Dense Linear Algebra Kernels on Heterogeneous Platforms: Redistribution Issue

Olivier Beaumont, Arnaud Legrand, Fabrice Rastello, Yves Robert

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 8512

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

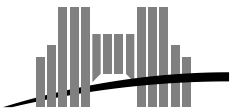
SPI

Dense Linear Algebra Kernels on Heterogeneous Platforms: Redistribution Issues

Olivier Beaumont
Arnaud Legrand
Fabrice Rastello
Yves Robert

Dec 2000

Research Report N° 2000-45



École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr

INRIA

Dense Linear Algebra Kernels on Heterogeneous Platforms: Redistribution Issues

Olivier Beaumont
Arnaud Legrand
Fabrice Rastello
Yves Robert

Dec 2000

Abstract

In this paper, we deal with redistribution issues for dense linear algebra kernels on heterogeneous platforms. In this context, processors speeds may well vary during the execution of a large kernel, which requires efficient strategies for redistributing the data along the computations. The strategy that we propose is to redistribute data after some well identified static phases and therefore, it is neither fully static nor fully dynamic. We present an optimal algorithm (under some assumptions) for redistributing data when performing matrix matrix multiplication.

Keywords: heterogeneous platforms, different-speed processors, load-balancing, data redistribution, matrix product

Résumé

Dans ce rapport, nous nous intéressons au problème des redistributions de données pour les noyaux d'algèbre linéaire adaptes aux plateformes hétérogènes. La vitesse des différents processeurs pouvant varier au cours du temps sur ce type de plateformes, il est important de mettre en œuvre des stratégies de redistributions efficaces afin de maintenir un bon équilibrage de charge tout au long du calcul. La stratégie hybride (ni complètement statique ni complètement dynamique) que nous proposons consiste a redistribuer les données après des phases d'équilibrages statiques bien delimitées. Nous présentons également un algorithme optimal (sous certaines hypothèses) pour la redistributions des données lors du calcul d'un produit de matrices.

Mots-clés: plateformes de calcul hétérogènes, processeurs de vitesses différentes, équilibrage de charge, redistribution de données, produit de matrices

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 major limitation to programming heterogeneous platforms arises from the additional difficulty of balancing the load when using processors running at different speeds. In this paper, we discuss the required framework to build an extension of the ScaLAPACK library capable of running on top of HNOWs or non-dedicated parallel machines. More precisely, we concentrate on dense linear algebra kernels such as matrix multiplication, or LU and QR decompositions. With processors running at different speeds, block-cyclic distribution is no longer enough; new data distribution schemes must be determined and analyzed.

In Section 2, we present several distribution schemes corresponding to different design strategies and to different modelings of the network. We show that deriving efficient distribution schemes turns out to be surprisingly difficult for all these problems. Technically, we will prove that the underlying optimization problems are NP-hard, and we will provide efficient (polynomial) approximations of the optimal solution.

When targeting networks of workstations, we need to consider that the network is not dedicated to a specific application, and therefore that processor speeds may vary during the execution of a large application. Therefore, it is necessary to be able to redistribute the data if changes in processor speeds occur. In Section 3, we discuss both advantages and drawbacks of static and dynamic strategies when performing large and regular applications (such as linear algebra) on top of HNOWs. In Section 4, we present a simple (but optimal under some assumptions) algorithm for the redistribution of data along the computations. This algorithm is well-suited to the distribution schemes discussed in Section 2. The proof of optimality is given in Section 5.

2 Matrix multiplication on heterogeneous networks

In this section, we present several algorithms and the corresponding distribution schemes for computing matrix multiplications (MM) on top of HNOWs. Those distributions schemes that have been studied extensively in [1, 2] and we refer to these papers for the proofs of the theorems presented in this section.

2.1 Two dimensional heterogeneous grids

One possible distribution scheme consists in mapping the processors on a grid. The idea here is to avoid rebuilding ScaLAPACK kernels from scratch; instead, we take advantage of the deep modularity of the library, and we modify only high-levels routines related to data distribution. The main drawback of this method is that we cannot give to each processor an amount of work in exact accordance to its computing power.

Assume first that the 2D grid is homogeneous: the $p \times p$ processors are identical. In that case, ScaLAPACK uses a block version of the outer product algorithm[1] described in [4, 5, 6], 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 $p \times p$ 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 Figure 1. For square $p \times p$ homogeneous 2D-grids,

[1] ScaLAPACK uses a two-dimensional grid rather than a linear array for scalability reasons [3].

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.

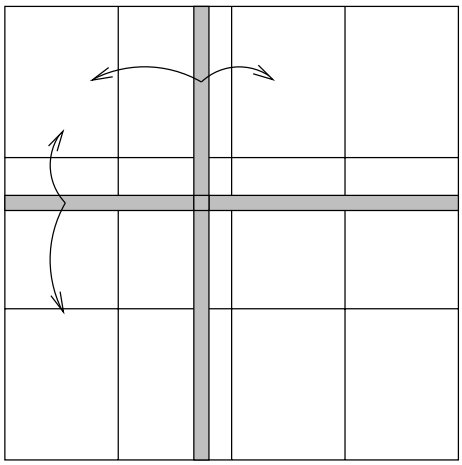


Figure 1: The MM algorithm on a 3×4 homogeneous 2D-grid.

2.2 Matrix Product on a 2D Heterogeneous Grid

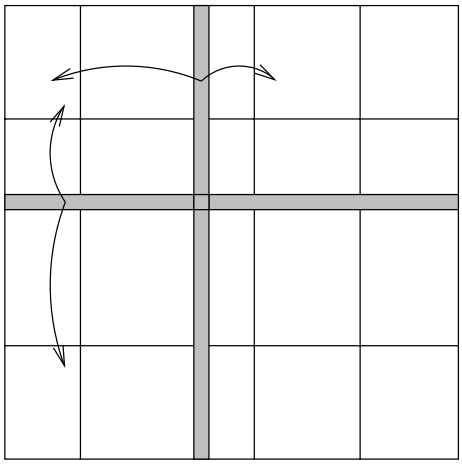
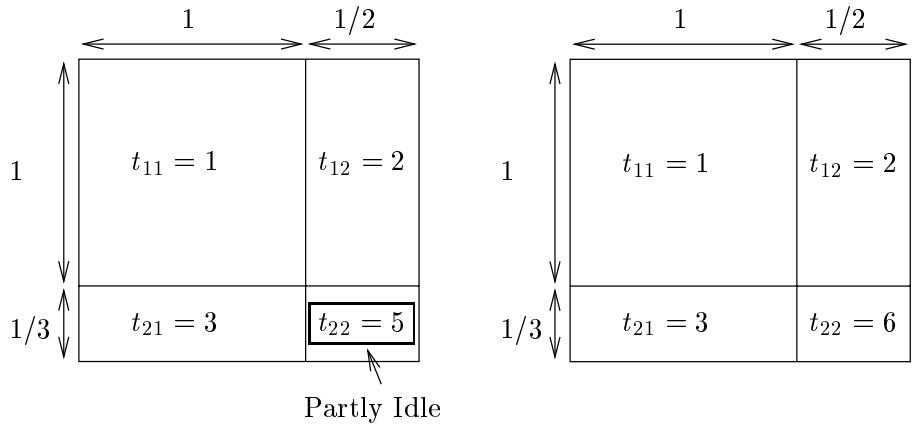


Figure 2: The MM algorithm on a 3×4 heterogeneous 2D-grid.

We can modify the previous MM algorithms for a heterogeneous grid. We keep the same framework (see Figure 2), but we want to balance the computing load so that each processor receives roughly an amount of work in accordance to its computing power. Because all C blocks require the same amount of arithmetic operations, each processor executes an amount of work which is proportional to the number of blocks that are allocated to it, hence proportional to the area of its rectangle. To parallelize the matrix product $C = AB$, we have to tile the C matrix into a 2D grid of $p \times p$ different-size rectangles, as shown in Figure 2. In general, this can not be done exactly, since the grid framework is too restrictive (see Figure 3).

The problem is therefore both to arrange the processors into a 2D $p \times p$ grid and to compute the area of the p^2 rectangles so as to balance the load during computations. If s_k denotes the number of blocks the



(a) Perfect balance is not possible

(b) Perfect balance is possible because the $T = (t_{ij})$ matrix is rank-1

Figure 3: Load balancing on a 2×2 grid

processor P_k is able to proceed within one time unit and $r_k \times c_k$ the area of its rectangle, $\frac{r_k c_k}{s_k}$ should not depend on k.

More precisely, the optimization problem can be stated as follows:

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

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

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

Indeed, since we know that perfect load-balance is not always possible, we try to maximize the number of elements that can be processed within one time unit, which can be expressed by $(\sum_{i=1}^{p} r_i)(\sum_{j=1}^{p} c_j)$.

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, 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

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

and

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

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

The technical proof of this result can be found in [1]. It states the intrinsic difficulty of static load balancing on heterogeneous platforms.

2.3 More general heterogeneous distributions

Another possible distribution scheme consists in balancing the load so that each processor receives an amount of work in plain accordance to its computing power. In order to balance the load perfectly, we need to remove the constraint stating that the processors have to be arranged into a 2D grid. When computing matrix product, each processor executes an amount of work which is proportional to the number of blocks that are allocated to it. Thus, the iteration space is split into rectangles and the workload of each processor is hence proportional to the area of its rectangle (see Figure 4 with $p = 13$ processors). Since we can balance the load perfectly, the question is: how to compute the *shape* of the rectangles so as to minimize the total execution time (i.e. the computational and the communication time)? We will consider two different modelings of the network. If all communications can occur in parallel, then the question is to minimize the maximal perimeter among all the rectangles. Conversely, if communications cannot occur in parallel, then the question is to minimize the sum of the perimeters among all the rectangles (see Figure 4).

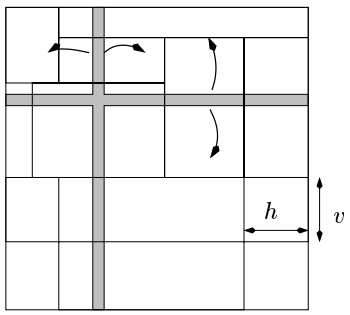


Figure 4: The MM algorithm on a heterogeneous platform.

Let s_i be the fraction of the total computing power corresponding to processor P_i, $1 \leq i \leq p$. Normalizing processor speeds, we have $\sum_{i=1}^{p} s_i = 1$. Normalizing the computing workload accordingly, we have to tile the unit square into p rectangles R_i of prescribed area s_i, $1 \leq i \leq p$.

Let $h_i \times v_i$ be the size of rectangle R_i, where $h_i v_i = s_i$. At each step of the MM algorithm, communications take place between processors: the *total* volume of data exchanged is proportional to the *sum* $\hat{C} = \sum_{i=1}^{p}(h_i + v_i)$ of the half perimeters of the p rectangles R_i.

If the communications can be handled in parallel, then the minimization of the time spent in communication can be expressed as the following optimization problem:

Definition 3 *PERI-MAX(s): Given p real positive numbers $s_1, \ldots, s_p$ s.t. $\sum_{i=1}^{p} s_i = 1$, find a partition of the unit square into p rectangles R_i of area s_i and of size $h_i \times v_i$, so that $\hat{M} = \max_{1 \leq i \leq p}(h_i + v_i)$ is minimized.*

Conversely, if communications cannot be handled in parallel, minimizing the amount of communications is equivalent to minimizing the total amount of data to be communicated, what leads to the following optimization problem.

Definition 4 *PERI-SUM(s): Given p real positive numbers $s_1, \ldots, s_p$ s.t. $\sum_{i=1}^{p} s_i = 1$, find a partition of the unit square into p rectangles R_i of area s_i and of size $h_i \times v_i$, so that $\hat{C} = \sum_{i=1}^{p}(h_i + v_i)$ is minimized.*

The decision problems associated to PERI-MAX and PERI-SUM are the following:

- PERI-SUM(s,K) Given p real positive numbers $s_1, \ldots, s_p$ s.t. $\sum_{i=1}^{p} s_i = 1$ and a positive real bound K, is there a partition of the unit square into p rectangles R_i of area s_i and of size $h_i \times v_i$, so that $\sum_{i=1}^{p}(h_i + v_i) \leq K$?
- PERI-MAX(s,K) Given p real positive numbers $s_1, \ldots, s_p$ s.t. $\sum_{i=1}^{p} s_i = 1$ and a positive real bound K, is there a partition of the unit square into p rectangles R_i of area s_i and of size $h_i \times v_i$, so that $\max_{1 \leq i \leq p}(h_i + v_i) \leq K$?

Theorem 2 *PERI-SUM(s,K) and PERI-MAX(s,K) are both NP-complete.*

A full-length version of the proof is available in the technical report [2].

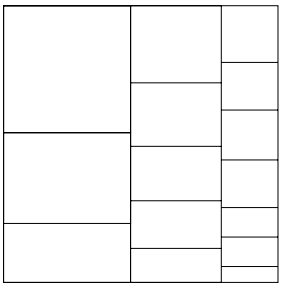


Figure 5: Tiling the unit square into columns of rectangles.

If we restrict the search to column-based partitionings, i.e. partitionings with the constraint that rectangles are assigned to columns (with possibly different numbers of rectangles per column, see Figure 5), we obtain two new problems, COL-PERI-SUM(s) and COL-PERI-MAX(s). Surprisingly, their complexity if not the same (unless P=NP):

Theorem 3 *COL-PERI-SUM(s) is polynomial and COL-PERI-MAX(s,K) is NP-complete.*

The optimal algorithm for COL-PERI-SUM can be found in [2] and it is based on a simple dynamic programming algorithm. This algorithm can be used as a heuristic for solving the PERI-SUM problem. Numerical tests proving the efficiency of this heuristic can be found in [2].
In Section 4, we concentrate on column based partitioning. We give a simple algorithm that computes a new column based partitioning if the relative speeds of processors (i.e. the area of the rectangle allocated to them) have changed during time. The algorithm we propose is optimal under some conditions.

3 Static vs. dynamic strategies

Distributing the computations (together with the associated data) can be performed either dynamically or statically, or a mixture of both [7]. On homogeneous platforms, static strategies already suffer from a wide range of problems:

1. Designing optimal (i.e. minimizing the completion time) scheduling algorithms is a NP-complete problem, except in some rare cases. To give a single example, even the central problem of scheduling unit-length tasks with unit-communication delays onto unlimited resources is NP-complete [8]. Heuristic methods rely on rules-of-thumb, such as the estimated length of the critical path. Few theoretical guarantees are provided in the literature.

2. Accurately estimating task execution times and communication delays is often difficult. For instance, it may not be practical to estimate communication delays at compile-time because of the run-time network contention delays.

3. Static methods cannot cope with possible (but unpredictable) variations in the processor speeds (e.g. due to variations in their load).

Dynamic load-balancing strategies are more flexible. Several sophisticated techniques have been reported in the literature (see [9] for an introduction or [10] for a recent survey). These techniques provide efficient solutions to the following problems [10]: load evaluation, profitability determination, work transfer vector calculation, task selection and task migration. The major disadvantage of dynamic load-balancing schemes is the run-time overhead due to various sources, including

- the load-information transfer among processors,

- the decision-making policies for the selection of processes and processors,
- and the communication delays incurred by task re-location (which may cause a costly data redistribution).

For heterogeneous platforms, the advantages and flexibility of dynamic load-balancing appear even more appealing. To reduce run-time overhead, simple selection strategies are often used, both for processor selection and for process selection. Processors are chosen according to paradigms based upon the idea *"use the past predict the future"*, i.e. use the already observed speed of computation of each machine to decide for the next distribution of work [11, 12, 13]. Processes are selected using a greedy strategy, picking up new tasks just as they terminate their current computation (e.g. using a classical master-slave implementation). Dynamic strategies look promising because the machine loads is self-regulated, hence self-balanced, despite any machine heterogeneity.

However, the true adversary of dynamic strategies on heterogeneous platforms are *data dependences*, which may well lead to slow the whole process down to the pace of the slowest processors, as shown in [1].

Thus, if we consider very regular problems (such as linear algebra algorithms), data dependences, in addition to communication costs and control overhead, have a severe impact on classic dynamic schemes. In contrathe workload of processors varyst, static strategies will suppress (or at least minimize) data redistributions and memory management overhead while preserving parallelism. Nevertheless, since processor speeds may vary during the execution of a large application, it is necessary to design strategies for data redistributions, in order to cope with the load imbalance that may appear.

4 Redistribution of data on heterogeneous platforms

We have proved that column-based distributions are well suited for balancing the load between processors of different speeds when performing matrix multiplications. We have also shown that, in the context of large and regular scientific applications executed on a non dedicated HNOW, both static and dynamic strategies fail to maintain a good load balancing. Nevertheless, a possible solution consists in performing large static phases and then to redistribute data if the load is not perfectly balanced, because of changes in processor speeds.

In this section, we study a strategy for redistributing data when a load imbalance occurs. Our aim is not to redistribute the data in order to minimize the total execution time. Indeed, such a redistribution would involve too many communications, possibly much more than what is needed to finish the matrix product. In this section, we concentrate on load balancing, and the redistribution scheme that we propose enables us to obtain a perfect load balancing. Conversely, we do not try to optimize the communication time when performing redistribution.

More precisely, we will change both the shape and the area of the rectangles so that the load would be balanced but we will modify neither their relative positions nor the number of column of processors. In the solution of COL-PERI-SUM with new relative speeds of the processors, their relative position may change, and the cost of the redistribution may be huge.

4.1 Elementary operations

In this section, we present the set of elementary operations used to transform a column based partitioning of the matrix that is not optimal (with respect to load balancing because processor speeds may have changed) into a column based partitioning that is optimal (with respect to load balancing) for actual processor speeds. Since we know the actual performances of the processors, we can compute for each of them δ_i, the difference between the number of blocks that processor P_i should hold in order to achieve perfect load balancing and the number of blocks that it actually holds (we can notice that with this definition $\sum_{i=1}^{p} \delta_i = 0$).

The elementary operations used to reach perfect load balancing are depicted in Figure 6. We restrict ourselves to operations that do not affect the MM algorithm. In particular, moving a whole column 6(a) or a whole row 6(b) from a processor to another one, or perform local slidings 6(d) of the border between two adjacent processors belonging to the same column of processors only changes the place where the components of C

will be computed. Conversely, the migration of part of a row from a processor to another one deeply affects the algorithm, since it changes the whole topology of the matrix distribution over processors (see 6(c)).

Therefore, the only elementary operations that we consider are migrations of a whole row or a whole row and local slidings of the border between two adjacent processors belonging to the same column of processors.

4.2 Impact of elementary operations on load balancing

The effects of the migration of a whole column of the matrices are depicted in Figure 6(a). It is the only elementary operation that modifies the load of a whole column of processors. We will prove in Section 4.5 that it is possible to obtain a perfect load balancing of each column C_k of processors (i.e. $\sum_{i \in C_k} \delta i = 0$) with a simple greedy algorithm.

The effects of the migration of a whole row of the matrices are depicted in Figure 6(b). This operation may be used for balancing the load between the processors of a same column. Unfortunately, as we noticed it (see Figure 6(c)), we cannot perform migrations of part of a row in each column of processors, and therefore, it is not possible to balance the load between all the processors using only migrations of rows and columns, as shown below.

If the initial load imbalance is $\begin{array}{|c|c|}\hline -1 & 1 \\ \hline 1 & -1 \\ \hline \end{array}$, where the values indicate the relative number of blocks δ_i the processors should receive to reach perfect load balancing, the best load balancing we can obtain using only rows and columns migrations is $\begin{array}{|c|c|}\hline -2 & 0 \\ \hline 0 & -2 \\ \hline \end{array}$ or $\begin{array}{|c|c|}\hline 0 & 2 \\ \hline 2 & 0 \\ \hline \end{array}$, and neither of those distributions is well balanced (i.e. $\forall i,\ \delta_i{=}0$).

In order to obtain a perfect load balancing, we therefore need to use the last elementary operation, i.e. the local sliding of the border between two adjacent processors (see Figure 6(d)). We will prove in Section 5 that it is always possible to obtain a perfect load balancing using only rows and columns migrations and local slidings of the borders between adjacent processors.

4.3 Cost of elementary operations

We need to make a few assumptions concerning the cost of communications between the processors in order to obtain a modeling of the communication network and to prove the optimality of the greedy algorithm presented in Section 4.5. First, we assert that latency can be neglected and therefore that the time required to send a message between two processors is proportional to the size of the message. The second assumption states that the time required to send a message between two processors does not depend upon those processors, i.e. that the communication network is homogeneous. The last assumption is less important: it states that the time required to perform the migration of a whole row of the matrices does not depend on the choice of the row (see Figure 9. In Section 4.6, we consider the modifications introduced by the relaxation of the last assumptions.

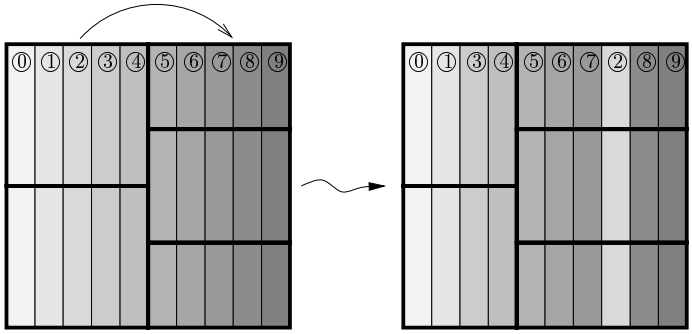
If the assumptions stated above hold true, we propose in Section 4.5 a greedy algorithm that finds the sequence of elementary operations that leads to perfect load balancing at a minimal communication cost.

4.4 Sequences of elementary operations

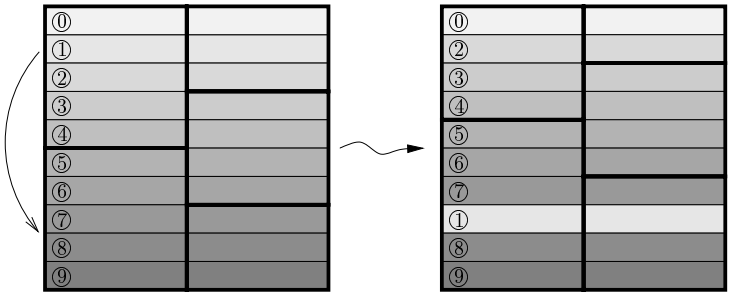
In this section, we study some sequences of elementary operations for balancing the load between processors. The three elementary operations that we have defined are commutative, and moreover, with the assumptions of Section 4.3, the cost of a sequence of elementary operations does not depend on which order they are performed.

An example of a sequence of elementary operations is given in Figure 7. The number associated to each processor P_i is δi, i.e. the relative number of blocks P_i has to receive in order to balance perfectly the load between all the processors.

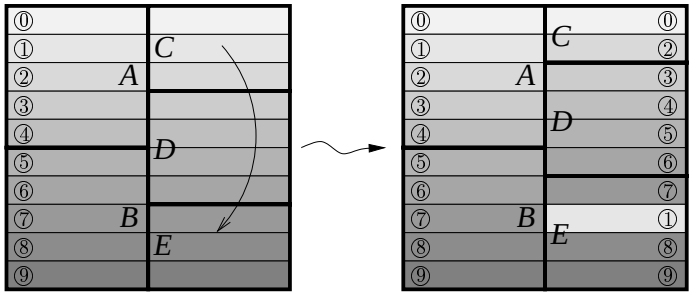
In this example, all columns have same width ($\frac{2n}{3}$). A column migration from C_3 to C_1 costs $2n$ elementary communications (an elementary communication consists in a point to point communication of one block of each matrices A, B and C). Then, all columns are well balanced, but we still have to balance the load between the processors belonging to same column. This can be done using only local slidings. In this case,



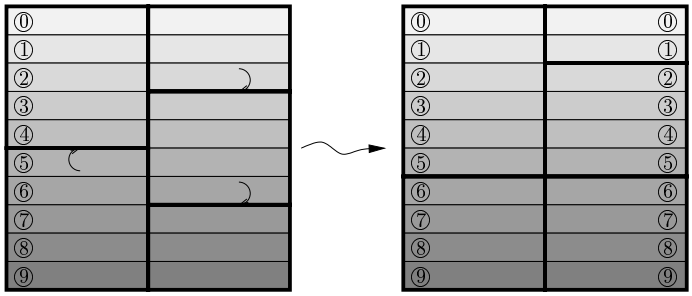
(a) Column Migration: Migration of a whole column of the matrices



(b) Row Migration: Migration of a whole row of the matrices



(c) migrating part of a row of the matrices deeply affects the broadcast of the column of A. Indeed, after the migration, processor C, D and E belong to the neighborhood of processor A and the messages from A to E are very small.



(d) Local Sliding: Local sliding of the border between two processors belonging to the same column.

Figure 6: Elementary operations used for redistribution

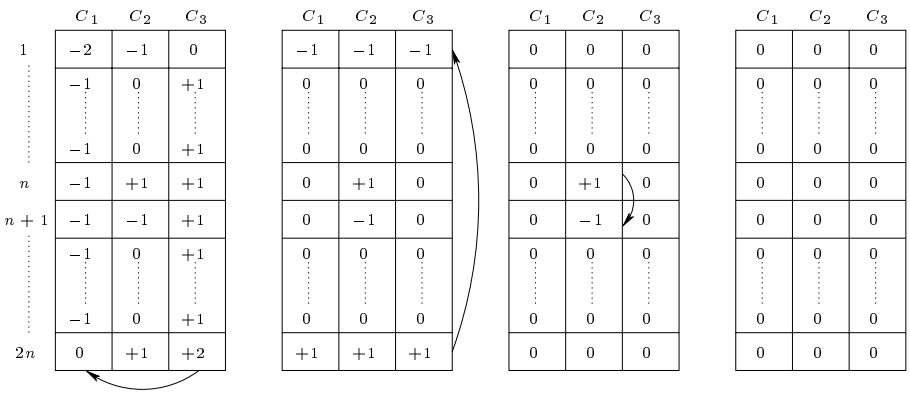


Figure 7: A sequence of elementary operations for balancing the load

we have to perform $(2n-1) + 2(n-1) + (2n-1) = 6n-4$ local slidings (the cost of a local sliding is $\frac{2n}{3}$ elementary communications), since this operation only affects the place of the border between adjacent processors. On the other hand, we can also perform the migration of a whole row of each matrices from the last row of processors to the first row of processors (what costs $2n$ elementary communications) and then just perform one local sliding between the median processors of column C_2. The cost of this solution is obviously much smaller.

We can notice that the choice of an optimal sequence of elementary operations is not always obvious. For instance, consider the redistribution of the data if the relative number of blocks needed by the processors is as depicted in Figure 8 (where + stands for 1 and − for −1). Each column of processors is well balanced so that we only need to define a sequence of row migrations and local slidings in order to reach perfect load balancing into each column.

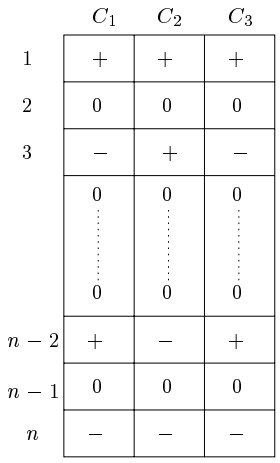


Figure 8: The choice of the sequence of elemntary operations may not be trivial.

Different possible sequences are discussed below:

- if we balance the load only with local slidings in each column of processors, the cost is $\frac{2n}{3}\ (4 + \big(4 + 2(n-4)\big) + 4) = \frac{2n}{3}\ (2n+4)$.
- if we first perform the migration of a whole row of the matrices from the first row of processors to the last row of processors, and then perform local slidings into each of the columns, then the overall cost becomes $2n + \frac{2n}{3}\ (3(n-4))$, what is even worse.

- if we first perform two row migrations from the first row of processors to the last row of processors and from row $n-2$ to row 3 and then perform local slidings into the second column to reach perfect load balancing, then the overall cost is $4n + \frac{2n}{3}$ $(2(n-4))$. This sequence is in fact optimal.

This simple example shows that the choice of the optimal sequence of elementary operations to be performed to reach perfect load balancing is not always trivial. Nevertheless, we present in next section a greedy algorithms that solves this problem.

4.5 An optimal greedy algorithm

Since the cost of a sequence of elementary operations does not depend on which order they are performed, and since all the operations commute, we organize the algorithm as follows. First, we perform column migrations so as to balance the workload between each columns of processors. Then, we can always reach perfect load balancing into each column using only local slidings (the cost of such a balancing can be easily determined, as shown in Section 5). Therefore, we perform a row migration if and only if the overall cost of the row migration and resulting local slidings necessary to reach perfect load balancing is lower than the initial cost of local slidings.

The whole process is expressed in Algorithm 1.

GREEDY BALANCING($D = (\delta_1, \ldots, \delta_p)$)
1: **While** D is not balanced between columns
2: Perform a column migration from the column C_k maximizing $\sum_{P_i \in C_k} \delta i$ to the column C_j minimizing $\sum_{P_i \in C_j} \delta i$.
3: **While** there exists a row migration so that the overall cost of the row migration and resulting local slidings necessary to reach perfect load balancing is lower than the initial cost of local slidings
4: Perform the row migration
5: Perform local slidings in each column to reach perfect load balancing

ALGO. 1: Greedy balancing algorithm.

Theorem 4 *With our assumptions on the cost of elementary operations (see Section 4.3), the sequence of elementary operations given by greedy algorithm defined above is optimal among sequences leading to a perfect load balancing .*

The proof of this theorem is very technical and is given in Section 5.

4.6 Relaxation of some assumptions

The greedy algorithm defined above is optimal under the assumptions we made in Section 4.3. As we noticed before, the cost of a row migration is not in general constant, since in some columns of processors, no communication may be required by a row migration (see Figure 9). Unfortunately, the greedy algorithm is not optimal if we relax the assumption stating that the cost of row migration does not depend on the choice of the row.

Let us consider the situation depicted in Figure 9.

We suppose that the width of C_3 (and other columns C_1 and C_2) is $\frac{2n}{3}$ blocks. If we consider that the cost of a row migration from row i to row j is 0 in a column of processors whenever the same processor holds rows i and j in this column, then, for instance, the cost of the migration of row 1 to row $n-2$ is only $\frac{2n}{3}$, since it does not require any communication in the first two columns of processors.

Thus, it can be proved that the best sequence of elementary operations consists in two row migrations (from 1 to $n-2$ and from $n+2$ to $2n$), for an overall cost of $2 \times \frac{2n}{3}$ elementary communications. On the other hand, the choice made by greedy algorithm consists in two row migrations (from 1 to $2n$ and from $n+2$ to $n-2$), for an overall cost of $4n$ elementary communications.

Thus, the greedy algorithm is no more optimal if we relax the assumption stating that the cost of row migration is constant.

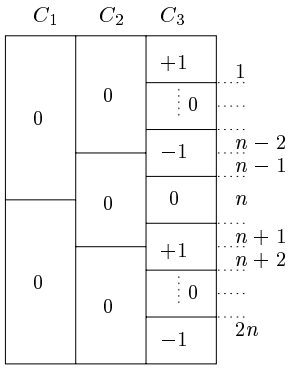


Figure 9: The greedy algorithm is not optimal if the cost of row migration is not constant.

5 Proof of the optimality of the greedy algorithm

5.1 A modeling of load imbalance

Let us consider a column based distribution of matrices over processors. Let n_{col} be the number of column of processors and w_i the width (i.e. the number of blocks) of the i-th column of processors. If $i \in [\![1, n_{col}]\!]$ and $j \in [\![1, n_i]\!]$, we will represent the j^{th} rectangle (processor) of column i by its abscissa $y_{i,j}$, its height $h_{i,j}$ and its current imbalance $\delta_{i,j}$ (expressed in number of blocks).

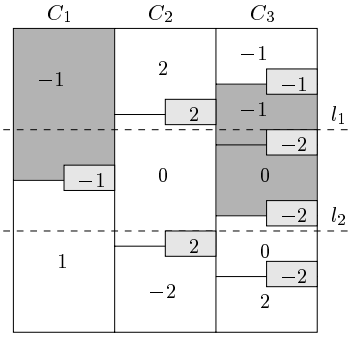


Figure 10: An unbalanced distribution

Since it is possible to balance the workload in columns of processors by performing only column migrations, we can suppose that each column of processors is balanced, i.e. $\forall i, \quad \sum_{j=1}^{n_i} \delta_{i,j} = 0$, and if we denote by $b_{i,j} = \sum_{k=1}^{j} \delta_{i,k} \quad (i \in [\![1, n_{col}]\!], \quad j \in [\![1, n_i - 1]\!])$, the cost for reaching perfect load balancing using only local slidings $LS(D)$ is

$$LS(D) = \sum_{i=1}^{n_{col}} \sum_{j=1}^{n_i} w_i |b_{i,j}| \tag{5.1}$$

Moreover, let us denote by $\langle\!\langle\, l_2, l_1 \,\rangle\!\rangle = \langle\!\langle\, l_1, l_2 \,\rangle\!\rangle$ $(l_1 < l_2)$ the following set

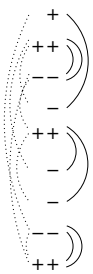
$$\{ b_t \mid t \in \langle\!\langle\, l_1, l_2 \,\rangle\!\rangle \,\} = \{ b_{i,j} \mid \quad i \in \langle\!\langle\, 1, n_{col} \,\rangle\!\rangle, y_{i,j} + h_{i,j} < l_2, \text{ and } [y_{i,j}, y_{i,j} + h_{i,j}[\cap [l_1, l_2[\neq \emptyset \quad \}. \tag{5.2}$$

$\langle\!\langle\, l_1, l_2 \,\rangle\!\rangle$ is depicted by grey zones in Figure 10. We can notice that if $a < b < c$, then $\langle\!\langle\, a, c \,\rangle\!\rangle = \langle\!\langle\, a, b \,\rangle\!\rangle \cup \langle\!\langle\, b, c \,\rangle\!\rangle$.

The effect of the migration of the row l_1 to row l_2 on $b_{i,j}$ can be expressed by

- If $l_1 < l_2$: $\forall t \in [\![l_1, l_2]\!]$: $b_t \leftarrow b_t + 1$
- If $l_2 < l_1$: $\forall t \in [\![l_2, l_1]\!]$: $b_t \leftarrow b_t - 1$

5.2 Canonical representation of row migrations



As noticed before, under our assumptions, the cost of a sequence of row migrations does not depend on which order they are performed. Therefore, a set of row migrations can be represented by a list of (abscissas, relative number of rows to be migrated), and it can be seen as a word on $\{+, -\}$, with as many $+$ as $-$.

Lemma 1 *Let u be a finite word on $\{+, -\}$ and let p be the smallest non empty prefix of u with as many $+$ as $-$. Then, if φ is the canonical morphism from $\{+, -\}$ into $\{(,)\}$, then $\varphi(p)$ is a well balanced expression of parentheses.*

Therefore, any set of row migrations can be expressed by a well balanced expression of parentheses. The following technical lemmas will be used for completing the proof of the optimality of greedy algorithm in Section 5.3.

Lemma 2 $\forall x \in \mathbb{R} : \forall k \in \mathbb{N} :$ $|x+k+1| - |x+k| \geqslant |x+1| - |x|$
$|x+k| - |x+k+1| \geqslant |x| - |x+1|$

Lemma 3 *Let $e_2, \ldots, e_n$ be a set of well balanced row migrations. If (l_1, l_2) (with $l_1 < l_2$) denotes a row migration (which does not affect the canonical representation $e_2, \ldots, e_n$) and if b'_t denotes the value of b_t after the set of row migrations $e_2, \ldots, e_n$, then*

$$\sum_{[\![l_1,l_2]\!]} w_t|b'_t + 1| < \sum_{[\![l_1,l_2]\!]} w_t|b'_t| \Rightarrow \sum_{[\![l_1,l_2]\!]} w_t|b_t + 1| < \sum_{[\![l_1,l_2]\!]} w_t|b_t|$$

$$\sum_{[\![l_1,l_2]\!]} w_t|b'_t| < \sum_{[\![l_1,l_2]\!]} w_t|b'_t + 1| \Rightarrow \sum_{[\![l_1,l_2]\!]} w_t|b_t| < \sum_{[\![l_1,l_2]\!]} w_t|b_t + 1|$$

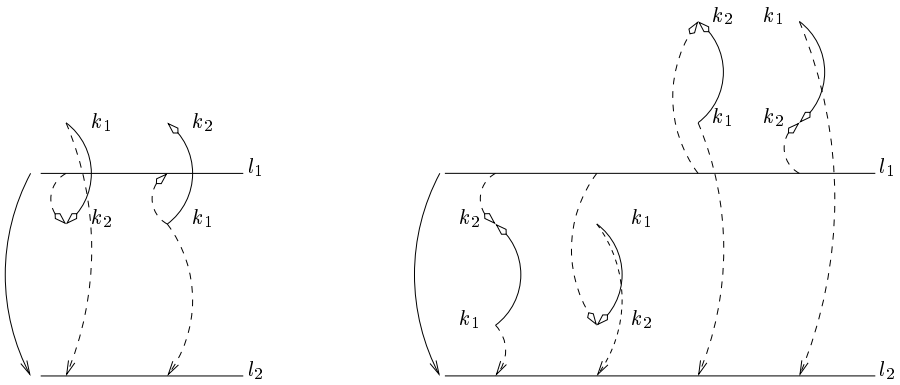
5.3 Optimality of greedy algorithm

The sketch of the proof is as follows. We suppose that greedy algorithm has determined at a given step that the best row migration is from row l_1 to l_2 (we can suppose that $l_1 < l_2$ by symmetry). We prove that there exists an optimal set of row migrations that affects row l_1, what achieves the proof.

Let E be the canonical representation of a set of optimal row migrations which does not affect l_1 and let us suppose that any set of row migrations affecting l_1 is worse. For each possible case, we will exhibit a set of non optimal row migrations E' that affects l_1 and use it to prove that the choice made by greedy algorithm was wrong.

Case 1 : Let us suppose that l_1 is enclosed by a row migration of E (Figure 11(a)). Let E be denoted by $E = \{(k_1, k_2), e_2, \ldots, e_n\}$ and let (k_1, k_2) be the row migration that encloses l_1. Let us denote by E' the set of row migrations $E' = \{(l_1, k_2), e_2, \ldots, e_n\}$ if $k_1 < k_2$ (or $E' = \{(k_1, l_1), e_2, \ldots, e_n\}$ if $k_2 < k_1$, the rest of the proof being unchanged). Since the cost of E' is greater than the cost of E, we will prove that the greedy algorithm would have chosen the row migration (k_1, l_2) instead of (l_1, l_2).

Let us denote by $b'_{i,j}$ the new value of $b_{i,j}$ after the row migrations $\{e_2, \ldots, e_n\}$.
Since $E' = \{(l_1, k_2), e_2, \ldots, e_n\}$ is worse than $E = \{(k_1, k_2), e_2, \ldots, e_n\}$, then



(a) l_1 is enclosed by a row migration of E

(b) l_1 is not enclosed by any row migrations of E

Figure 11: Possible cases for the position of l_1 with respect to row migrations of E.

$$\sum_{\langle\!\langle k_1,k_2\rangle\!\rangle} w_t|b'_t+1| < \sum_{\langle\!\langle k_1,l_1\rangle\!\rangle} w_t|b'_t| + \sum_{\langle\!\langle l_1,k_2\rangle\!\rangle} w_t|b'_t+1|$$

$$\sum_{\langle\!\langle k_1,l_1\rangle\!\rangle} w_t|b'_t+1| < \sum_{\langle\!\langle k_1,l_1\rangle\!\rangle} w_t|b'_t| \qquad \text{and thus (lemma 3)}$$

$$\sum_{\langle\!\langle k_1,l_1\rangle\!\rangle} w_t|b_t+1| < \sum_{\langle\!\langle k_1,l_1\rangle\!\rangle} w_t|b_t| \qquad \text{and}$$

$$\sum_{\langle\!\langle k_1,l_2\rangle\!\rangle} w_t|b_t+1| < \sum_{\langle\!\langle k_1,l_1\rangle\!\rangle} w_t|b_t| + \sum_{\langle\!\langle l_1,l_2\rangle\!\rangle} |b_t+1|$$

Therefore, the greedy algorithm would have chosen the row migration (k_1, l_2) instead of (l_1, l_2), what achieves the proof in this case.

Case 2 : Let us suppose that l_1 is not enclosed by any row migration of E (Figure 11(b)). We suppose that (l_1, l_2) encloses one of the row migration of E and that both row migrations occur in the same sense (the proofs in other possible situations depicted in Figure 11(b) are very similar). Let us denote E by $\{(k_1, k_2), e_2, \ldots, e_n\}$, where (k_1, k_2) is enclosed in (l_1, l_2). Then, $E' = \{(l_1, k_2), e_2, \ldots, e_n\}$ is a non optimal choice of row migrations, and we will prove (as previously) that the greedy algorithm would have chosen the row migration (k_1, l_2) instead of (l_1, l_2).

Let us denote by $b'_{i,j}$ the new value of $b_{i,j}$ after the row migrations $\{e_2, \ldots, e_n\}$.
Since $E' = \{(l_1, k_2), e_2, \ldots, e_n\}$ is worse than $E = \{(k_1, k_2), e_2, \ldots, e_n\}$, then

$$\sum_{\langle\!\langle l_1,k_1\rangle\!\rangle} w_t|b'_t| + \sum_{\langle\!\langle k_1,k_2\rangle\!\rangle} w_t|b'_t+1| < \sum_{\langle\!\langle l_1,k_2\rangle\!\rangle} w_t|b'_t+1|$$

$$\sum_{\langle\!\langle l_1,k_1\rangle\!\rangle} w_t|b'_t| < \sum_{\langle\!\langle l_1,k_1\rangle\!\rangle} w_t|b'_t+1| \quad \text{and thus (lemma 3)}$$

$$\sum_{\langle\!\langle l_1,k_1\rangle\!\rangle} w_t|b_t| < \sum_{\langle\!\langle l_1,k_1\rangle\!\rangle} w_t|b_t+1| \quad \text{and}$$

$$\sum_{\langle\!\langle l_1,k_1\rangle\!\rangle} w_t|b_t| + \sum_{\langle\!\langle k_1,l_2\rangle\!\rangle} w_t|b_t+1| < \sum_{\langle\!\langle l_1,l_2\rangle\!\rangle} w_t|b_t+1|$$

Therefore, the greedy algorithm would have chosen the row migration (k_1, l_2) instead of (l_1, l_2), what achieves the proof in this case.

Therefore, we have proved that the greedy algorithm we proposed in Section 4.5 is optimal under the assumptions we made in Section 4.3.

6 Conclusion

The major limitation to programming large linear algebra kernels on top of HNOWs arises from the additionnal difficulties to distribute data so that the load is balanced with processors running at different speeds and to keep a good load balancing if changes in processor speeds occur during the execution.
Unfortunately, static strategies are not well suited to possible changes in processor speeds and dynamic strategies fail to obtain a good load balancing for regular applications (such as linear algebra). Thus, the method we propose to reach (and then maintain) a good load balancing consists in redistributing data after some well identified static phases.
The algorithm that we propose for redistributing data is well suited to column based distributions used for distributing data when performing matrix multiplication. Under some assumptions concerning the communication network, we have proved that this algorithm enables to find a perfectly balanced distribution at a minimal redistribution cost.

References

[1] O. Beaumont, V. Boudet, A. Legrand, F. Rastello, Y. Robert, Heterogeneity considered harmful to algorithm designers, Tech. Rep. RR-2000-24, LIP, ENS Lyon (Jun. 2000).

[2] O. Beaumont, V. Boudet, F. Rastello, Y. Robert, Matrix-matrix multiplication on heterogeneous platforms, Tech. Rep. RR-2000-02, LIP, ENS Lyon, short version appears in the proceedings of ICPP'2000. (Jan. 2000).

[3] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley, ScaLAPACK Users' Guide, SIAM, 1997.

[4] R. Agarwal, F. Gustavson, M. Zubair, A high performance matrix multiplication algorithm on a distributed-memory parallel computer, using overlapped communication, IBM J. Research and Development 38 (6) (1994) 673–681.

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

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

[7] B.A.Shirazi, A. Hurson, K. Kavi, Scheduling and load balancing in parallel and distributed systems, IEEE Computer Science Press, 1995.

[8] P. Chrétienne, C. Picouleau, Scheduling with communication delays: a survey, in: P. Chrétienne, E. C. Jr., J. Lenstra, Z. Liu (Eds.), Scheduling Theory and its Applications, John Wiley & Sons, 1995, pp. 65–89.

[9] N. Shivaratri, P. Krueger, M. Singhal, Load distributing for locally distributed systems, IEEE Computer 25 (12) (1992) 33–44.

[10] J. Watts, S. Taylor, A practical approach to dynamic load balancing, IEEE Transactions on Parallel and Distributed Systems 9 (3) (1998) 235–248.

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

[12] M. Cierniak, M. J. Zaki, W. Li, Customized dynamic load balancing for a network of workstations, Journal of Parallel and Distributed Computing 43 (1997) 156–162.

[13] F. Berman, High-performance schedulers, in: I. Foster, C. Kesselman (Eds.), The Grid: Blueprint for a New Computing Infrastructure, Morgan-Kaufmann, 1999, pp. 279–309.