Data Allocation Strategies for Dense Linear Algebra Kernels on Heterogeneous Two-dimensional Grids.

Olivier Beaumont, Vincent Boudet, Fabrice Rastello and Yves Robert
LIP, UMR CNRS–ENS Lyon–INRIA 5668
Ecole Normale Supérieure de Lyon
F - 69364 Lyon Cedex 07
e-mail: Vincent.Boudet@ens-lyon.fr

Abstract

We study the implementation of dense linear algebra computations, such as matrix multiplication and linear system solvers, on two-dimensional (2D) grids of heterogeneous processors. For these operations, 2D-grids are the key to scalability and efficiency. The uniform block-cyclic data distribution scheme commonly used for homogeneous collections of processors limits the performance of these operations 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 2D-grids with respect to the performance of the processors. The practical usefulness of these strategies is fully demonstrated by experimental data for a heterogeneous network of workstations.

Key words: heterogeneous network, heterogeneous grid, different-speed processors, load-balancing, data distribution, data allocation, numerical libraries.

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 *as well as* more recent ones. In addition, parallel machines used in a multi-user environment exhibit the very characteristics of a HNOW: different loads imply different processor speeds when running a parallel application, even though all processors are identical. Note that multi-user parallel machines are interesting in this context because they may exhibit a better communication-to-computation ratio than Ethernet-based networks.

The major limitation to programming heterogeneous platforms arises from the additional difficulty of balancing the load when using processors running at different speed. In this paper, we explore several possibilities to implement linear algebra kernels on HNOWs. We have been exploring data allocation strategies for HNOWs arranged as a unidimensional (linear) array in previous papers [8, 9]. Arranging the processors along a two-dimensional grid turns out to be surprisingly difficult. There are two complicated problems to solve: (i) how to arrange the heterogeneous processors along a 2D-grid; (ii) how to distribute matrix blocks to the processors once the grid is built. The major contribution of this paper is to provide an efficient solution to both problems, thereby providing the required framework to build an extension of the ScaLAPACK library [6] capable of running on top of HNOWs or non-dedicated parallel machines.

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. In Section 3 we summarize existing algorithms for matrix multiplication and dense linear solvers on 2D (homogeneous) grids. In Section 4 we propose data allocation strategies for implementing the previous kernels on 2D heterogeneous grids. We give some final remarks and conclusions in Section 5. Due to the lack of space we do not report any MPI experiments in this paper:

see [7] for such experiments.

2. Framework

2.1. Static Versus Dynamic Strategies

Because we have a library designer's approach, we target static strategies to allocate data and computations to the processors. In fact, 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, communication costs and control overhead may well lead to slow the whole process down to the pace of the slowest processors. 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 data allocation.

However, to be successful, static strategies must obey a more refined model than standard block-cyclic distributions: such distributions are well-suited to processors of equal speed but lead to a great load imbalance between processors of different speed.

2.2. Machine Model

Our machine model is either a heterogeneous network of workstations of different speeds, or a parallel computer with multiple users. In this latter case, we assume different loads on the different processors, thereby considering the parallel computer as a heterogeneous machine (and calling it a HNOW as well). In both cases, we have to model the communication links. For HNOWS interconnected with a standard Ethernet network, all communications are inherently sequential, while for Myrinet or switched networks, independent communications can take place in parallel. In all cases, we consider that the communications performed by one processor are sequential.

In any case, we will configure the HNOW as a (virtual) 2D grid for scalability reasons [10]. We come back to this point in Section 3 when describing the well-known blocked matrix multiplication and LU or QR decomposition algorithms.

2.3. Related Work

There is a great deal of papers in the literature dealing with dynamic schedulers to distribute the computations (together with the associated data) onto heterogeneous platforms. Most schedulers use naive 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 [4] and the more specialized references [2, 13] for further details. Several scheduling and mapping heuristics have been proposed to map task graphs onto HNOWs [23, 24, 21, 18]. Scheduling tools such as Prophet [25] or AppLeS [4] are available (see also the survey paper [22]).

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 Lastovetky [19] 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 4 direct neighbors to communicate with. We come back on this distribution in Section 3.1.2.

3. Linear Algebra Kernels on 2D Grids

In this section we briefly recall the algorithms implemented in the ScaLAPACK library [6] on 2D homogeneous grids. Then we discuss how to modify the two-dimensional block-cyclic distribution which is used in ScaLAPACK to cope with 2D heterogeneous grids.

3.1. Matrix-matrix Multiplication

3.1.1 Homogeneous Grids

For the sake of simplicity we restrict to 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, 17, 20]. 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 $c_{i,j} += a_{i,k} \times b_{k,j}$.

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 algoritm [20]). Moreover, on a homogeneous grid, broadcasts are performed as independent ring broadcasts (along the rows and the columns), hence they can be pipelined.

Of course, ScaLAPACK uses a blocked version of this algorithm to squeeze the most out state-of-the-art processors with pipelined arithmetic units and multilevel memory hierarchy [16, 10]. Each matrix coefficient in the description above is replaced by a $r \times r$ square block, where optimal values of r depend on the communication-to-computation ratio of the target computer.

Finally, 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. An example is given in Figure 1 with $p = q = 4$ and $\lceil \frac{n}{r} \rceil = 10$.

1	2	3	4	1	2	3	4	1	2
5	6	7	8	5	6	7	8	5	6
9	10	11	12	9	10	11	12	9	10
13	14	15	16	13	14	15	16	13	14
1	2	3	4	1	2	3	4	1	2
5	6	7	8	5	6	7	8	5	6
9	10	11	12	9	10	11	12	9	10
13	14	15	16	13	14	15	16	13	14
1	2	3	4	1	2	3	4	1	2
5	6	7	8	5	6	7	8	5	6

Figure 1. Processors are numbered from 1 to 16. This figure represents the distribution of 10×10 matrix blocks onto 4×4 processors.

3.1.2 Heterogeneous Grids

Suppose now we have a $p \times q$ grid of heterogeneous processors. Instead of distributing the $r \times r$ matrix blocks cyclically along each grid dimension, we distribute *block panels* cyclically along each grid dimension. A block panel is a rectangle of consecutive $B_p \times B_q$ $r \times r$ blocks. See Figure 2 for an example with $B_p = 4$ and $B_q = 3$: this panel of 12 $r \times r$ blocks will be distributed cyclically along both dimensions of the 2D grid. The previous cyclic dimension for homogeneous grids obviously corresponds to the case $B_p = p$ and $B_q = q$. Now, the distribution of individual blocks is no longer purely cyclic but remains periodic. We illustrate in Figure 3 how block panels are distributed on the 2D-grid.

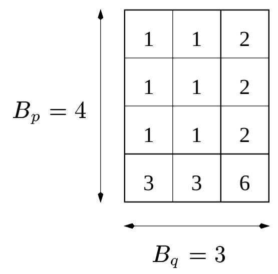

Figure 2. A block panel with $B_p = 4$ and $B_q = 3$. Each processor is labeled by its cycle-time, i.e the (normalized) time it needs to compute one $r \times r$ block: the processor labeled 1 is twice faster than the one labeled 2, hence it is assigned twice more blocks within each panel.

How many $r \times r$ blocks should be assigned to each processor within a panel ? Intuitively, as in the case of uni-dimensional 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 Figure 2, we have a 2×2 grid of processors of respective cycle-time $t_{1,1} = 1$, $t_{1,2} = 2$, $t_{2,1} = 3$ and $t_{2,2} = 6$. The allocation of the $B_p \times B_q = 4 \times 3 = 12$ blocks of the panel perfectly balances the load amongst 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

1	1	2	1	1	2	1	1	2	1
1	1	2	1	1	2	1	1	2	1
1	1	2	1	1	2	1	1	2	1
3	3	6	3	3	6	3	3	6	3
1	1	2	1	1	2	1	1	2	1
1	1	2	1	1	2	1	1	2	1
1	1	2	1	1	2	1	1	2	1
3	3	6	3	3	6	3	3	6	3
1	1	2	1	1	2	1	1	2	1
1	1	2	1	1	2	1	1	2	1

Figure 3. Allocating 4×3 panels on a 2×2 grid (processors are labeled by their cycle-time). There is a total of 10×10 matrix blocks.

not hold, additional communications will be needed, as illustrated in Figure 4.

Translated in terms of $r \times r$ matrix blocks, the above conditions mean that each processor P_{ij}, $1 \leq j \leq q$ in the i-th grid row must receive the same number r_i of blocks. Similarly, P_{ij}, $1 \leq i \leq p$ must receive c_j blocks. This condition does hold in the example of Figure 3, hence each processor only communicates with its direct neighbors.

Unfortunately, and in contrast with the uni-dimensional case, the additional constraints induced by the communication pattern may well prevent to achieve a perfect load balance amongst processors. Coming back to Figure 2, 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 Figure 2, 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 $r_i \times c_j \times t_{ij}$. To have a perfect load balance, we have to fulfill the following equations:

$$r_1 \times t_{11} \times c_1 = r_1 \times t_{12} \times c_2 = r_2 \times t_{21} \times c_1 = r_2 \times t_{22} \times c_2$$

$$\text{that is} \quad r_1 c_1 = 2 r_1 c_2 = 3 r_2 c_1 = 6 r_2 c_2.$$

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 uni-dimensional scheme); next we balance the load between columns (using the uni-dimensional scheme again, weighting each column by the inverse of the harmonic mean of the cycle-times of the processors within the column, see below). This is the "heterogeneous block cyclic distribution" of Kalinov and Lastovetky [19], which leads to the solution of Figure 4. 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.

⋮	⋮
1	2
1	5
1	2
3	2
1	2
1	2
1	2
3	5

Figure 4. The distribution of Kalinov and Lastovetky. Two consecutive columns are represented here. Processors have two west neighbors instead of one.

We use the example to explain with further details how the heterogeneous block cyclic distribution of Kalinov and Lastovetky [19] works. First they balance the load in each processor column independently, using the uni-dimensional 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 5 out of every 7 matrix rows, while P_{22} (cycle-time $t_{22} = 5$) should receive the remaining 2 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 have a library designer's approach, 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 Lastovetky'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. Solving this optimization problem is the objective of Section 4.

3.2. 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.

3.2.1 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 and assume that the LU factorization of the $k \times b$ first columns has proceeded with $k \in \{0, 1, \ldots \frac{N-1}{b}\}$. During the next step, the algorithm factors the next panel of r 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 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-r 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 $\bar{A} \leftarrow \bar{A} - LU$ that 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

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 [15, 10, 5] for a detailed performance analysis of the right-looking variants, that demonstrates their good scalability property. The parallelization of the QR decomposition is analogous [12, 11]

3.2.2 Heterogeneous Grids

For the implementation of the LU and QR decomposition algorithms on a heterogeneous 2D grid, we modify the ScaLAPACK $CYCLIC(r)$ distribution very similarly as for the matrix-matrix multiplication problem. The intuitive reason is the following: as pointed out before, the core of the LU and QR decompositions is a rank-r update, hence the techniques for the outer-product matrix algorithm naturally apply.

We still use block panels made up with several $r \times r$ matrix blocks. The block panels are distributed cyclically along both dimensions of the grid. The only modification if 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:

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

Say we use a panel with $B_p = 8$ ad $B_q = 6$, i.e. a panel composed of 48 blocks. Using the methods described below (see Section 4), we assign he blocks as follows:

- Within each panel column, the first processor row receives 6 blocks and the second processor rows receives 2 blocks
- Out of the 6 panel columns, the first grid column receives 4 and the second grid column receive 2

This allocation is represented in Figure 5. 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 processors execute the same amount of (independent) computations at each step of the algorithm. For the LU and QR decomposition algorithms, the ordering of the columns is quite important. In the example, the first processor column operates like 6 processors of cycle-time 1 and 2 processors of cycle-time 3, which is equivalent to a single processor A of cycle-time $\frac{3}{20}$; the second processor column operates like 6 processors of cycle-time 2 and 2 processors of cycle-time 5, which is equivalent to a single processor B of cycle-time $\frac{5}{17}$. The uni-dimensional algorithm allocates the six panel columns as $ABAABA$, and we retrieve the allocation of Figure 5.

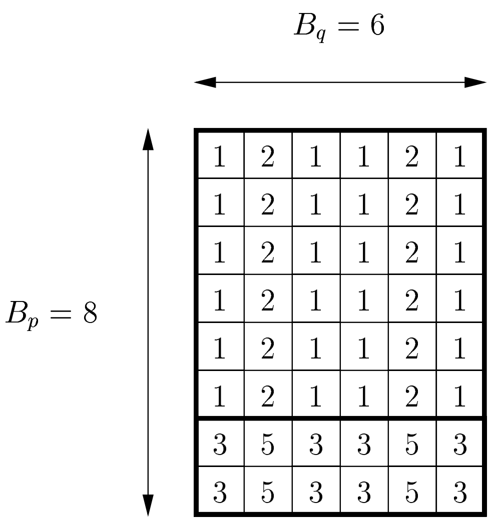

Figure 5. Allocation of the blocks within a block panel with $B_p = 8$ and $B_q = 6$. Each processor of the 2×2 grid is labeled by its cycle-time.

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 it is not the case. 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. Solving the 2D Heterogeneous Grid Allocation Problem

4.1. 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 $Z = XY$ 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 re-number the processors as P_{ij}, with cycle-time t_{ij}, $1 \leq i \leq p, 1 \leq j \leq q$. 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 Z matrix: see Figure 6 for an example.

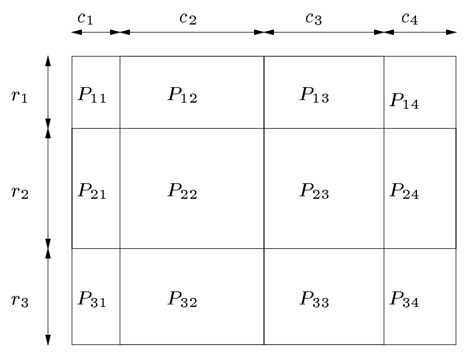

Figure 6. Allocating computations to processors on a 3×4 grid

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 Z, 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 is 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

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

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):

Objective Obj_1: $\min_{(\sum_i r_i=1; \sum_j c_j=1)} \{r_i \times t_{ij} \times c_j\}$

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 Opt_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 $r_i \times t_{ij} \times c_j \leq 1$ to ensure that P_{ij} can process its block within one cycle. Since the total number of data elements being processed is $(\sum_{i=1}^{p} r_i) \times \left(\sum_{j=1}^{q} c_j\right)$, we get the (equivalent) optimization problem:

Objective Obj_2: $\max_{r_i \times t_{ij} \times c_j \leq 1} \{(\sum_i r_i) \times (\sum_j c_j)\}$

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_i's 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:

$$\max_{r_i \times t_{ij} \times c_j \leq 1} \{(\sum_{i=1}^{p} r_i) \times (\sum_{j=1}^{q} c_j)\}$$
$$= \max_{r_i}\{ \max_{c_j \,\mathrm{with}\, r_i \times t_{ij} \times c_j \leq 1} \{(\sum_{i=1}^{p} r_i) \times (\sum_{j=1}^{q} c_j)\}\}$$
$$= \max_{r_i}\{(\sum_{i=1}^{p} r_i) \times \max_{c_j \,\mathrm{with}\, r_i \times t_{ij} \times c_j \leq 1} \{(\sum_{j=1}^{q} c_j)\}\}$$
$$= \max_{r_i}\{(\sum_{i=1}^{p} r_i) \times \max_{c_j \leq \frac{1}{r_i \times t_{ij}}} \{(\sum_{j=1}^{q} c_j)\}\}$$
$$= \max_{r_i}\{(\sum_{i=1}^{p} r_i) \times (\sum_{j=1}^{q} \min_i\{\frac{1}{r_i \times t_{ij}}\})\}$$
$$= \max_{r_i}\{(\sum_{i=1}^{p} r_i) \times (\sum_{j=1}^{q} \frac{1}{\max_i\{r_i \times t_{ij}\}})\}$$

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 an heuristic, is the main objective of Section 4.3.

Conjecture We make guesses about the NP-completeness of this problem. The reason is very simple. The number of transpositions of the $p \times q$ processors is exponential. Furthermore, when we have choose a transposition of the processors, find the correct values for the r_i and the c_j appears as an hard problem.

The 2D load-balancing problem In the next sections we give a solution to the 2D load-balancing problem which can be stated as follows: given $n = p \times q$ processors, how to 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-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.

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 introduce a heuristic to give a fast but sub-optimal solution to the 2D load-balancing problem.

4.2. Reduction to Non-Decreasing Arrangements

The arrangement of the processors along the grid is a degree of freedom of the problem. For example when using a Myrinet network [14] we can define every desired topology for a fixed degree (number of neighbors) of the interconnection graph. Hence, finding a good arrangement is a key step of the load-balancing problem.

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

Theorem 1 *There exists a non-increasing arrangement which is optimal.*

Proof The proof works as follows:

1. Let the $p \times q$ cycle-times be denoted as $t_1, t_2, t_{p\times q}$
2. Consider an optimal arrangement of the grid. There is no reason that the optimal arrangement be a non-decreasing 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 non-decreasing arrangement

We need a few definitions:

Definition 1 Arrangement *An arrangement is a one-to-one mapping*

$$\sigma : \left(\begin{array}{rcl} \{1\ldots pq\} & \mapsto & \{1\ldots p\} \times \{1 \ldots q\} \\ k & \to & \sigma(k) = (i,j) \end{array} \right)$$

which assigns a position to each processor in the grid.

Non-decreasing arrangement *An arrangement σ is non-decreasing if $t_{\sigma^{-1}(i,j)} \le t_{\sigma^{-1}(i',j')}$ for all $(1,1) \le (i,j) \le (i',j') \le (p,q)$*

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 of correct transpositions that modifies σ into a non-decreasing arrangement. To prove this, we use a *weight function* W that quantifies the distance to the "non-decreasing-ness", so to speak: a correct transposition will decrease the weight of the arrangement it is applied to. We search for

$$W : \left(\begin{array}{rcl} \Sigma_{p,q} & \to & \mathcal{N} \\ \sigma & \to & W(\sigma) \end{array} \right)$$

(where $\Sigma_{p,q}$ is the set of all arrangements and $\mathcal{N}$ represents the set of positive integers) such that for each correct transposition τ, $W(\tau(\sigma)) < W(\sigma)$. We choose

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

To check that W has the desired property, let σ be an arrangement such that $(i,j) \le (i',j')$ 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

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

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 $r_1 \ge r_2 \ge \ldots r_p$ and $c_1 \ge c_2 \ge \ldots c_q$. If σ is non-decreasing, we are done. Otherwise, there exists $(1,1) \le (i,j) < (p,q)$ such that either $t_{\sigma^{-1}(i+1,j)} < t_{\sigma^{-1}(i,j)}$ or $t_{\sigma^{-1}(i,j+1)} < t_{\sigma^{-1}(i,j)}$. The proof is the same in both cases, hence assume that $t_{\sigma^{-1}(i+1,j)} < t_{\sigma^{-1}(i,j)}$. Let $k = \sigma^{-1}(i,j)$ and $l = \sigma^{-1}(i+1,j)$: 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, which achieves the quantity $\mathcal{O} = (\sum r_i \times \sum c_j$ subject to

$\max_{i,j}(r_i c_j t_{\sigma^{-1}(i,j)}) \leq 1$. We have $r_i \geq r_{i+1}$ and $t_{\sigma^{-1}(i+1,j)} < t_{\sigma^{-1}(i,j)}$, hence

$$r_i c_j t_{(\tau\circ\sigma)^{-1}(i,j)} = r_i c_j t_{\sigma^{-1}(i+1,j)} \leq r_i c_j t_{\sigma^{-1}(i,j)} \leq 1$$
$$r_{i+1} c_j t_{(\tau\circ\sigma)^{-1}(i+1,j)} = r_{i+1} c_j t_{\sigma^{-1}(i,j)} \leq r_i c_j t_{\sigma^{-1}(i,j)} \leq 1$$

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

4.3. Solution 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 non-decreasing 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 $r_1, \ldots, r_p$ and $c_1, \ldots, c_q$ be the solution to the optimization problem Obj_1

4.3.1 Spanning Trees

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 bipartite graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. There are $p+q$ vertices labeled with r_i and c_j and the graph is complete. 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

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

The spanning tree $\mathcal{T}$ is said to be *acceptable* if and only if all the remaining inequalities are satisfied: $\forall (r_i, c_j) \in \mathcal{E}, \;\; r_i t_{i,j} c_j = 1.$. 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:

Algorithm We generate all the spanning trees of $\mathcal{G}$. For a given tree $\mathcal{T}$, we first impose that $r_1 = 1$, then by walking on the tree we find the values for the other r_i and c_j. For example, if c_3 is connected to r_1 in $\mathcal{T}$, then we take $c_3 = \frac{1}{r_1 t_{13}}$. When we have a value for all r_i and c_j, we check if the tree is acceptable. Finally, we select the acceptable tree that maximizes $\sum_{1\leq i\leq p} r_i \sum_{1\leq j\leq q} c_j$.

Justification 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 ($(r_i, c_j) \in \mathcal{E}'$) iff $r_i c_j t_{i,j} = 1$. If $\mathcal{U}$ is connected, we are done. Otherwise, we decrease the number of connected components in $\mathcal{U}$, assigning new values to some r_i and c_j, but without decreasing the quantity $\sum_{1\leq i\leq p} r_i \sum_{1\leq j\leq q} c_j$. Let $\mathcal{V}'$ be a connected component of $\mathcal{U}$. Suppose (without any loss of generality) that $\mathcal{V}' = \{r_1, \ldots, r_{p'}, c_1, \ldots, c_{q'}\}$ with $p' < p$.

First case If $q' = q$, it means that for all $1 \leq j \leq q$, $r_{p'+1} c_j t_{p'+1,j} < 1$. Let $\alpha = \min_{1\leq j\leq q} \frac{1}{r_{p'+1} c_j t_{p'+1,j}}$. Then, $r_{p'+1}$ can be increased by a factor of α, and the product $\sum_{1\leq i\leq p} r_i \sum_{1\leq j\leq q} c_j$ is increased too. Moreover, the vertex $r_{p'+1}$ becomes connected to the component $\mathcal{V}'$. So the number of connected components is decreased by one.

Second case If $q' < q$, it means that for all $(p'+1, 1) \leq (i,j) \leq (p, q')$ and for all $(1, q'+1) \leq (i,j) \leq (p', q)$, $r_i c_j t_{i,j} < 1$. Let $\alpha_r = \min_{(1,q'+1)\leq(i,j)\leq(p',q)} \frac{1}{r_i c_j t_{i,j}}$ and $\alpha_c = \min_{(p'+1,1)\leq(i,j)\leq(p,q')} \frac{1}{r_i c_j t_{i,j}}$. We also introduce the notations $R_a = \sum_{1\leq i\leq p'} r_i$, $R_b = \sum_{p'+1\leq i\leq p} r_i$, $C_a = \sum_{1\leq j\leq q'} c_j$ and $C_b = \sum_{q'+1\leq j\leq q} c_j$.

Now, we have two possibilities to increase the connectivity of the graph: either increase R_b and decrease C_b by a factor of α_r or decrease R_b and increase C_b by a factor of α_c. We must check that at least one of this solution does increase the product $(R_a + R_b)(C_a + C_b)$. Indeed, consider the function

$$f : \begin{pmatrix} \mathcal{R}_+^* & \longmapsto & \mathcal{R}_+^* \\ \lambda & \longrightarrow & f(\lambda) = (\lambda R_b + R_a)(\frac{C_b}{\lambda} + C_a) \end{pmatrix}$$

Note that $f'(1) = R_b C_a - R_a C_b$. Moreover, f is a continuous function that is first decreasing to a minimum and then increasing. Therefore,

- if $f'(1) \geq 0$, then for all $\lambda \geq 1$, $f(\lambda) \geq f(1)$. In particular, $f(\alpha_r) > f(1)$.
- if $f'(1) \leq 0$, then for all $\lambda \leq 1$, $f(\lambda) \geq f(1)$. 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 $\sum_{1\leq i\leq p} r_i \sum_{1\leq j\leq q} c_j$. We conclude that there does exist an acceptable spanning tree whos value is the optimal solution.

To summarize, 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.

In the rest of this section we deal with two particular simple cases.

4.3.2 Case of a 2×2 Grid

For small size grids, we can find analytical solutions to the optimization problem. As an example, we will solve in this section the 2×2 case.

In the 2×2 case, we want to maximize (see the definition of Obj_2) the quantity $(r_1 + r_2)(\frac{1}{\max(r_1t_{11}, r_2t_{21})} + \frac{1}{\max(r_1t_{12}, r_2t_{22})})$. We normalize our problem by letting $r_1 = 1$ and $r_2 = r$. We have to maximize

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

There are three cases to study:

First case $0 \leq r \leq min(\frac{t_{11}}{t_{21}}, \frac{t_{12}}{t_{22}})$: 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 $max(\frac{t_{11}}{t_{21}}, \frac{t_{12}}{t_{22}}) \leq r \leq +\infty$: 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 $min(\frac{t_{11}}{t_{21}}, \frac{t_{12}}{t_{22}}) \leq r \leq max(\frac{t_{11}}{t_{21}}, \frac{t_{12}}{t_{22}})$: by symmetry, we can suppose that $\frac{t_{11}}{t_{21}} \leq \frac{t_{12}}{t_{22}}$. 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 2 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

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

4.3.3 Rank-1 Matrices

If the matrix $(t_{ij})_{1\leq i\leq p, 1\leq j\leq q}$ 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 $2 \leq i \leq p$ and $c_j = \frac{1}{t_{1j}}$ for $2 \leq i \leq q$. All the $p \times q$ inequalities $r_it_{ij}c_j$ are equalities, which means that all processors are fully utilized:

$$r_it_{ij}c_j = \frac{1}{t_{i1}}t_{ij} \times \frac{1}{t_{j1}}t_{ij} = 1,$$

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 do not exist, we can intuitively say that the optimal arrangement is the "closest one" to a rank-1 matrix.

4.4. Polynomial Heuristic

In this section, we study a polynomial heuristic to find an arrangement of the processors (and a solution to the corresponding optimization problem) that leads to a good load-balancing. The polynomial heuristic is based on the approximation of the arrangement matrix by a rank-1 matrix, for which the problem can easily be solved (see 4.3.3).

4.4.1 Arrangement of the processors

We are given the processors cycle times as input to the heuristic. We arrange the processors cycle times in the matrix T as follows

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

This arrangement usually leads to an arrangement matrix T which is close to a rank-1 matrix.
For instance, if we consider the case of 9 processors with cycle times $1, 2, \ldots, 9$, the arrangement matrix T we obtain is

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

4.4.2 Computing the r_i's and the c_j's

As noticed above (see 4.3.3), the computation of the r_i's and the c_j's is obvious when T is a rank-1

matrix. Our aim is to approximate T by a rank-1 matrix and then to compute the r_i's and the c_j's corresponding to this matrix.
Let us denote by

$$USV^t = T^{inv}$$

the singular value decomposition of T^{inv}, where $T^{inv} = (\frac{1}{t_{i,j}})_{i,j}$. It is known that the best approximation (in the sense of the l^2-norm) of T^{inv} by a rank-1 matrix is given by

$$sab^t,$$

where a and b are respectively the left and right singular vectors associated to s, the largest singular value of T^{inv}.
Thus, if we set

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

we can expect that

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

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 $r_i t_{i,j} c_j \leq 1$ 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 $\forall i,j,\ r_i t_{i,j} c_j \leq 1$, and such that

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

Let us consider again the case of 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

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

The mean value of B is 0.8302, what means that on average, the processors work 83.02% of time, and the value of the objective function $(\sum r_i)(\sum c_j)$ is 2.4322.

4.4.3 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 in 4.4.2. In the case of our example, we obtain

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

This suggests to choose another arrangement matrix T for the processor 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_i's and the c_j's as shown in 4.4.2 and we restart the process. We consider that the process as converged when no modification occurs in the matrix T.
In the case of 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).
The 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

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

4.4.4 Numerical 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 a $n \times n$ grid.
Figure 7 displays the evolution with n of the average work load (the mean value of B) of the processors (when the convergence is reached).
Figure 8 displays the evolution with n of the ratio

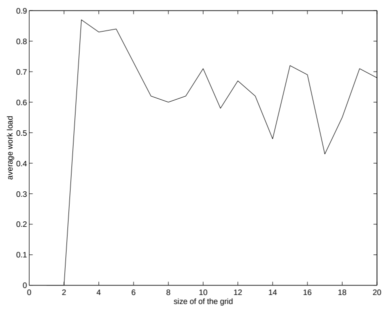

Figure 7. average work load

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

Figure 9 displays the evolution with n of the aver-

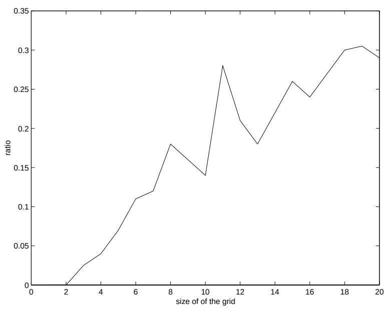

Figure 8. τ

age number of steps necessary to reach the convergence.

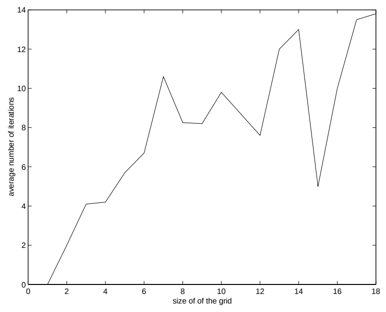

Figure 9. number of iterations

4.4.5 Concluding remarks on the polynomial heuristic

The polynomial heuristic 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 with n^2 processors (nevertheless, one usually obtain satisfying results after a few steps only).

5. Conclusion

In this paper, we have discussed static allocation strategies to implement matrix-matrix products and dense linear system solvers on heterogeneous computing platforms. Extending the standard ScaLAPACK block-cyclic distribution to heterogeneous 2D grids turns 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 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 polynomial heuristic.

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*, 38(6):673–681, 1994.

[2] S. Anastasiadis and K. C. Sevcik. Parallel application scheduling on networsk of workstations. *Journal of Parallel and Distributed Computing*, 43:109–124, 1997.

[3] D. Arapov, A. Kalinov, A. Lastovetsky, and I. Ledovskih. Experiments with mpc: efficient solving regular problems on heterogeneous networks of computers via irregularization. In *Irregular'98*, LNCS 1457. Springer Verlag, 1998.

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

[5] L. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. Scalapack: A portable linear algebra library for distributed-memory computers - design issues and performance. In *Supercomputing '96*. IEEE Computer Society, 1996.

[6] 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.

[7] V. Boudet, A. Petitet, F. Rastello, and Y. Robert. Data allocation strategies for dense linear algebra kernels on heterogeneous two-dimensional grid. In

Parallel and Distributed Computing and Systems conference (PDCS'99). IASTED Press, 1999.

[8] V. Boudet, F. Rastello, and Y. Robert. Algorithmic issues for (distributed) heterogeneous computing platforms. In R. Buyya and T. Cortes, editors, *Cluster Computing Technologies, Environments, and Applications (CC-TEA'99)*. CSREA Press, 1999. Extended version available as LIP Technical Report RR-99-19.

[9] V. Boudet, F. Rastello, and Y. Robert. A proposal for a heterogeneous cluster ScaLAPACK (dense linear solvers). In H. R. Arabnia, editor, *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*. CSREA Press, 1999. Extended version available as LIP Technical Report RR-99-17.

[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 Communications*, 97:1–15, 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*, 5:173–184, 1996.

[12] E. Chu and A. George. QR factorization of a dense matrix on a hypercube multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 11:990–1028, 1990.

[13] M. Cierniak, M. J. Zaki, and W. Li. Scheduling algorithms for heterogeneous network of workstations. *The Computer Journal*, 40(6):356–372, 1997.

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

[15] J. Dongarra, R. van de Geijn, and D. Walker. Scalability issues in the design of a library for dense linear algebra. *Journal of Parallel and Distributed Computing*, 22(3):523–537, 1994.

[16] J. J. Dongarra and D. W. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 37(2):151–180, 1995.

[17] G. Fox, S. Otto, and A. Hey. Matrix algorithms on a hypercube i: matrix multiplication. *Parallel Computing*, 3:17–31, 1987.

[18] M. Iverson and F. zgner. Dynamic, competitive scheduling of multiple dags in a distributed heterogeneous environment. In *Seventh Heterogeneous Computing Workshop*. IEEE Computer Society Press, 1998.

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

[20] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[21] M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Seventh Heterogeneous Computing Workshop*. IEEE Computer Society Press, 1998.

[22] H. Siegel, H. Dietz, and J. Antonio. Software support for heterogeneous computing. *ACM Computing Surveys*, 28(1):237–239, 1996.

[23] G. Sih and E. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–187, 1993.

[24] M. Tan, H. Siegel, J. Antonio, and Y. Li. Minimizing the aplication execution time through scheduling of subtasks and communication traffic in a heterogeneous computing system. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):857–1871, 1997.

[25] J. Weissman and X. Zhao. Scheduling parallel applications in distributed networks. *Cluster Computing*, 1(1):109–118, 1998.