Static Data Allocation and Load Balancing Techniques for Heterogeneous Systems

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

August 2001

Abstract

In this paper, we review static data allocation and load balancing techniques for heterogeneous platforms. Data distribution techniques must obey a more refined model than standard block-cyclic distributions to equally balance the load between processors of different speeds. We show that deriving efficient distribution schemes is a rather simple task for linear networks but turns out to be surprisingly difficult for multi-dimensional problems. We also discuss several variants of the master-slave paradigm for different-speed processors.

Corresponding author
Yves Robert
LIP, Ecole Normale Superieure de Lyon
69364 Lyon Cedex 07, France
e-mail: Yves.Robert@ens-lyon.fr

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 [19] or MPI [39] experiment (possibly all night long) is a cheap alternative to buying supercomputer hours. The idea is to make use of *all* available resources, namely slower machines *in addition to* more recent ones.

The major limitation to programming heterogeneous platforms arises from the additional difficulty of balancing the load when using processors running at different speeds. This paper is devoted to discussing static data allocation techniques that will enable to achieve this goal. With processors running at different speeds, block-cyclic distribution, as used in the ScaLAPACK library [8] or in High Performance Fortran [29], is no longer enough; new data distribution schemes must be determined and analyzed. We show that deriving efficient distribution schemes is a rather simple task for linear networks but turns out to be surprisingly difficult for multi-dimensional problems. We also discuss several variants of the master-slave paradigm for different-speed processors: for some variants, simple solutions can be designed, while for some other variants the complexity is still unknown (to the best of our knowledge).

The rest of the paper is organized as follows. In Section 2, we target heterogeneous networks of workstations configured as linear arrays. We investigate data allocation schemes to evenly balance the workload for several computational kernels on such platforms. Next, in Section 3, we discuss in depth the matrix multiplication algorithm, and we show the intrinsic difficulty of static allocation schemes for heterogeneous clusters. Then, in Section 4, we revisit the master-slave paradigm with different-speed slaves. We discuss related work in Sections 3.2.6 and 4.5. Finally, we give some remarks and conclusions in Section 5.

2 Static Data Allocation Strategies for Linear Arrays

In this section we target heterogeneous networks of workstations configured as linear arrays. We investigate data allocation schemes to evenly balance the workload for several computational kernels on such platforms. We start with an example (a simple image processing kernel) to explain why classical block-cyclic distribution is not suited to different-speed processors (Section 2.1). Next, we deal with the simple problem of distributing independent chunks of computations to linear arrays of heterogeneous processors (Section 2.2). We use this result to design optimal allocation schemes for finite-difference computations over a tiled iteration space (Section 2.3). Finally, we tackle the implementation of dense linear solvers, for which we propose an optimal data distribution in Section 2.4.

2.1 A Motivating Example

The main body of a "red sweep" in a distance transform [38] or path planning [7] algorithm is expressed in Algorithm 1.

Algorithm 1 The main body of a "red sweep".

For $i = 1$ **To** n **Do**
  For $j = 1$ **To** n **Do**
    $a(i,j) = f\big(a(i-1,j-1), a(i-1,j), a(i-1,j+1), a(i,j-1), a(i,j)\big)$

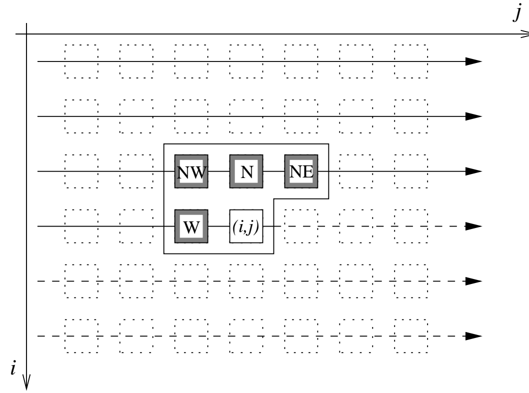

Figure 1: The updating mask in a forward sweep.

Each pixel (or position in path planning) (i, j) is updated using a mask made up with its north-west, north, north-east and west neighbors (see Figure 1). Because of the dependences, all pixels cannot be updated simultaneously; however there is some potential parallelism in the algorithm, as illustrated by Figure 2, where time-steps for ASAP (*as soon as possible*) execution are reported.

P_1	0	1	2	3	4	5	6	7	8	9	...
P_2	2	3	4	5	6	7	8	9	...		
P_3	4	5	6	7	8	9	...				
P_4	6	7	8	9	...						
P_5	8	9	...								
...	...										

Figure 2: The first 9 time-steps for ASAP execution with 5 homogeneous processors of cycle-time $t = 1$. No communication time is taken into account.

How should pixel rows be distributed to processors? Assume first that we have p homogeneous processors. A good strategy is to use a cyclic allocation of rows to processors. As soon as a new pixel (i, j) is computed by the processor that is assigned row i, the new value is sent to the processor that is assigned row $i + 1$. Computation progresses smoothly in parallel.

Assume now that we target a heterogeneous platform. Consider a toy example with 3 processors P_1, P_2 and P_3 of respective (normalized) cycle-times $t_1 = 1$, $t_2 = 2$ and $t_3 = 4$. This means that P_1 is twice as fast as P_2 and four times faster than P_3: one pixel is executed within 1 time-step by P_1, within 2 time-steps by P_2 and within 4 time-steps by P_3. Suppose that the same cyclic allocation is used. Row 1 is assigned to P_1, which starts computing at time-step $t = 0$. At $t = 2$, P_1 has updated the first two pixels of row 1, so P_2 can start executing row 2. It takes 4 time-steps to P_2 to update the first two pixels of row 2. At step $t = 6$, P_3 starts updating its first pixel in row 3. But processor P_1 will compute its row much faster than the others; when finished, it will start executing row 4. But P_1 cannot execute this row any faster that P_3 executes its own, because of the vertical dependences. Very soon, both P_1 and P_2 will be forced to operate at the slower speed of P_3.

However, more sophisticated distributions can be chosen. For instance, we can allocate rows by chunks of 7 consecutive rows. Out of each chunk, the first 4 rows go to P_1, the next 2 go to P_2 and the last row goes to P_3. The intuitive idea is to assign to each processor an amount of work that is

P_1	0	1	3	6	10	14	18	22	26	30 ...
P_1	2	4	7	11	15	19	23	27	31	...
P_1	5	8	12	16	20	24	28	32	...	
P_1	9	13	17	21	25	29	33	...		
P_2	14	18	22	26	30	34	...			
P_2	20	24	28	32	36	...				
P_3	26	30	34	38	...					
...	...									

Figure 3: Execution with the static periodic allocation, where $t_1 = 1$, $t_2 = 2$ and $t_3 = 4$

inversely proportional to its cycle-time. Note that this new distribution requires a new scheduling to be efficient: processors should not progress row by row any longer; instead, they must give priority to updating new pixels in their last row as soon as possible, as illustrated in Figure 3, where the computational wavefront appears at time-step $t = 22$: the computation progresses downwards along a diagonal line, and the work is perfectly balanced amongst the three processors, despite their different speeds.

To summarize, this simple example is intended to show the need to resort to more complicated allocation strategies than the standard block-cyclic schemes advocated for homogeneous platforms.

2.2 Distributing Independent Chunks

Given B independent chunks of computations, each of equal size (i.e. each requiring the same amount of work), how can we assign these chunks to p physical processors $P_1, P_2, \ldots, P_p$ of respective cycle-times $t_1, t_2, \ldots, t_p$, so that the workload is best balanced? Here the execution time is understood as the number of time units needed to perform one chunk of computation, i.e. each processor P_i executes each computation chunk within t_i time units. Then how to distribute chunks to processors? The intuition is that the load of P_i should be inversely proportional to t_i. Since the loads (i.e. number of chunks) on each processor must be integers, we use the following algorithm to solve the problem, where c_i denotes the number of chunks allocated to processor P_i. Thus, the overall execution obtained with an allocation $C = (c_1, c_2, \ldots, c_p)$ is given by $\max_{1 \le i \le p} c_i t_i$.

Algorithm 2 gives the optimal allocation [5]. It can be applied to simple load balancing problems such as matrix product on a processor ring: indeed, such a program can be decomposed into successive communication-free steps.

Each step consists of a bunch of independent chunks that can be distributed using Algorithm 2. Consider a toy example with 3 processors of respective cycle-times $t_1 = 3$, $t_2 = 5$ and $t_3 = 8$. We aim to compute the product $C = A \times B$, where A and B are of size 2496×2496. The matrices can be decomposed into 78×78 square blocks of size 32×32 (32 is a typical block size for cache-based workstations [8]). Hence, $B = 78$ blocks of columns have to be distributed among the processors, and $B = 78$ independent chunks will be computed at each step. Table 1 applies Algorithm 2 to this load balancing problem.

A, B and C share the same layout: P_1 owns the first c_1 blocks of columns (i.e. $M[*, 1 : bc_1]$), P_2 owns the following c_2 blocks (i.e. $M[*, bc_1 + 1 : b(c_1 + c_2)]$), and P_3 owns the remaining c_3 blocks of columns (i.e. $M[*, b(c_1 + c_2) + 1 : b(c_1 + c_2 + c_3)]$). At step $0 \le t < \frac{n}{b}$, the processor P_i that owns the tth bloc of columns of A broadcasts it to all the other processors. Thus, each processor owns a block of columns ($A[*, bt + 1 : b(t + 1)]$) and some parts of the tth block of rows of B (resp.

Algorithm 2 Optimal distribution for B independent chunks, over p processors of cycle-times $t_1, \ldots, t_p$.

DISTRIBUTE($B, t_1, t_2, \ldots, t_p$)

{Initialization: Approximate the c_i so that $c_i \times t_i \approx Constante$ and }

{$c_1 + c_2 + \ldots + c_p \leq B$}

1: **For** $i = 1$ **To** p **Do**

2: $c_i = \left\lfloor \frac{\frac{1}{t_i}}{\sum_{i=1}^{p} \frac{1}{t_i}} \times B \right\rfloor$

{Iteratively increment some c_i that minimize the execution time until $\sum_{k=1}^{p} c_i < B$}

3: **While** $\sum_{i=1}^{p} c_i < B$ **Do**

4: $i = \text{argmin}_{1 \leq j \leq p}(t_j(c_j + 1))$

5: $c_j = c_j + 1$

6: **Return** $(c_1, c_2, \ldots, c_k)$

Steps	c_1	c_2	c_3	$\max_i(c_i t_i)$
$\sum c_i = 76$ (initialization)	39	23	14	117
$\sum c_i = 77$	40	23	14	120
$\sum c_i = 78 = B$	40	24	14	120

Table 1: Steps of Algorithm 2 for 3 processors with $t_1 = 3$, $t_2 = 5$ and $t_3 = 8$, and $B = 78$.

$B[bt+1 : b(t+1), 1 : bc_1]$, $B[bt+1 : b(t+1), bc_1+1 : b(c_1+c_2)]$ and $B[bt+1 : b(t+1), b(c_1+c_2)+1 : b(c_1+c_2+c_3)]$). Each processor P_k updates independently its C submatrix with these blocks of columns and of rows. A few different steps for matrix multiplication are represented in Figure 4. The previous computation being has a cost proportional to $c_k t_k$ for each processor P_k, hence the whole processing is optimally load-balanced.

2.3 Finite-Difference Computations

Consider the two-dimensional rectangular iteration space represented in Figure 5. Tiles are rectangular, and their edges are parallel to the axes. All tiles have the same fixed size. The size of the tiled iteration space is $N_1 \times N_2$. Dependences between tiles are summarized by the vector pair

$$\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}.$$

In other words, the computation of a tile cannot be started before both its left and lower neighbor tiles have been executed. Such computations are representative of finite-difference methods [44].

Assume that there are p different-speed processors P_i of cycle-time t_1, $1 \leq i \leq p$. Assume also that columns of tiles are assigned to processors, so as to increase both the locality and the granularity of the computations. When targeting a homogeneous array, a natural way to allocate tile columns to physical processors is to use a pure cyclic allocation [37, 25, 3]. For heterogeneous arrays, we use a periodic allocation derived from Algorithm 2. More precisely, blocks of contiguous columns are cyclically allocated to the processors. The period is a parameter B given by the user (for instance $B = N_2$, the total number of columns). We use Algorithm 2 to determine the number c_i of columns associated to P_i, and we sort the quantities $c_i t_i$. At the price of some index changes, let $c_1 t_1 \leq c_2 t_2 \leq \ldots\ldots c_p t_p$. Then we assign the first c_1 columns to P_1, the next c_2 columns to

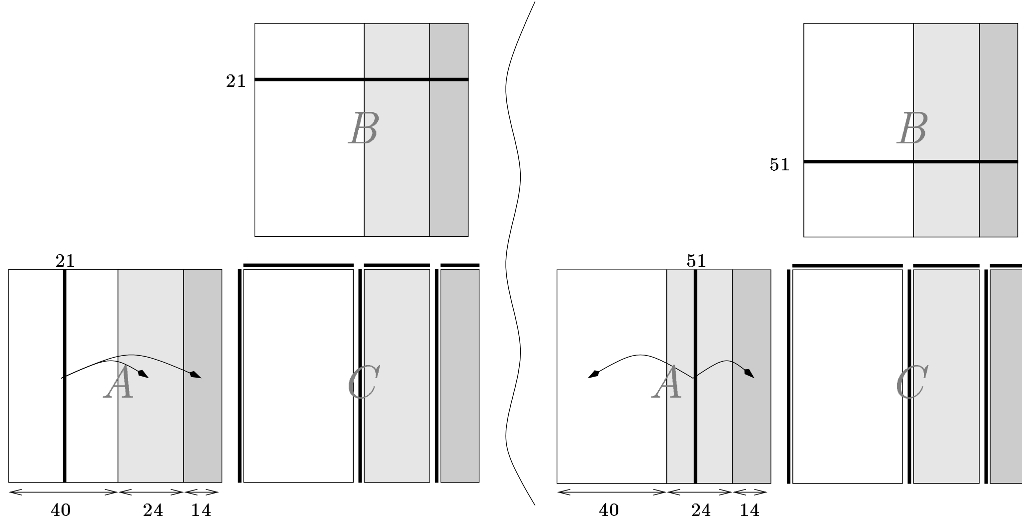

Figure 4: Different steps (21 and 51) of matrix multiplication on a platform made of 3 heterogeneous processors of respective cycle-times $t_1 = 3$, $t_2 = 5$ and $t_3 = 8$. All indices in the Figure are block numbers.

P_2, and so on, until B columns are assigned. If needed, we iterate the process for the remaining columns. This allocation is illustrated in Figure 6, again using 3 processors such that $t_1 = 3$, $t_2 = 5$ and $t_3 = 8$. For $B = 10$, Algorithm 2 returns $c_1 = 5$, $c_2 = 3$ and $c_3 = 2$.

As in the motivating example (for which the same allocation would be used), the scheduling must not be the same as for homogeneous processors: rather than computing its tile columns one after the other, each processor processes tiles horizontally, so as not to delay the next processors. Owing to the condition $c_1 t_1 \leq c_2 t_2 \leq \ldots\ldots c_p t_p$, processors will be kept fully active after the initialization phase: while, say, P_1 is computing the third row of its c_1 columns, P_2 is computing the second row of its c_2 columns, and P_3 is computing the first row of its c_3 columns.

In a word, our solution consists in allocating panels of B tile columns to the p processors in a periodic fashion. Inside each panel, processors receive an amount of columns inversely proportional to their speed. Within each panel, the work is well-balanced, and dependences do not slow down the execution.

2.4 Linear System Solvers

The previous data allocation technique is not adapted for LU and QR decompositions, as explained below. Roughly speaking, the LU decomposition algorithm works as follows on a homogeneous linear array [13]. The preferred distribution is a `CYCLIC(b)` distribution of columns. At each step, the processor that owns the pivot panel factors it and broadcasts it to all the processors, which update their remaining column blocks. At the next step, the next block of b columns becomes the pivot panel, and the computation progresses.

Because the largest fraction of the work takes place in the update, we would like to load-balance the work so that the update is best balanced. Consider the first step. After the factorization of the first block, all updates are independent chunks: here a chunk consists of the update of a single block of b columns. If the matrix size is $n = B \times b$, there remains $B - 1$ chunks to be updated. We can use Algorithm 2 to distribute these independent chunks. But the size of the matrix shrinks as the computation goes on. At the second step, the number of blocks to update is only $B - 2$. If we

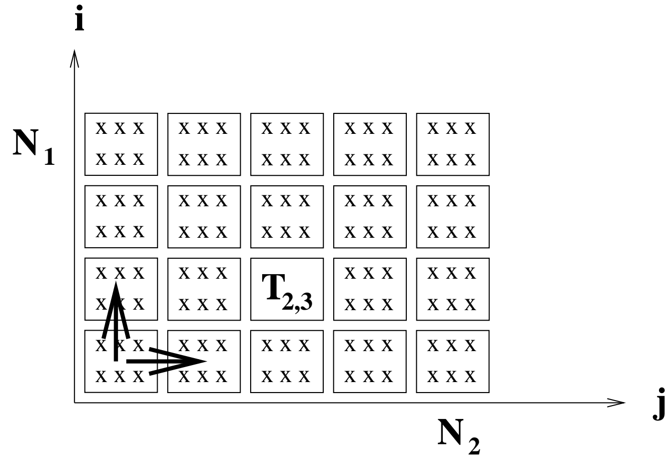

Figure 5: A tiled iteration space with horizontal and vertical dependences.

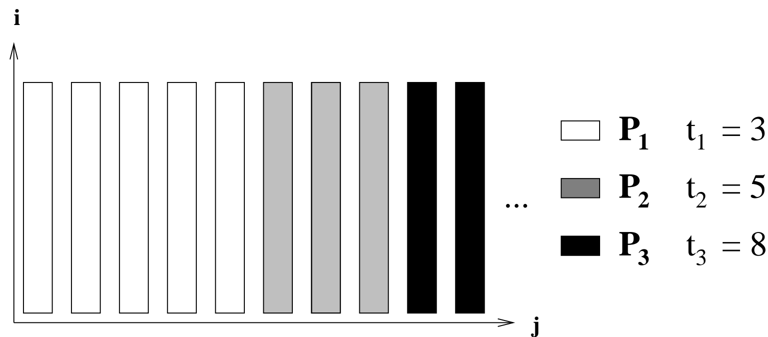

Figure 6: The periodic allocation for 3 processors with $t_1 = 3$, $t_2 = 5$ and $t_3 = 8$, and $B = 10$.

want to distribute these chunks independently of the first step, redistribution of data will have to take place between the two steps, and this will incur a lot of communications. Rather, we search for a static allocation of column blocks to processors that will remain the same throughout the computations, as the decomposition progresses. We aim at balancing the updates of all steps with the same allocation. As illustrated in Figure 7, we need a distribution that is kind of repetitive (because the matrix shrinks) but not fully cyclic (because processors have different speeds).

Looking closer at the successive updates, we see that only column blocks of index $i+1$ to B are updated at step i. Hence our objective is to find a distribution such that for each $i \in \{2, \ldots, B\}$, the amount of blocks in $\{i, \ldots, B\}$ owned by a given processor is approximately inversely proportional to its cycle-time (proportional to its speed). To derive such a distribution, we use the Algorithm 2 without the initialization phase of lines 1-2. This algorithm was introduced in [10] and is best explained using the former toy example.

2.4.1 A Dynamic Programming Algorithm

Consider an example with 3 processors of (relative) cycle-times $t_1 = 3$, $t_2 = 5$, and $t_3 = 8$. In Figure 8, we report the allocations found by the algorithm up to $B = 10$. The entry "Selected processor" denotes the rank of the processor chosen to build the next allocation. At each step, "Selected processor" is computed so that the cost of the allocation is minimized. The cost of the allocation is computed as follows: let us denote by c_i the number of chunks allocated to processor P_i,

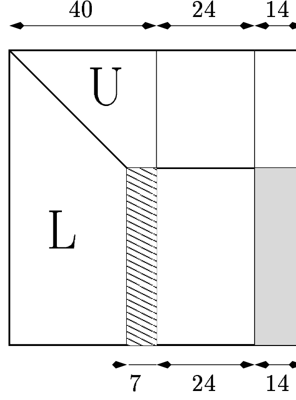

Figure 7: 33rd step of LU decomposition (indices are block numbers): with the former distribution, the computation becomes less balanced. Here, after factoring block 33, processor 1 has 7 updates and works for $7 \times 3 = 21$ units of time, while processor 2 works $24 \times 5 = 120$ units of time.

then the execution time for an allocation $\mathcal{C} = (c_1, c_2, \dots, c_p)$ is $\max_{1 \leq i \leq p} c_i t_i$ (the maximum is taken over all processor execution times). So, the average cost to execute one chunk is $\hat{\mathcal{C}} = \frac{\max_{1 \leq i \leq p} c_i t_i}{\sum_{i=1}^{p} c_i}$.

Cycle-time	$t_1 = 3$	$t_2 = 5$	$t_3 = 8$		
Chunk Size	c_1	c_2	c_3	Cost	Selected proc.
0	0	0	0		P_1
1	1	0	0	3	P_2
2	1	1	0	2.5	P_1
3	2	1	0	2	P_3
4	2	1	1	2	P_1
5	3	1	1	1.8	P_2
6	3	2	1	1.67	P_1
7	4	2	1	1.71	P_1
8	5	2	1	1.87	P_2
9	5	3	1	1.67	P_3
10	5	3	2	1.6	

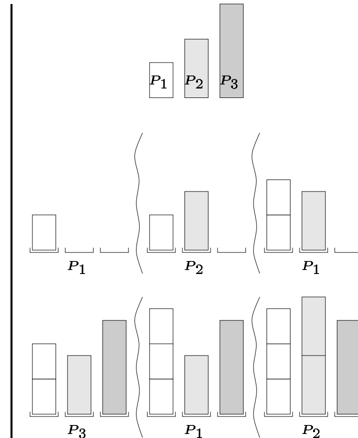

Figure 8: Running the dynamic programming algorithm with 3 processors: $t_1 = 3$, $t_2 = 5$, and $t_3 = 8$.

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

Of course, if we are to allocate 10 chunks, we can use Algorithm 2 and find that 5 chunks should be given to processor P_1, 3 to P_2 and 2 to P_3. But the dynamic programming algorithm returns the optimal allocation for any subset of chunks $[1, s]$, where $s \leq B$ (see [10] for the proof). Note that the cost of the allocations is not a decreasing function of B.

2.4.2 Application to LU Decomposition

For LU decomposition we allocate slices of B blocks of width b to processors, as illustrated in Figure 9. B is a parameter to be discussed below. For a matrix of size $nb \times nb$, we can simply let $B = n$, i.e. define a single slice.

Within each slice, we use the dynamic programming algorithm in a "reverse" order. In other words, the k-th chunk (for $1 \le k \le B$) is allocated to processor $\sigma(B - k + 1)$. Consider the toy example in Table 1 with 3 processors of relative speed $t_1 = 3$, $t_2 = 5$ and $t_3 = 8$. The dynamic programming algorithm allocates chunks to processors as shown in Table 8. Hence, the obtained pattern is $(P_3 P_2 P_1 P_1 P_2 P_1 P_3 P_1 P_2 P_1)$ (see Figure 10 for the detailed allocation within a slice). As illustrated in Figure 9, 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 $B = 10$. See Figure 10 for the detailed allocation within a slice, together with the cost of the updates.

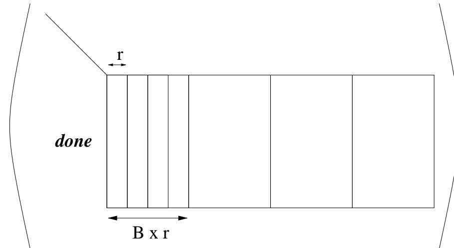

Figure 9: Allocating slices of B chunks.

This periodic data allocation scheme is at the heart of the extension of the ScaLAPACK library devoted to dense linear solvers such as LU or QR factorizations, as proposed in [5]. Note that a major advantage of a fully static distribution with a fixed parameter B is that we can use the current ScaLAPACK release with little programming effort. In the homogeneous case with p processors, we use a `CYCLIC(b)` distribution for the matrix data, and we define p processes. In the heterogeneous case, we still use a `CYCLIC(b)` distribution for the data, but we define B processes which we allocate to the p physical processors according to the load-balancing strategy.

3 Static Data Allocation Strategies for Geometric Problems

Contrarily to the uni-dimensional case, static strategies turn out to be surprisingly difficult for multi-dimensional problems. To illustrate this difficulty, we take the example of matrix multiplication (MM) problem.

3.1 MM Algorithms

In this section we briefly describe how to implement a parallel (or distributed) MM algorithm on a heterogeneous platform. We adopt an abstract view by assuming that we have a collection of p heterogeneous computing resources $P_1, P_2, \dots, P_p$. If each computing resource P_i reduces to a single

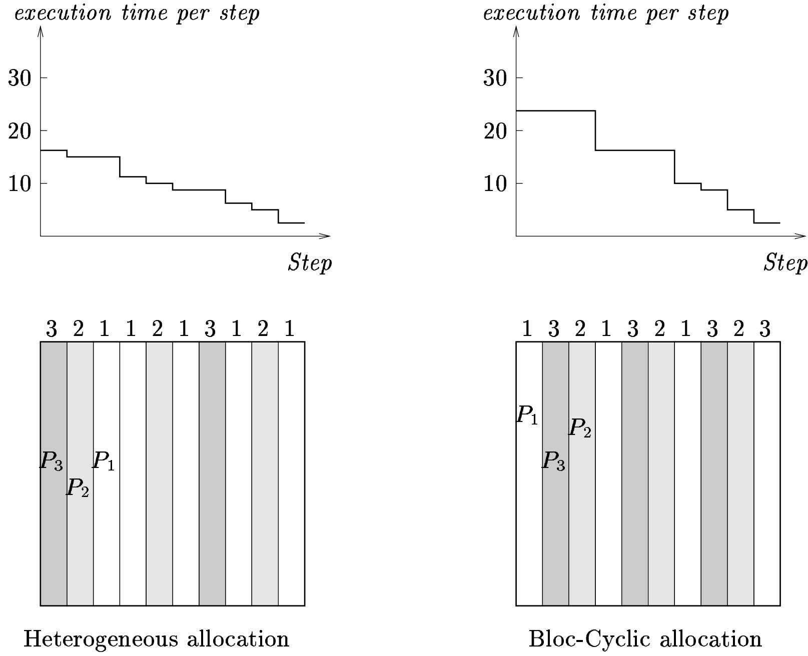

Figure 10: Comparison of two different distributions for the LU-decomposition algorithm on a heterogeneous platform made of 3 processors of relative cycle-time 3, 5 and 8. The first distribution is the one given by our algorithm, the second one is the cyclic distribution. The total number of chunks is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

processor, we are dealing with a heterogeneous network of workstations or PCs (HNOW). When each computing resource P_i is itself a heterogeneous cluster or a parallel machine, we are targeting a metacomputing environment, made up from a collection of clusters. The high-level algorithmic description is the same for all target machines. However, our model will have to cope with different hypotheses on communication issues. We come back to the impact of communication modeling in Section 3.2.2. Before dealing with heterogeneous resources, we briefly summarize existing algorithms for homogeneous machines.

3.1.1 Homogeneous Grids

We start by briefly recalling the MM algorithm implemented in the ScaLAPACK library [8] on 2D 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, 33]. Consider a 2D processor grid of size $p \times q$, and assume for a while 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 $P_{i,1:q}$.
- each processor $P_{k,j}$ (for all $j \in \{1, .., q\}$) vertically broadcasts $b_{k,j}$ to processors $P_{1:p,j}$.
- so that each processor $P_{i,j}$ can independently update $c_{i,j} = c_{i,j} + a_{i,k} b_{k,j}$.

This current version of the ScaLAPACK library 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, 12]. Each matrix coefficient in the description above is replaced by a $b \times b$ square block, where optimal values of b depend on the memory hierarchy and on the communication-to-computation ratio of the target computer. Hence from now, a matrix of size $nb \times nb$ is virtually considered as a $n \times n$ matrix where atoms are $b \times b$ blocks. Finally, a level of virtualization is added: usually, the number of blocks n^2 is much greater than the number of processors pq. 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.

To prepare for the description of the heterogeneous version, we introduce another "logical" description of the algorithm:

- We 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 $b \times b$ block, and the unit of computation is the updating of one block, i.e. a matrix multiplication of size b.
- 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 C matrix is 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 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 11. 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 $nb \times nb$), it turns out that all rectangles are identical squares of $\frac{n}{p} \times \frac{n}{p}$ blocks.

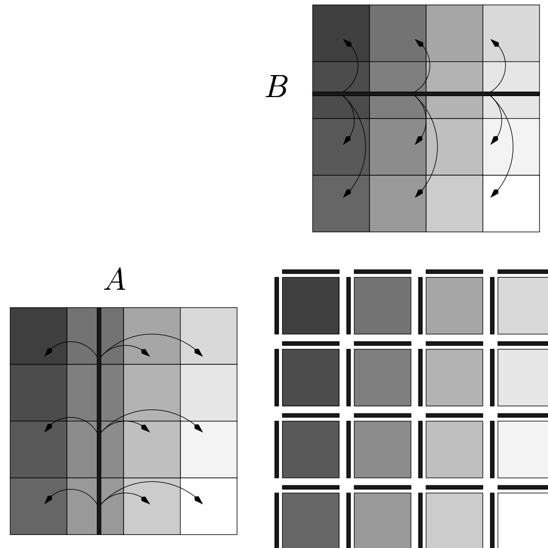

Figure 11: One step of the MM algorithm on a 4×4 homogeneous 2D-grid.

On Figure 11, we see that the total amount of communications performed by the MM algorithm is proportional to the sum of the half-perimeters of the rectangles allocated to the processors: more precisely, at each step each processor responsible for a rectangle of $h \times v$ blocks must receive (vertically) h blocks of matrix B and (horizontally) v blocks of matrix A. This explains why the allocated rectangles are identical squares for $p \times p$ homogeneous 2D-grids: because the (half)-perimeter of a rectangle of fixed area is minimized when it is a square, this choice does minimize the communication volume.

There are other homogeneous MM algorithms: for instance Cannon's algorithm [33] (whose main drawback is to require an initial permutation of matrices A and B) replaces all the horizontal and vertical broadcasts by nearest-neighbor shifts. The total communication volume at each step is the same, but the communications are different. Still, all processors independently update their rectangle of C blocks at each step.

3.1.2 Heterogeneous Platforms

How to modify the previous MM algorithms for a heterogeneous platform? The idea is to keep the same framework: at each step, one pivot column and one pivot row are communicated to all processors, and independent updates take place. However, with different-speed processors, we cannot distribute same size rectangles from the C matrix to the processors. Intuitively, we want to balance the computing load so that each processor receives 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 $C = A \times B$, we have to tile the C matrix into p non-overlapping rectangles, each rectangle being assigned to one processor. Figure 12 shows an example with 13 different-speed computing resources.

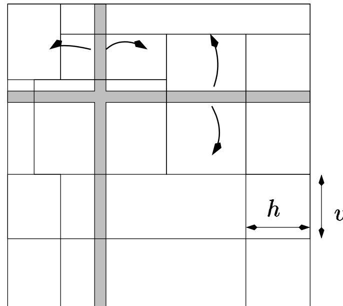

Figure 12: The MM algorithm on a heterogeneous platform.

The question is: how to compute the *area* and *shape* of these p rectangles so as to minimize the total execution time? As usual with parallel algorithms, there are two non-independent and maybe conflicting goals: (i) load-balancing computations; (ii) minimizing communication overhead. Goal (i) is related to the area of the rectangles that are allocated to the processors, while goal (ii) is related to their shapes. We discuss areas and shapes in the next section, in order to formally state (and try to solve) this difficult optimization problem.

3.2 The Heterogeneous MM Optimization Problem

Consider a matrix product $C = AB$, where A, B and C are square matrices of $n \times n$ square blocks of size b. Assume that we have p computing resources $P_1, P_2, \ldots, P_p$ of (relative) cycle-times $t_1, t_2, \ldots, t_p$. We start with load-balancing issues before dealing with communication overhead.

3.2.1 Load Balancing

To perfectly load-balance the computation, each processor should receive an amount of work in accordance to its computing power. If, say, P_2 is twice faster than P_1 ($t_1 = 2t_2$), then P_2 should be assigned twice as many elements as P_1. In other words, the *area* of its rectangle should be the double of that of P_1. Let s_i be the area of the rectangle $R_i = h_i \times v_i$ allocated to processor P_i.

Obviously, the first equation is $\sum_{i=1}^{p} s_i = n^2$, in order to obtain a true partition of the C matrix. Next, since P_i processes its rectangle within $s_i t_i$ time-steps, we have

$$s_1 t_1 = s_2 t_2 = \ldots = s_p t_p.$$

The last constraint is to write s_i as $s_i = h_i v_i$, where h_i and v_i are the number of rows and columns of R_i. These equations do not always have integer solutions, which means that a perfect load balancing of the computations is not always possible.

However, we are not really interested in an exact solution. A more concrete and interesting question is the following: given the p computing resources, how to compute the respective area of the rectangles R_i so that the workload is asymptotically optimally balanced: the larger the matrix size (expressed in blocks), the more accurate the tiling into rectangles. This question translates into the following system: given $t_1, \ldots, t_p$, search for real unknowns s_i, h_i and v_i, $1 \le i \le p$, such that:

$$\begin{cases} (1) & s_1 t_1 = s_2 t_2 = \ldots = s_p t_p \\ (2) & \sum_{i=1}^{p} s_i = 1 \\ (3) & \text{The } p \text{ rectangles of size } h_i \times v_i \text{ (where } h_i v_i = s_i) \text{ tile the unit square} \end{cases}$$

Condition (1) ensures that the area of the rectangle R_i allocated to processor P_i is inversely proportional to its cycle-time. Condition (2) is for normalization: the sum of the areas of the p rectangles is that of the unit square, a necessary condition for condition (3) to hold. Note that, as expected, conditions (1) and (2) allow to compute the s_i: we obtain

$$s_i = \frac{\frac{1}{t_i}}{\sum_{i=1}^{p} \frac{1}{t_i}}$$

We see that s_i is computed from the harmonic mean of the t_i, and it is not an integer ($0 < s_i < 1$ as soon as $p \ge 2$).

There are always solutions to the normalized problem. For instance we fulfill condition (3) by choosing to tile the unit square into p horizontal slices of height $v_i = s_i$ (and width $h_i = 1$), or into p vertical slices of width $h_i = s_i$ (and height $v_i = 1$). This degree of freedom comes from the fact that load balancing imposes constraints on the area of the rectangles R_i, but not on their shapes. Shapes come into the story when discussing communication issues, as explained below.

Finally, note that it is straightforward to retrieve a solution of the original problem (tiling a matrix of $n \times n$ blocks) from the solution of the normalized problem: we simply multiply by n all the h_i and the v_i, getting $h_i(n) = nh_i$ and $v_i(n) = nv_i$; then we round up values to integers $h'_i(n)$ and $v'_i(n)$ while preserving the constraint $\sum_{i=1}^{p} h'_i(n) = \sum_{i=1}^{p} v'_i(n) = n$ (there are many possible variations). Therefore we derive a *generic* solution to the original problem, which is valid for all values of the parameter n.

3.2.2 Communication Overhead

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, so we keep the value of $\hat{C}$ as stated.

Sequential communications 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 HNOW 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: it is proportional to the communication time needed at each step of the MM algorithm, with the underlying hypothesis that the network is homogeneous. In this section, we do not investigate further the situation where different-speed links are available between the processors (see Section 3.2.5 for a pointer).

Parallel communications Conversely, some communications can occur in parallel, or some efficient broadcast mechanisms can be used, 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 use a column-based allocation as depicted in Figure 13: vertical communications are performed in parallel in all columns, and broadcasts or at least scatters can be performed horizontally.

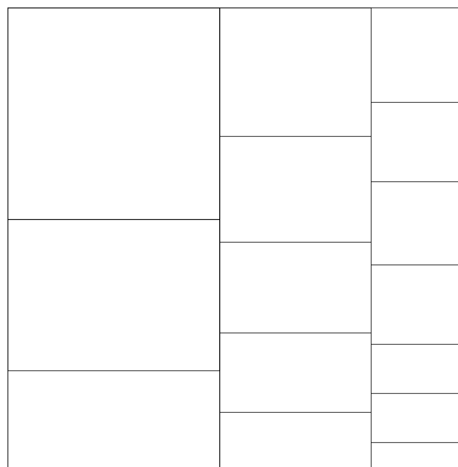

Figure 13: Tiling the unit square into columns of rectangles.

Collections of clusters Finally, in a metacomputing context, inter-cluster communications are typically one order of magnitude slower than intra-cluster communications, so we may want to adopt a two-level scheme: we assign rectangles to clusters as described in Figure 14, while inside each cluster some master-slave mechanism could be provided.

Optimization criteria It seems that minimizing the total communication volume is the most important optimization problem, because of its wide potential applicability. Also, forgetting about MM algorithms for a while, 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 while load-balancing the work amounts to solving exactly the same optimization problem.

The rest of the paper is devoted to solving the MM optimization problem using the total communication volume $\hat{C}$ as the objective function to be minimized. We formally state this optimization problem in Section 3.2.3. For the sake of completeness, we discuss some extensions of the problem in Section 3.2.5.

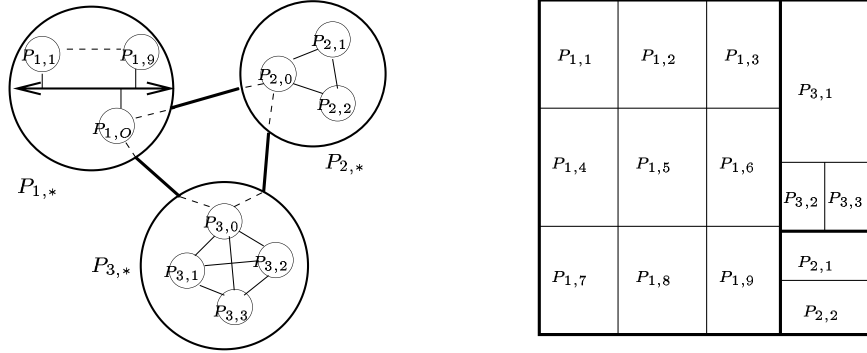

Figure 14: Two level allocation scheme for a collection of clusters; in this example, one processor within each cluster, namely $P_{i,0}$, is dedicated to inter-cluster communications.

3.2.3 The MM Optimization Problem

This section is devoted to solving the MM optimization problem using the total communication volume $\hat{C}$ as the objective function to be minimized. We have p computing resources P_i, $1 \leq i \leq p$. Each P_i is assigned a rectangle R_i of prescribed area s_i, 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 minimize the total communication volume $\hat{C}$. The abstract optimization problem is the following:

Definition 1 (MM-OPT(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.*

Given the solution (or an approximation of the solution) of MM-OPT(s), we scale and round up the values to the nearest integers so as to derive a concrete solution for matrices of given size n. As stated above, the integer solution will be asymptotically optimal. There is an obvious lower bound for MM-OPT(s):

Lemma 1 *For all solutions of MM-OPT(s), $\hat{C} \geq 2 \sum_{i=1}^{p} \sqrt{s_i}$.*

Proof The half-perimeter of each rectangle R_i will be always larger that $2\sqrt{s_i}$, the value when it is a square. Of course, tiling the unit square into p squares of area s_i is not always possible (think of the problem of tiling the unit square into two squares of same area 0.5), so this lower bound is not always tight. ■

As already mentioned, it is easy to solve MM-OPT(s) when using a square two-dimensional grid of homogeneous processors ($s_{ij} = 1/p^2$ for $1 \leq i, j \leq p$); Even if the set of homogeneous processors ($s_i = s$ for $1 \leq i \leq p$) cannot be arranged as a perfect square ($\sqrt{p} \notin \mathbb{N}$) the problem is intuitively polynomial [9]. However, with heterogeneous processors, the MM-OPT(s) optimization problem turns out to be difficult: indeed, the decision problem associated to MM-OPT is NP-complete [6]. More important than the proof, this result itself clearly demonstrates the intrinsic difficulty of static load-balancing on heterogeneous platforms while minimizing communication cost.

3.2.4 Column-Based Heuristics

In this section we introduce a polynomial heuristic to solve the MM-OPT problem. We consider the more constrained problem MM-COL(s) where we impose that the tiling is made up of processor columns, as illustrated in Figure 15. In other words, MM-COL(s) is the restriction of MM-OPT(s) to column-based partitions.

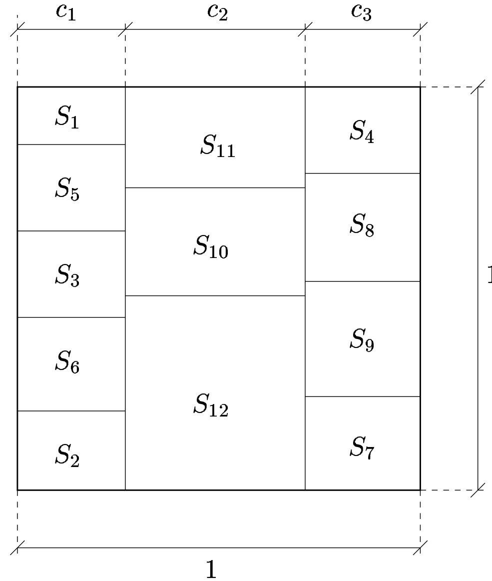

Figure 15: Column-based partitioning of the unit square: $\mathcal{C} = 3$, $k_1 = 5$, $k_2 = 3$ and $k_3 = 4$.

Framework We describe the MM-COL(s) problem more formally: we aim at tiling the unit square into $\mathcal{C}$ columns (where $\mathcal{C}$ is yet to be determined) of width $c_1, \dots, c_{\mathcal{C}}$. Each column C_i is partitioned itself into k_i rows (to be determined too) of respective area $s_{\sigma(i,1)}, \dots, s_{\sigma(i,k_i)}$. Of course, the final partitioning has $\sum_{i=1}^{\mathcal{C}} k_i = p$ rectangles, and all the areas $s_1, \dots, s_p$ are represented once and only once. The goal is to build such a partitioning, subject to the minimization of the sum of the rectangle perimeters.

Algorithm We describe our algorithm which is based upon the dynamic programming paradigm. The main points are the following:

1. Re-index the variables $s_1, \dots, s_p$ such that $s_1 \leq s_2 \leq \dots \leq s_p$.
2. Iteratively build the function $f_{\mathcal{C}}^{perimeter}$, by incrementing the value of $\mathcal{C}$ from 1 to the desired value. For $q \in \{1, \dots, p\}$, $f_{\mathcal{C}}^{perimeter}(q)$ represents the total perimeter of an optimal column-based partitioning of a rectangle of height 1 and width $(\sum_{i=1}^{q} s_i)$ into q rectangles of respective area $s_1, \dots, s_q$, using $\mathcal{C}$ columns.

To help understand the derivation, we apply the algorithm on the following example: we have $p = 8$ areas of values $(0.05, 0.05, 0.08, 0.1, 0.1, 0.12, 0.2, 0.3)$. The results of the algorithm are described in Table 2 and the resulting partitioning is depicted in Figure 16. Each column $\mathcal{C}_i$ contributes to the sum of the half-perimeters as follows: 1 for the vertical line, and $k_i c_i$ for the k_i horizontal lines of length c_i.

	q=1	q=2	q=3	q=4	q=5	q=6	q=7	q=8
$\mathcal{C}=1$	1.05 | 0	1.2 | 0	**1.54 | 0**	2.12 | 0	2.9 | 0	4 | 0	5.9 | 0	9 | 0
$\mathcal{C}=2$		2.1 | 1	2.28 | 2	2.56 | 2	2.94 | 3	**3.5 | 3**	4.38 | 4	5.76 | 5
$\mathcal{C}=3$			3.18 | 2	3.38 | 3	3.66 | 4	4 | 4	4.58 | 5	**5.5 | 6**
$\mathcal{C}=4$				4.28 | 3	4.48 | 4	4.78 | 5	5.2 | 6	5.88 | 7
$\mathcal{C}=5$					5.38 | 4	5.6 | 5	5.98 | 6	6.5 | 7
$\mathcal{C}=6$						6.5 | 5	6.8 | 6	7.28 | 7
$\mathcal{C}=7$							7.7 | 6	8.1 | 7
$\mathcal{C}=8$								9 | 7

Table 2: Table containing the values of $f_{\mathcal{C}}^{perimeter}(q)$ and a separated by |. $f_{\mathcal{C}}^{perimeter}(q) = \min_{a\in[\mathcal{C}-1,q-1]}\left(1 + \left(\sum_{a<i\leq q} s_i\right) \times (q-a) + f_{\mathcal{C}-1}^{perimeter}(a)\right)$ and a is the value minimizing the previous expression. Bold entries correspond to the optimal solution.

In the example, the optimal partitioning is obtained for 3 columns ($f_3^{perimeter}(8) = 5.5$, smallest entry of the column $q = 8$ in Table 2). The first column of width $c_3 = s_7 + s_8 = 0.5$ is composed of 2 elements. The second column of width $c_2 = s_4 + s_5 + s_6 = 0.32$ is composed of 3 elements. Then the last column of width $c_1 = s_1 + s_2 + s_3 = 0.18$ is made up with the smallest 3 elements (see Figure 16).

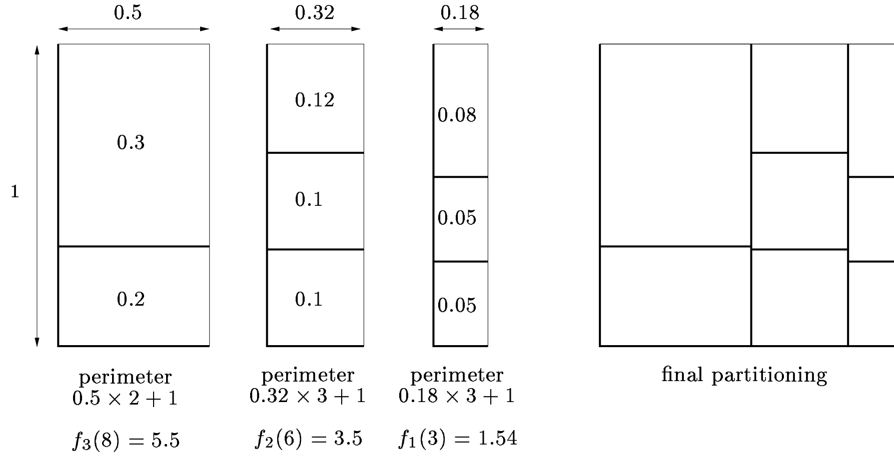

Figure 16: Optimal column-based partitioning for the example. Thicker lines correspond to the sum of the half-perimeters. We obtain $\hat{C} = 5.5$.

The worst-case complexity of the algorithm is $O(p^3)$. Note that in practice the complexity will be lower than the worst-case analysis shows, because $f_{\mathcal{C}}^{perimeter}(p)$ is a function that is first decreasing and then increasing as $\mathcal{C}$ varies. All the functions $f_{\mathcal{C}}^{perimeter}$ will not be built, the expected cost will be $p^2\mathcal{C}_{opt} \approx p^{2.5}$.

The final partitioning corresponding to the function $f_{\mathcal{C}_{opt}}^{perimeter}(p) = \min_{1\leq\mathcal{C}\leq p} f_{\mathcal{C}}^{perimeter}(p)$ is found using a simple algorithm which corresponds to tracking (backwards) the bold entries in Table 2. The unit square is partitioned into $\mathcal{C}_{opt}$ columns. The ith column contains the rectangles $s_{d_i+1}, \ldots, s_{d_i+k_i}$ with $d_i = k_1 + k_2 + \ldots + k_{i-1}$:

Algorithm 3 Construction of functions $f_{\mathcal{C}}^{perimeter}$ ($f_{\mathcal{C}}^{cut}(q)$ corresponds to the total number of blocs in the first $\mathcal{C}-1$ columns). So that, there remains $q - f_{\mathcal{C}}^{cut}(q)$ blocs in the column $\mathcal{C}$.

PARTITION($s_1, \ldots, s_p$)

1: $S = 0$
2: **For** $q = 1$ **To** p **Do**
3: $\quad S = S + s_q$
4: $\quad f_1^{perimeter}(q) = 1 + S \times q$
5: $\quad f_1^{cut}(q) = 0$
6: **For** $\mathcal{C} = 2$ **To** p **Do**
7: $\quad$ **For** $q = \mathcal{C}$ **To** p **Do**
8: $\quad\quad f_{\mathcal{C}}^{perimeter}(q) = \min_{a \in [\mathcal{C}-1, q-1]} \left(1 + \sum_{q-a<i\leq q} s_i(q-a) + f_{\mathcal{C}-1}^{perimeter}(a) \right)$
9: $\quad\quad f_{\mathcal{C}}^{cut}(q) = q - a_{opt}$ *{where a_{opt} reaches the minimum in the previous expression}*
10: **Return** (f_*^{cut})

Algorithm 4 Construction of the optimal solution from functions $f_{\mathcal{C}}^{cut}$

RE-BUILD($f_*^{cut}, \mathcal{C}_{opt}$)

1: $q = p$
2: **For** $\mathcal{C} = \mathcal{C}_{opt}$ **DownTo** 2 **Do**
3: $\quad k_{\mathcal{C}} = q - f_{\mathcal{C}}^{cut}(q)$
4: $\quad q = f_{\mathcal{C}}^{cut}(q)$
5: $k_1 = q$
6: **Return** $(k_1, \ldots, k_{\mathcal{C}_{opt}})$

Optimality This algorithm leads to the optimal column-based partitioning, as shown in [6]. Also, the following guarantee can be established (again, see [6] for a proof): let $r = \frac{\max s_i}{\min s_i}$, and let $\hat{C}$ denote the sum of the half-perimeters of the rectangles obtained with the optimal column-based partitioning. Then,

$$\frac{\hat{C}}{2\sum\sqrt{s_i}} \leq \sqrt{r}(1 + \frac{1}{\sqrt{p}})$$

Hence, if $r = 1$, i.e. all the processors have the same speed, the column-based partitioning is asymptotically optimal. On the other hand, if r is large, i.e. one processor is much faster than another, the bound is very pessimistic. For all practical purposes, we believe that the column-based algorithm is the best trade-off between efficiency and simplicity.

3.2.5 Extensions of the Model

As pointed out in Section 3.2.2, minimizing the total volume of communications $\hat{C}$ seems to be a very natural goal. However, other objective functions could be selected, because the target computing platform may influence the way communications are implemented.

Objective function for parallel communications If *all* communications could be performed in parallel, the bottleneck would come from the processor which sends/receives the largest messages. To model such a situation, a possible objective function would be to minimize $\hat{M} = \max_{i=1}^{p}(h_i + v_i)$ instead of $\hat{C} = \sum_{i=1}^{p}(h_i + v_i)$. Unfortunately, the problem remains NP-complete with this objective function [4].

Objective function for heterogeneous networks Another extension of the model could come from the modeling of the network. It is natural to consider the case of a heterogeneous network, where processors communicate through different-speed links. This problem is difficult too: it is obviously NP-complete if we use different-speed processors, and if we weight the cost of each communication with a factor proportional to the bandwidth of the communication link (because it is more complicated than MM-OPT). But interestingly, the problem remains NP-complete, even when using homogeneous processors, i.e. a heterogeneous network linking processors computing at the same speed [35]. Still, we believe that it is possible to modify the column-based heuristics presented in Section 3.2.4 to design a MM algorithm targeted to a heterogeneous network linking different-speed processors.

3.2.6 Related Work

We survey related papers from the literature in this section. They range into two categories: papers dealing with linear algebra on heterogeneous platforms on one hand, and papers covering geometric optimization problems similar to MM-OPT(s) on the other hand.

Linear Algebra on Heterogeneous Platforms Several authors have dealt with the *static* implementation of MM algorithms on heterogeneous platforms. One simple approach is given by Kalinov and Lastovetky [27]. Their idea is to achieve a perfect load-balance as follows: first they take a fixed layout of processors arranged as a collection of processor columns; then the load is evenly balanced *within* each processor column independently; next the load is balanced *between* columns; this is the "heterogeneous block cyclic distribution" of [27], which we illustrate in Figure 17, where we have $p = 8$ areas of values $(0.05, 0.05, 0.08, 0.1, 0.1, 0.12, 0.2, 0.3)$. This simple scheme is likely to give better results than a straightforward partitioning into horizontal slices.

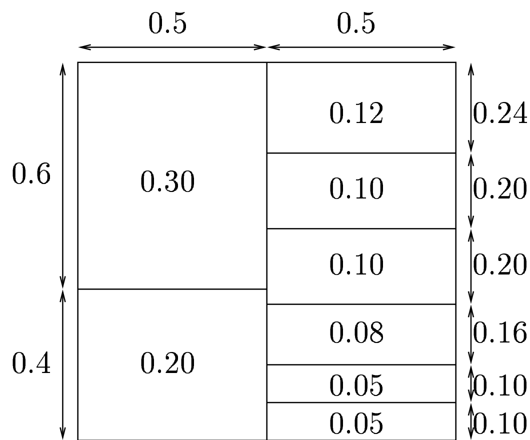

Figure 17: The distribution of Kalinov and Lastovetky. Both columns are partitioned independently. Then the final partition is made according to the total column weights: 0.5 versus 0.5. The total cost is $\hat{C} = 6$, to be compared to the cost $\hat{C} = 8$ of a partitioning into 8 horizontal slices.

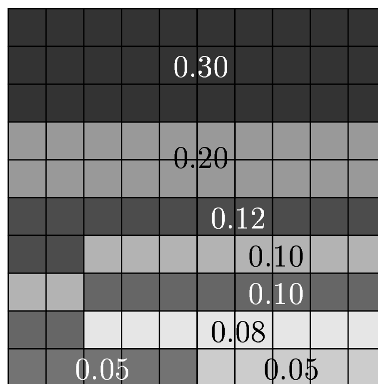

Figure 18: Illustrating contiguous block allocation. The cost is as high as $\hat{C} = 8.2$.

Another approach is proposed by Crandall and Quinn [14]. First they compare a contiguous block allocation (see Figure 18) to horizontal slicing; next they introduce a better processor arrangement: they introduce a recursive algorithm to tile the iteration space (i.e. partitioning the unit square) into p rectangles of prescribed area $s_1, s_2, \ldots, s_p$ with $\sum_{i=1}^{p} s_i = 1$, so that the total communication volume is kept small. The algorithm works recursively as follows: if at some stage their remains some rectangle R of size $h.v$ to partition into q rectangles of prescribed area $s_{i_1}, s_{i_2}, \ldots, s_{i_q}$, where $\sum_{j=1}^{q} s_{i_j} = h \times v$, then partition R along its shortest dimension into two rectangles R_1 and R_2, where R_1 contains the largest $\lceil \frac{q}{2} \rceil$ rectangles and R_2 the other ones. For instance if $h \leq v$ and $s_{i_1} \geq s_{i_2} \geq \ldots \geq s_{i_q}$, we obtain a rectangle R_1, for the $\lceil \frac{q}{2} \rceil$ largest rectangles, of area $h \times v_1$, where $v_1 = \frac{\sum_{j=1}^{\lceil \frac{q}{2} \rceil} s_{i_j}}{\sum_{j=1}^{q} s_{i_j}} \times v$; rectangle R_2 is for the remaining rectangles and of area $h \times v_2$, where $v_2 = v - v_1$ (see Figure 19 for an illustration).

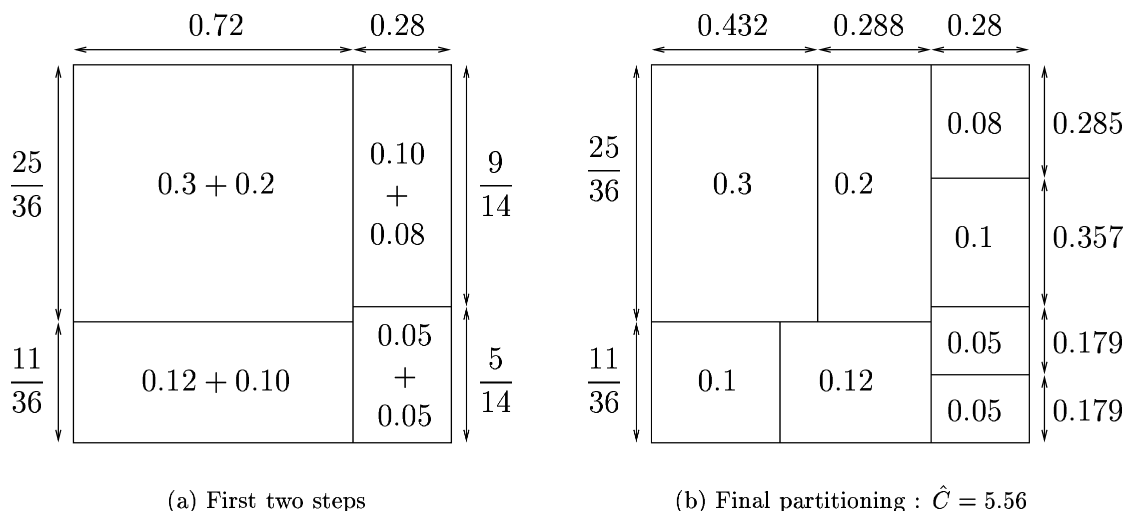

Figure 19: Illustrating the recursive partitioning of Crandall and Quinn (obtained in three steps in this example).

Kaddoura, Ranka and Wang [26] refine the previous recursive algorithm and provide several variations. They report several numerical simulations.

Problems Similar to MM-OPT(s) There are several problems related to MM-OPT(s) in the literature:

- The most similar problem is the restriction of MM-OPT(s) to an homogeneous set of processors: how to cut a square into p equal-area pieces using glass cuts. Bose et al. [9] show that the optimal solution consists of orthogonal cuts. Then let $m = \lfloor \sqrt{p} \rfloor$, and $c = m + 1$ if $r \geq m(m+1)$, $c = m$ or-else. Bose et al. conjecture that the optimal solution is a column-based partitioning made up of c columns; $k \mod c$ columns would contain $\lfloor \frac{k}{c} \rfloor + 1$ rectangles and $c - k \mod c$ columns would contain $\lfloor \frac{k}{c} \rfloor$ rectangles. However, the authors provide a guaranted bound that proves the asymptotical optimality of this partitioning.

- An other related problem is the following: how to tile the unit square into p rectangles of same area so as to minimize the maximum perimeter of these rectangles? This problem is

shown to be polynomial by Kong et al. [31, 30]: the optimal solution is one of the following two arrangements: let either $m = \lfloor \sqrt{p} \rfloor$ or $m = \lceil \sqrt{p} \rceil$, and use m columns composed of $\lfloor \frac{p}{m} \rfloor$ or $\lceil \frac{p}{m} \rceil$ rectangles. This problem is motivated by a data-allocation problem which is related to ours in the following sense: assume that we have p equal-speed processors and that we aim at minimizing the largest amount of communications made by one processor. Because the above arrangements are optimal, we have a polynomial solution to this problem.

The heterogeneous counterpart of this problem is the following: given p different-speed processors, how to allocate data so that the length of the largest communication is optimized? in terms of tiling, how to tile the unit square into p non-overlapping rectangles of prescribed area $s_1, \ldots, s_p$ whose sum is 1 so that the largest perimeter is minimized? This interesting problem is NP-complete too [4], which again shows the intrinsic difficulty of designing heterogeneous parallel algorithms!

- Another related problem is to find the minimum partition of a rectangle with interior points: given a rectangle R and a finite set P of points located inside R, find a set of line segments that partition R into rectangles such that every point in P is on the boundary of some rectangle. The goal is to minimize the total length of the introduced line segments. This problem is shown NP-complete in [36, 21, 22], where approximation algorithms are given. The link with our problem is that the objective function is the same, but the original motivation in [21, 22] was a VLSI routing problem (and the constraints are quite different).

- There are several NP-complete geometric optimization problems that are listed in [15]. One example is the minimum rectangle tiling problem [28]: given an $n \times n$ array A of non-negative numbers, and a positive integer p, find a partition of A into p non-overlapping rectangular subarrays, such that the maximum weight of any rectangle in the partition is minimized (the weight of a rectangle is the sum of its elements).

4 Master-Slave Tasking

Master-slave tasking is a simple yet widely used technique to execute independent tasks under the centralized supervision of a control processor. In the standard implementation of master-slave, the tasks are executed by identical processors (the slaves). We revisit the master-slave paradigm in the framework of heterogeneous computing resources: slave processors have different computation speeds. We present several scenarios to model the communication pattern between the master and the slaves: In all cases, such communications will take place in exclusive mode on a dedicated hardware resource (such as a serial bus). To give a single motivation, this framework applies to any Monte Carlo simulation since one must run large numbers of identical, independent simulations for different values of the random number generator seed. Monte Carlo simulations are widely used in various areas such as cellular microphysiology [41], reactor simulations [42] or studies on conformations of proteins [40].

In Section 4.1 we state two different variants of the master-slave problem: (i) with communications only before the dispatching of the tasks, and (ii) with communications both before and after the processing of the tasks. We give in Section 4.3 a polynomial time algorithm that solves the first problem. The second problem seems intrinsically more difficult and we prove in Section 4.4.2 that it is at least as difficult as a problem known to be open. We also prove a guaranteed approximation algorithm for the real problem in Section 4.4. and we briefly survey related work in Section 4.5.

4.1 Problem statement

The target architectural platform is represented in Figure 20. The master M and the p slaves $P_1, P_2, \ldots, P_p$ communicate through a shared medium, typically a bus, that can be accessed only in exclusive mode. At a given time-step, at most one processor can communicate with the master, either to receive data from the master or to send results back to the master.

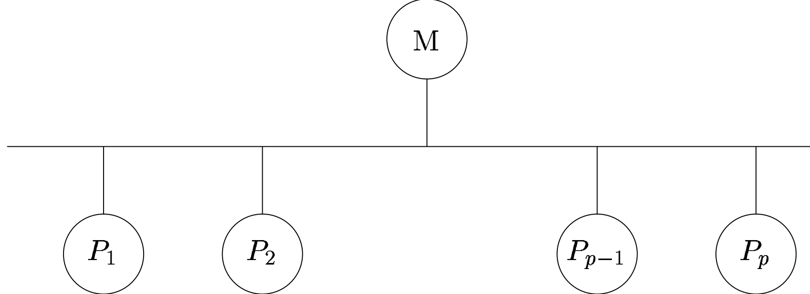

Figure 20: The target master-slave architecture.

We assume that the master holds a pool of independent tasks to be processed by the p slaves. All tasks are of same-size, i.e. they represent the same amount of processing. Tasks are considered to be atomic (execution cannot be preempted once initiated). Processors are heterogeneous; more precisely, slave P_i requires t_i units of time to process a single task. We say that t_i is the "cycle-time" of processor P_i. Each P_i will execute c_i tasks (where c_i is to be determined) from the pool. Regardless of the hypotheses concerning communication costs, there are two (related) optimization problems:

Definition 2 (MinTime(C)) *Given a total number of tasks C, determine the best allocation of tasks to slaves, i.e. the allocation $\mathcal{C} = \{c_1, c_2, \ldots, c_p\}$ s.t. $\sum_{i=1}^{p} c_i = C$ which minimizes the total execution time.*

Definition 3 (MaxTasks(T)) *Given a time bound T, determine the best allocation of tasks to slaves, i.e. the allocation $\mathcal{C} = \{c_1, c_2, \ldots, c_p\}$ s.t. all processors complete their execution within T units of time and $\sum_{i=1}^{p} c_i = C$ is maximized.*

In the paper, we concentrate on solving the second problem MaxTasks(T). Given the solution to this problem, we find a solution to MinTime(C) by using binary search on T, calling MaxTasks(T) several times, and returning the smallest value of T for which the answer is at least C.

We now state some specific hypotheses for the communication costs. For each modeling of these communication costs, we analytically formulate the MaxTasks(T) problem.

4.1.1 Without any communication cost

Assume first that there is no communication cost at all. It is not difficult to solve both previous problems using a greedy algorithm. The solution of problem MaxTasks(T) is straightforward: we let $c_i = \left\lfloor \frac{T}{t_i} \right\rfloor$ for all i, $1 \leq i \leq p$, which obviously defines the optimal solution.

4.1.2 With an initial scattering of data

The formulation of this problem is taken from Andonie et al. [2], who study the implementation of distributed backpropagation neural networks on heterogeneous networks of workstations, using the

PVM library [19]. The training of the neural network is divided into computational phases. At each phase, the training pattern is distributed among the slaves, which are different-speed processors. Before executing any task, each slave must receive some data file from the master processor. Because the communication medium is exclusive, it it not relevant to distinguish whether the data file is the same for all slaves (then the master executes a broadcast operation) or whether it is different (then the master executes a scatter operation): we only assume that each slave must receive the same amount of data, and that each reception costs t_{com} units of time. In the model of Andonie et al. [2], there is no communication cost paid to send the results back to the master. In general, when the slaves compute "yes/no" results, the cost of returning the results may well be neglected in front of the cost of the initial scatter and/or of the computations. Note that we deal with another model, including communication costs both before and after the tasks, in Section 4.2.

Due to the constraint on the communication medium, the p messages will be sent one after the other. Obviously, it cannot hurt to send the messages as soon as possible, i.e. at time steps 0, t_{com}, $2t_{\text{com}}$, ... , $(p-1)t_{\text{com}}$. The problem is then to determine the ordering of the p messages, i.e. the permutation σ of $\{1, 2, \ldots, p\}$ such that slave P_i receives the message at time $\sigma(i)t_{\text{com}}$. We are ready to state the optimization problem analytically:

Definition 4 (MaxTasks1(T)) *Given a time bound T, determine the best allocation of tasks to slaves, i.e. a permutation σ and an allocation $\mathcal{C} = \{c_1, c_2, \ldots, c_p\}$ s.t. all processors complete their execution within T units of time and the total number of tasks is maximized:*

$$\max \left(\sum_{i=1}^{p} c_i \;\middle|\; \begin{array}{c} \sigma \textit{ permutation and} \\ \forall i \in [1,p] : \sigma(i)t_{com} + c_i t_i \leq T \end{array} \right)$$

4.2 With initial and final communications

As pointed out above, it is natural to assume that after the processing of their tasks, slave processors will send some data back to the master. Because this message may well have a different size than the message initially sent by the master, we model this situation by using two communication costs, t^1_{com} for the messages sent by the master to the slaves, and t^2_{com} for the messages sent by the slaves to the master.

As above, we look for a permutation σ_1 which determines the ordering of the initial messages from the host: the host sends data to slave P_i at time $\sigma_1(i)t^1_{\text{com}}$. But we also look for a second permutation σ_2 which determines the ordering of the final messages sent back to the host: given a time bound T, slave P_i sends data back to the host at time $T - \sigma_2(i)t^2_{\text{com}}$. This formulation is without any loss of generality: some slave processor P_i might send its message earlier than this bound, but we can always shift the communication pattern as stated, i.e. delay some messages, as illustrated Figure 21. We are ready to state the optimization problem analytically:

Definition 5 (MaxTasks2(T)) *Given a time bound T, determine the best allocation of tasks to slaves, i.e. two permutations σ_1 and σ_2, and an allocation $\mathcal{C} = \{c_1, c_2, \ldots, c_p\}$ s.t. all processors complete their execution within T units of time and the total number of tasks is maximized:*

$$\max \left(\sum_{i=1}^{p} c_i \;\middle|\; \begin{array}{c} \sigma_1, \sigma_2 \textit{ permutations and} \\ \forall i \in [1,p] : \sigma_1(i)t^1_{com} + c_i t_i + \sigma_2(i)t^2_{com} \leq T \end{array} \right)$$

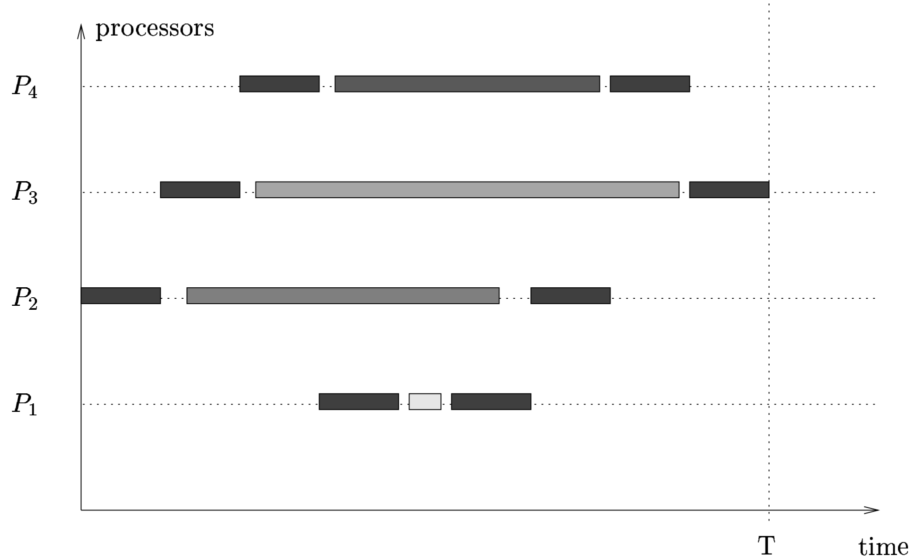

Figure 21: Delaying messages sent back to the host.

4.3 Solution with an initial scattering of data

4.3.1 Restricted search

To (partially) solve the MaxTasks1(T) problem of Section 4.1.2, Andonie et al. [2] restrict the search to allocations where the fastest processors start computing first. They use a dynamic programming algorithm to solve the optimization problem MinTime(C). With our setting for problem MaxTasks1(T), this amount to sort the cycle-times as $t_1 \leq t_2 \leq \ldots \leq t_p$ and to let $\sigma(i) = i$ for $1 \leq i \leq p$. The intuition is that fastest processors execute tasks more rapidly than the others, hence should work longer.

However, the intuition is misleading in some cases. Assume for instance two slave processors ($p = 2$) with $t_1 = 5$ and $t_2 = 9$ and let $t_{\text{com}} = 1$. For the time bound $T = 28$, it is better to start the slow processor P_2 first: P_2 can then execute three tasks: $t_{\text{com}} + 3t_2 = 28 \leq T$; the fast processor, although started at time-step $2t_{\text{com}} = 2$, can execute five tasks: $2t_{\text{com}} + 5t_1 = 27 \leq T$. If we start the fast processor first, it cannot execute more than 5 tasks, while the second processor can execute only 2.

4.3.2 Matching techniques

The optimal solution to the MaxTasks1(T) problem can be found using a weighted-matching algorithm. The idea is to draw a complete bipartite graph with $2p$ vertices, as shown in Figure 22. Vertices on the left represent processors, while vertices on the right represent possible values for the permutation σ. The edge from vertex P_i to vertex S_j is weighted with the maximum number of tasks that P_i can execute if $\sigma(i) = j$, namely $\left\lfloor \frac{T - jt_{\text{com}}}{t_i} \right\rfloor$. Extracting a matching from the graph enables to assign a different value of σ for each processor, thereby guaranteeing that σ is indeed a permutation. In fact, there is a one-to-one correspondence between matchings and permutations. Because the total weight of a given matching is the total number of tasks that can be executed for the corresponding choice of the permutation, our problem reduces to finding the maximum weighted matching in the bipartite graph. Efficient (polynomial) algorithms exist to solve this problem, see [20, 43]. More precisely, the complexity of finding the maximum weighted matching

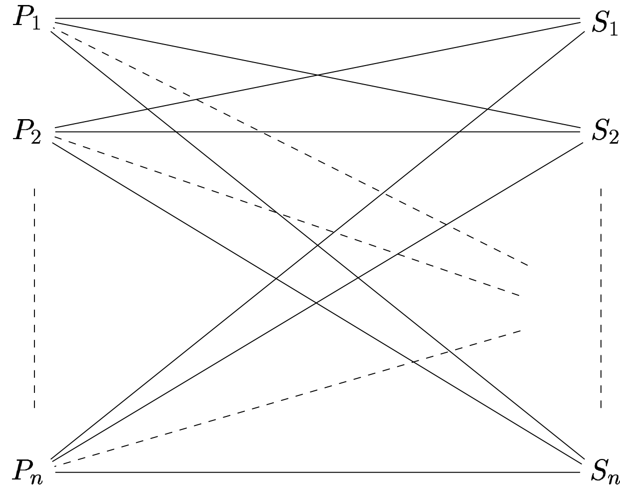

Figure 22: Bipartite graph for MaxTasks1(T).

in a bipartite graph with $2p$ vertices is of order $O(p^3)$.

We work out the following example: assume $p = 3$ processors of cycle-times $t_1 = 4$, $t_2 = 5$ and $t_3 = 9$. Let $t_{\text{com}} = 1$, and consider the time bound $T = 118$. The weighted bipartite graph is shown Figure 23. The maximum weighted matching is unique, it corresponds to the permutation $\sigma(1) = 2$, $\sigma(2) = 3$ and $\sigma(3) = 1$. The total number of tasks is $29 + 23 + 13 = 65$.

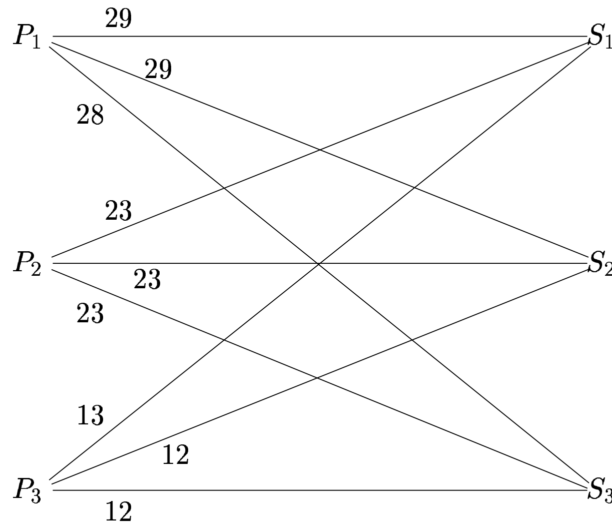

Figure 23: Bipartite graph with $p = 3$, $t_1 = 4$, $t_2 = 5$, $t_3 = 9$, $t_{\text{com}} = 1$, and $T = 118$.

To conclude this section, we formally state our result:

Proposition 1 *The optimal solution to the* MaxTasks1(T) *problem with initial messages and p processors can be found in time $O(p^3)$ using the above weighted-matching algorithm.*

4.4 Solution with initial and final communications

The solution to the MaxTasks2(T) problem with initial and final messages turns out to be surprisingly difficult. In fact, we do not know of any polynomial algorithm for the general case. We can give

an efficient guaranteed approximation using matching techniques, as explained in Section 4.4.1. In Section 4.4.2, we give some remarks about the complexity of MaxTasks2(T).

4.4.1 Matching techniques

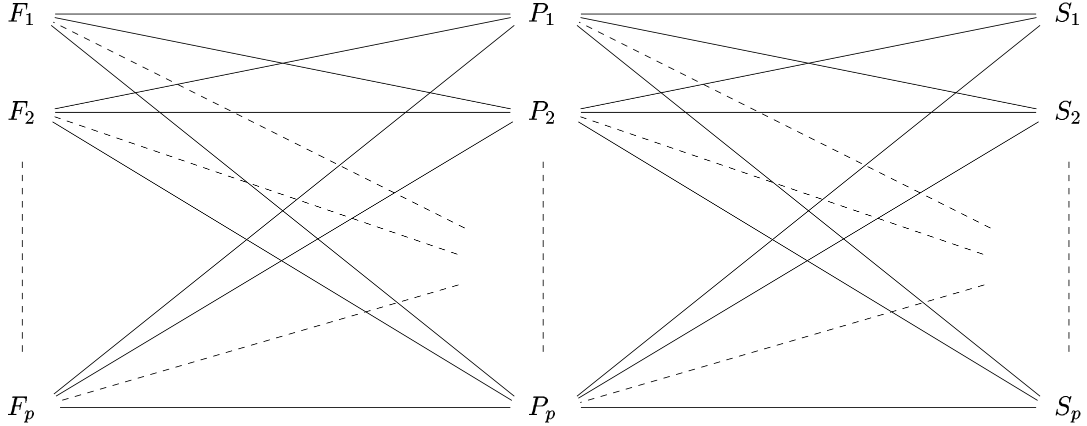

Figure 24: Bipartite graph with initial and final communications.

To take both permutations σ_1 and σ_2 into account, we build a bipartite graph $G = (V, E)$ with $3p$ vertices (i.e. $|V| = 3p$), as shown Figure 24. The p leftmost vertices F_i correspond to the first permutation σ_1, the p center vertices P_i correspond to processors, and the p rightmost vertices S_i correspond to the second permutation σ_2. Rather than a matching, we extract a 2-factor from the graph [20, 43]: more precisely, we select a subset E' of $2p$ edges so that in the graph $G = (V, E')$ each vertex F_i or S_i is exactly of degree 1, and each vertex P_i is exactly of degree 2. The complexity of extracting 2-factor from the graph with $3p$ vertices is of order $O(p^3)$ again, since we can solve independently the maximum weighted matching in both bipartite graph with $2p$ vertices (on the left-hand size and on the right-hand size in Figure 24) in time of order $O(p^3)$.

The problem is that edge weights cannot be determined fully accurately; the inequality $\sigma_1(i)t^1_{\text{com}} + c_i t_i + \sigma_2(i)t^2_{\text{com}} \leq T$ translates into

$$c_i \leq \left\lfloor \frac{T - \sigma_1(i)t^1_{\text{com}} - \sigma_2(i)t^2_{\text{com}}}{t_i} \right\rfloor,$$

and we need to know *both* $\sigma_1(i)$ and $\sigma_2(i)$ to compute c_i. Instead, we use the approximation

$$\left\lfloor \frac{T/2 - \sigma_1(i)t^1_{\text{com}}}{t_i} \right\rfloor + \left\lfloor \frac{T/2 - \sigma_2(i)t^2_{\text{com}}}{t_i} \right\rfloor.$$

This approximation enables us to weight the edges as follows: the edge between F_j and P_i is weighted as $\left\lfloor \frac{T/2 - j t^1_{\text{com}}}{t_i} \right\rfloor$ while the edge between P_i and S_k is weighted as $\left\lfloor \frac{T/2 - k t^2_{\text{com}}}{t_i} \right\rfloor$.

Theorem 1 *The previous approximation leads to tasks allocations that differs at most by p tasks from the optimal solution.*

4.4.2 Some remarks about the complexity of MaxTasks2(T)

We have not found any polynomial algorithm for the general case, and we have not been able to prove the NP-completeness of MaxTasks2(T). Nevertheless, we can formulate a few remarks about

the intrinsic difficulty of MaxTasks2(T). First, an exhaustive search of all possible permutations would have a complexity of order $O((p!)^2)$, which is impossible in practice as soon as $p \geq 9$ (about 44 minutes on a Pentium III 550 MHz for $p = 9$ and more than one day for $p = 10$). Moreover, the problem seems to be difficult even for very simple instances of MaxTasks2(T), as shown below. Indeed, let us consider the following open problem (polynomial vs. NP-complete) problem in combinatorial optimization (see [24]):

Permutation Sums:
Instance: Let $a_1 \leq a_2 \leq \ldots \leq a_p$ be p positive integers satisfying $\sum_{i=1}^{p} a_i = p(p+1)$.
Question: Are there two permutations σ_1 and σ_2 of $\{1, 2, \ldots, n\}$ such that $\forall i \in [1, p] : \sigma_1(i) + \sigma_2(i) = a_i$.

Let us build the following instance of MaxTasks2(T):
Instance: Let $T = 3 \max_i a_i$, $t_{\text{com}} = 1$ and let $t_i = T - a_i$, $\forall i \in [1, p]$ denote the cycle time of processor P_i.
Question: Is it possible to perform p tasks within T units of time?

Let us consider the above instance of MaxTasks2(T). Since $t_i = T - a_i$ and $T = 3 \max_i a_i$, $2t_i > T$, it is impossible to execute more than one task with a given processor within T units of time. Thus, our instance of MaxTasks2(T) is very simple (with respect to the general formulation) since $\forall i, \ \ c_i \leq 1$. Indeed, the question can be stated as follows:

Is it possible to find two permutations σ_1 and σ_2 such that $\forall i \in [1, p] : \sigma_1(i) + \sigma_2(i) + t_i \leq T$.

Nevertheless, this instance of MaxTasks2(T) is as difficult as Permutation Sums. More precisely, if Permutation Sums is proved to be NP-complete then MaxTasks2(T) is also NP-complete and if MaxTasks2(T) can be solved in polynomial time, then it proves that Permutation Sums can also be solved in polynomial time. Indeed, let us suppose that

$$\exists \sigma_1, \sigma_2 \ \text{ such that } \ \forall 1 \leq i \leq p, \ \ \sigma_1(i) + \sigma_2(i) + t_i \leq T.$$

Then, $\forall 1 \leq i \leq p, \ \ \sigma_1(i) + \sigma_2(i) \leq a_i$.
Moreover, let us suppose that

$$\exists i, \ \ \sigma_1(i) + \sigma_2(i) < a_i,$$

then

$$p(p+1) = \sum_{i=1}^{p} (\sigma_1(i) + \sigma_2(i)) < \sum_{i=1}^{p} a_i = p(p+1),$$

which is absurd. Thus, finally

$$\forall i \in [1, p] : \sigma_1(i) + \sigma_2(i) = a_i.$$

Thus, we can expect that the general instance of MaxTasks2(T) is intrinsically difficult.

4.5 Related Work

Several theoretical papers deal with complexity results for parallel machine problems with a server, establishing complexity (NP-completeness) results [23, 32, 11] and providing guaranteed approximations [34]. Before processing, each job must be loaded on a machine, which takes a certain setup time. All these setups have to be done by a single server which can handle at most one job at a time. Our first problem (with initial messages only) is a very special instance of this class of server problems.

Our second problem (with initial and final messages) is a special instance of the job-shop scheduling problem (see problem SS18 in [18]) where each job consists of only three tasks, the first and last of which having to be executed by the two machines dedicated to communications. Because this instance is very specific, we do not know its complexity (polynomial versus NP-complete).

5 Conclusion

In this paper, we have discussed how to extend static strategies for the distribution of regular computations and data in the framework on heterogeneous computations. We have showed that deriving efficient distribution schemes is a rather simple task for linear networks but turns out to be surprisingly difficult for multi-dimensional problems. We have discussed several variants of the master-slave paradigms, which also demonstrate the complexity of load balancing with different-speed processor, even in the simple framework presented here.

In the context of heterogeneous computations, static strategies have several drawbacks. Indeed, the network may be non-dedicated in some situations, and possible variations in the processor speeds have to be taken into account. Nevertheless, the solution may well not involve fully dynamic strategies, but rather to consider mixed strategies. We believe that the problems due to variations in processor speeds can be addressed by remapping data and computations from time to time between well identified static phases.

In conclusion, we believe that the design of efficient data allocation techniques for regular problems, similar to the ScaLAPACK library on homogeneous platforms, relies both on the design of efficient static distribution schemes (to cope with heterogeneity) and on the design of efficient remapping strategies (to cope with speed variations during the computations). However, the difficulty of algorithmic issues is not to be underestimated. Data decomposition, scheduling heuristics, and load balancing were known to be difficult problems in the context of classical parallel architectures. They become extremely difficult in the context of heterogeneous clusters, not to mention metacomputing platforms.

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] R. Andonie, A.T. Chronopoulos, D. Grosu, and H. Galmeanu. Distributed backpropagation neural networks on a PVM heterogeneous system. In *Parallel and Distributed Computing and Systems Conference (PDCS'98)*, pages 555–560. IASTED Press, 1998.

[3] Rumen Andonov and Sanjay Rajopadhye. Optimal orthogonal tiling of two-dimensional iterations. *Journal of Parallel and Distributed Computing*, 45(2):159–165, 1997.

[4] 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/`.

[5] Olivier Beaumont, Vincent Boudet, Antoine Petitet, Fabrice Rastello, and Yves Robert. A proposal for a heterogeneous cluster scalapack (dense linear solvers). *IEEE Trans. Computers*, 2001, to appear.

[6] Olivier Beaumont, Vincent Boudet, Fabrice Rastello, and Yves Robert. Matrix multiplication on heterogeneous platforms. *IEEE Trans. Parallel Distributed Systems*, 2001, to appear.

[7] F. Bitz and H.T. Kung. Path planning on the warp computer: using a linear systolic array in dynamic progamming. *Inter. J. Computer Math*, 25:173–188, 1988.

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

[9] Prosenjit Bose, Jurek Czyzowicz, and Dominic Lessard. Cutting rectangles in equal area pieces. In Mike Soss, editor, *Proceedings of the 10th Canadian Conference on Computational Geometry*, pages 94–95, Montréal, Québec, Canada, 1998. School of Computer Science, McGill University.

[10] Pierre Boulet, Jack Dongarra, Yves Robert, and Frédéric Vivien. Static tiling for heterogeneous computing platforms. *Parallel Computing*, 25:547–568, 1999.

[11] P. Brucker, C. Dhaenens-Flipo, S. Knust, S.A. Kravchenko, and F. Werner. Complexity results for parallel machine problems with a single server. Technical Report Reihe P, No. 219, Fachbereich Mathematik Informatik, Universität Osnabrück, 2000.

[12] 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).

[13] 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.

[14] 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.

[15] P. Crescenzi and V. Kann. A compendium of NP optimization problems. World Wide Web document, URL: `http://www.nada.kth.se/~viggo/wwwcompendium/wwwcompendium.html`.

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

[19] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine: A Users'Guide and Tutorial for Networked Parallel Computing*. The MIT Press, 1996.

[20] M. Gondran and M. Minoux. *Graphs and Algorithms*. John Wiley & Sons, 1984.

[21] T.F. Gonzalez and S. Zheng. Improved bounds for rectangular and guilhotine partitions. *J. Symbolic Computation*, 7:591–610, 1989.

[22] T.F. Gonzalez and S. Zheng. Approximation algorithm for partitioning a rectangle with interior points. *Algorithmica*, 5:11–42, 1990.

[23] N. Hall, C.N. Potts, and C. Sriskandarajah. Parallel machine scheduling with a common server. *Discrete Applied Mathematics*, 102:223–243, 2000.

[24] Steve Hedetniemi. Open Problems in Combinatorial Optimization. World Wide Web document, URL: `http://www.cs.clemson.edu/~hedet/algorithms.html`.

[25] K. Högstedt, L. Carter, and J. Ferrante. Determining the idle time of a tiling. In *Principles of Programming Languages*, pages 160–173. ACM Press, 1997. Extended version available as Technical Report UCSD-CS96-489, and on the WEB at http://www.cse.ucsd.edu/∼carter.

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

[27] 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.

[28] S. Khanna, S. Muthukrishnan, and M. Paterson. On approximating rectangle tiling and packing. In *Proc. 9th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 384–393. ACM Press, 1998.

[29] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[30] T.Y. Kong, D.M. Mount, and W. Roscoe. The decomposition of a rectangle into rectangles of minimal perimeter. *SIAM J. Computing*, 17(6):1215–1231, 1988.

[31] T.Y. Kong, D.M. Mount, and M. Wermann. The decomposition of a square into rectangles of minimal perimeter. *Discrete Applied Mathematics*, 16:239–243, 1987.

[32] S.A. Kravchenko and F. Werner. Parallel machine scheduling problems with a single server. *Mathematical Computational Modelling*, 26:1–11, 1997.

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

[34] H. Lee and M. Guignard. A hybrid bounding procedure for the workload allocation problem on parallel unrelated machines with setups. *Journal of the Operational Research Society*, 47:1247–1261, 1996.

[35] A. Legrand. Algorithmique parallèle: environnements hétérogènes et non-dédiés. Master's thesis, Ecole Normale Supérieure de Lyon, June 2000. Available at www.ens-lyon.fr/∼yrobert.

[36] A. Lingas, R.Y. Pinter, R.L. Rivest, and A. Shamir. Minimum edge length partitioning of rectilinear polygons. In *Proc. 20th Ann. Allerton Conference on Communication, Control and Computing*, 1982.

[37] H. Ohta, Y. Saito, M. Kainaga, and H. Ono. Optimal tile size adjustment in compiling general DOACROSS loop nests. In *1995 International Conference on Supercomputing*, pages 270–279. ACM Press, 1995.

[38] A. Rosenfeld and A.C. Kak. *Digital Picture Processing*. Academic Press, New York, 1982.

[39] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI the complete reference*. The MIT Press, 1996.

[40] Kizhake Soman, Robert Fraczkiewicz, Christian Mumenthaler, Berthold von Freyberg, and Thomas Schaumannand Werner Braun. FANTOM - (Fast Newton - Raphson Torsion Angle Minimizer). World Wide Web document, URL: `http://www.scsb.utmb.edu/fantom/fm_home.html`. a program for "the calculation of conformations of linear and cyclic polypeptides and proteins with low conformational energies including distance and dihedral angle constraints from nuclear magnetic resonance experiments or for modeling purposes.".

[41] J.R. Stiles, T.M. Bartol, M.M. Salpeter, and M.M. Salpeter. Monte Carlo simulation of neuromuscular transmitter release using MCell, a general simulator of cellular physiological processes. *Computational Neuroscience*, pages 279–284, 1998.

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

[43] D.B. West. *Introduction to Graph Theory.* Prentice Hall, 1996.

[44] Michael Wolfe. *High Performance Compilers For Parallel Computing.* Addison-Wesley, 1996.