Algorithmic Issues on Heterogeneous Computing Platforms

Pierre Boulet[1], Jack Dongarra[2,3], Fabrice Rastello[4], Yves Robert[4] and Frédéric Vivien[5]

[1] LIFL, , Universit de Lille, 59655 Villeneuve d'Ascq Cedex, France
[2] Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301, USA
[3] Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
[4] LIP, UMR CNRS–ENS Lyon–INRIA 8512, ENS Lyon, 69364 Lyon Cedex 07, France
[5] ICPS, Universit de Strasbourg, Pôle Api, 67400 Illkirch, France
e-mail: `Firstname.Lastname@ens-lyon.fr, dongarra@cs.utk.edu`

October 1998

Abstract

This paper discusses some algorithmic issues when computing with a heterogeneous network of workstations (the typical poor man's parallel computer). Dealing with processors of different speeds requires to use more involved strategies than block-cyclic data distributions. Dynamic data distribution is a first possibility but may prove impractical and not scalable due to communication and control overhead. Static data distributions tuned to balance execution times constitute another possibility but may prove inefficient due to variations in the processor speeds (e.g. because of different workloads during the computation). We introduce a static distribution strategy that can be refined on the fly, and we show that it is well-suited to parallelizing scientific computing applications such as finite-difference stencils or LU decomposition.

Key words: heterogeneous networks, distributed-memory, different-speed processors, tiling, communication-computation overlap, mapping, LU decomposition.

1 Introduction

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

The major limitation to programming heterogeneous platforms arises from the additional difficulty of balancing the load when using processors running at different speed. Distributing the computations (together with the associated data) can be performed either dynamically or statically, or a mixture of both. At first sight, we may think that dynamic strategies like a greedy algorithm are likely to perform better, because the machine loads will be self-regulated, hence self-balanced, if processors pick up new tasks just as they terminate their current computation. However, data dependences may lead to slow the whole process down to the pace of the slowest processor, as our examples taken from standard linear algebra kernels will demonstrate. In fact, the more constrained the problem (because of the data dependences), the more efficient a static distribution of the work. A simple approach to statically distributing independent chunks of computations is to let the number of chunks allocated to each processor be inversely proportional to its speed. Coupled with a refined data allocation (block-cyclic distributions are not enough), this approach leads to satisfactory results, as proven both theoretically and experimentally (through MPI experiments) in this paper.

The rest of the paper is organized as follows. In Section 2 we give pointers to the abundant literature on dynamic allocation strategies, and we propose a static allocation strategy, based upon a simple dynamic-programming algorithm, to evenly distribute independent chunks of computations to different-speed processors. Section 3 is devoted to a first example where greedy algorithms fail because of data dependences, while static allocation schemes provide good results. In Section 4 we move to a second, more involved, example, i.e. partitioning a finite-difference stencil computation. In Section 5 we discuss how to implement LU decomposition on a heterogeneous network of workstations. We give some final remarks and conclusions in Section 6.

2 A strategy for static load balancing

The problem of making effective use of heterogeneous networks of workstations has received a considerable attention in the recent years. We only give a few pointers to the existing literature in this section.

When targeting non-dedicated workstations with unpredictable workload and possible failures, dynamic allocation strategies seem unavoidable. Dynamic strategies range from simple master-slave paradigms (where each processor picks up a new computational chunk as soon as it returns from its current work) to more sophisticated task allocation strategies, most of which *"use the past predict the future"*, i.e. use the currently observed speed of computation of each machine to decide for the next distribution of work. See the survey paper of Berman [3] and the more specialized references [1, 8, 9, 12, 13, 15] for further details. Performance prediction models for heterogeneous networks are dealt with in [10, 18] (among others). Limitations of dynamic strategies come from two main factors:

- Dependences may keep fast processors idle, as illustrated in Sections 3 and 4.
- A dynamic allocation of tasks may induce a large overhead due to the redistribution of data, as illustrated in Section 5.

Static strategies are less general than dynamic ones but constitute an efficient alternative for heavily constrained problems. The basis for such strategies is to distribute computations to processors so that the workload is evenly balanced, and so that no processor is kept idle by data dependences. To illustrate the static approach, consider the following simple problem: given M independent chunks of computations, each of equal size (meaning each requiring the same amount of work), how can we assign these chunks to p physical processors P_1, P_2, ..., P_p of respective execution times t_1, t_2, ..., 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. A difficulty arises when stating the problem: how accurate are the estimated processor speeds? won't they change during program execution? We come back on estimating processor speeds later, and we assume for a while that each processor P_i will indeed execute 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: P_i receives c_i chunks so that

$$c_i \times t_i = Constant = K.$$

We get $c_1 t_1 = c_2 t_2 = \ldots = c_p t_p = K$ where $c_1 + c_2 + \ldots + c_p = M$. Hence:

$$K \times \sum_{i=1}^{p} \frac{1}{t_i} = M \quad \Longrightarrow \quad c_i = \frac{\frac{1}{t_i}}{\sum_{i=1}^{p} \frac{1}{t_i}} \times M.$$

This strategy leads to a perfect load balance when M is a multiple of $C = \mathrm{lcm}(t_1, t_2, \ldots, t_p) \sum_{i=1}^{p} \frac{1}{t_i}$. Consider first a toy example with 3 processors with $t_1 = 3$, $t_2 = 5$ and $t_3 = 8$. We have $\mathrm{lcm}(t_1, t_2, \ldots, t_p) = 120$, $\sum_{i=1}^{p} \frac{1}{t_i} = \frac{79}{120}$, hence $C = 79$. We obtain the static allocation of Table 1.

Execution-time	$t_1 = 3$	$t_2 = 5$	$t_3 = 8$
Chunks $M = 79$	$c_1 = 40$	$c_2 = 24$	$c_3 = 15$

Table 1: Static allocation for 3 processors with $t_1 = 3$, $t_2 = 5$ and $t_3 = 8$

Consider now a more concrete example, taken from a heterogeneous network at the University of Tennessee: see Table 2. We get $L = \mathrm{lcm}(t_1, t_2, \ldots, t_p) = 34,560,240$ and $C = L \sum_{i=1}^{7} \frac{1}{t_i} = 8,469,789$. Needless

Name	nala	bluegrass	dancer	donner	vixen	rudolph	zazu	simba
Description	Ultra 2	SS 20	SS 5	SS 5	SS 5	SS 10	SS1 4/60	SS1 4/60
Execution time t_i	11	26	33	33	38	40	528	530
Chunks $M = 8,469,789$	3,141,840	1,329,240	1,047,280	1,047,280	909480	864,006	65,455	65,208

Table 2: Static allocation for 8 Sun workstations.

to say, such a large amount of chunks is not feasible in practice, so we should derive a method to allocate smaller pieces of work.

Let B, a constant fixed by the user, denote the number of chunks to be allocated to the p processors. The execution time, for an allocation $\mathcal{C} = (c_1, c_2, \ldots, c_p)$ such that $\sum_{i=1}^{p} c_i = B$, is $\max_{1 \le i \le p} c_i t_i$ (the maximum is taken over all processor execution times), so that the average cost to execute one chunk is

$$cost(\mathcal{C}) = \frac{\max_{1 \le i \le p} c_i t_i}{B}$$

and we aim at finding the allocation $\mathcal{C}$ of minimal cost. A naive and sub-optimal solution would be to approximate the rational expressions $c_i = \frac{\frac{1}{t_i}}{\sum_{i=1}^{p} \frac{1}{t_i}} \times B$ by rounding them, while enforcing the condition $\sum_{i=1}^{p} c_i = B$. Using the former toy example with 3 processors, $t_1 = 3$, $t_2 = 5$, and $t_3 = 8$, take $B = 9$: $c_1 = 4.56$, $c_2 = 2.73$ and $c_3 = 1.71$, so a possible approximation is $\mathcal{C} = (4, 3, 2)$, whose cost is 1.78. Note that the optimal solution is $\mathcal{C} = (5, 3, 1)$, whose cost is 1.67. A costly but optimal solution is obtained by solving the integer linear programming problem

$$\min \gamma \text{ s.t. } \begin{cases} \sum_{i=1}^{p} c_i = B \\ c_i t_i \le \gamma, 1 \le i \le p \end{cases}$$

However, there is a nice dynamic programming approach that leads to the best allocation. It is best explained using the former toy example again:

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

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

In Table 3, 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. For instance at step 4, i.e. to allocate a 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 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 $(c_1, c_2, c_3) = (2, 1, 1)$.

Proposition 1 *(see [4]) The dynamic programming algorithm returns the optimal allocation for any number of chunks up to B.*

Proof As already said, we build a function that, given a best allocation with a total number of chunks equal to $n-1$, computes a best allocation with a total number of chunks equal to n. Once we have this function, we start with an initial value $n=0$, and we compute a best allocation for each increasing value of n up to $n=B$. First we characterize the best allocations for a given chunk size s:

Lemma 1 *Let $\mathcal{C}=(c_1,\dots,c_p)$ be an allocation, and let $s=\sum_{1\leq i\leq p} c_i$ be the total number of chunks. Let $m=\max_{1\leq i\leq p} c_i t_i$ denote the maximum computation time over all processors. If $\mathcal{C}$ verifies*

$$\forall i, 1\leq i\leq p,\ t_i c_i \leq m \leq t_i(c_i+1), \tag{1}$$

then it is optimal for the total number of chunks s.

Proof Take an allocation verifying the above condition 1. Suppose that it is not optimal. Then there exists a better allocation $\mathcal{C}'=(c'_1,\dots,c'_p)$ with $\sum_{1\leq i\leq p} c'_i = s$, such that

$$m' = \max_{1\leq i\leq p} c'_i t_i < m.$$

By definition of m, there exists i_0 such that $m=c_{i_0}t_{i_0}$. We can then successively derive

$$\begin{aligned}
c_{i_0}t_{i_0} &= m > m' \geq c'_{i_0}t_{i_0}\\
c_{i_0} &> c'_{i_0}\\
\exists i_1, c_{i_1} &< c'_{i_1} \quad \left(\text{because} \sum_{1\leq i\leq p} c_i = s = \sum_{1\leq i\leq p} c'_i\right)\\
c_{i_1}+1 &\leq c'_{i_1}\\
t_{i_1}(c_{i_1}+1) &\leq t_{i_1}c'_{i_1}\\
m &\leq m' \quad (\text{by definition of } m \text{ and } m')
\end{aligned}$$

which contradicts the non-optimality of the original allocation. ■

There remains to build allocations satisfying Condition (1). The following algorithm suffices:

- For the chunk size $s=0$, take the optimal allocation $(0,0,\dots,0)$.
- To derive an allocation $\mathcal{C}'$ verifying equation (1) with chunk size s from an allocation $\mathcal{C}$ verifying (1) with chunk size $s-1$, add 1 to a well-chosen c_j one that verifies

$$t_j(c_j+1) = \min_{1\leq i\leq p} t_i(c_i+1). \tag{2}$$

  In other words, let $c'_i = c_i$ for $1\leq i\leq p, i\neq j$, and $c'_j = c_j+1$.

Lemma 2 *This algorithm is correct.*

Proof We have to prove that allocation $\mathcal{C}'$, given by the algorithm, verifies Equation (1).

Since allocation $\mathcal{C}$ verifies equation (1), we have $t_i c_i \leq m \leq t_j(c_j+1)$. By definition of j from Equation (2), we have

$$m' = \max_{1\leq i\leq p} t_i c'_i = \max\left(t_j(c_j+1), \max_{1\leq i\leq q, i\neq j} t_i c_i\right) = t_j c'_j.$$

We then have $t_j c'_j \leq m' \leq t_j(c'_j+1)$ and

$$\begin{aligned}
&\forall i\neq j, 1\leq i\leq q,\\
&\qquad \mathbf{t_i c'_i} = t_i c_i \leq m \leq \mathbf{m'} \leq t_j c'_j = \min_{1\leq i\leq p} t_i(c_i+1) \leq t_i(c_i+1) = \mathbf{t_i(c'_i+1)},
\end{aligned}$$

so the resulting allocation does verify Equation (1). ■

This completes the proof of the proposition. ■

The complexity of the dynamic programming algorithm is $O(pB)$, where p is the number of processors and B, the upper bound on the number of chunks. Note that the cost of the allocations is not a decreasing function of B. However, the algorithm "converges" to the perfectly balanced solution: for a number of chunks equal to $C = 79$, we retrieve the optimal solution $(c_1, c_2, c_3) = (40, 24, 15)$ whose cost is $\frac{120}{79} = 1.52$. When processor speeds are accurately known and guaranteed not to change during program execution, the dynamic programming approach provides the best possible load balancing of the processors. Let us discuss the relevance of both hypotheses:

Estimating processor speed. There are too many parameters to accurately predict the actual speed of a machine for a given program, even assuming that the machine load will remain the same throughout the computation. Cycle-times must be understood as *normalized cycle-times* [9], i.e application-dependent elemental computation times, which are to be computed via small-scale experiments (repeated several times, with an averaging of the results).

Changes in the machine load. Even during the night, the load of a machine may suddenly and dramatically change because a new job has just been started. The only possible strategy is to "use past to predict future": we can compute performance histograms during the current computation, these lead to new estimates of the t_i, which we use for the next allocation.

In a word, a possible approach is to slice the total work into phases. We use small-scale experiments to compute a first estimation of the t_i, and we allocate chunks according to these values for the first phase. During the first phase we measure the actual performance of each machine. At the end of the phase we collect the new values of the t_i, and we use these values to allocate chunks during the second phase, and so on. Of course a phase must be long enough, say a couple of seconds, so that the overhead due to the communication at the end of each phase is negligible. Each phase corresponds to B chunks, where B is chosen by the user as a trade-off: the larger B, the more even the predicted load, but the larger the inaccuracy of the speed estimation.

3 Tiling

To understand how dependences may prevent dynamic strategies to reach a good efficiency, we use the simple example of a tiled computation over a rectangular iteration space. Tiling has been studied by several authors and in different contexts, and we refer the reader to the papers by Högsted, Carter, and Ferrante [14] and by Calland, Dongarra, and Robert [6], which provide a review of the existing literature. Briefly, the idea is to partition the iteration space of a loop nest with uniform dependences into tiles whose shape and size are optimized according to some criterion (such as the communication-to-computation ratio). Once the tile shape and size are defined, the tiles must be distributed to physical processors and the final scheduling must be computed. Tiling is a widely used technique to increase the granularity of computations and the locality of data references. The larger the tiles, the more efficient are the computations performed using state-of-the-art processors with pipelined arithmetic units and a multilevel memory hierarchy (this feature is illustrated by recasting numerical linear algebra algorithms in terms of blocked Level 3 BLAS kernels [11, 7]). Another advantage of tiling is the decrease in communication time (which is proportional to the surface of the tile) relative to the computation time (which is proportional to the volume of the tile). These two features make tiling a very useful technique for programming Networks of Workstations (NOWs): as soon as the tiles are large enough, communications can be fully overlapped by (independent) computations.

Consider the two-dimensional rectangular iteration space represented in Figure 1. Tiles are rectangular, and their edges are parallel to the axes. All tiles have the same fixed size. Tiles are indexed as $T_{i,j}$, $0 \le i < N_1$, $0 \le j < N_2$. 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. Given a tile $T_{i,j}$, we call both tiles $T_{i+1,j}$ and $T_{i,j+1}$ its successors, whenever the indices make sense.

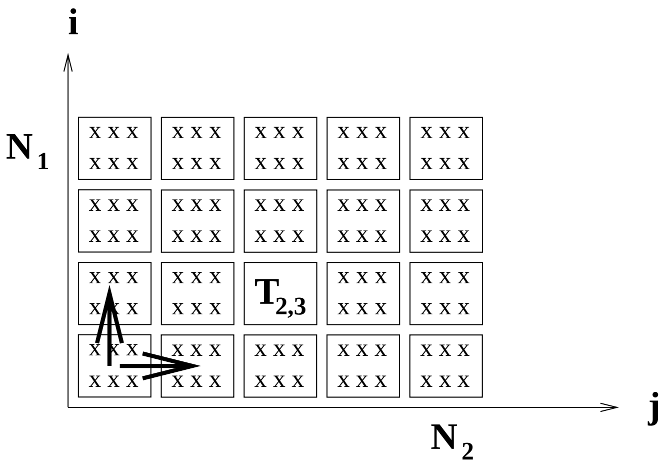


Figure 1: A tiled iteration space with horizontal and vertical dependences.

There are p available processors, numbered from 1 to p. Let t_q the time needed by processor P_q to execute a tile, for $1 \leq q \leq p$. While we assume the computing resources are heterogeneous, we assume the communication network is homogeneous: if two adjacent tiles T and T' are not assigned to the same processor, we pay the same communication overhead T_{com}, whatever the processors that execute T and T'. This is a crude simplification because the network interfaces of heterogeneous systems are likely to exhibit very different latency characteristics. However, because communications can be overlapped with independent computations, they eventually have little impact on the performance, as soon as the granularity (the tile size) is chosen large enough. This theoretical observation has been verified during our MPI experiments (see below).

Tiles are assigned to processors by using a scheduling σ and an allocation function *proc*, both to be determined. Tile T is allocated to processor proc(T), and its execution begins at time-step $\sigma(T)$. Formally, the constraints induced by the dependences are the following: for each tile T and each of its successors T', we have

$$\begin{cases} \sigma(T) + t_{\text{proc}(T)} \leq \sigma(T') & \text{if proc}(T) = \text{proc}(T') \\ \sigma(T) + t_{\text{proc}(T)} + T_{\text{com}} \leq \sigma(T') & \text{otherwise} \end{cases}$$

To increase the granularity and to minimize the number of communications, columns of tiles (rather than single tiles) are allocated to processors. So a computational chunk will be a tile column. Columnwise allocations are asymptotically optimal [5]. When targeting a homogeneous NOW, a natural way to allocate tile columns to physical processors is to use a pure cyclic allocation [16, 14, 2] (in HPF words, this is a `CYCLIC(1)` distribution of tile columns to processors). For heterogeneous NOWs, we use a periodic allocation based upon the ideas of Section 2.

3.1 Theoretical heuristic

Let $L = \text{lcm}(t_1, t_2, \ldots, t_p)$ and consider an iteration space with L columns: if we allocate $\frac{L}{t_i}$ columns of tiles to processor i, as suggested in Section 2, all processors need the same number of time-steps to compute all their tiles: the workload is perfectly balanced. Of course, we must schedule the different tiles allocated to a same processor so that the processors do not remain idle, waiting for other processors because of dependence constraints.

We introduce a heuristic that allocates the tiles to processors by blocks of columns whose size is computed according to the above discussion. This heuristic produces an asymptotically optimal allocation: the ratio of its makespan over the optimal execution time tends to 1 as the number of tiles (the domain size) increases.

In a columnwise allocation, all the tiles of a given column of the iteration space are allocated to the same processor. When contiguous columns are allocated to the same processor, they form a block. When a processor is assigned several blocks, the scheduling is the following:

1. Blocks are computed one after the other, in the order defined by the dependences. The computation of the current block must be completed before the next block is started.
2. Tiles inside each block are computed in a rowwise order: if, say, 3 consecutive columns are assigned to a processor, it will execute the three tiles in the first row, then the three tiles in the second row, and so on. Note that (given 1.) this strategy is the best to minimize the latency (for another processor to start next block as soon as possible).

We prove in [4] that dependence constraints do not slow down the execution of two consecutive blocks (of adequate size) by two different-speed processors. To state our heuristic, let $P_1, \dots, P_p$ be P processors that respectively execute a tile in time $t_1, \dots, t_p$. We allocate blocks of column to processors by panels of size $C = L \times \sum_{i=1}^{p} \frac{1}{t_i}$, where $L = \mathrm{lcm}(t_1, t_2, \dots, t_p)$ columns. For the first panel, we assign the block B_1 of the first L/t_1 columns to P_1, the block B_2 of the next L/t_2 columns to P_2, and so on until P_p receives the last L/t_p columns of the panel. We repeat the same scheme with the second panel (columns $C+1$ to $2C$) first, and so on until all columns are allocated (note that the last panel may be incomplete). As already said, processors will execute blocks one after the other, row by row within each block.

Proposition 2 *(see [4]) Our heuristic is asymptotically optimal: letting T be its makespan, and T_{opt} be the optimal execution time, we have*

$$\lim_{N_2 \to +\infty} \frac{T}{T_{opt}} = 1.$$

The two main advantages of our heuristic are (i) its regularity, which leads to an easy implementation; and (ii) its guarantee: it is theoretically proved to be close to the optimal. However, we need to adapt it to deal with practical cases, because the number $C = L \times \sum_{i=1}^{p} \frac{1}{t_i}$ of columns in a panel may be too large, as shown by the example of Table 2.

3.2 Practical heuristic

We choose the "best" block sizes while bounding the total number of columns B, using the dynamic programming algorithm of Section 2. B should be chosen large enough to balance processor loads but not too large compared to the total number of columns, to ensure that T is closed to T_{opt} (see the proof of Proposition 2). In other words, B is a trade-off between idle time and latency. Note that small values of B may lead to very good performances (the cost is not a decreasing function of B). Finally, an accurate estimation of processor speeds is not mandatory if we re-estimate these frequently.

3.2.1 MPI Experiments

We report several experiments on the network of workstations presented in Table 2. We study different columnwise allocations. Our simulation program is written in C using the MPI library for communication. It is not a full tiling program, because we have not inserted the code required to deal with the boundaries of the computation domain. The domain has 100 rows and a number of columns varying from 200 to 1000 by steps of 100. An array of doubles is communicated for each communication; its size is the square root of the tile area.

We are reporting experiments for a single tile size, because results are not sensitive to this parameter (as soon as the tile is large enough so that communications can be overlapped). This is confirmed by the correlated curves represented in Figure 2, where the execution times of three of our modified heuristics are measured for different tile sizes.

The actual communication network is an Ethernet network. It can be considered as a bus, not as a point-to-point connection ring; hence our model for communication is not fully correct. However, this configuration has little impact on the results, which correspond well to the theoretical predictions. The workstations were not dedicated to us, but the experiments were run at hours when we expected to be the only users. Still, the

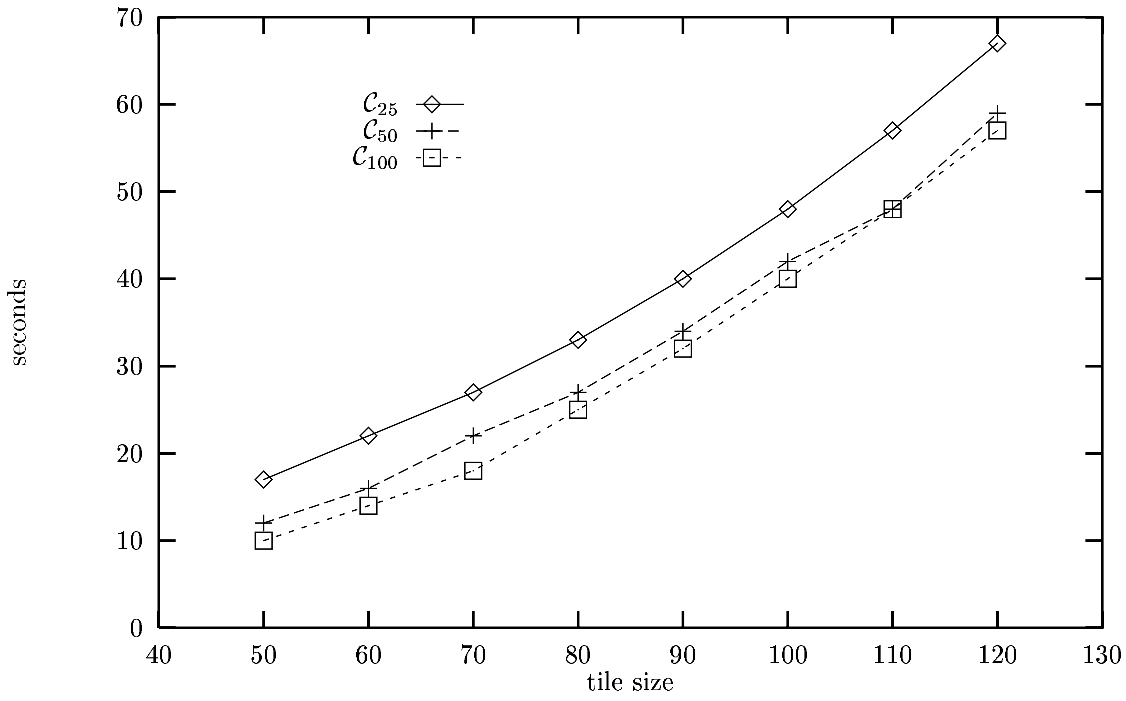


Figure 2: Our modified heuristics with different tile sizes. C_u denotes the best allocation algorithm for a panel size upper bound equal to u.

load did vary from time to time: timings reported in the figures are the average of several measures, from which aberrant data have been suppressed. In Figures 3 and 5, we show for reference the sequential time as measured on the fastest machine, "nala".

Cyclic Allocations. We have experimented with cyclic allocations on the 6 fastest machines, on the 7 fastest machines, and on all 8 machines. Because cyclic allocation is optimal when all processors have the same speed, this will be a reference for other simulations. We have also tested a block cyclic allocation with block size equal to 10, in order to see whether the reduced amount of communication helps. Figure 3 presents the results[1] for these 6 allocations (3 purely cyclic allocations using 6, 7, and 8 machines, and 3 block-cyclic allocations).

We comment on the results of Figure 3 as follows:

- With the same number of machines, a block size of 10 is better than a block size of 1 (pure cyclic).
- With the same block size, adding a single slow machine is disastrous, and adding the second one only slightly improves the disastrous performances.
- Overall, only the block cyclic allocation with block size 10 and using the 6 fastest machines gives some speedup over the sequential execution.

We conclude that cyclic allocations are not efficient when the computing speeds of the available machines are very different. For the sake of completeness, we show in Figure 4 the execution times obtained for the same domain (100 rows and 1000 columns) and the 6 fastest machines, for block cyclic allocations with different block sizes. We see that the block-size has a small impact on the performances.

We point out that cyclic allocations would be the outcome of a greedy master-slave strategy. Indeed, processors will be allocated the first P columns in any order. Re-number processors according to this initial

[1]Some results are not available for 200 columns because the panel size is too large.

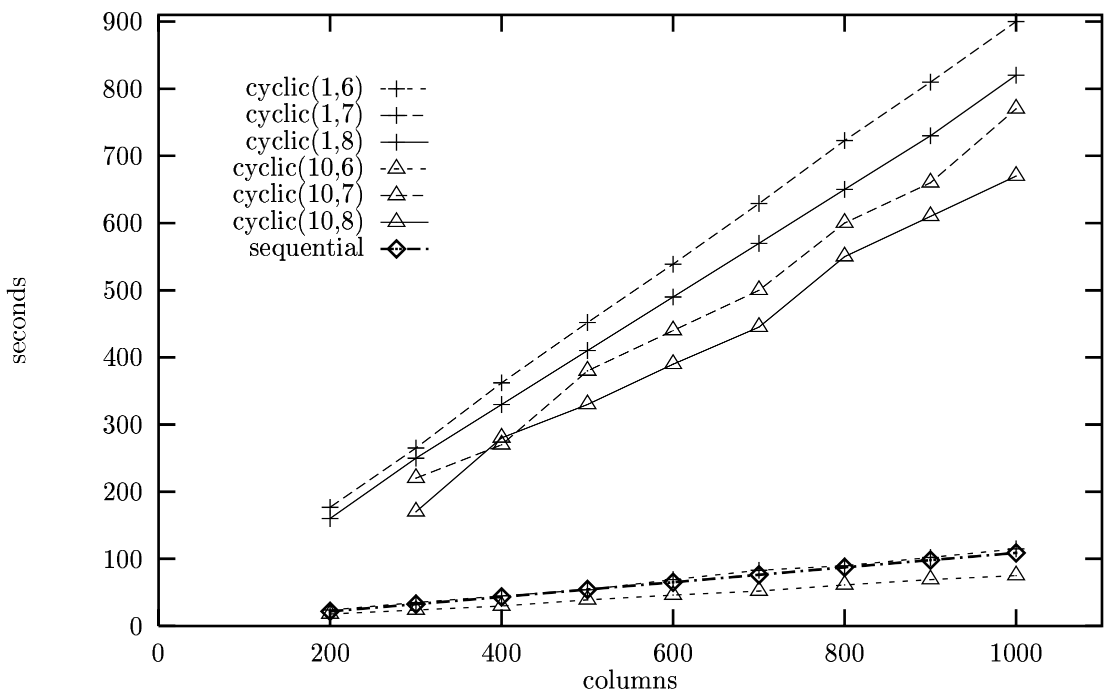


Figure 3: Experimenting with cyclic and block-cyclic allocations: cyclic(b,m) corresponds to a block cyclic allocation with block size b, using the m fastest machines of Table 2.

assignment. Then throughout the computation, P_j will return after P_{j-1} and just before P_{j+1} (take indices modulo p) , because of the dependences. Hence computations would only progress at the speed of the slowest processor, with a cost $\frac{\max t_p}{p}$.

Using our heuristics. Consider now our heuristics. In Table 4, we show the block sizes computed by the algorithm described in Section 3.2 for different upper bounds of the panel size. The best allocation computed with bound u is denoted as $\mathcal{C}_u$. The time needed to compute these allocations is completely negligible with respect to the computation times (a few milliseconds versus several seconds).

	nala	bluegrass	dancer	donner	vixen	rudolph	zazu	simba	*cost*	panel
$\mathcal{C}_{25}$	7	3	2	2	2	2	0	0	4.44	18
$\mathcal{C}_{50}$	15	6	5	5	4	4	0	0	4.23	39
$\mathcal{C}_{100}$	33	14	11	11	9	9	0	0	4.18	87
$\mathcal{C}_{150}$	52	22	17	17	15	14	1	1	4.12	139

Table 4: Block sizes for different panel size bounds.

Figure 5 presents the results for these allocations. Here are some comments:

- Each of the allocations computed by our heuristic is superior to the best block-cyclic allocation.
- For 1000 columns and allocation $\mathcal{C}_{150}$, we obtain a speedup of 2.2 (and 2.1 for allocation $\mathcal{C}_{50}$), which is very satisfying (see below).

The optimal cost for our workstation network is $cost_{opt} = \frac{L}{C} = \frac{34{,}560{,}240}{8{,}469{,}789} = 4.08$. Note that the cost of $\mathcal{C}_{150}$, $cost(\mathcal{C}_{150}) = 4.12$, is very close to the optimal cost. The peak theoretical speedup is equal to $\frac{\min_i t_i}{cost_{opt}} = 2.7$. For 1000 columns, we obtain a speedup equal to 2.2 for $\mathcal{C}_{150}$. This is satisfying considering

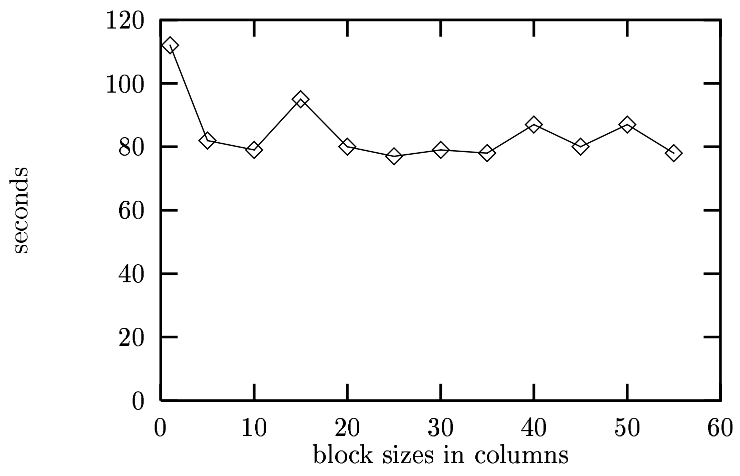


Figure 4: Cyclic allocations with different block sizes.

that we have here only 7 panels, so that side effects still play an important role. Note also that the peak theoretical speedup has been computed by neglecting all the dependences in the computation and all the communications overhead.

3.2.2 Load-balancing on the fly

Our allocation algorithm can be used to dynamically balance the load of the computation. Indeed, one has just to measure the elapsed time for some current computations, recompute a new allocation based on these computation times (which only costs a few milliseconds) and redistribute the data accordingly.

The main problem while doing this is to determine the frequency at which this reallocation should take place. Indeed, a redistribution could be costly (though, in general, it could be limited to neighbor-to-neighbor communications). On the other hand, the highest the frequency of the reallocation , the highest the reactiveness to load changes. As a conclusion, selecting the frequency of reallocations is highly application- and network-dependent.

4 Finite-difference stencil computation

4.1 Framework

In this section, we briefly sketch how to tile a one-dimensional relaxation problem on a heterogeneous NOW. Figure 6 illustrates the Fermi-Pasta-Ulam model for one-dimensional crystals [17]. Let x_i^t denote the coordinate of the i^{th} molecule at time t. According to the model, we have

$$\begin{aligned} x_i^t &= 2x_i^{t-1} + x_i^{t-2} + \frac{(dt)^2}{m} f(x_{i-1}^{t-1}, x_i^{t-1}, x_{i+1}^{t-1}) \\ &= g(x_{i-1}^{t-1}, x_i^{t-1}, x_{i+1}^{t-1}, x_i^{t-2}) \end{aligned}$$

The main kernel of a program that computes the Fermi-Pasta-Ulam model for a one dimensional string of l molecules and during T time-steps is the following (we assume a special processing on the boundaries):

```
do $t = 1, T$
    do $i = 1, l$
        $B[t,i] = g(B[t-1,i-1], B[t-1,i], B[t-1,i+1], B[t-2,i], A[t,i])$
```

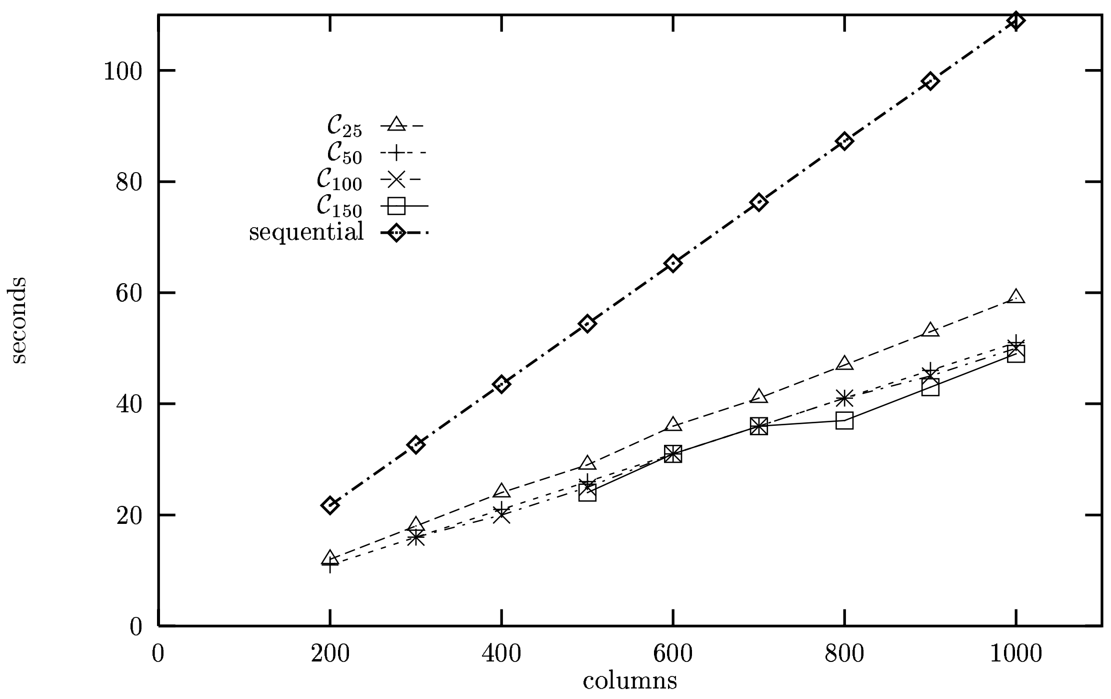
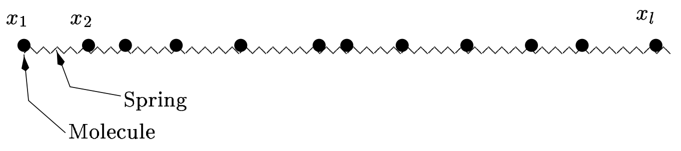


Figure 5: Experimenting with our modified heuristics.



$$m\vec{a} = \sum \vec{F}$$

$$m\ddot{x}_i = k(x_{i+1} - x_i) + K(x_{i+1} - x_i)^\beta + k(x_{i-1} - x_i) + K(x_{i-1} - x_i)^\beta$$

Figure 6: The Fermi-Pasta-Ulam model of a one dimensional crystal.

In this kernel, $B[t,i]$ stores the current position of molecule i at step t, while $A[t,i]$ represents some purely local data. The dependence vectors in this loop can be summarized by the set

$$\left\{ \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 2 \\ 0 \end{pmatrix} \right\}.$$

Because of the conflicting dependence vectors $(1,1)^t$ and $(1,-1)^t$, the iteration space must be rotated (skewed) before tiling techniques can be applied. The new loop nest is the following:

```
do $t = 1, T$
    do $i' = t, l + t - 1$
        $B[t, i' - t + 1] = g(B[t-1, i' - t], B[t-1, i' - t + 1],$
                $B[t-1, i' - t + 2], B[t-2, i' - t + 1], A[t, i' - t + 1])$
```

The new dependence vectors are

$$\left\{ \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 2 \\ 2 \end{pmatrix} \right\}.$$

Because all the components of the new dependence vectors are nonnegative, the two loops are now permutable, and we can tile the loop nest to increase the granularity: in the tiled iteration space of Figure 7, each circle represents a tile instead of a single computation node.

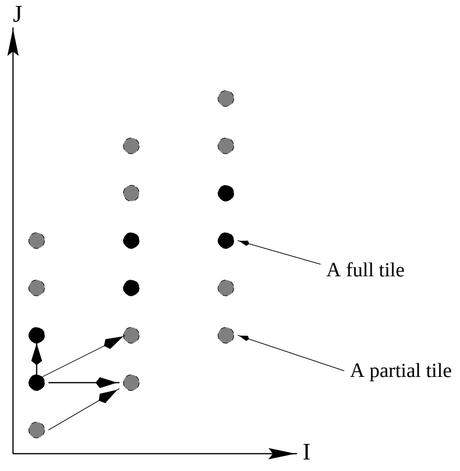


Figure 7: The tiled iteration space.

Dependence vectors between tiles are outlined in Figure 7, they correspond to the set

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

In other words, the computation of a tile cannot be started before its left, lower and lower-left neighbor tiles have been executed: the diagonal dependence vector $(1,1)^t$ turns out to be redundant. Still, the diagonal communication must be routed, either horizontally and then vertically, or the other way round, and it may even be combined with any of the other two messages (induced by dependence vectors $(0,1)^t$ and $(1,0)^t$).

To summarize, we end up with a tiled iteration space whose shape is a parallelogram and whose dependence vectors are the pair $(0,1)^t$ and $(1,0)^t$. This turns out to be a very general situation when solving finite-difference problems.

4.2 Solution for a parallelogram-shaped iteration space

Our solution is similar to that for the rectangular-shaped problem: columns of tiles are distributed to the processors. More precisely, the panel size B is chosen such that the load is best balanced and that some idle time is not created by too large blocks of columns. However, because the domain has been skewed, dependences further constrain the problem, and there is a technical condition (C) to enforce so that no processor is kept idle. To characterize a parallelogram-shaped iteration space, we use the notations of [14]:

- There are N columns of M tiles.
- The rise parameter r relates the slope (in reference to the horizontal axis) of the top and bottom boundaries of the iteration space. For example in Figure 7, the rise is $r = \frac{7}{5}$ (tiles are of size 7×5).

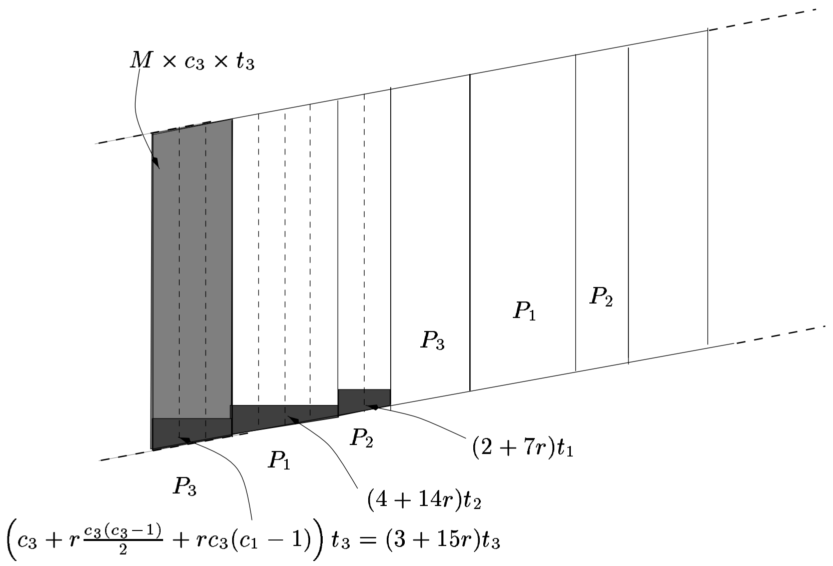
Let t_j denote the time needed by processor P_j to execute a single tile[2]. Reorder processors so that $c_1 t_1 \leq c_2 t_2 \leq \ldots \leq c_p t_p$). We need to ensure that P_p never gets idle, i.e. that the startup latency is smaller than the time for P_p to compute a block of columns. This condition can be safely over-estimated by the following inequality (where $c_{p+1} = c_1$):

[2]Hence the time needed by P_j to execute a tile column is Mt_j, which is not coherent with the notations of Section 3. This is motivated by the (technical) need to track elementary paths in the skewed domain (see the proof of Proposition 3).

Condition (C)

$$Mt_pc_p \geq \sum_{j=1}^{p} \left(\left(c_j + r\frac{c_j(c_j-1)}{2} + rc_j(c_{j+1}-1) \right) t_j + t_{comm} \right)$$

This formula is explained in Figure 8. Note that condition (C) will always hold for large domains.



$M \times c_3 \times t_3$
P_3
P_1
P_2
P_3
P_1
P_2
$(2+7r)t_1$
$(4+14r)t_2$
$\left(c_3 + r\frac{c_3(c_3-1)}{2} + rc_3(c_1-1)\right) t_3 = (3+15r)t_3$

Figure 8: The last processor never remains idle (except at the beginning and at the end of the whole computation) if the time to compute the dark-shadowed part (over estimation of the start-up latency) is smaller than the time to compute the light-shadowed part. In that particular example, $(t_1, t_2, t_3) = (3,5,5)t_{calc}$, the total number of chunks is taken to be $B = 9$, so $(c_1, c_2, c_3) = (4,3,2)$. The condition is then $15Mt_{calc} \geq 3t_{comm} + (37+152r)t_{calc}$. So if $r = 1$ and $t_{comm} = t_{calc}$, the condition becomes $M \geq 12.8$, which in practice will always be true.

Proposition 3 *(see [4]) Our heuristic is asymptotically optimal when condition (C) holds.*

Again, a greedy strategy would lead to a cyclic allocation, so that computation would progress only at the pace of the slowest processor.

5 LU decomposition

The last algorithm that we deal with in this paper is the (blocked) LU decomposition taken from the ScaLAPACK library [7].

5.1 Algorithm

The LU decomposition algorithm works as follows: at each step, the pivot processor processes the pivot panel (a block of r columns) and broadcasts it to all the processors, which update their remaining columns. For next step, the next r columns become the pivot panel, and the computation progresses. The preferred distribution for a homogeneous NOW is a $CYCLIC(r)$ distribution of columns, where r is typically chosen as $r = 32$ or $r = 64$. We want to use our modified heuristic to distribute B chunks of data to processors, where a chunk is a block of r consecutive columns (see Figure 9).

Because the largest fraction of the work takes place in the update, we load-balance the work so that the update is best balanced. Given the fact that the size of the matrix shrinks as the computation goes on, we use the dynamic programming for $s = 0$ to $s = B$ in a reverse order, as illustrated below. Consider the toy

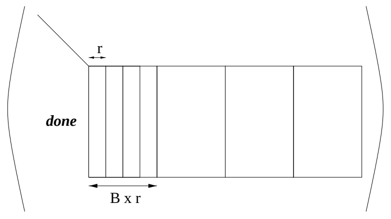


Figure 9: Allocating slices of B chunks.

Chunk number	1	2	3	4	5	6	7	8	9	10
Processor number	1	2	1	3	1	2	1	1	2	3

Table 5: Static allocation for $B = 10$ chunks.

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 5. The allocation of chunks to processors is obtained by reading the second line of Table 5 from right to left: $(3, 2, 1, 1, 2, 1, 3, 1, 2, 1)$ (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 update in the first slice at each step: when there are the initial 10 chunks, next when only 9 chunks remain, ..., finally when only the last chunk remain. The update of the other slices remain 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 update.

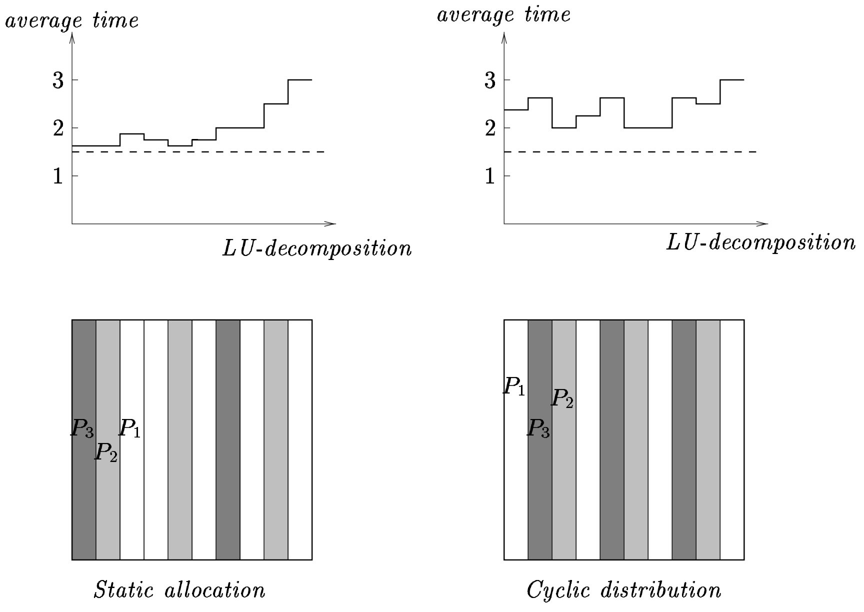


Figure 10: Comparison of two different distributions for the LU-decomposition algorithm on a heterogeneous platform made of 3 processors of relative speed 3, 5 and 8. The first distribution is the one given by our algorithm, the second one is the cyclic distribution. The total number of chunks is $B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

Name	farot	arquebuse	smirnoff	loop	cornas	xeres
Description	Ultra 1	Sparc 20	Sparc 5	Sparc 5	Sparc 5	ELC
Execution time t_i	210	361	646	573	660	555

Table 6: The 6 Sun workstations used for LU decomposition. Processor speeds have been measured by computing the same LU decomposition on each machine separately.

5.2 PVM Experiments

In this section, we report several experiments on the network of workstations presented in Table 6. More precisely, we compare the ScaLAPACK implementation of the cyclic(6) allocation to a PVM implementation of our column distribution. Column blocks are of size $32 \times n$, where n (the matrix size) varies from 288 to 3168 by step of 288. Results are reported in Figures 11 and 12. We start with the following comments:

- Asymptotically, the computation time for LU-decomposition with cyclic distribution is imposed by the slowest processor. In the example, cyclic distribution will lead to the same execution time as if computed with 6 identical processors of duration 660.
- Asymptotically, the best distribution should be equivalent to the computation on 6 identical processors of speed $\frac{6}{\frac{1}{210}+\frac{1}{361}+\dots+\frac{1}{573}} = 424.3$.

In our PVM experiments, B is taken to be 9: hence *farot* has 3 processes, *arquebuse* has 2 processes, *cornas*, *smirnoff*, *xeres* and *loop* have 1 process each. Such a distribution is asymptotically equivalent to having 6 identical processors of speed $\frac{\max(3\times 210, 2\times 361, 646, 573, 660, 555)}{9} \times 6 = 481.33$.

Hence, if we define the speedup as the ratio $\frac{\text{Computation time with our distribution}}{\text{Computation time with a cyclic distribution}}$, the best theoretical speedup would be $\frac{660}{481.3} = 1.37$. However, with our choice of B, we cannot expect a speedup greater than $\frac{660}{424.3} = 1.56$.

In practice, we do obtain a speedup closed to the theoretical speedup for large matrices (see Figures 11 and 12). We point out that this was obtained with a very small programming effort, because we did not modify anything in the ScaLAPACK routines, but only declared several PVM processes per machine. We did pay a high overhead and memory increase for managing these processes, so that a refined implementation would lead to better results.

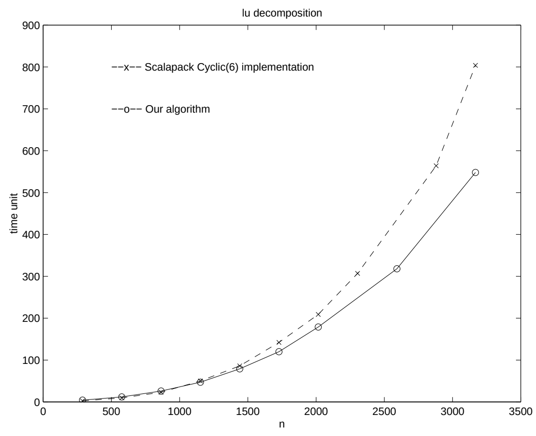


Figure 11: Experimental results for LU-decomposition using 6 heterogeneous workstations.

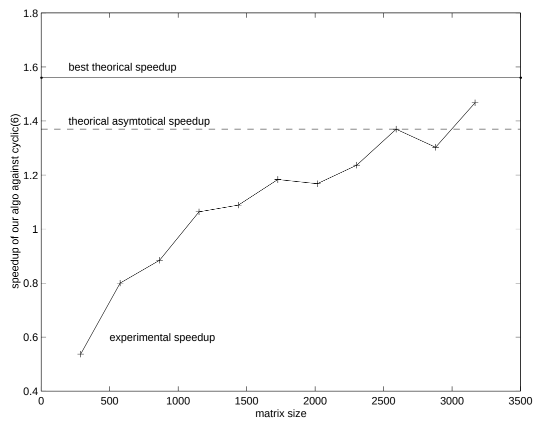


Figure 12: Speedup of the experiments of Figure 11: we report the ratio of the computation time with our new distribution (with $B = 9$ blocks of 32 columns) over the computation time with a regular *cyclic*(32) distribution (used in ScaLAPACK).

6 Conclusion

In this paper, we have discussed algorithmic techniques to deal with heterogeneous computing platforms. Such platforms are likely to play an important role in the future. We have introduced static allocation strategies that are asymptotically optimal. We have modified these strategies to allocate column chunks of reasonable size, and we have reported successful experiments on two heterogeneous networks of workstations. The practical significance of the modified heuristics should be emphasized: even when processor speeds are inaccurately known, allocating small but well-balanced chunks turns out to be quite successful, while dynamic greedy strategies fail.

References

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

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

[3] 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, 1998.

[4] P. Boulet, J. Dongarra, Frédéric Rastello, Yves Robert, and Frédéric Vivien. Algorithmic issues for heterogeneous computing platforms. Technical Report RR-98-49, LIP, ENS Lyon, 1998. Available at `www.ens-lyon.fr/LIP/lip/publis/publis.us.html`.

[5] P. Boulet, J. Dongarra, Yves Robert, and Frédéric Vivien. Tiling for heterogeneous computing platforms. Technical Report UT-CS-97-373, University of Tennessee, Knoxville, 1997.

[6] P.Y. Calland, J. Dongarra, and Y. Robert. Tiling with limited resources. In L. Thiele, J. Fortes, K. Vissers, V. Taylor, T. Noll, and J. Teich, editors, *Application Specific Systems, Achitectures, and Processors, ASAP'97*, pages 229–238. IEEE Computer Society Press, 1997. Extended version available on the WEB at http://www.ens-lyon.fr/~yrobert.

[7] 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).

[8] Michal Cierniak, Mohammed J. Zaki, and Wei Li. Customized dynamic load balancing for a network of workstations. *Journal of Parallel and Distributed Computing*, 43:156–162, 1997.

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

[10] Val Donaldson, Francine Berman, and Ramamohan Pandri. Program speedup in a heterogeneous computing network. *Journal of Parallel and Distributed Computing*, 21:316–322, 1994.

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

[12] Xing Du and Xiadong Zhang. Coordinating parallel processes on networks of workstations. *Journal of Parallel and Distributed Computing*, 46:125–135, 1997.

[13] Andrew S. Grimshaw, Jon B. Weissman, Emily A. West, and Ed. C. Loyot Jr. Metasystems: an approach combining parallel processing and heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 21:257–270, 1994.

[14] 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.

[15] Maher Kaddoura and Sanjay Ranka. Run-time support fo parallelization of data-parallel applications on adaptive and nonuniform computational environments. *Journal of Parallel and Distributed Computing*, 43:163–168, 1997.

[16] 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.

[17] M. Remoissenet. *Waves called solitons.* Springer Verlag, 1994.

[18] Yong Yan, Xiadong Zhang, and Yongsheng Song. An effective and practical performance model for parallel computing on nondedicated heterogeneous now. *Journal of Parallel and Distributed Computing*, 38:63–80, 1996.