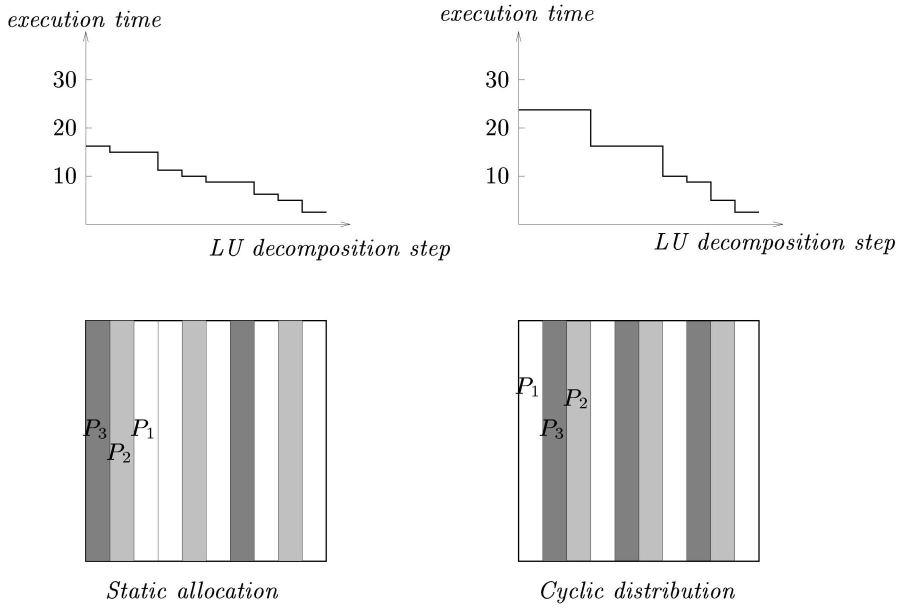


Fig. 5. Comparison of two different distributions for the LU-decomposition algorithm on a heterogeneous platform made of three processors of relative cycle-time 3, 5, and 8. The first distribution is the one given by our algorithm, the second one is the cyclic distribution. The total number of chunks is $B = 10$.

$B_p = 4$

$P_{1,1}$	$P_{1,1}$	$P_{1,2}$
$P_{1,1}$	$P_{1,1}$	$P_{1,2}$
$P_{1,1}$	$P_{1,1}$	$P_{1,2}$
$P_{2,1}$	$P_{2,1}$	$P_{2,2}$

$B_q = 3$

Fig. 6. A block panel with $B_p = 4$ and $B_q = 3$ for a 2×2 processors grid of respective cycle-time $t_{1,1} = 1$, $t_{1,2} = 2$, $t_{2,1} = 3$, and $t_{2,2} = 6$. The processor $P_{1,1}$ is twice as fast as the processor $P_{1,2}$, hence, it is assigned twice as many blocks within each panel.

4.1.1 Matrix Multiplication

Homogeneous Grids. For the sake of simplicity, we focus here on the multiplication $C = AB$ of two square $n \times n$ matrices A and B. In that case, ScaLAPACK uses the outer product algorithm described in [1], [21], [30]. Consider a 2D processor grid of size $p \times q$. Assume first that $n = p = q$. In that case, the three matrices share the same layout over the 2D grid: Processor $P_{i,j}$ stores $a_{i,j}$, $b_{i,j}$, and $c_{i,j}$. Then, at each step k:

- Each processor $P_{i,k}$ (for all $i \in \{1, .., p\}$) horizontally broadcasts $a_{i,k}$ to processors $P_{i,*}$.
- Each processor $P_{k,j}$ (for all $j \in \{1, .., q\}$) vertically broadcasts $b_{k,j}$ to processors $P_{*,j}$,

so that each processor $P_{i,j}$ can independently compute $c_{i,j} = c_{i,j} + a_{i,k} \times b_{k,j}$.

A block version of this algorithm is used in the current version of the ScaLAPACK library because it is scalable, efficient, and it does not need any initial permutation (unlike Cannon's algorithm [30]). Moreover, on a homogeneous grid, broadcasts are performed as independent ring broadcasts (along the rows and the columns), hence, they can be pipelined. As already pointed out, ScaLAPACK uses a blocked version of this basic algorithm. Each matrix coefficient in the description above is replaced by a $b \times b$ square block. A level of virtualization is added: usually, the number of blocks $\lceil \frac{n}{r} \times \rceil \lceil \frac{n}{r} \rceil$ is much greater than the number of processors $p \times q$. Thus, blocks are scattered in a cyclic fashion along both grid dimensions so that each processor is responsible for updating several blocks at each step of the algorithm. In other words, ScaLAPACK uses a `CYCLIC(b)` allocation in both grid dimensions.

4.1.2 Heterogeneous Grids

Suppose now we have a $p \times q$ grid of heterogeneous processors. Instead of distributing the $b \times b$ matrix blocks cyclically along each grid dimension, we distribute *block panels* cyclically along each grid dimension. A block panel is a $B_p \times B_q$ rectangle of consecutive $b \times b$ blocks. See Fig. 6 for an example with $B_p = 4$ and $B_q = 3$: This panel will be distributed cyclically along both dimensions of the 2D grid. The previous cyclic dimension for homogeneous grids obviously corresponds to the case $B_p = B_q = 1$. Now, the distribution of individual blocks is no longer purely cyclic, but remains periodic. We illustrate in Fig. 7 how block panels are distributed on the 2D-grid.

$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$
$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$
$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$
$P_{2,1}$	$P_{2,1}$	$P_{2,2}$	$P_{2,1}$	$P_{2,1}$	$P_{2,2}$	$P_{2,1}$
$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$
$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$
$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$
$P_{2,1}$	$P_{2,1}$	$P_{2,2}$	$P_{2,1}$	$P_{2,1}$	$P_{2,2}$	$P_{2,1}$
$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$
$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$

Fig. 7. Allocating 4×3 panels on a 2×2 grid of processors of respective cycle-time $t_{1,1} = 1$, $t_{1,2} = 2$, $t_{2,1} = 3$, and $t_{2,2} = 6$. There are a total of 10×10 matrix blocks.

How many $b \times b$ blocks should be assigned to each processor within a panel? Intuitively, as in the case of unidimensional grids, the workload of each processor (i.e., the number of block per panel it is assigned to) should be inversely proportional to its cycle-time. In the example of Fig. 6, the allocation of the $B_p \times B_q = 4 \times 3 = 12$ blocks of the panel perfectly balances the load among the four processors.

There is an important condition to enforce when assigning blocks to processors within a block panel. We want each processor in the grid to communicate only with its four direct neighbors. This implies that each processor in a grid row is assigned the same number of matrix rows. Similarly, each processor in a grid column must be assigned the same number of matrix columns. If these conditions do not hold, additional communications will be needed, as illustrated in Fig. 8.

Translated in terms of $b \times b$ matrix blocks, the above conditions mean that each processor P_{ij}, $1 \le j \le q$, in the ith grid row must receive the same number r_i of blocks. Similarly, P_{ij}, $1 \le i \le p$, must receive c_j blocks. This condition does hold in the example of Fig. 7, where $(c_1, c_2) = (2, 1)$ and $(r_1, r_2) = (3, 1)$, hence, each processor only communicates with its direct neighbors.

Unfortunately, and in contrast with the unidimensional case, the additional constraints induced by the communication pattern may well prevent the achievement of perfect load balance amongst processors. Coming back to Fig. 6, we did achieve a perfect load balance, owing to the fact that the processor cycle-times could be arranged in the rank-1 matrix

$$\begin{pmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 6 \end{pmatrix}.$$

For instance, change the cycle-time of $P_{2,2}$ into $t_{22} = 5$. If we keep the same allocation as in Fig. 6, P_{22} remains idle every sixth time-step. Note that there is no solution to perfectly balance the work. Indeed, let r_1, r_2, c_1, and c_2 be the number of blocks assigned to each row and column grid. Processor P_{ij} computes $r_i \times c_j$ blocks in time $r_i \times c_j \times t_{ij}$.

⋮	⋮
$P_{1,1}$	$P_{1,2}$
$P_{1,1}$	$P_{2,2}$
$P_{1,1}$	$P_{1,2}$
$P_{2,1}$	$P_{1,2}$
$P_{1,1}$	$P_{1,2}$
$P_{1,1}$	$P_{1,2}$
$P_{1,1}$	$P_{1,2}$
$P_{2,1}$	$P_{1,2}$

Fig. 8. The distribution of Kalinov and Lastovetsky. Two consecutive columns are represented here. Processor have two west neighbors instead of one.

To have a perfect load balance, we have to fulfill the following equations:

$$r_1 \times t_{11} \times c_1 = r_1 \times t_{12} \times c_2 = r_2 \times t_{21} \times c_1 = r_2 \times t_{22} \times c_2$$
$$\text{that is } \; r_1 c_1 = 2r_1 c_2 = 3r_2 c_1 = 5r_2 c_2.$$

We derive $c_1 = 2c_2$, then $r_1 = 3r_2 = \frac{5}{2}r_2$, hence, a contradiction. Note that we have not taken into account the additional condition $(r_1 + r_2) \times (c_1 + c_2) = 12$, stating that there are 12 blocks within a block panel: It is impossible to perfectly load-balance the work, whatever the size of the panel.

If we relax the constraints on the communication pattern, we can achieve a perfect load-balance as follows: First, we balance the load in each processor column independently (using the unidimensional scheme); next, we balance the load between columns (using the unidimensional scheme again, weighting each column by the inverse of the harmonic mean of the processors cycle-times within the column, see below). This is the "heterogeneous block cyclic distribution" of Kalinov and Lastovetsky [28], which leads to the solution of Fig. 8. Because processor $P_{2,2}$ has two west neighbors instead of one, at each step of the algorithm, it is involved in two horizontal broadcasts instead of one.

We use the example to explain, with further detail, how the heterogeneous block cyclic distribution of Kalinov and Lastovetsky [28] works. First, they balance the load in each processor column independently, using the unidimensional scheme. In the example, there are two processors in the first grid column with cycle-times $t_{11} = 1$ and $t_{21} = 3$, so P_{11} should receive three times more matrix rows than P_{21}. Similarly, for the second grid column, P_{12} (cycle-time $t_{12} = 2$) should receive five out of every seven matrix rows, while P_{22} (cycle-time $t_{22} = 5$) should receive the remaining two rows. Next, how to distribute matrix columns? The first grid column operates as a single processor of cycle-time $2\frac{1}{1+\frac{1}{3}} = \frac{3}{2}$. The second grid column operates as a single processor of cycle-time $2\frac{1}{\frac{1}{2}+\frac{1}{5}} = \frac{20}{7}$. So, out of every 61 matrix columns. we assign 40 to the first processor column and 21 to the second processor column.

Because we do not want to rebuild ScaLAPACK from scratch, we do not want the number of horizontal and vertical communications to depend upon the data distribution. For large grids, the number of horizontal neighbors of a given processor cannot be bounded a priori if we use Kalinov and Lastovetsky's approach. We enforce the grid communication pattern (each processor only communicates with its four direct neighbors) to minimize communication overhead. The price to pay is that we have to solve a difficult optimization problem to load-balance the work as efficiently as possible and that our algorithm may not lead to perfect load balancing because of the topology constraint. Solving this optimization problem is the objective of Section 4.3. In [5], more general data distribution strategies are proposed which lead to perfect load balancing while minimizing communication overhead. Those schemes nevertheless require writing completely new routines for linear algebra kernels and cannot rely on top of ScaLAPACK.

4.2 The LU and QR Decompositions

We first recall the ScaLAPACK algorithm for the LU or QR decompositions on a homogeneous 2D-grid. We discuss next how to implement them on a heterogeneous 2D-grid.

4.2.1 Homogeneous Grids

In this section, we briefly review the direct parallelization of the right-looking variant of the LU decomposition. We assume that the matrix A is distributed onto a two-dimensional grid of (virtual) homogeneous processors. We use a `CYCLIC(b)` decomposition in both dimensions. The right-looking variant is naturally suited to parallelization and can be briefly described as follows: Consider a matrix A of order $n = M \times b$ and assume that the LU factorization of the $k \times b$ first columns has been proceeded with $0 \leq k \leq M - 1$. During the next step, the algorithm factors the next panel of b columns, pivoting if necessary. Next, the pivots are applied to the remainder of the matrix. The lower trapezoid factor just computed is broadcast to the other process columns of the grid using an increasing-ring topology so that the upper trapezoid factor can be updated via a triangular solve. This factor is then broadcast to the other process rows using a minimum spanning tree topology so that the remainder of the matrix can be updated by a rank-b update. This process continues recursively with the updated matrix. In other words, at each step, the current panel of columns is factored into L and the trailing submatrix $\bar{A}$ is updated. The key computation is this latter rank-b update, $\bar{A} \leftarrow \bar{A} - LU$, which can be implemented as follows:

1. The column processor that owns L broadcasts it horizontally (so there is a broadcast in each processor row).
2. The row processor that owns U broadcasts it vertically (so there is a broadcast in each processor column).
3. Each processor locally computes its portion of the update.

$B_q = 6$

	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$
	$P_{2,1}$	$P_{2,2}$	$P_{2,1}$	$P_{2,1}$	$P_{2,2}$	$P_{2,1}$
	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$
$B_p = 8$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$
	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$
	$P_{2,1}$	$P_{2,2}$	$P_{2,1}$	$P_{2,1}$	$P_{2,2}$	$P_{2,1}$
	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$
	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$	$P_{1,1}$	$P_{1,2}$	$P_{1,1}$

Fig. 9. Allocation of the blocks within a block panel with $B_p = 8$ and $B_q = 6$ for a grid of processors of respective cycle-time $t_{1,1} = 1$, $t_{1,2} = 2$, $t_{2,1} = 3$, and $t_{2,2} = 5$.

The communication volume is thus reduced to the broadcast of the two row and column panels and matrix $\bar{A}$ is updated in place (this is known as an outer-product parallelization). Load balance is very good. The simplicity of this parallelization, as well as its expected good performance, explains why the right-looking variants have been chosen in ScaLAPACK [10]. See [18], [10], [7] for a detailed performance analysis of the right-looking variants, which demonstrates their good scalability property. The parallelization of the QR decomposition is analogous [12], [11].

4.2.2 Heterogeneous Grids

For the implementation of the LU and QR decomposition algorithms on a heterogeneous 2D grid, we modify the ScaLAPACK `CYCLIC(b)` distribution very similarly as for the matrix multiplication problem. The intuitive reason is the following: As pointed out before, the core of the LU and QR decompositions is a rank-b update, hence, the load-balancing techniques for the outer-product matrix algorithm naturally apply. However, periodic distributions must be used to take into account the shrinking of the matrix during the elimination.

We still use block panels made up of several $b \times b$ matrix blocks. The block panels are distributed cyclically along both dimensions of the grid. The only modification is that the order of the blocks within a block panel becomes important. Consider the previous example with four processors laid along a 2×2 grid as follows:

$$T = \begin{pmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 5 \end{pmatrix}.$$

Say we use a panel with $B_p = 8$ and $B_q = 6$, i.e., a panel composed of 48 blocks. Using the methods described below (see Section 4.3), we assign the blocks as follows, corresponding to the approximation of T by a rank-1 matrix $\left(\begin{smallmatrix} 1 & 2 \\ 3 & 6 \end{smallmatrix}\right)$:

- Within each panel column, the first processor row receives six blocks and the second processor rows receives two blocks ($(r_1, r_2) = (6, 2)$).
- Out of the six panel columns, the first grid column receives four and the second grid column receives two of them ($(c_1, c_2) = (4, 2)$).

This allocation is represented in Fig. 9. We need to explain how we have allocated the six panel columns. For the matrix multiplication problem, the ordering of the blocks within the panel was not important because all of the processors execute the same number of (independent) computations at each step of the algorithm. For the LU and QR decomposition algorithms, the ordering of columns and rows is quite important. In the example, the first processor column operates like six processors of cycle-time 1 and two processors of cycle-time 3, which is equivalent to a single processor $P_{*,1}$ of cycle-time $\frac{3}{20}$; the second processor column operates like six processors of cycle-time 2 and two processors of cycle-time 5, which is equivalent to a single processor $P_{*,2}$ of cycle-time $\frac{5}{17}$. The unidimensional algorithm allocates the six panel columns as $(P_{*,1}P_{*,2}P_{*,1}P_{*,1}P_{*,2}P_{*,1})$. Identically, the vertical allocation pattern is $(P_{1,*}P_{2,*}P_{1,*}P_{1,*}P_{1,*}P_{2,*}P_{1,*}P_{1,*})$ and we retrieve the allocation of Fig. 9.

To conclude this section, we have a difficult load-balancing problem to solve. First, we do not know which is the best layout of the processors, i.e., how to arrange them to build an efficient 2D grid. In some cases (rank-1 matrices), we are able to load-balance the work perfectly, but, in most cases, this is not possible. Next, once the grid is built, we have to determine the number of blocks that are assigned to each processor within a block panel. Again, this must be done so as to load-balance the work because processors have different speeds. Finally, the panels are cyclically distributed along both grid dimensions. The rest of the paper is devoted to a solution to this difficult load-balancing problem.

4.3 The 2D Heterogeneous Grid Allocation Problem

4.3.1 Problem Statement and Formulation

Consider n processors $P_1, P_2, \ldots, P_n$ of respective cycle-times $t_1, t_2, \ldots, t_n$. The problem is to arrange these processors along a two-dimensional grid of size $p \times q \leq n$ in order to compute the product $C = AB$ of two $N \times N$ matrices as fast as possible. We need some notations to formally state this objective.

Consider a given arrangement of $p \times q \leq n$ processors along a two-dimensional grid of size $p \times q$. Let us renumber the processors as P_{ij}, with cycle-time t_{ij} (with $1 \leq i \leq p$ and $1 \leq j \leq q$). Assume that processor P_{ij} is assigned a block of r_i rows and c_j columns of data elements, meaning that it is responsible for computing $r_i \times c_j$ elements of the C matrix: see Fig. 10 for an example.

There are two (equivalent) ways to compute the efficiency of the grid:

- Processor P_{ij} is scheduled to evaluate rectangular data block $r_i \times c_j$ of the matrix C, which it will process within $r_i \times c_j \times t_{ij}$ units of time. The total execution time t_{exe} is taken over all processors:

  $$t_{exe} = \max_{i,j}\{r_i \times t_{ij} \times c_j\}.$$

  t_{exe} must be normalized to the average time t_{ave} needed to process a single data element: Since there are a total of N^2 elements to compute, we enforce that $\sum_{i=1}^{p} r_i = N$ and that $\sum_{j=1}^{q} c_j = N$. We get

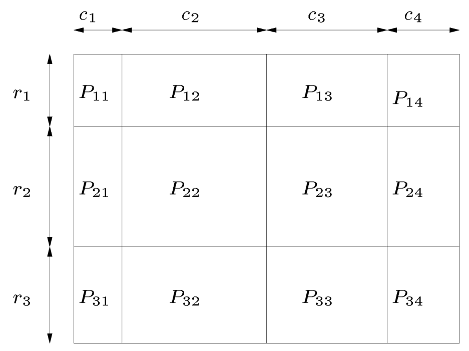


Fig. 10. Allocating computations to processors on a 3×4 grid.

$$t_{ave} = \frac{\max_{i,j}\{r_i \times t_{ij} \times c_j\}}{\left(\sum_{i=1}^{p} r_i\right) \times \left(\sum_{j=1}^{q} c_j\right)}.$$

We are looking for the minimum of this quantity over all possible integer values r_i and c_j. We can simplify the expression for t_{ave} by searching for (nonnegative) rational values r_i and c_j which sum up to 1 (instead of N):

$$\textbf{Objective } Obj_1 : \min_{(\sum_i r_i=1;\sum_j c_j=1)} \max_{i,j}\{r_i \times t_{ij} \times c_j\}.$$

Given the rational values r_i and c_j returned by the solution of the optimization problem Obj_1, we scale them by the factor N to get the final solution. We may have to round up some values, but we do so while preserving the relation $\sum_{i=1}^{p} r_i = \sum_{j=1}^{q} c_j = N$. Stating the problem as Obj_1 renders its solution generic, i.e., independent of the parameter N.

- Another way to tackle the problem is the following: What is the largest number of data elements that can be computed within one time unit? Assume again that each processor P_{ij} of the $p \times q$ grid is assigned a block of r_i rows and c_j columns of data elements. We need to have $r_i \times t_{ij} \times c_j \le 1$ to ensure that P_{ij} can process its block within one cycle. Since the total number of data elements being processed is $(\sum_{i=1}^{p} r_i) \times (\sum_{j=1}^{q} c_j)$, we get the (equivalent) optimization problem:

$$\textbf{Objective } Obj_2 : \max_{r_i \times t_{ij} \times c_j \le 1} \left\{ \left(\sum_i r_i\right) \times \left(\sum_j c_j\right) \right\}.$$

Again, the rational values r_i and c_j returned by the solution of the optimization problem Obj_2 can be scaled and rounded to get the final solution.

Although there are $p+q$ variables r_i and c_j, there are only $p+q-1$ degrees of freedom: If we multiply all r_is by the same factor λ and divide all c_j by λ, nothing changes in Obj_2. In other words, we can impose $r_1 = 1$, for instance, without loss of generality.

We can further manipulate Obj_2 as follows:

$$\begin{aligned}
&\max_{r_i \times t_{ij} \times c_j \le 1} \left\{ \left(\sum_{i=1}^{p} r_i\right) \times \left(\sum_{j=1}^{q} c_j\right) \right\} \\
&= \max_{r_i} \left\{ \max_{c_j \text{ with } r_i \times t_{ij} \times c_j \le 1} \left\{ \left(\sum_{i=1}^{p} r_i\right) \times \left(\sum_{j=1}^{q} c_j\right) \right\} \right\} \\
&= \max_{r_i} \left\{ \left(\sum_{i=1}^{p} r_i\right) \times \max_{c_j \text{with } r_i \times t_{ij} \times c_j \le 1} \left\{ \left(\sum_{j=1}^{q} c_j\right) \right\} \right\} \\
&= \max_{r_i} \left\{ \left(\sum_{i=1}^{p} r_i\right) \times \max_{\forall i, c_j \le \frac{1}{r_i \times t_{ij}}} \left\{ \left(\sum_{j=1}^{q} c_j\right) \right\} \right\} \\
&= \max_{r_i} \left\{ \left(\sum_{i=1}^{p} r_i\right) \times \left(\sum_{j=1}^{q} \min_i \left\{ \frac{1}{r_i \times t_{ij}} \right\} \right) \right\} \\
&= \max_{r_i} \left\{ \left(\sum_{i=1}^{p} r_i\right) \times \left(\sum_{j=1}^{q} \frac{1}{\max_i\{r_i \times t_{ij}\}} \right) \right\}.
\end{aligned}$$

We obtain an expression with only p variables (and $p-1$ degrees of freedom). This last expression does not look very friendly, though. Solving this optimization problem, optimally or through a heuristic, is the main objective of the following sections.

The 2D Load-Balancing Problem. In the next section, we give a solution to the 2D load-balancing problem which can be stated as follows: Given $n = p \times q$ processors, how do we arrange them along a 2D grid of size $p \times q$ so as to optimally load-balance the work of the processors for the matrix multiplication problem? Note that solving this problem will, in fact, lead to the solution of many linear algebra problems, including dense linear system solvers.

The problem is even more difficult to tackle than the optimization problem stated above because we do not assume the processors arrangement as given. We search among all possible arrangements (layouts) of the $p \times q$ processors as a $p \times q$ grid and, for each arrangement, we must solve the optimization problem Obj_1 or Obj_2.

4.3.2 NP-Completeness

In this section, we formally state the previous optimization problem and prove its NP-completeness. In the presentation, processor speeds come in more handy than cycle-times: If processor P_i has (relative) cycle-time t_i, we use its speed $s_i = 1/t_i$ rather than t_i. We start with the definition of the 2D load-balancing problem, where the mapping f represents the processor arrangement along the grid and the r_i and c_j are the variables used in Obj_1:

Definition 1. *MAX-GRID(s): Given p^2 real positive numbers $s_1, \ldots, s_{p^2}$, find*

$$r_1, \ldots, r_p, c_1, \ldots, c_p,$$

and a one-to-one mapping f from $[1,p] \times [1,p]$ to $[1,p^2]$ so that

$$\forall (i,j) \in [1,p] \times [1,p], \quad r_i c_j \le s_{f(i,j)}$$

and $(\sum_{i=1}^{p} r_i)(\sum_{j=1}^{p} c_j)$ is maximal.

The decision problem associated to the optimization problem MAX-GRID is the following:

Definition 2. *MAX-GRID(s, K): Given p^2 real positive numbers $s_1, \ldots, s_{p^2}$ and a real positive number K, does there exist*

$$r_1, \ldots, r_p, c_1, \ldots, c_p,$$

and a one-to-one mapping f from $[1,p] \times [1,p]$ to $[1,p^2]$ so that

$$\forall (i,j) \in [1,p] \times [1,p], \quad r_i c_j \leq s_{f(i,j)}$$

and

$$\left(\sum_{i=1}^{p} r_i\right)\left(\sum_{j=1}^{p} c_j\right) \geq K.$$

Theorem 1. *MAX-GRID(s, K) is NP-complete.*

Proof. The proof is lengthy and technical. We use several lemmas. First, we select the following NP-complete problem for the reduction:

Lemma 1.

$$2P\text{-}eq \leq_P MAX\text{-}GRID,$$

where 2P-eq is defined as follows:

Definition 3. *2-Partition-Equal (2P-eq). Given a set of p integers, $\mathcal{A} = \{a_1, \ldots, a_p\}$, is there a partition of $\{1, \ldots, p\}$ into two subsets $\mathcal{A}_1$ and $\mathcal{A}_2$ such that*

$$\sum_{i \in \mathcal{A}_1} a_i = \sum_{i \in \mathcal{A}_2} a_i \text{ and } card(\mathcal{A}_1) = card(\mathcal{A}_2) \text{ ?}$$

Since *2P-eq* is known to be NP-Complete [22], Lemma 1 will complete the proof of Theorem 1.

(a) Reduction: $2P\text{-}eq \leq_P$ MAX-$(\mathbf{s}, \mathbf{K})$. Here, we consider an arbitrary instance of the 2-Partition-Equal problem, i.e., a set $\mathcal{A} = \{a_1, \ldots, a_{2n}\}$ of $2n$ integers. We have to polynomially transform this instance into an instance of the MAX-GRID problem which has a solution iff the original instance of 2-Partition-Equal has one solution.

Define $\{b_1, \ldots, b_{2n}\}$ as $\forall i, b_i = (a_i + 2n \max_k a_k)$. Thus, $2n \max a_i \leq b_i \leq (2n+1) \max a_i$. We build the instance of the MAX-GRID problem (denoted by $\text{MAX-GRID}(b_1, \ldots, b_{2n}, K)$), where

$$\left\{\begin{array}{rcl} K & = & (4n \max a_i)^2 + \sum_{i=1}^{2n} b_i + \frac{(\sum_{i=1}^{2n} b_i)^2}{4(4n \max a_i)^2}, \\ s_1 & = & (4n \max a_i)^2, \\ s_{i+1} & = & b_i, \ \forall i, \ 1 \leq i \leq 2n, \\ s_i & = & 1, \ \forall i, \ 2n+2 \leq i \leq (n+1)^2. \end{array}\right.$$

In what follows, we show that a solution is necessarily as depicted in Fig. 11, where the restriction of f to $([2, n+1], 1) \bigcup (1, [2, n+1])$ defines a one-to-one mapping with $[2, 2n+1]$ and

$$\left\{\begin{array}{rrcl} & r_1 c_1 & = & (4n \max a_i)^2, \\ \forall i, \ 2 \leq i \leq n+1, & r_i & = & \frac{b_{\sigma(i,1)}}{c_1}, \\ \forall j, \ 2 \leq j \leq n+1, & c_j & = & \frac{b_{\sigma(1,j)}}{r_1}, \end{array}\right.$$

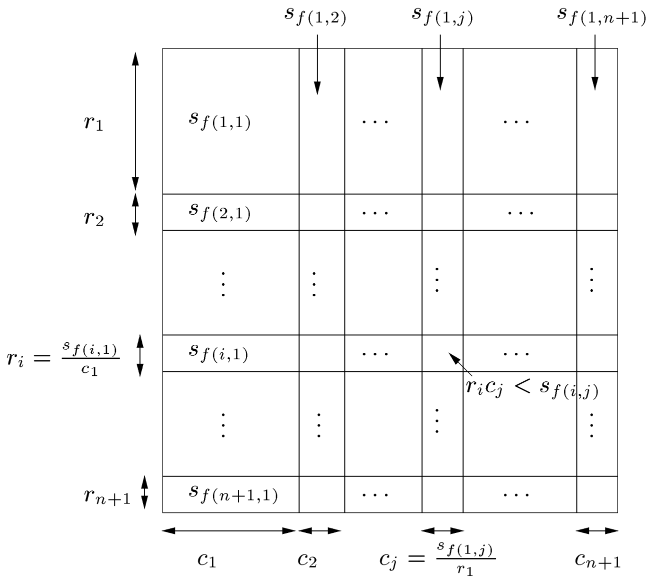


Fig. 11. Solution of $\text{MAX-GRID}(b_1, b_2, \ldots, b_{2n}, K)$.

Thus, we can check that

$$\left(\sum_{i=1}^{p} r_i\right)\left(\sum_{j=1}^{p} c_j\right) \geq K \Longleftrightarrow \sum_{2}^{n+1} b_{f(i,1)} = \sum_{2}^{n+1} b_{f(1,j)}.$$

The intuitive reason is the following: Since s_1 is much larger than other areas, the best choice is r_1 and c_1 so that $r_1 c_1 = s_1$. Similarly, because $b_i \gg 1$, it would be better to put the b_is on the same row or the same column as s_1. Otherwise, one of the b_is would be in the middle of the grid and the area assigned to the corresponding processor would be much smaller than the one it is able to process. Finally, the area of the grid will be maximal when the grid is balanced, i.e., if the b_is satisfy *2P-eq*. All these points will be made clear in the proof below.

Lemma 2. *If $(\sum_{i=1}^{p} r_i)(\sum_{j=1}^{p} c_j) \geq K$, then the area assigned to the processor of speed s_1 is at least $11n^2 \max {a_i}^2$.*

Proof. Up to a permutation, suppose that $f^{-1}(1) = (1,1)$, i.e., that the area assigned to the processor of speed s_1 is $r_1 c_1$. Since $\sum_{i \geq 2} s_i \leq 5n^2 \max {a_i}^2$ and because

$$\left(\sum_{i=1}^{p} r_i\right)\left(\sum_{j=1}^{p} c_j\right) \leq r_1 c_1 + \sum_{i \geq 2} s_i,$$

we have

$$r_1 c_1 \geq 11n^2 \max {a_i}^2.$$

□

Lemma 3. *Let f be any one-to-one mapping from $[1, n+1] \times [1, n+1]$ to $[1, (n+1)^2]$ s.t. $f(1,1) = 1$. Moreover, let us suppose that $r_1 c_1 \geq 11n^2 \max {a_i}^2$ (Lemma 2). Then, $(\sum_{i=1}^{p} r_i)(\sum_{j=1}^{p} c_j)$ is maximal if and only if*

$$\begin{cases} \forall i \geq 2, & r_i c_1 = s_{\sigma(i,1)}, \\ \forall j \geq 2, & c_j r_1 = s_{\sigma(1,j)}. \end{cases}$$

Proof. By definition,

$$\begin{cases} \forall i \geq 2, & r_i c_1 \leq s_{\sigma(i,1)} \\ \forall j \geq 2, & c_j r_1 \leq s_{\sigma(1,j)}, \end{cases}$$

and $(\sum_{i=1}^{p} r_i)(\sum_{j=1}^{p} c_j)$ is maximal when each r_i and c_j is maximal.

Moreover, if

$$\begin{cases} \forall i \geq 2, & r_i c_1 = s_{\sigma(i,1)} \\ \forall j \geq 2, & c_j r_1 = s_{\sigma(1,j)}, \end{cases}$$

then all other conditions $r_i c_j \leq s_{f(i,j)}$ are automatically fulfilled. Indeed,

$$\forall (i,j) \neq (1,1), \quad 1 \leq s_{f(i,j)} \leq (2n+1) \max a_i$$

and, thus,

$$\begin{aligned} r_i c_j &= \frac{s_{f(i,1)} s_{f(1,j)}}{x_1 y_1}, \\ &\leq \frac{(2n+1)^2 \max {a_i}^2}{11 n^2 \max {a_i}^2} \\ &\leq 1 \\ &\leq s_{f(i,j)}. \end{aligned}$$

□

Therefore, f being given, the maximal value for $(\sum_{i=1}^{p} r_i)(\sum_{j=1}^{p} c_j)$ is

$$S(f) = \left(r_1 + \frac{\sum_2^{n+1} s_{f(i,1)}}{c_1} \right) \left(c_1 + \frac{\sum_2^{n+1} s_{f(1,j)}}{r_1} \right).$$

Let us set

$$p = r_1 c_1, \; S_1 = \sum_2^{n+1} s_{f(i,1)} \text{ and } S_2 = \sum_2^{n+1} s_{f(1,j)}.$$

Thus,

$$S(f) = p + (S_1 + S_2) + \frac{S_1 S_2}{p}.$$

Lemma 4.

$$S(f) \geq K \Longleftrightarrow (p = (4n \max a_i)^2) \; and \; \left(S_1 = S_2 = \frac{\sum b_i}{2} \right)$$

and, therefore, $S(f) \geq K$ if and only if there exists a solution to the original instance of the 2P-eq problem.

Proof. By construction, $S_1 + S_2 \leq \sum b_i$ and, therefore, $S_1 S_2 \leq \frac{(\sum b_i)^2}{4}$. Moreover, $S_1 S_2 = \frac{(\sum b_i)^2}{4}$ if and only if $S_1 = S_2 = \sum b_i$. Thus,

$$S(f) \leq p + \sum b_i + \frac{(\sum b_i)^2}{4p}.$$

Let us consider the function g defined by

$$g(p) = p + \frac{(\sum b_i)^2}{4p}.$$

This function is a decreasing rather than increasing function of p and reaches the minimum value for $p = \frac{\sum b_i}{2}$. Since

$$\frac{\sum b_i}{2} \gg 11 n^2 \max {a_i}^2 \leq p \leq (4n \max a_i)^2,$$

g is maximal for $p = (4n \max a_i)^2$. In other words,

$$\begin{cases} S(\sigma) \leq (4n \max a_i)^2 + \sum b_i + \frac{(\sum b_i)^2}{4(4n \max a_i)^2} = K \\ S(\sigma) = K \text{ if and only if } (p = (4n \max a_i)^2) \\ \text{and } (S_1 = S_2 = \frac{\sum b_i}{2}). \end{cases}$$

□

Thus, MAX-GRID$(b_1, \ldots, b_{2n}, K)$ has a solution iff $2P$-$eq(b_1, \ldots, b_{2n})$ has a solution and, therefore, iff the original instance of $2P$-$eq(a_1, \ldots, a_{2n})$ has a solution.

(b) Conciseness of the Transformation. The last element of the proof is to prove that our instance of the MAX-GRID problem has a size polynomial in the size of the original instance of the *2P-eq* problem.

Lemma 5. *Define $MAX = \max_k a_k$ as above and let $c(a)$ and $c(b)$ denote, respectively, the encoding of the data a and b. Then,*

$$Length(c(b)) = O(Length(c(a))^2).$$

Proof.

$$\begin{aligned} &\text{Length}(c(a)) = \sum_k \log(a_k) \geq \log(\text{MAX}) + (n-1)\log(\min_k a_k) \\ &\geq (n-1)\log 2 + \log \text{MAX}. \\ &\text{Length}(c(b)) = \sum_k \log(b_k) = \sum_k \log\Big(2n \text{ MAX}\Big(1 + \frac{a_k}{n\text{MAX}}\Big)\Big) \\ &\leq 1 + n(\log n + \log 2) + n \log \text{MAX}. \end{aligned}$$

Therefore,

$$\text{Length}(c(b)) = O(\text{Length}(c(a))^2).$$

□

This achieves the proof of the NP-completeness of MAX-GRID. □

4.3.3 Searching for the Optimal Solution

For small grid sizes, we may want to search for the optimal solution even though the previous result shows that an exponential cost is unavoidable (unless P = NP). We start with a useful result to reduce the number of arrangements to be searched. Next, we derive an algorithm to solve the optimization problem Obj_1 or Obj_2 for a fixed (given) arrangement. Despite the reduction, we still have an exponential number of arrangements to search for. Even worse, for a fixed arrangement, our algorithm exhibits an exponential cost. Therefore, we shall introduce a heuristic in the next section, in order to give a fast but suboptimal solution to the 2D load-balancing problem.

Reduction to Nondecreasing Arrangements. In this section, we show that we do not have to consider all the possible arrangements; instead, we reduce the search to "nondecreasing arrangements". A nondecreasing arrangement on a $p \times q$ grid is defined as follows: In every grid row, the cycle-times are increasing: $t_{ij} \leq t_{i,j+1}$ for all $1 \leq j \leq q-1$. Similarly, in every grid column, the cycle-times are increasing: For all $1 \leq i \leq p-1$, $t_{ij} \leq t_{i+1,j}$.

Proposition 3. *There exists a nonincreasing arrangement which is optimal.*

Proof. The proof works as follows:

1. Let the pq cycle-times be denoted as $t_1, t_2, \ldots, t_{pq}$.
2. Consider an optimal arrangement on the grid of size $p \times q$. Note that there is no reason that the optimal arrangement must be a nondecreasing arrangement.
3. Show that some well-chosen "correct" transpositions can be applied to the arrangement while preserving the optimality of the solution. The correct transpositions will make the arrangement "closer" to a nondecreasing arrangement.
4. Show that the number of steps to reach a nondecreasing arrangement is finite.

We need a few definitions:

Definition 4.

- ***(Arrangement)*** *An arrangement is a one-to-one mapping*

$$\sigma : \begin{pmatrix} [1, pq] & \mapsto & [1,p] \times [1,q] \\ k & \rightarrow & \sigma(k) = (i,j) \end{pmatrix}$$

which assigns a position to each processor in the grid.

- ***(Nondecreasing arrangement)*** *An arrangement σ is nondecreasing if $t_{\sigma^{-1}(i,j)} \leq t_{\sigma^{-1}(i',j')}$ for all $(1,1) \leq (i,j) \leq (i',j') \leq (p,q)$.*
- ***(Correct transposition)*** *Let σ be an arrangement. If $\sigma(k) < \sigma(l)$ and $t_k > t_l$, the transposition $\tau(k,l)$ which transposes the values of $\sigma(k)$ and $\sigma(l)$ is said to be correct.*

Given any arrangement σ, there exists a suite of correct transpositions that modifies σ into a nondecreasing arrangement. To prove this, we use a *weight function* W that quantifies the distance to the "nondecreasingness." We will show that a correct transposition will decrease the weight of the arrangement it is applied to. We choose

$$W(\sigma) = \sum_{i,j} t_{\sigma^{-1}(i,j)} \times (p+q-i-j).$$

Let us check that, for each correct transposition τ, $W(\tau(\sigma)) < W(\sigma)$. Let σ be an arrangement such that $(i,j) \leq (i',j')$ and $t_{\sigma^{-1}(i,j)} > t_{\sigma^{-1}(i',j')}$. Let $k = \sigma^{-1}(i,j)$ and $l = \sigma^{-1}(i',j')$: By hypothesis, the transposition $\tau = \tau(k,l)$ is correct. Let $\sigma' = \tau \circ \sigma$. We have

$$\begin{aligned} W(\sigma') &= W(\sigma) + (t_{\sigma^{-1}(i',j')} - t_{\sigma^{-1}(i,j)})(p+q-i-j) \\ &\qquad + (t_{\sigma^{-1}(i,j)} - t_{\sigma^{-1}(i',j')})(p+q-i'-j') \\ &= W(\sigma) - ((i'-i)+(j'-j))(t_{\sigma^{-1}(i,j)} - t_{\sigma^{-1}(i',j')}) \\ &< W(\sigma). \end{aligned}$$

Consider an optimal arrangement σ and let $r_1, \ldots, r_p$ and $c_1, \ldots, c_q$ be the solution to the optimization problem Obj_1. Since an equivalent solution is obtained by transposing two columns or two rows of the arrangement, we can assume that $r_1 \geq r_2 \geq \ldots r_p$ and $c_1 \geq c_2 \geq \ldots c_q$. If σ is nondecreasing, we are done. Otherwise, there exists $(1,1) \leq (\alpha,\beta) < (p,q)$ such that either $t_{\sigma^{-1}(\alpha+1,\beta)} < t_{\sigma^{-1}(\alpha,\beta)}$ or $t_{\sigma^{-1}(\alpha,\beta+1)} < t_{\sigma^{-1}(\alpha,\beta)}$. The proof is the same in both cases, hence, assume that $t_{\sigma^{-1}(\alpha+1,\beta)} < t_{\sigma^{-1}(\alpha,\beta)}$. Let $k = \sigma^{-1}(\alpha,\beta)$ and $l = \sigma^{-1}(\alpha+1,\beta)$: By hypothesis, the transposition $\tau = \tau(k,l)$ is correct. Let $\sigma' = \tau \circ \sigma$. We want to show that σ' is as good as σ (in the sense of Obj_2), so we need to show that, for all i and j, $r_i c_j t_{\sigma'^{-1}(i,j)} \leq 1$.

Indeed, we have $r_\alpha \geq r_{\alpha+1}$ and $t_{\sigma^{-1}(\alpha+1,\beta)} < t_{\sigma^{-1}(\alpha,\beta)}$, hence,

$$\begin{cases} r_\alpha c_\beta t_{(\tau\circ\sigma)^{-1}(\alpha,\beta)} = r_\alpha c_\beta t_{\sigma^{-1}(\alpha+1,\beta)} \leq r_\alpha c_\beta t_{\sigma^{-1}(\alpha,\beta)} \leq 1 \\ r_{\alpha+1} c_\beta t_{(\tau\circ\sigma)^{-1}(\alpha+1,\beta)} = r_{\alpha+1} c_\beta t_{\sigma^{-1}(\alpha,\beta)} \leq r_\alpha c_\beta t_{\sigma^{-1}(\alpha,\beta)} \leq 1. \end{cases}$$

Therefore, σ' is optimal, too, and $W(\sigma') < W(\sigma)$. If σ' is not nondecreasing, we repeat the process, which converges in a finite number of steps, because there is a finite number of weight values. □

Spanning Trees for a Given Arrangement. In this section, we show how to solve the optimization problem Obj_1 or Obj_2 for a given arrangement. For small size problems, all the possible nondecreasing arrangements can be generated, hence, we have an exponential but feasible solution to the 2D load balancing problem. Let σ be a given arrangement on a $p \times q$ grid and let $(r_1, \ldots, r_p, c_1, \ldots, c_q)$ be the solution to the optimization problem Obj_1.

Consider the optimization problem Obj_1. We have to maximize the quadratic expression $(\sum_{1\leq i\leq p} r_i)(\sum_{1\leq j\leq q} c_j)$ under $p \times q$ inequalities $r_i t_{ij} c_j \leq 1$. We have $p+q-1$ degrees of freedom. The objective of this section is to show that, for at least $p+q-1$, inequalities are in fact equalities. We use a graph-oriented approach to this purpose.

We consider the following complete bipartite graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, composed on one side of p vertices labeled with r_i and the other side by q vertices labeled with c_j. The weight of the edge (r_i, c_j) is t_{ij}. Given a spanning tree $\mathcal{T} = (\mathcal{V}, \mathcal{E}')$ of the graph $\mathcal{G}$, if we start from $r_1 = 1$, we can (uniquely) determine all the values of the r_i and c_j by following the edges of $\mathcal{T}$, enforcing the equalities

$$\forall (r_i, c_j) \in \mathcal{E}', \quad r_i t_{i,j} c_j = 1.$$

The spanning tree $\mathcal{T}$ is said to be *acceptable* if and only if all the remaining inequalities are satisfied: $\forall (r_i, c_j) \in \mathcal{E}, \ r_i t_{i,j} c_j \leq 1$. The value of an acceptable spanning tree is $(\sum r_i)(\sum c_j)$. We claim that the solution of Obj_1 is obtained with the acceptable spanning tree of maximal value.

This leads to the following algorithm:

- We generate recursively all the spanning trees of $\mathcal{G}$ of root r_1.
- Iteratively, we add vertices to the tree and, by enforcing the condition $r_i t_{ij} c_j = 1$, we evaluate the corresponding labels.
- Let us define *an acceptable subtree* as a subgraph of an acceptable tree. Then, during the recursive process, it might be possible to detect that a subtree is not acceptable so that all the trees are not built.

Finally, we select the acceptable tree that maximizes $\sum_{1\le i\le p} r_i \sum_{1\le j\le q} c_j$.

Correctness of the Algorithm. To justify the previous algorithm, consider an optimal solution to Obj_1 and draw the bipartite graph $\mathcal{U} = (\mathcal{V}, \mathcal{E}')$ corresponding to the equalities: $\mathcal{U}$ has $p + q$ vertices labeled with r_i and c_j. There is an edge between vertices r_i and c_j ($(r_i, c_j) \in \mathcal{E}'$) if and only if $r_i c_j t_{i,j} = 1$. Let us show that this graph is connected, so suppose it is not.

Suppose (without any loss of generality) that $\mathcal{V}' = \{r_1, \ldots, r_{p'}, c_1, \ldots, c_{q'}\}$ is a connected component of $\mathcal{U}$ and that $p' < p$.

First, suppose that $q' = q$. It means that, for all $1 \le j \le q$, $r_{p'+1} c_j t_{p'+1,j} < 1$. Then, $r_{p'+1}$ can be increased by a factor of α, with $\alpha = \min_{1\le j\le q} \frac{1}{r_{p'+1} c_j t_{p'+1,j}}$, and the so-built solution is strictly better (since the product $\sum_{1\le i\le p} r_i \sum_{1\le j\le q} c_j$ is increased), which contradicts the optimality of the initial solution.

Consequently, $q' < q$ and $p' < p$. It means that, for all

$$(p'+1, 1) \le (i,j) \le (p, q')$$

and for all

$$(1, q'+1) \le (i,j) \le (p', q), \quad r_i c_j t_{i,j} < 1.$$

Let

$$\begin{cases} \alpha_r &= \min_{(i>p' \text{ and } j\le q')} \frac{1}{r_i c_j t_{i,j}} \\ \alpha_c &= \min_{(i\le p' \text{ and } j>q')} \frac{1}{r_i c_j t_{i,j}}. \end{cases}$$

We also introduce the notations

$$\begin{cases} R_a &= \sum_{i\le p'} r_i \\ R_b &= \sum_{i>p'} r_i \\ C_a &= \sum_{j\le q'} c_j \\ C_b &= \sum_{j>q'} c_j. \end{cases}$$

Now, we have two possibilities to increase the connectivity of the graph: Either increase the elements corresponding to R_b by a factor of α_r (but decrease the elements corresponding to C_b so as to maintain the acceptable solution) or increase the elements corresponding to C_b by a factor of α_c (but decrease the element corresponding to R_b by a factor $\frac{1}{\alpha_c}$). As we will see, at least one of these solutions does increase the product $\sum_{1\le i\le p} r_i \sum_{1\le j\le q} c_j = (R_a + R_b)(C_a + C_b)$, which contradicts the optimality hypothesis of the initial solution.

Indeed, consider the function f defined by $f(\lambda) = (\lambda R_b + R_a)(\frac{C_b}{\lambda} + C_a)$ Note that $f'(1) = R_b C_a - R_a C_b$. Moreover, f is a continuous function that is first strictly decreasing to a minimum and then strictly increasing. Therefore:

- If $f'(1) \ge 0$, then, for all $\lambda \ge 1$, $f(\lambda) > f(1)$. In particular, $f(\alpha_r) > f(1)$.
- If $f'(1) \le 0$, then, for all $\lambda \le 1$, $f(\lambda) > f(1)$. In particular, $f(\frac{1}{\alpha_c}) > f(1)$.

One of the previous two solutions will indeed increase the connectivity of $\mathcal{U}$ while preserving the objective function $\sum_{1\le i\le p} r_i \sum_{1\le j\le q} c_j$. We conclude that there does exist an acceptable spanning tree whose value is the optimal solution.

In summary, given an arrangement, we are able to compute the solution to the optimization problem. The cost is exponential because there is an exponential number of spanning trees to check for acceptability. Still, our method is constructive and can be used for problems of limited size.

Case of a 2×2 Grid. For small size grids, we can find analytical solutions to the optimization problem. As an example, we explicitly show the solution for a 2×2 grid. In that case, we want to maximize (see the definition of Obj_2) the quantity $(r_1 + r_2)(\frac{1}{\max(r_1 t_{11}, r_2 t_{21})} + \frac{1}{\max(r_1 t_{12}, r_2 t_{22})})$. We normalize our problem by letting $r_1 = 1$ and $r_2 = r$. We have to maximize

$$(1+r)\left(\frac{1}{max(t_{11}, rt_{21})} + \frac{1}{max(t_{12}, rt_{22})}\right).$$

There are three cases to study:

- *First case* $0 \le r \le min(\frac{t_{11}}{t_{21}}, \frac{t_{12}}{t_{22}})$: In this interval, the value of the expression varies as an increasing function of r. So, the maximum on this interval is obtained for $r = min(\frac{t_{11}}{t_{21}}, \frac{t_{12}}{t_{22}})$.
- *Second case* $max(\frac{t_{11}}{t_{21}}, \frac{t_{12}}{t_{22}}) \le r \le +\infty$: In this interval, the expression varies as a decreasing function of r. So, the maximum is obtained for $r = max(\frac{t_{11}}{t_{21}}, \frac{t_{12}}{t_{22}})$.
- *Third case* $min(\frac{t_{11}}{t_{21}}, \frac{t_{12}}{t_{22}}) \le r \le max(\frac{t_{11}}{t_{21}}, \frac{t_{12}}{t_{22}})$: By symmetry, we can suppose that $\frac{t_{11}}{t_{21}} \le \frac{t_{12}}{t_{22}}$. So, our expression is now $(1+r)(\frac{1}{rt_{21}} + \frac{1}{t_{12}})$. This function is first decreasing and then increasing. Hence, the maximum is obtained on the bounds of the interval, i.e., for $r = \frac{t_{11}}{t_{21}}$ or $r = \frac{t_{12}}{t_{22}}$.

In conclusion, there are two possible values for the maximum, namely $r = \frac{t_{11}}{t_{21}}$ or $r = \frac{t_{12}}{t_{22}}$. For the objective function, we obtain the value

$$\max\left(\left(1 + \frac{t_{11}}{t_{21}}\right)\left(\frac{1}{t_{11}} + \frac{1}{\max(t_{12}, \frac{t_{11}t_{22}}{t_{21}})}\right),\right.$$
$$\left.\left(1 + \frac{t_{12}}{t_{22}}\right)\left(\frac{1}{\max(t_{11}, \frac{t_{12}t_{21}}{t_{22}})} + \frac{1}{t_{12}}\right)\right).$$

Rank-1 Matrices. If the matrix $(t_{ij})_{1\le i\le p, 1\le j\le q}$ is a rank-1 matrix, then the optimal arrangement for the 2D load-balancing problem is easy to determine. Assume without loss of generality that $t_{11} = 1$. We let $r_1 = c_1 = 1$, $r_i = \frac{1}{t_{i1}}$ for $2 \le i \le p$ and $c_j = \frac{1}{t_{1j}}$ for $2 \le i \le q$. All the $p \times q$ inequalities $r_i t_{ij} c_j$ are equalities, which means that all processors are fully utilized:

$$r_i t_{ij} c_j = \frac{1}{t_{i1}} t_{ij} \times \frac{1}{t_{1j}} = 1,$$

because the 2×2 determinant

$$\begin{vmatrix} t_{11} & t_{1j} \\ t_{i1} & t_{ij} \end{vmatrix}$$

is zero (with $t_{11} = 1$). No idle time occurs with such a solution; the load-balancing is perfect.

Unfortunately, given $p \times q$ integers, it is very difficult to know whether they can be arranged into a rank-1 matrix of size $p \times q$. If such an arrangement does not exist, we can intuitively say that the optimal arrangement is the "closest one" to a rank-1 matrix.

4.3.4 Heuristic Solution

In this section, we study a heuristic solution to find an arrangement of the processors and a solution to the corresponding optimization problem that leads to good load-balancing. As pointed out above, if the processors cycle-times can be arranged into a rank-1 matrix, it is easy to compute the r_is and the c_js so that no idle time occurs. In general, this is not possible. Nevertheless, basic linear algebra techniques enable us to find both a reasonable arrangement (close to a rank-one matrix) and reasonable values for the r_is and the c_js (so that the idle time is low).

The heuristic works as follows: First, it determines a reasonable (nondecreasing) arrangement matrix for processor cycle-times. Then, it computes approximate values for the r_is and the c_js corresponding to this arrangement matrix. Finally, an iterative refinement of the arrangement matrix is proposed which computes a new arrangement matrix which better fits with the values of the r_is and the c_js computed during the second step.

Initial Arrangement of the Processors. We are given the processors cycle-times as input to the heuristic. Our aim is to find an arrangement so that the resulting matrix is close to a rank-one matrix. We arrange the processors cycle-times in the matrix T as follows:

$$\begin{cases} \forall i, \ \forall 1 \le j < q, & t_{i,j} \le t_{i,j+1} \\ \forall 1 \le i < p, & t_{i,q} \le t_{i+1,1}. \end{cases}$$

For instance, if we consider the case of nine processors with cycle-times $(1, 2, \ldots, 9)$, the arrangement matrix T that we obtain is

$$T = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}.$$

This arrangement is not optimal, but it is nondecreasing. We will refine it using SVD techniques [23] in order to obtain a better arrangement.

Computing the r_is and the c_js. As noticed above, the computation of the r_is and the c_js is obvious when T is a rank-1 matrix. The SVD decomposition provides the best approximation (in the sense of the 2-norm) of a given matrix by a rank-one matrix. It is therefore natural to use it to find the best approximation of the arrangement matrix defined above and then to compute the r_is and the c_js corresponding to this approximation.

Let us denote by $U \Sigma V^t = S$ the singular value decomposition of S, where $S = (\frac{1}{t_{i,j}})_{i,j}$. The best approximation of S by a rank-1 matrix is given by sab^t, where a and b are, respectively, the left and right singular vectors associated with s, the largest singular value of S. Thus, if we set

$$\begin{cases} \forall i, & r_i = sa_i \\ \forall j, & c_j = b_j, \end{cases}$$

we can expect that

$$\forall i, j, \qquad r_i t_{i,j} c_j \simeq 1.$$

We consider T^{inv} rather than T since the approximation by the rank-1 matrix is usually better on the largest components. This way, we better approximate the components of T corresponding to processors with low time cycle.

In order to ensure that the inequalities $r_i t_{i,j} c_j \le 1$ are fulfilled, we divide each c_j by the largest component of the jth column of the matrix $r_i t_{i,j} c_j$. Then, in order to avoid idle time, we divide each r_i by the largest component of the ith row of the matrix $r_i t_{i,j} c_j$. Thus, we obtain two vectors r and c satisfying $\forall i, j, \ r_i t_{i,j} c_j \le 1$ and such that

$$\begin{cases} \forall i, \exists j & r_i t_{i,j} c_j = 1 \\ \forall j, \exists i & r_i t_{i,j} c_j = 1. \end{cases}$$

Consider again the matrix

$$T = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}.$$

We obtain

$$r = \begin{pmatrix} 1.1661 \\ 0.3675 \\ 0.2100 \end{pmatrix}, c = \begin{pmatrix} 0.6803 \\ 0.4288 \\ 0.2859 \end{pmatrix},$$

and

$$T_{exe} = (r_i t_{i,j} c_j)_{i,j} = \begin{pmatrix} 0.7933 & 1 & 1 \\ 1 & 0.7879 & 0.6303 \\ 1 & 0.7203 & 0.5402 \end{pmatrix}.$$

The value of the objective function $(\sum r_i)(\sum c_j)$ is 2.4322.

Note that this allocation can easily be improved by using the process described in "*Correctness of the algorithm*" of Section 4.3.3.

Iterative Refinement. In this section, we propose an iterative refinement in order to obtain a better arrangement of the processors (and a better solution to the corresponding optimization problem). In order to determine the new arrangement matrix, we compute the matrix $T^{opt} = (\frac{1}{r_i \, c_j})_{i,j}$. T^{opt} is a rank-1 matrix whose components are the processor cycle-times that are optimal with respect to the vectors r and c computed above. In the case of our example, we obtain

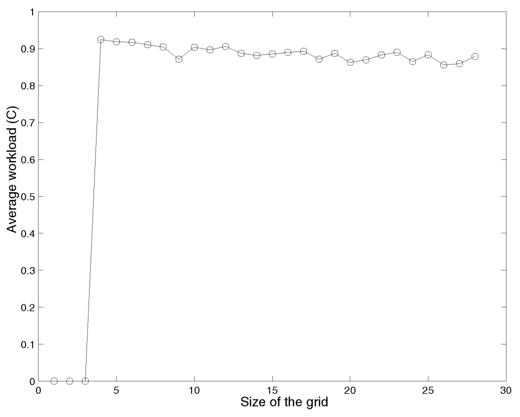


Fig. 12. Evaluation of the allocation. Comparison with the absolute lower bound.

$$T^{opt} = \begin{pmatrix} 1.2606 & 2.0000 & 3.0000 \\ 4.0000 & 6.3464 & 9.5195 \\ 7.0000 & 11.1061 & 16.6592 \end{pmatrix}.$$

This suggests the choice of another arrangement matrix T for the processors' cycle-times that better fits to the matrix T^{opt}. More precisely, we derive the new matrix T from the following conditions:

$$\forall i,j,k,l, \quad t_{i,j} \leq t_{k,l} \Longleftrightarrow t^{opt}_{i,j} \leq t^{opt}_{k,l}.$$

Then, we compute the r_is and the c_js as shown in the previous paragraph and we restart the process. We consider that the process has converged when no modification occurs in the matrix T. In our example, after the second step, the arrangement matrix T becomes

$$T = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 7 \\ 6 & 8 & 9 \end{pmatrix}$$

and the value of the objective function $(\sum r_i)(\sum c_j)$ is 2.5065 (instead of 2.4322). Convergence is obtained after three steps. The value of the objective function $(\sum r_i)(\sum c_j)$ is then 2.5889 and the corresponding arrangement matrix is

$$T = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 6 & 8 \\ 5 & 7 & 9 \end{pmatrix}.$$

Simulation Results. In this section, we present the results of our algorithm for processors with random cycle-times in $[0,1]$. We consider the arrangement of n^2 processors into an $n \times n$ grid. Fig. 12 displays the evolution with n of the objective function

$$\widehat{C}(r_1, r_2, \ldots, r_p, c_1, \ldots, c_q) = \frac{(\sum_{i=1}^{p} r_i)(\sum_{j=1}^{q} c_j)}{\sum_{i,j} \frac{1}{t_{ij}}}.$$

Indeed, during one unit of time:

- The amount of work done using this allocation $\widehat{C}(r_1, r_2, \ldots, r_p, c_1, \ldots, c_q)$ is $(\sum_{i=1}^{p} r_i)(\sum_{j=1}^{q} c_j)$.
- The maximal amount of work that each processor P_{ij} could do (not necessarily reachable) is $\frac{1}{t_{ij}}$.

Fig. 13 displays the evolution with n of the ratio

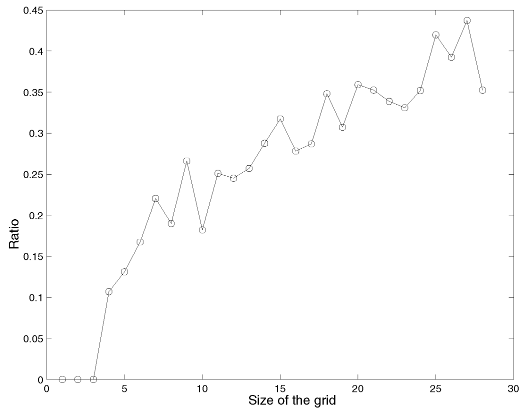


Fig. 13. Evolution of the ratio τ.

$$\tau = \frac{((\sum r_i)(\sum c_j))_{\text{after convergence}}}{((\sum r_i)(\sum c_j))_{\text{after the first step}}} - 1.$$

Finally, Fig. 14 displays the evolution with n of the average number of steps necessary to reach convergence.

Concluding Remarks on the Heuristic Solution. The heuristic solution gives satisfying results. Nevertheless, the algorithm does not converge to an optimal solution with respect to the optimization problem. Moreover, it seems that the number of steps of the iterative process grows with n and, therefore, involves more than $O(n^3)$ flops to arrange n^2 processors into an $n \times n$ grid. Nevertheless, one usually obtains satisfying results after only a few steps.

5 MPI Experiments

5.1 Estimating Processor Speed

There are too many parameters to accurately predict the actual speed of a machine for a given program, even assuming that the machine load will remain the same throughout the computation. Cycle-times must be understood as *normalized cycle-times* [13], i.e., application-dependent elemental computation times, which are to be computed via small-scale experiments (repeated several times, with an averaging of the results).

We ranked the computers to be used with respect to performance by measuring the time necessary to multiply two matrices of order 500 in double precision arithmetic.

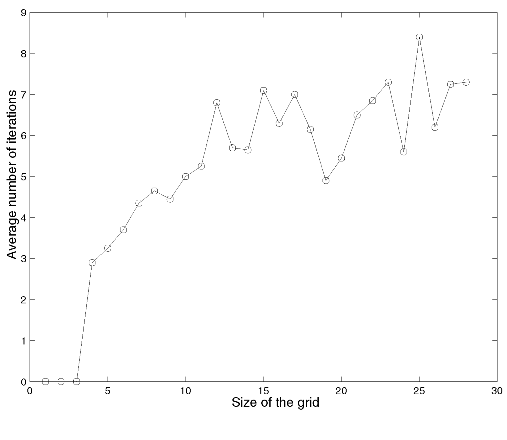


Fig. 14. Number of iterations.

TABLE 3
Specifications of the Nine Heterogeneous Processors

Name	Mhz	500x500 Matrix Multiply (Mflops)
texas	507	362
alabama	507	357
mississipi	498	357
onyx	431	305
cotnari	351	250
cuervo	200	134
muddy-waters	402	287
pink-floyd	402	284
petit-gris	200	128

This sorting criterion is certainly adequate for parallel dense linear algebra such as the LU and QR decompositions studied in this paper. On the Intel Pentium processor-based computers, we used the ATLAS [37] software, which automatically generates an efficient BLAS implementation for this architecture in particular. For example, on a 500 Mhz Pentium III computer, ATLAS 3.0 achieves a performance of 362 Mflops for multiplying two matrices of order 500. The higher level of cache was flushed between every timing measure and we used the operating system wallclock timer. On the Sun Sparc workstations, we used the BLAS library supplied by the vendor (sunperf) and we similarly measured their performance. Table 3 lists the computers we used in our experiments, their peak performance, and the performance they achieved with respect to our ranking test.

5.2 Numerical Results

In this section, we compare the classical block cyclic distribution of ScaLAPACK with the heterogeneous 2D distribution that we propose. The tests have been performed both on matrix multiplication and QR decomposition, using the nine processors listed in Table 3. The speed of the slowest processor is 128, while the speed of the fastest processor is 362. The ratio $\frac{362}{128} \approx 2.8$ shows that the processor set is reasonably heterogeneous: In fact, it mixes machines that are two years older than others.

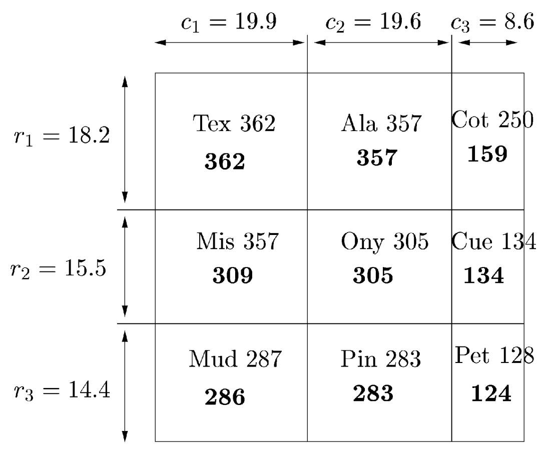


Fig. 15. Heterogeneous distribution. Bold entries represent the actual work allocated to the processors.

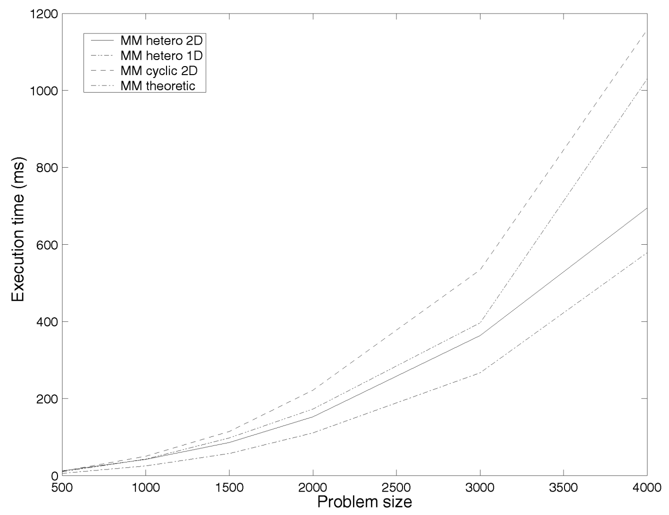


Fig. 16. Comparison of different allocations for matrix multiplication using nine processors.

For the largest experiment with nine processors, the heuristic of Section 4.3.4 leads to the distribution depicted in Fig. 15. This distribution theoretically enables us to update $(\sum_i r_i) \times (\sum_j c_j) = 2318.44$ elements at each time step. With the classical homogeneous distribution, the whole process is slowed down to the pace of the slowest processor and, therefore, we can only expect to update $9 \times 128 = 1,152$ elements at each time step. The theoretical improvement is therefore $\frac{2318.44}{1152} = 2.01$.

We display in Figs. 16 and 17 the time of matrix multiplication and QR decomposition for different problem sizes. There are three experimental plots in each figure, corresponding to our unidimensional heterogeneous distribution, our two-dimensional heterogeneous distribution, and the ScaLAPACK two-dimensional block-cyclic distribution. We also report the theoretical execution time expected for the two-dimensional heterogeneous distribution (if communication costs could be neglected). For each kernel, the heterogeneous strategy on a 2D grid is better

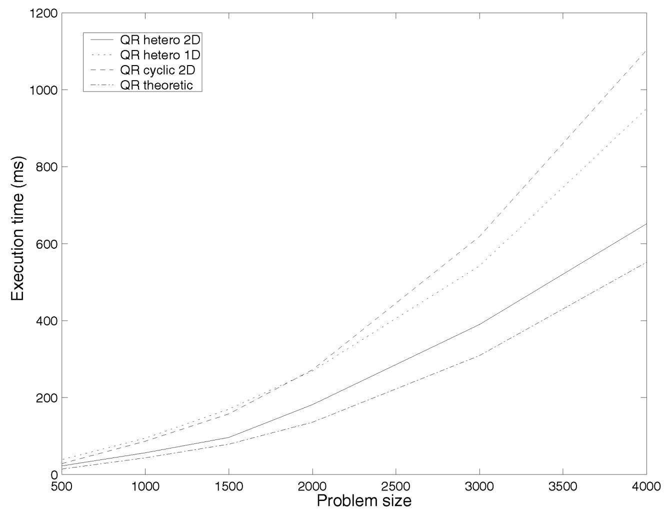


Fig. 17. Comparison of different allocations for QR decomposition using nine processors.

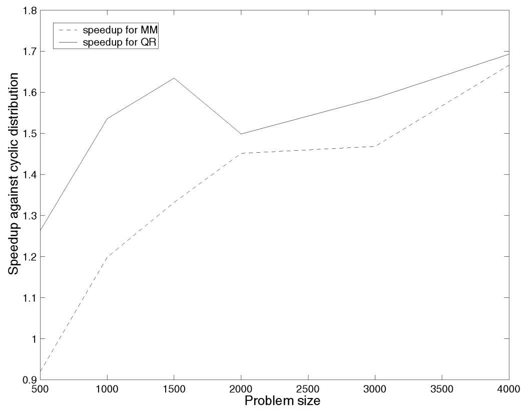


Fig. 18. Evolution of the speedup against cyclic distribution with the matrix size for matrix multiplication and QR decomposition using nine processors. The theoretical upper bound is 2.01.

than the heterogeneous strategy for a unidimensional grid, which in turn is better than with the block-cyclic distribution on a 2D grid.

Of course, the results displayed in Figs. 16 and 17 show that this theoretical speedup is not reached. This is due to the communications involved in the algorithms. All the computations have been performed with a slow network (100-base Ethernet) and the cost of communications is important, especially when involved matrices are small. Nevertheless, we can observe in Fig. 18 that the largest the involved matrices, the closer the actual speedup to the theoretical improvement.

6 Conclusion

In this paper, we have discussed data allocation strategies to implement matrix products and dense linear system solvers on heterogeneous computing platforms. Such platforms are likely to play an important role in the future. We have shown both theoretically and experimentally (through MPI experiments) that our data allocation algorithms were quite satisfactory. They form the basis for a successful extension of the ScaLAPACK library to heterogeneous platforms.

We point out that extending the standard ScaLAPACK block-cyclic distribution to heterogeneous 2D grids has turned out to be surprisingly difficult. In most cases, a perfect balancing of the load between all processors cannot be achieved and deciding how to arrange the processors along the 2D grid is a challenging NP-complete problem. But, we have formally stated the optimization problem to be solved and we have presented both an exact solution (with exponential cost) and a heuristic solution.

We believe that the original motivation for this work, namely the extension of ScaLAPACK, will prove very important in the (much larger) context of metacomputing: Dense linear algebra algorithms are the prototype of tightly coupled kernels that need to be implemented efficiently on collections of distributed and heterogeneous platforms: We view them as a perfect testbed before experimenting on more challenging computational problems on the grid. Future work will indeed be related to using *collections* of heterogeneous clusters rather than a single one. Implementing linear algebra kernels on several collections of workstations or parallel servers, scattered all around the world and connected through fast but nondedicated links, would give rise to a "Computational Grid ScaLAPACK." Our results constitute a first step toward achieving this ambitious goal.

Acknowledgments

The authors thank the reviewers whose comments and suggestions have greatly improved the presentation of the paper. Antoine Petitet was a member of the Innovative Computing Laboratory, Computer Science Department, University of Tennessee at Knoxville while this work was conducted.

References

[1] R. Agarwal, F. Gustavson, and M. Zubair, "A High Performance Matrix Multiplication Algorithm on a Distributed-Memory Parallel Computer, Using Overlapped Communication," *IBM J. Research and Development,* vol. 38, no. 6, pp. 673-681, 1994.

[2] S. Anastasiadis and K.C. Sevcik, "Parallel Application Scheduling on Networks of Workstations," *J. Parallel and Distributed Computing,* vol. 43, pp. 109-124, 1997.

[3] D. Arapov, A. Kalinov, A. Lastovetsky, and I. Ledovskih, "A Parallel Language and Its Programming System for Heterogeneous Networks," *Concurrency: Practice and Experience,* vol. 12, no. 13, pp. 1317-1343, 2000.

[4] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi, *Complexity and Approximation.* Berlin: Springer, 1999.

[5] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert, "Partitioning a Square into Rectangles: Np-Completeness and Approximation Algorithms," Technical Report RR-2000-10, LIP, ENS Lyon, Feb. 2000.

[6] F. Berman, "High-Performance Schedulers," *The Grid: Blueprint for a New Computing Infrastructure,* I. Foster and C. Kesselman, eds., pp. 279-309, Morgan-Kaufmann, 1999.

[7] L. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley, "Scalapack: A Portable Linear Algebra Library for Distributed-Memory Computers—Design Issues and Performance," *Proc. Supercomputing '96,* 1996.

[8] 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.

[9] P. Boulet, J. Dongarra, Y. Robert, and F. Vivien, "Static Tiling for Heterogeneous Computing Platforms," *Parallel Computing,* vol. 25, pp. 547-568, 1999.

[10] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley, "ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers—Design Issues and Performance," *Computer Physics Comm.,* vol. 97, pp. 1-15, 1996 (also LAPACK Working Note #95).

[11] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R.C. Whaley, "The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines," *Scientific Programming,* vol. 5, pp. 173-184, 1996.

[12] E. Chu and A. George, "QR Factorization of a Dense Matrix on a Hypercube Multiprocessor," *SIAM J. Scientific and Statistical Computing,* vol. 11, pp. 990-1028, 1990.

[13] M. Cierniak, M.J. Zaki, and W. Li, "Scheduling Algorithms for Heterogeneous Network of Workstations," *The Computer J.,* vol. 40, no. 6, pp. 356-372, 1997.

[14] P.E. Crandall, "The Limited Applicability of Block Decomposition in Cluster Computing," *Proc. Fourth IEEE Int'l Symp. High Performance Distributed Computing,* pp. 102-109, 1995.

[15] P.E. Crandall and M.J. Quinn, "Block Data Decomposition for Data-Parallel Programming on a Heterogeneous Workstation Network," *Proc. Second Int'l Symp. High Performance Distributed Computing,* pp. 42-49, 1993.

[16] P. Crescenzi and V. Kann, "A Compendium of NP Optimization Problems," http://www.nada.kth.se/~viggo/wwwcompendium/wwwcompendium.html, 2001.

[17] D.E. Culler and J.P. Singh, *Parallel Computer Architecture: A Hardware/Software Approach.* San Francisco: Morgan Kaufmann, 1999.

[18] J. Dongarra, R. van de Geijn, and D. Walker, "Scalability Issues in the Design of a Library for Dense Linear Algebra," *J. Parallel and Distributed Computing,* vol. 22, no. 3, pp. 523-537, 1994.

[19] J.J. Dongarra and D.W. Walker, "Software Libraries for Linear Algebra Computations on High Performance Computers," *SIAM Review,* vol. 37, no. 2, pp. 151-180, 1995.

[20] *The Grid: Blueprint for a New Computing Infrastructure,* I. Foster and C. Kesselman, eds. Morgan-Kaufmann, 1999.

[21] G. Fox, S. Otto, and A. Hey, "Matrix Algorithms on a Hypercube I: Matrix Multiplication," *Parallel Computing,* vol. 3, pp. 17-31, 1987.

[22] M.R. Garey and D.S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness.* W.H. Freeman, 1991.

[23] G.H. Golub and C.F. Van Loan, *Matrix Computations,* second ed. Johns Hopkins, 1989.

[24] T.F. Gonzalez and S. Zheng, "Approximation Algorithm for Partitioning a Rectangle with Interior Points," *Algorithmica,* vol. 5, pp. 11-42, 1990.

[25] M. Grigni and F. Manne, "On the Complexity of the Generalized Block Distribution," *Proc. Parallel Algorithms for Irregularly Structured Problems, Third Int'l Workshop IRREGULAR '96,* pp. 319-326, 1996.

[26] M. Iverson and F. Özgüner, "Dynamic, Competitive Scheduling of Multiple Dags in a Distributed Heterogeneous Environment," *Proc. Seventh Heterogeneous Computing Workshop,* 1998.

[27] M. Kaddoura, S. Ranka, and A. Wang, "Array Decompositions for Nonuniform Computational Environments," *J. Parallel and Distributed Computing,* vol. 36, no. 2, pp. 91-105, 1996.

[28] A. Kalinov and A. Lastovetsky, "Heterogeneous Distribution of Computations while Solving Linear Algebra Problems on Networks of Heterogeneous Computers," *Proc. HPCN Europe 1999,* P. Sloot, M. Bubak, A. Hoekstra, and B. Hertzberger, eds., pp. 191-200, 1999.

[29] S. Khanna, S. Muthukrishnan, and M. Paterson, "On Approximating Rectangle Tiling and Packing," *Proc. Ninth Ann. ACM-SIAM Symp. Discrete Algorithms,* pp. 384-393, 1998.

[30] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing.* Benjamin/Cummings, 1994.

[31] A. Lingas, R.Y. Pinter, R.L. Rivest, and A. Shamir, "Minimum Edge Length Partitioning of Rectilinear Polygons," *Proc. 20th Ann. Allerton Conf. Comm., Control, and Computing,* 1982.

[32] M. Maheswaran and H.J. Siegel, "A Dynamic Matching and Scheduling Algorithm for Heterogeneous Computing Systems," *Proc. Seventh Heterogeneous Computing Workshop,* 1998.

[33] H.J. Siegel, H.G. Dietz, and J.K. Antonio, "Software Support for Heterogeneous Computing," *ACM Computing Surveys,* vol. 28, no. 1, pp. 237-239, 1996.

[34] G.C. Sih and E.A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Trans. Parallel and Distributed Systems,* vol. 4, no. 2, pp. 175-187, Feb. 1993.

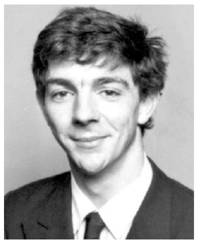
[35] M. Tan, H.J. Siegel, J.K. Antonio, and Y.A. Li, "Minimizing the Application Execution Time through Scheduling of Subtasks and Communication Traffic in a Heterogeneous Computing System," *IEEE Trans. Parallel and Distributed Systems,* vol. 8, no. 8, pp. 857-871, Aug. 1997.

[36] J.B. Weissman and X. Zhao, "Scheduling Parallel Applications in Distributed Networks," *Cluster Computing,* vol. 1, no. 1, pp. 109-118, 1998.

[37] R.C. Whaley and J. Dongarra, "Automatically Tuned Linear Algebra Software," *Proc. Supercomputing '98,* 1998.

Olivier Beaumont obtained the PhD thesis from the Université de Rennes in January 1999. He is currently an associate professor in the Computer Science Laboratory LIP at ENS Lyon. His main research interests are parallel algorithms on distributed memory architectures.

Vincent Boudet is currently a PhD student in the Computer Science Laboratory LIP at ENS Lyon. He is mainly interested in algorithm design and in compilation-parallelization techniques for distributed memory architectures.



Antoine Petitet received the Engineer of Computer Science degree from ENSEEIHT Toulouse, France, in 1990. He was awarded the PhD degree in computer science from the University of Tennessee, Knoxville, in 1996. He is currently a benchmark engineer at Sun France. His research interests primarily focus on parallel computing, numerical linear algebra and the design of parallel numerical software libraries for distributed memory concurrent computers.

Fabrice Rastello obtained the PhD degree from the Ecole Normale Supérieure de Lyon in September 2000. He is currently working as an engineer for ST Microelectronics in Grenoble. His main research interests are parallel algorithms for distributed memory architectures and automatic compilation/parallelization techniques.

Yves Robert obtained the PhD degree from the Institut National Polytechnique de Grenoble in January 1986. He is currently a full professor in the Computer Science Laboratory LIP at ENS Lyon. He is the author of three books, 60+ papers published in international journals, and 70+ papers published in international conferences. His main research interests are parallel algorithms for distributed memory architectures and automatic compilation/parallelization techniques. He is a member of the ACM and the IEEE.