Laboratoire de l'Informatique du Parallélisme

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

Static LU Decomposition on Heterogeneous Platforms

Olivier Beaumont
Arnaud Legrand
Fabrice Rastello
Yves Robert

Dec 2000

Research Report N° 2000-44

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

Static LU Decomposition on Heterogeneous Platforms

Olivier Beaumont
Arnaud Legrand
Fabrice Rastello
Yves Robert

Dec 2000

Abstract

In this paper, we deal with algorithmic issues on heterogeneous platforms. We concentrate on dense linear algebra kernels, such as matrix multiplication or LU decomposition. Block cyclic distribution techniques used in ScaLAPACK are no longer sufficient to balance the load among processors running at different speeds. The main result of this paper is to provide a static data distribution scheme that leads to an asymptotically perfect load balancing for LU decomposition, thereby providing solid foundations toward the design of a cluster-oriented version of ScaLAPACK.

Keywords: heterogeneous platforms, different-speed processors, load-balancing, LU decomposition

Résumé

Dans ce rapport, nous nous intéressons au problème de la distributions de données pour des noyaux d'algèbre linéaire (tels que le produit de matrices ou la décomposition LU) adaptés aux plateformes hétérogènes. Les distributions cycliques par blocs utilisées dans ScaLAPACK ne sont plus adaptées à de telles plateformes et ne permettent pas d'obtenir un bon équilibrage de charge. Le résultat principal de cet article porte sur une technique de distribution des données permettant un équilibrage de charge asymptotiquement optimal pour les décompositions LU, et pouvant donc servir de bases solides à la mise en œuvre d'une version de ScaLAPACK adaptée aux grappes de grappes.

Mots-clés: plateformes de calcul hétérogènes, processeurs de vitesses différentes, équilibrage de charge, décomposition LU

1 Introduction

Extending static load-balancing techniques to cope with heterogeneous resources has a great practical significance: the future of parallel computing platforms is likely to be described by the key-words *distributed* and *heterogeneous*. Indeed, heterogeneous networks of workstations (HNOWs) are ubiquitous in university departments and companies, and they represent the typical poor man's parallel computer: running a large PVM or MPI experiment (possibly all night long) is a cheap alternative to buying supercomputer hours. The idea is to make use of all available resources, namely slower machines *in addition to* more recent ones. Furthermore, linking geographically scattered heterogeneous clusters of processors will give rise to *metacomputing grids*, as described in Foster and Kesselman's book [9].

The major limitation to programming heterogeneous clusters arises from the additional difficulty of balancing the load when using processors running at different speeds (see the survey paper of Berman [5], and the references therein). In this paper, we explain how to cope with this difficulty for some classical dense linear algebra kernels. The rest of the paper is organized as follows. In Section 2, we deal with matrix multiplication: we prove that designing static allocation schemes for this simple computational kernel is intrinsically difficult. In Section 3, we briefly review classical data distribution schemes for LU decomposition on homogeneous platforms, and we explain how to cope with heterogeneous clusters. We introduce a two-dimensional allocation scheme which ensures a good load-balancing throughout the algorithm. The problem is more difficult than for matrix multiplication, because the size of the matrix to be updated reduces at each step: a good load balancing for the first steps may well prove not suited to later updates. In Section 4, we prove the asymptotical optimality of the data distribution introduced in Section 3. To the best of our knowledge, this is the first result available in the literature on the complexity of LU decomposition with different-speed processors. Finally, we give some remarks and conclusions in Section 5.

2 Matrix Multiplication

In this section, we briefly survey previous results that we have obtained for matrix multiplication (MM) on top of HNOWs. We refer to [2, 3] for all proofs.

2.1 Matrix Multiplication on Homogeneous Grids

Assume first that we target a 2D homogeneous: the $p \times q$ processors are identical. In that case, ScaLAPACK uses a block version of the outer product algorithm[1] described in [1, 10, 13], which can be summarized as follows:

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

When the processor set is heterogeneous, extending this distribution turns out to be surprisingly difficult. The problem is (i) to arrange different-speed processors along the grid (all permutations must be investigated); and (ii), once the layout is given, to compute the best partition of the matrix to achieve the best

[1] ScaLAPACK uses a two-dimensional grid rather than a linear array for scalability reasons [6].

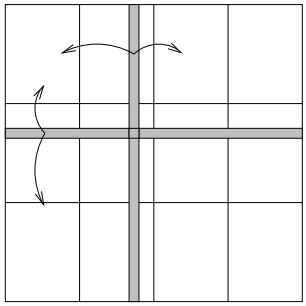


Figure 1: The MM algorithm on a 3×4 homogeneous 2D-grid.

possible load-balancing. In general, heterogeneous 2D grids do not lead to perfectly load-balanced solutions, and the above problem is NP-hard [2].

Fortunately, 2D grids are not likely to play a role in the future as important as they have done so far (from the Illiac IV to the Intel Paragon). If we relax the topology constraint stating that processors have to be arranged in a 2D grid, perfect load balancing can be easily achieved. Unfortunately, load-balancing alone is not the key of success for a parallel algorithm: communication overhead plays an important role too.

2.2 Matrix Multiplication on Heterogeneous Platforms

Consider again the product $C = A \times B$ with p heterogeneous processors. As before, the three matrices A, B and C are identically partitioned into p (superposed) rectangles, and there is a one-to-one mapping between these rectangles and the processors (each processor is responsible for updating its rectangle). But we relax the constraint stating that the processors have to be arranged into a 2D grid. See Figure 2 with $p = 13$ processors: in this example some processors have up to six direct neighbors.

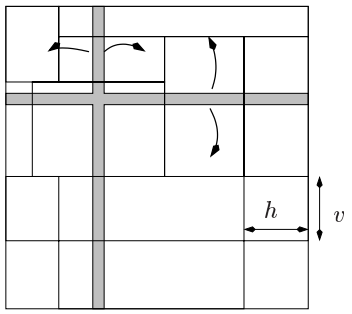


Figure 2: The MM algorithm on a heterogeneous platform.

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, $1 \leq i \leq p$. The question is: how to compute the *shape* of these p rectangles so as to minimize the total execution time?

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. In fact, this is not exactly true: because the pivot row and columns are not sent to the processors that own them, we should subtract 2 from $\hat{C}$, 1 for the horizontal communications and 1 for the vertical ones. Since minimizing $\hat{C}$ or $\hat{C} - 2$ is equivalent, we keep the value of $\hat{C}$ as stated. Minimizing $\hat{C}$ seems to be a very natural goal, because it represents the total volume of communications. For instance it is natural to assume that communications will be mostly sequential on a heterogeneous network of workstations where processors are linked by a simple Ethernet network; also, there

will be little or none computation/communication overlap on such a platform. In that context, minimizing the total communication volume is the main objective.

Conversely, some communications can occur in parallel, if the computing resources are linked through a dedicated high-speed network, and if parallel communication links are provided. In that context, we may want to minimize the *maximal* amount of communications to be performed by each processor, so that the objective function becomes $\hat{M} = \max_{1 \leq i \leq p}(h_i + v_i)$.

Once a solution to either optimization problem has been found, we derive the allocation of data elements to processor P_i by rounding up the values $n \times h_i$ and $n \times v_i$. Both optimization problems have a wide potential applicability. Forgetting about MM algorithms, consider the implementation of any application (such as a finite-difference scheme) where heterogeneous processors communicate boundary elements at each step (the communication scheme need not be nearest-neighbor, it can be anything): minimizing the total communication volume, or the maximal amount of communications performed by one processor, while load-balancing the work, amounts to solving exactly the same optimization problems.

Note that the problem of evenly distributing the load is easy to solve in this new context of partitioning the iteration space without any topological constraint on the communication grid: we can always tile the unit square into p horizontal slices of height $s_1, s_2, \ldots, s_p$. The difficulty now is to minimize the objective function, i.e. the communication overhead.

2.3 Complexity Results

Distributing matrices on a heterogeneous set of processors turns out to be equivalent to tiling the unit square with p rectangles R_i of prescribed area s_i (that represent the relative speed of processors), $1 \leq i \leq p$ where $\sum_{i=1}^{p} s_i = 1$. The shape of each R_i is the degree of freedom: we want to tile the unit square so as to solve the following optimization problems, depending upon the underlying communication network:

Definition 1 *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.*

Definition 2 *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.*

Theorem 1 *The decision problems associated to both previous optimization problems are NP-complete.*

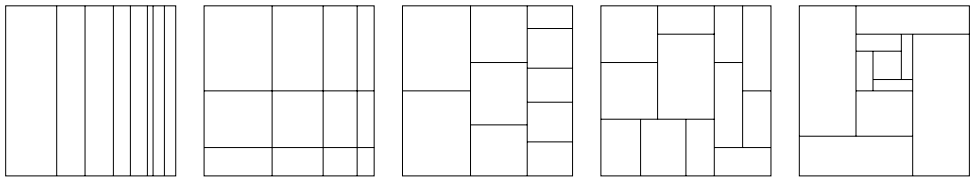


Figure 3: Taxonomy of unit square partitions

A full-length version of the proof is available in [3]. Figure 3 depicts a taxonomy of partitions of the unit square. Many optimization problems can be defined, depending on the partition type and on the underlying communication network; their complexity highly depends upon these parameters. For example, if we restrict the search to column-based partitions, i.e. partitions with the constraint that rectangles are assigned to columns (with possibly different numbers of rectangles per column, see Figure 3), 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 2 *COL-PERI-SUM(s) is polynomial while COL-PERI-MAX(s) is NP-complete.*

There are several "natural" heuristics to approximate PERI-SUM and PERI-MAX. However, proving approximation bounds turns out to be very technical. To approximate PERI-SUM and PERI-MAX, we propose two column-based heuristics in [3]. Both are very simple to implement and their efficiency has been proved through extensive experimental tests. In [4], a more complicated recursive heuristic for PERI-SUM has been considered, for which a nice approximation bound is provided. Complexity results for all problems are summarized in Table 1. Although several partitioning algorithms have been proposed in the literature by Crandall and Quinn [8], Kalinov and Lastovetky [12], and Kaddoura, Ranka and Wang [11], the complexity results stated in Table 1 are the first available (to the best of our knowledge).

<table>
<tr><th></th><th>1D</th><th>2D</th><th>Column-based</th><th>Recursive</th><th>Unconstrained</th></tr>
<tr><td>$\sum$</td><td rowspan="2">Polynomial</td><td rowspan="2">NP-hard
[2]</td><td>Polynomial [3]</td><td>???</td><td>NP-hard.
Guaranteed
heuristic with 5/4
bound. [4]</td></tr>
<tr><td>max</td><td>NP-hard
[2]
Guaranteed
heuristic with
$2/\sqrt{3}$ bound.</td><td>???</td><td>NP-hard.
Guaranteed
heuristic with $2/\sqrt{3}$
bound. [4]</td></tr>
</table>

Table 1: Complexity results.

3 Data Distribution Schemes for LU Decomposition

In this section, we tackle a more complicated kernel than MM, namely LU decompostion[2]. We show how to extend the well-known blocked algorithm provided in the ScaLAPACK library [6] to cope with heterogeneous resources.

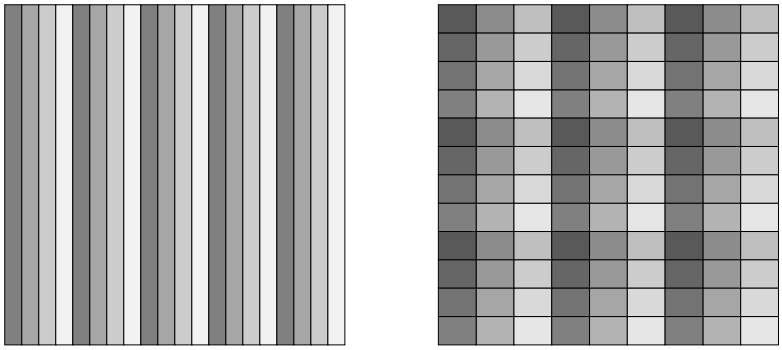
3.1 LU Decomposition on Homogeneous Platforms

The LU decomposition works as follows: at each step, the pivot processor processes the pivot panel (i.e. a block of r columns) and broadcasts it to other processors, so that they update their remaining columns. At the next step, the next r columns become the new pivot panel. Since most time is spent during the update phase, we need to distribute data to the processors so that the load is well balanced during all update phases, throughout the elimination.

The LU algorithm resemble the previous MM algorithm, except that at each step, the fraction of the matrix which remains to be processed evenly decreases both in width and height. This shrinking of the matrix imposes a dramatic change in the data distribution scheme. Indeed, if we use the distribution scheme described in Section 2.1, the processors in the left corner will be extensively used during the first n/pr steps, and then will be left idle, which would lead to a catastrophic load balancing.

This difficulty already arises in the context of homogeneous processors. ScaLAPACK [6] uses a block-cyclic distribution in both row and column dimensions. Indeed, mono-dimensional distributions are not scalable; $CYCLIC(r)$ distribution of both columns and rows (see Figure 4(b)) are preferred when the number of processors becomes large. On homogeneous NOWs, such a 2D block-cyclic distribution leads to a perfect load balancing of the update at each step.

[2] All results hold for QR decomposition, whose computation and communication patterns are quite similar to those of LU decomposition.



(a) Mono-dimensional block-cyclic allocation

(b) Bi-dimensional block-cyclic allocation

Figure 4: Homogeneous distributions for LU décompositions

3.2 LU Decomposition on Heterogeneous 1D Arrays

As discussed above, the heterogeneous distribution schemes discussed in Sections 2.2 and 2.3 are not suited to LU decomposition, because load-balancing must be ensured at each phase of the elimination. In this section, we concentrate on 1D heterogeneous grids (we will re-use the results for 2D heterogeneous grids later on). In this case, there exists a simple way to compute the best allocation.

We start from the following dynamic programming algorithm:

OPTIMAL_ALLOCATION($(s_1, \ldots, s_p)$, B)
1: $\mathcal{C} = (c_1, \ldots, c_p) = (0, \ldots, 0)$
2: **For** $b = 1$ **to** B **Do**
3: $\quad i = \text{argmin}_{1 \leqslant j \leqslant p}((c_j + 1)/s_j)$
4: $\quad \mathcal{A}(b) = i$; $c_i \leftarrow c_i + 1$
5: **Return** $\mathcal{A}$

ALGO. 1: Dynamic programming algorithm for the optimal allocation of B independent identical chunks on p heterogeneous processors of relative speeds $s_1, \ldots, s_p$.

For each value of $b \leq B$, let $\mathcal{C}^{(b)} = (c_1^{(b)}, \ldots, c_p^{(b)})$ denote the allocation of the first $b = \sum_{i=1}^{p} c_i$ chunks computed by the algorithm. This allocation is optimal [7]:

Theorem 3 *Given p processors with relative speeds $s_1, \ldots, s_p$, Algorithm 1 returns the optimal allocation of b independent chunks for each b, $1 \leq b \leq B$.*

Let us follow the execution of Algorithm 1 with 3 processors P_1, P_2 and P_3 of relative cycle times 3, 5 et 8 (i.e. relative speeds 0.506, 0.304 and 0.19). Assume $B = 10$. The b-th chunk is allocated to processor $P_{\mathcal{A}(b)}$, where $\mathcal{A}(b) = (1, 2, 1, 3, 1, 2, 1, 1, 2, 3)$.

For LU decomposition we allocate slices of B column blocks to processors, as illustrated in Figure 5. B is the period of the distribution, and can be chosen as a parameter by the user: we provide the best allocation for all values of B. When B is equal to the total number n of column blocks in the matrix, the allocation is the best possible. Within each slice of B consecutive blocks, we use the dynamic programming algorithm in a "reverse" order. Given B, we take the output $\mathcal{A}(B)$ of Algorithm 1 and we reverse it, as shown in Figure 6: the b-th chunk (for $1 \leq b \leq B$) is allocated to processor number $B - b + 1$. As illustrated in Figure 5, at a given step there are several slices of at most B chunks, and the number of chunks in the first slice decreases as the computation progresses (the leftmost chunk in a slice is computed first and then there only remains $B - 1$ chunks in the slice, and so on). In the example, the reversed allocation best balances the updates in

the first slice at each step: at the first step when there are the initial 10 chunks and 9 updates (the first chunk is not updated), but also at the second step when only 8 updates remain, and so on. The updating of the other slices remains well-balanced by construction, since their size does not change, and we keep the best allocation for $B = 10$. In Figure 6, in addition to the detailed allocation within a slice, we also show the cost of the updates: for example, we need only 16 time units to process 10 chunks with our allocation, while the standard block-cyclic allocation requires 24.

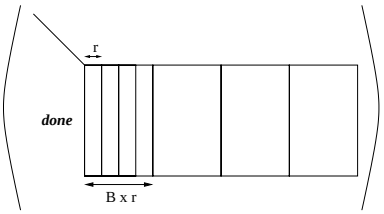


Figure 5: Allocating slices of B chunks, where a chunk is a block of r columns.

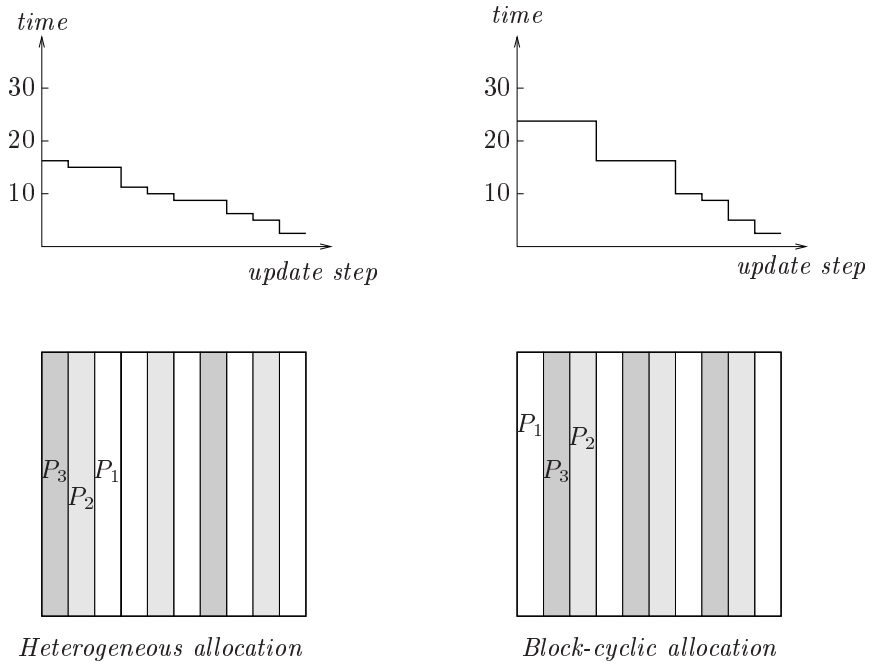


Figure 6: Comparison of the distribution given by Algorithm 1 and block-cyclic distribution for 3 processors with cycle time 3, 5 et 8, with $B = 10$.

3.3 LU Decomposition on Heterogeneous Platforms

During the decomposition, the trailing matrix that remains to be updated shrinks in both dimensions. We will use the same trick as for the unidimensional case: we compute the best allocation for $n \times n$ blocks, where n ranges from 1 to the total number of blocks in a dimension, and then we reverse it, so as to get a balanced allocation throughout the algorithm. Therefore, for the sake of convenience, we may reverse the process and concentrate on allocating chunks for $n \times n$ blocks, where n grows (but if we do so, we have to keep in mind that the final allocation will be the mirror in both dimensions of the result).

First we note that heterogeneous 2D grids are not likely to lead to well-balanced allocations throughout the algorithm, as illustrated in Figure 7. This is not surprising, because we were not able to achieve a

good load-balancing for the simpler MM kernel. At step 1 (actually the last step of the LU decomposition algorithm, since we have conceptually reversed the order), we have to allocate a single block in a given row and given column. Intuitively, we want to assign this block to the fastest processor, and this is the choice made in Figure 7. At step 2, we allocate another row and another column, assigning a total of 4 blocks (including the previous block). In Figure 7, we show that the best choice is to assign all four blocks to the fastest processor. At step 3, we pick another row and another column, allocating a total of 9 blocks, and so on. The objective is to have the allocation balanced at each step. In Figure 7, we show that it is not possible to find an allocation which is optimal for the first three steps. We would need to perform on-the-fly redistributions, which would be very costly.

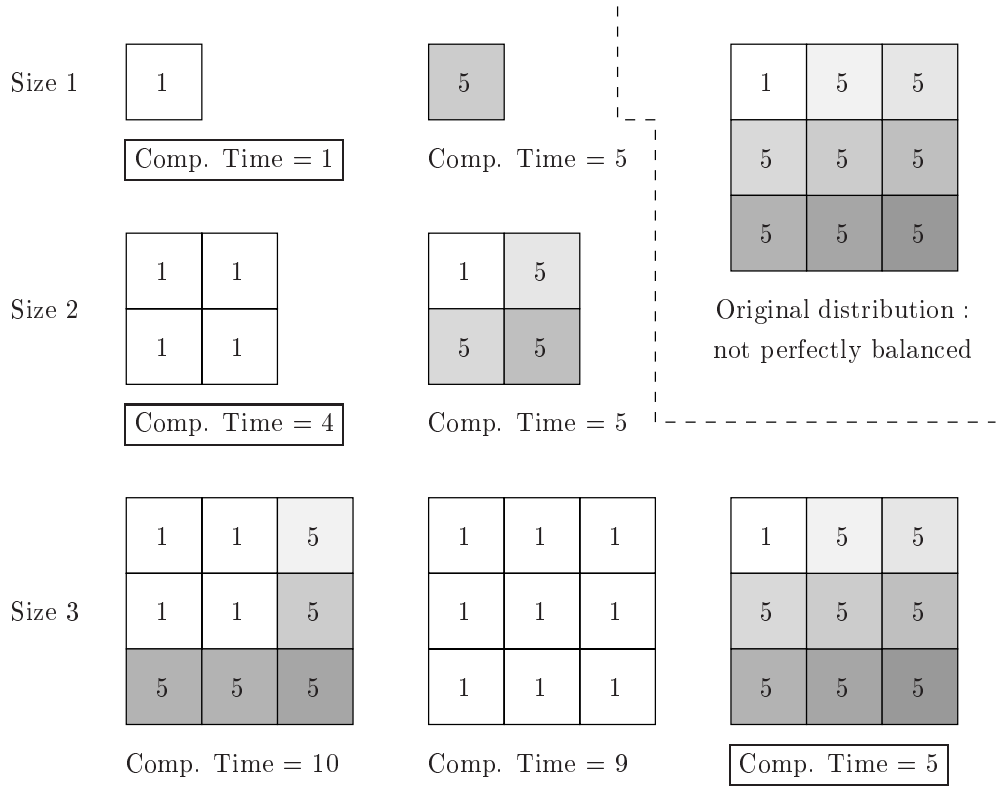


Figure 7: A 2D partition for a 3×3 processor grid. Relative processor cycle-times are $1, 5, 5, 5, 5, 5, 5, 5$ and 5. It is not possible to find a distribution which is optimal at each step: the optimal solution at step 3 cannot be obtained from any of the optimal distributions at step 2.

The previous load-balancing problem is due to the geometric rigidity of the two-dimensional processor grid. To circumvent the problem, we come back to column-based partitions, which provided perfectly balanced allocations for the MM kernel. The idea is to consider a column-based partition as a perfectly balanced *virtual* heterogeneous 2D grid by extending processors boundaries as shown in Figure 8. In this example, we started from a column-based partition for 10 processors, which we virtually transformed into a 8×3 grid. Recall that it is absolutely necessary to target a 2D layout of the processors to minimize communications; 1D allocations are not scalable.

Let r_i and c_j denote the height and width of the resulting decomposition into rectangles along the (virtual) 2D heterogeneous grid, as depicted in Figure 8. The idea is to use the dynamic programming scheme introduced in Section 3.2. Because this scheme leads to a uni-dimensional allocation, we use it twice, in both dimensions, and we merge the results, as explained below.

Consider the example represented in Figure 9. We started with 7 heterogeneous processors, whose initial rectangles are indicated with solid lines in the leftmost part of the figure. We virtualize the partition into a 6×3 grid, as indicated by the dash lines in the figure. The heights and widths of the new rectangles

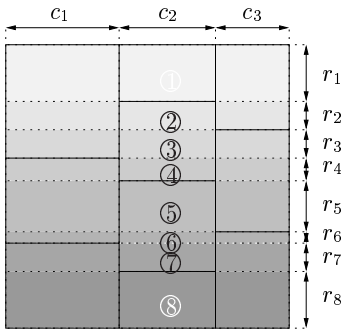


Figure 8: Virtualizing a column-based partition

are the following: $r = (\frac{3}{8}, \frac{5}{72}, \frac{7}{45}, \frac{7}{80}, \frac{7}{144}, \frac{5}{18})$ and $c = (\frac{1}{2}, \frac{8}{25}, \frac{9}{50})$. Running Algorithm 1 twice, we obtain the sequences $r_1, r_6, r_1, r_3, r_6, r_1, r_1, r_6, r_4, r_3, r_1, r_6, r_2, \ldots$ and $c_1, c_2, c_1, c_3, c_1, c_2, c_1, c_2, c_1, c_3, c_1, c_2, c_1, \ldots$. The resulting data distribution is depicted in the rightmost part of Figure 9. In the picture, we show the actual allocation, starting from the right-bottom of the matrix.

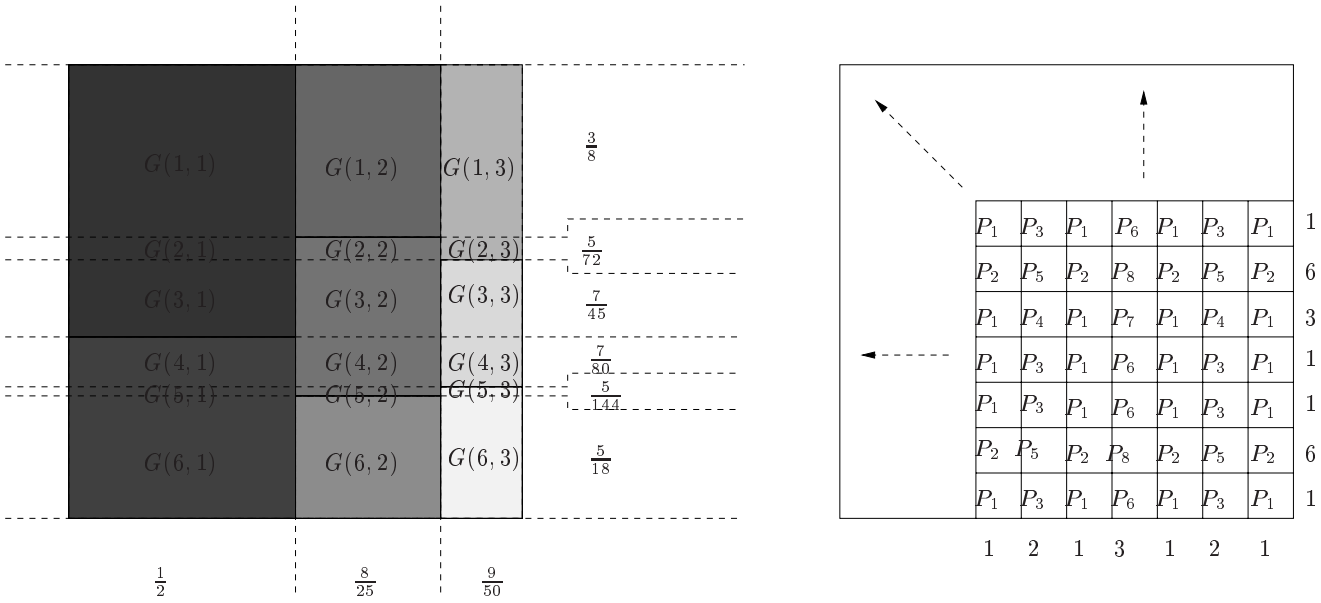


Figure 9: Data allocation for LU decomposition: the 2D dynamic programming scheme.

The following theorem asserts the asymptotical optimality (when the size of the matrix becomes large) of the above data distribution scheme:

Theorem 4 *The 2D dynamic programming algorithm defined above yields an asymptotically optimal allocation for LU decomposition.*

Proof Let $u_{i,j}$ denote the smallest rectangular zone owned by processor P_{ij} and $l_{i,j}$ denote its largest one. Thus, P_{ij} is responsible for updating those rectangular zones whose heights are labeled $r_{u_{i,j}}, r_{u_{i,j}+1}, \ldots, r_{l_{i,j}}$,

and whose width is c_j. Since this column-based distribution is perfectly balanced,

$$\left(\sum_{k=u_{i,j}}^{l_{i,j}} r_k \right) c_j = s_{ij} = \frac{1}{t_{ij}}. \tag{3.1}$$

Thus, the computational time required by the t^{th} step of the algorithm is larger than

$$T^{min}_{compl}(t) = \alpha t^2 \max_{i,j} \left(\sum_{k=u_{i,j}}^{l_{i,j}} r_k t_{ij} c_j \right) = \alpha t^2.$$

Now, $\forall t \in [\![1, n]\!]$, let $r_k^{(t)}$ denote the rows of (row) index k to be updated in the trailing sub-matrix $A_{[t,n]\times[t,n]}$ at step t. Similarly, let $c_j^{(t)}$ denote the columns of (column) index j in $A_{[t,n]\times[t,n]}$. We have

$$r_k^{(t)} t = r_k t + \varepsilon_k^{(t)} \text{ and } c_j^{(t)} t = c_j t + \eta_j^{(t)} \text{ with } \varepsilon_k^{(t)} \text{ et } \eta_j^{(t)} \in [-1, 1]. \tag{3.2}$$

The overall computational time for step t is therefore

$$\begin{aligned}
T_{compl}(t) &= \alpha t^2 \max_{i,j} \left(\sum_{k=u_{i,j}}^{l_{i,j}} r_k^{(t)} t_{ij} c_j^{(t)} \right) \\
&= \alpha t^2 \max_{i,j} \left(\frac{\sum_{k=u_{i,j}}^{l_{i,j}} r_k^{(t)}}{\sum_{k=u_{i,j}}^{l_{i,j}} r_k} \cdot \frac{c_j^{(t)}}{c_j} \right) && \text{(using (3.1))} \\
&= \alpha t^2 \max_{i,j} \left(\frac{\sum_{k=u_{i,j}}^{l_{i,j}} r_k + \varepsilon_k^{(t)}/t}{\sum_{k=u_{i,j}}^{l_{i,j}} r_k} \cdot \frac{c_j + \eta_j^{(t)}/t}{c_j} \right) && \text{(using (3.2))} \\
&= \alpha t^2 \max_{i,j} \left(\left(1 + \frac{\sum_{k=u_{i,j}}^{l_{i,j}} \varepsilon_k^{(t)}/t}{\sum_{k=u_{i,j}}^{l_{i,j}} r_k} \right) \cdot \left(1 + \frac{\eta_j^{(t)}/t}{c_j} \right) \right)
\end{aligned} \tag{3.3}$$

Thus,

$$\begin{aligned}
\sum_{t=1}^{n} T_{compl}(t) &= \sum_{t=1}^{n} \alpha t^2 \max_{i,j} \left(\left(1 + \frac{\sum_{k=u_{i,j}}^{l_{i,j}} \varepsilon_k^{(t)}/t}{\sum_{k=u_{i,j}}^{l_{i,j}} r_k} \right) \left(1 + \frac{\eta_j^{(t)}/t}{c_j} \right) \right) \\
&\leqslant \sum_{t=1}^{n} \alpha t^2 \left(1 + \frac{K_1}{t} \right) \left(1 + \frac{K_2}{t} \right) && \text{where } K_1 \text{ and } K_2 \text{ are constants} \\
&\leqslant \alpha \left(\sum_{t=1}^{n} t^2 + \sum_{t=1}^{n} (K_1 + K_2) t + \sum_{t=1}^{n} K_1 K_2 \right) && \text{depending neither on } n \text{ not on } t \\
\sum_{t=1}^{n} T_{compl}(t) &\leqslant \left(\sum_{t=1}^{n} \alpha t^2 \right) \left(1 + \mathcal{O}\left(\frac{1}{n} \right) \right) = \left(\sum_{t=1}^{n} T^{min}_{compl}(t) \right) \left(1 + \mathcal{O}\left(\frac{1}{n} \right) \right) && \text{, and then}
\end{aligned} \tag{3.4}$$

Therefore, the 2D dynamic programming heuristic yields to an asymptotically optimal solution for load-balancing the update phases throughout the execution of the LU decomposition algorithm. ■

The total communication overhead is given by

$$\begin{aligned} T_{comm}(t) &= \beta \sum_{i,j} t\left(c_j^{(t)} + \sum_{k=u_{i,j}}^{l_{i,j}} r_k^{(t)} \right) \\ &= \beta \sum_{i,j} \left(t\left(c_j + \sum_{k=u_{i,j}}^{l_{i,j}} r_k \right) + \left(\eta_j^{(t)} + \sum_{k=u_{i,j}}^{l_{i,j}} \epsilon_k^{(t)} \right) \right) \quad \text{(using (3.2))} \\ &= \beta t \left(\sum_r h_r + v_r \right) + \mathcal{O}(1) \end{aligned}$$

Thus,

$$\begin{aligned} \sum_{t=1}^{n} T_{comm}(t) &= \sum_{t=1}^{n} \beta t \left(\sum_r h_r + v_r \right) + \mathcal{O}(1) \\ &\leqslant \left(\sum_{t=1}^{n} \beta t \right) \left(\sum_r h_r + v_r \right) \left(1 + \mathcal{O}\left(\frac{1}{n} \right) \right) \end{aligned} \tag{3.5}$$

■

We note that the overall communication cost is roughly twice smaller for LU decomposition than for matrix multiplication, just as for homogeneous platforms. Since we did not explicitly use the fact that the initial partitioning was column-based, Theorem 4 can be straightforwardly extended to more general (recursive or unconstrained) partitionings.

4 Simulations

In this section, we compare the efficiency of several data distribution schemes for LU decomposition through some simulations. We compare the (expected) computational time of LU decomposition when the following data allocation schemes are used:

Block-cyclic This is the classical 1D block-cyclic allocation (see Figure 4(a)),

MM column-based partition This is the full block column-based partitioning designed for matrix multiplication (see sections 2.2 and 2.3). The load is expected to be well-balanced in the first phases of the decomposition.

2D greedy heuristic . This heuristic is based on the column-based partitioning obtained for MM. It builds the allocation by choosing iteratively (in a greedy fashion) the horizontal and vertical zones that minimize the computational time. In the example of Figure 10, we illustrate the first three steps. At step 1, the block is chosen from the fastest processor in (virtual) grid position ($\textcircled{1}$,$\mathcal{A}$), i.e. processor 1. At step 2 we select a block from processor 4 and at step 3 a block from processor 2.

2D dynamic programming algorithm . This is the approach described in Section 3.3.

In Figures 11 and 12, we compare the ratio between the (estimated) total execution time for the previous four data distribution schemes against the absolute lower bound (that of a perfect load balancing, which cannot be reached in general, as shown in Figure 7). In the first simulation (see Figure 11), the processor cycle times are 1, 5, 5, 5, 5, 5, 5, 5 and 5. The results of the simulation with another set of processors, with relative speeds 0.1, 0.1, 0.12, 0.15, 0.15, 0.18 and 0.2, are depicted in Figure 12. In both cases, the 2D dynamic programming leads to the best results and is checked to be asymptotically optimal.

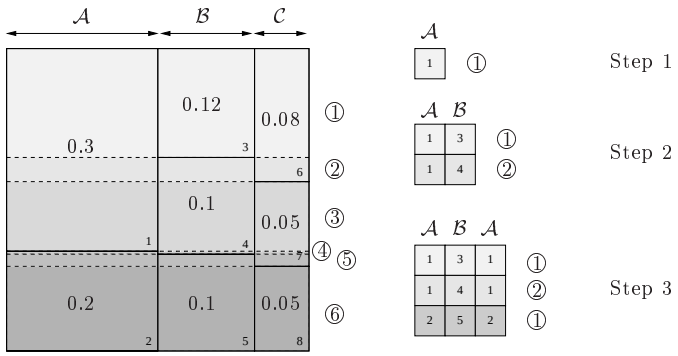


Figure 10: 2D Greedy heuristic

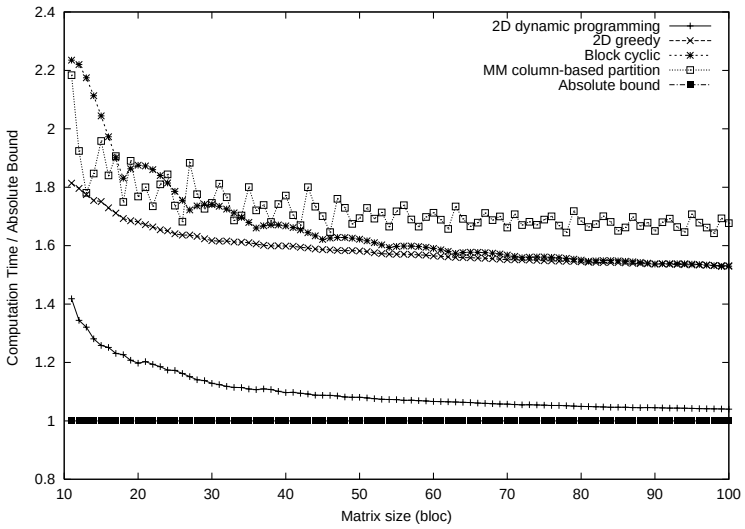


Figure 11: Comparison of various data allocation schemes for processors with cycle-time 1, 5, 5, 5, 5, 5, 5, 5 and 5.

5 Conclusion

In this paper, we have shown that designing efficient static data distribution strategies on heterogeneous platforms turns out to be a very difficult problem, even on simple (though important) computational kernels such as matrix multiplication or LU decomposition.

We have explained how static data distribution schemes for matrix multiplication could be extended to LU decomposition. platforms. The dynamic programming algorithms that we propose for LU decomposition are (i) optimal for 1D distributions; and (ii) asymptotically optimal for general 2D distributions. Furthermore, since the distribution scheme used for LU decomposition consists in a permutation of both rows and columns of the distribution scheme for matrix multiplication, it can be used for both kernels (exactly as in the homogeneous case, where the CYCLIC(r) allocation of LU can be used for matrix multiplication instead of a full-block allocation).

In the context of heterogeneous computations, purely static strategies may encounter several difficulties. Indeed, the network may or may not be dedicated the user, and unpredictible variations in the processor speeds may occur. We believe that load imbalance problems due to changes in the processor speeds can be addressed by remapping both data and computations between well identified static phases. All the heuristics that we have developed yield column based distributions, thus leading to a unified framework for enabling

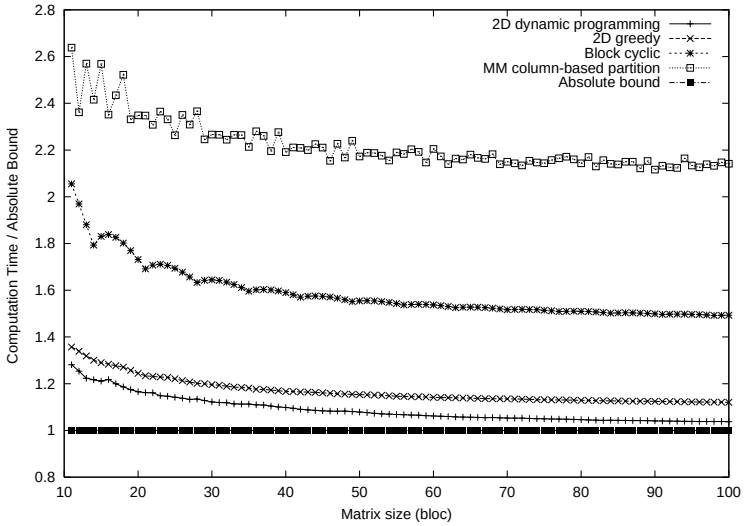


Figure 12: Comparison of various data allocation schemes for processors with relative speed 0.1, 0.1, 0.12, 0.15, 0.15, 0.18 and 0.2.

remapping strategies. For tightly-coupled linear algebra kernels, purely dynamic approaches would be killed by redistribution costs, so static strategies deployed within dynamic phases could prove the right tradeoff.

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] O. Beaumont, V. Boudet, A. Legrand, F. Rastello, and Y. Robert. Heterogeneity considered harmful to algorithm designers. Technical Report RR-2000-24, LIP, ENS Lyon, June 2000. Available at `www.ens-lyon.fr/LIP/`.

[3] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert. Matrix-matrix multiplication on heterogeneous platforms. Technical Report RR-2000-02, LIP, ENS Lyon, January 2000. Short version appears in the proceedings of ICPP'2000.

[4] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert. Partitioning a square into rectangles: NP-completeness and approximation algorithms. Technical Report RR-2000-10, LIP, ENS Lyon, February 2000.

[5] 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.

[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] Pierre Boulet, Jack Dongarra, Fabrice Rastello, Yves Robert, and Frédéric Vivien. Algorithmic issues on heterogeneous computing platforms. *Parallel Processing Letters*, 9(2):197–213, 1999.

[8] P.E. Crandall and M.J. Quinn. Block data decomposition for data-parallel programming on a heterogeneous workstation network. In *2nd International Symposium on High Performance Distributed Computing*, pages 42–49. IEEE Computer Society Press, 1993.

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

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

[11] M. Kaddoura, S. Ranka, and A. Wang. Array decomposition for nonuniform computational environments. *Journal of Parallel and Distributed Computing*, 36:91–105, 1996.

[12] 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.

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