A Proposal for a Heterogeneous Cluster ScaLAPACK (Dense Linear Solvers)

Vincent Boudet, Fabrice Rastello, Yves Robert

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 5668

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

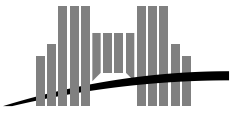
SPI

A Proposal for a Heterogeneous Cluster ScaLAPACK (Dense Linear Solvers)

Vincent Boudet, Fabrice Rastello and Yves Robert

February 1999

Research Report N° 1999-17



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

INRIA

A Proposal for a Heterogeneous Cluster ScaLAPACK (Dense Linear Solvers)

Vincent Boudet, Fabrice Rastello and Yves Robert

February 1999

Abstract

This paper discusses some algorithmic issues when computing with a heterogeneous network of workstations (the typical poor man's parallel computer). How is it possible to efficiently implement numerical linear algebra kernels like those included in the ScaLAPACK library ? Dealing with processors of different speeds requires to use more involved strategies than purely static 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). There is a challenge in determining a trade-off between the data distribution parameters and the process spawning and possible migration (redistribution) policies. We introduce a semi-static distribution strategy that can be refined on the fly, and we show that it is well-suited to parallelizing several kernels of the ScaLAPACK library such as LU or QR decomposition.

Keywords: heterogeneous networks, distributed-memory, different-speed processors, scheduling, mapping, numerical libraries.

Résumé

Dans ce rapport, nous nous intéressons à des problèmes algorithmiques liés à l'exécution de programmes sur un réseau de stations hétérogène (la machine parallèle du programmeur pauvre). Comment implémenter les algorithmes de calcul linéaire, de manière efficace, comme ceux inclus dans la librairie ScaLAPACK? Si dans le cadre d'un réseau de machines hétérogènes, une distribution cyclique purement statique des données est souvent optimale, elle n'est pas du tout adaptée à cette nouvelle configuration. Une distribution dynamique ne constitue pas non plus la solution à notre problème, à cause du surcoût de communications lié à la présence d'un maître ou à la présence incessante de redistributions. Une solution purement statique reste limitée à de courtes exécutions durant lesquelles la charge des processeurs ne varie pas. Nous proposons donc un algorithme semi-statique, quasi optimal dans le cas ou la charge des processeurs ne varie pas, et permettant toutefois des redistributions au vol de temps en temps le cas échéant. Ainsi, nous montrons par des tests effectués sur 2 plateformes différentes que cette approche constitue probablement une solution bien adaptée à la parallélisation de plusieurs noyaux de la librairie ScaLAPACK, comme par exemple les décompositions LU ou QR.

Mots-clés: plateforme hétérogène, mémoire distribuée, processeurs de vitesses différentes, ordonnancement, distribution, librairies de calcul numérique.

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. In this paper, we explore several possibilities to implement linear algebra kernels on heterogeneous networks of workstations (NOWs). Our goal is to come up with a first proposal for a heterogeneous "Cluster ScaLAPACK" library devoted to dense linear system solvers: how to efficiently implement numerical kernels like LU or QR decompositions on a heterogeneous NOW?

Consider a heterogeneous NOW: whereas programming a large application made up of several loosely-coupled tasks can be performed rather easily (because these tasks can be dispatched dynamically on the available processors), implementing a tightly-coupled algorithm (such as a linear system solver) requires carefully tuned scheduling and mapping strategies. 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 are likely to perform better, because the machine loads will be self-regulated, hence self-balanced, if processors pick up new tasks just as they terminate their current computation. However, data dependences, communication costs and control overhead may well lead to slow the whole process down to the pace of the slowest processors. On the other hand, static strategies will suppress (or at least minimize) data redistributions and control overhead during execution. To be successful, static strategies must obey a more refined model than standard block-cyclic distributions: such distributions are well-suited to processors of equal speed but would lead to a great load imbalance with processors of different speed. The design of static strategies that achieve a good load balance on a heterogeneous NOW is one of the major achievements of this paper.

The rest of the paper is organized as follows. In Section 2 we discuss a purely static strategy to allocate data and computations to heterogeneous processors. We propose an allocation based upon a dynamic-programming algorithm, to evenly distribute independent chunks of computations to different-speed processors. We analyze this strategy both theoretically and experimentally: in Section 3 we report PVM experiments run for LU and QR on two different heterogeneous NOWs. These experiments fully demonstrate the usefulness of our approach. In Section 4, we investigate several dynamic allocation strategies for implementing dense linear solvers on a heterogeneous NOW. Our goal is both to compare the static and dynamic approaches, and to present refined strategies based upon a mixture of static and dynamic allocations. We give some final remarks and conclusions in Section 5.

2 Static distribution

In this section we investigate static strategies for implementing ScaLAPACK routines on heterogeneous NOWs. 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. We start with the simple problem of distributing independent chunks of computations to processors, and we propose an optimal solution for that problem in Section 2.1. We use this result to tackle the implementation of linear solvers. We propose an optimal data

distribution in Section 2.2.

2.1 Distributing independent chunks

To illustrate the static approach, consider the following simple problem: given M 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, ..., 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. Since the loads must be integers, we use the following algorithm to solve the problem:

Algorithm 2.1: Optimal distribution for M independent chunks, over p processors of speed $t_1, \ldots, t_p$

Initialization: Approximate the c_i so that $c_i \times t_i \approx Constant$, and $c_1 + c_2 + \ldots + c_p \leq M$.
$forall\, i \in \{1, \ldots, p\}$, $c_i = \left\lfloor \frac{\frac{1}{t_i}}{\sum_{i=1}^{p} \frac{1}{t_i}} \times M \right\rfloor$.
Iteratively increment some c_i until $c_1 + c_2 + \ldots + c_p = M$
for $m = c_1 + c_2 + \ldots + c_p$ **to** M
    find $k \in \{1, \ldots, p\}$ such that $t_k \times (c_k + 1) = \min\{t_i \times (c_i + 1))\}$
    $c_k = c_k + 1$

Proposition 1 *Algorithm 2.1 gives the optimal allocation.*

Proof Consider an optimal allocation denoted by $o_1, \ldots, o_p$, and let j be such that $\forall i \in \{1, \ldots, p\}$, $o_j t_j \geq o_i t_i$. To prove the correctness of the algorithm, we prove the invariant

$$(C) : \forall i \in \{1, \ldots, p\}, c_i t_i \leq o_j t_j$$

After the initialization, $c_i \leq \frac{\frac{1}{t_i}}{\sum_{k=1}^{p} \frac{1}{t_k}} \times M$. We have $M = \sum_{k=1}^{p} o_k \leq o_j t_j \times \sum_{k=1}^{p} \frac{1}{t_k}$. Hence, $c_i t_i \leq \frac{M}{\sum_{k=1}^{p} \frac{1}{t_k}} \leq o_j t_j$, and condition (C) holds.

We use an induction to prove that condition (C) holds after each incrementation. Suppose that at a given step some c_k will be incremented. Before that step, $\sum_{i=1}^{p} c_i < M$, hence there exists $k' \in \{1, \ldots, p\}$ such that $c_{k'} \leq o_{k'}$. We have $t_{k'}(c_{k'} + 1) \leq t_{k'} o_{k'} \leq t_j o_j$, and the choice of k implies that $t_k(c_k + 1) \leq t_{k'}(c_{k'} + 1)$. Condition (C) does hold after the incrementation.

Finally, the time needed to compute the M chunks with the allocation $c_1, \ldots, c_p$ is $\max\{t_i \times c_i\} \leq o_j t_j$, and our allocation is optimal. ■

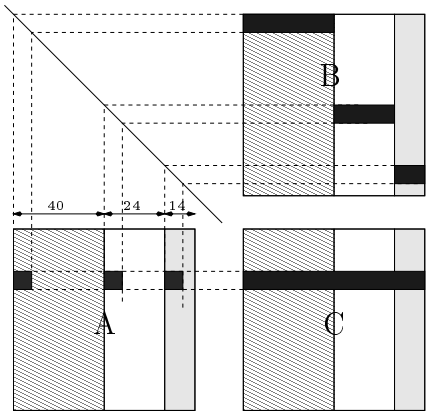
Complexity and use Since, $c_1 + c_2 + \ldots + c_p \geq M - p$, there are at most p steps of incrementation, so that the complexity of Algorithm 2.1 is[1] $O(p^2)$. This algorithm can only be applied to simple load

[1] Using a naive implementation. The complexity can be reduced down to $O(p \log(p))$ using ad-hoc data structures.

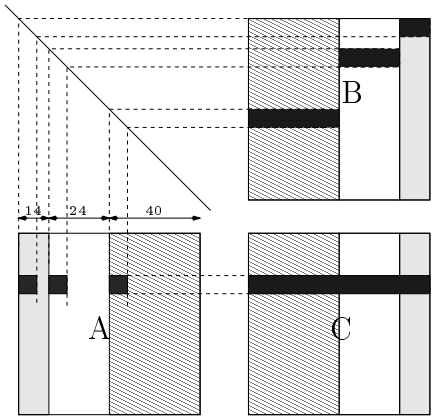
balancing problems such as matrix-matrix product on a processor ring. Indeed, such an algorithm can be decomposed into successive communication-free steps. The communication between steps are reduced to a simple shift across the ring of processes. Each step consists of a bunch of independent chunks that can be distributed using Algorithm 2.1. 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 blocksize for cache-based workstations [2]). Hence, $M = 78$ blocks of columns have to be distributed among the processors, and $M = 78$ independent chunks will be computed at each step. Table 1 applies Algorithm 2.1 to this load balancing problem. A few different steps for matrix multiplication are represented in Figure 1. Our simple allocation is quite sufficient for matrix multiplication, because each step is optimally load-balanced.

Steps	c_1	c_2	c_3	$\max_i(c_i t_i)$
Init, m=76	39	23	14	117
m=77	40	23	14	120
m=M=78	40	24	14	120

Table 1: Steps of algorithm 2.1 for 3 processors with $t_1 = 3$, $t_2 = 5$ and $t_3 = 8$, and $M = 78$



1^{st} **step**:
- p_1: $C[*, 1:40] += A[*, 1] \times B[1, 1:40]$;
- p_2: $C[*, 41:64] += A[*, 41] \times B[41, 41:64]$
- p_3: $C[*, 65:78] += A[*, 65] \times B[65, 65:78]$.



39^{th} **step**:
- p_1: $C[*, 1:40] += A[*, 39] \times B[39, 1:40]$

...

Figure 1: Different steps 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.

When processor speeds are accurately known and guaranteed not to change during program execution, the previous 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* [4], 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. See the survey paper of Berman [1] for further details.

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. We come back to estimating processor speeds in Section 4.5.

2.2 Linear solvers

Whereas the previous solution is well-suited to matrix multiplication, it does not perform efficiently for LU decomposition. Roughly speaking, the LU decomposition algorithm works as follows for a heterogeneous NOW: blocks of r columns are distributed to processors in a cyclic fashion. This is a $CYCLIC(r)$ distribution of columns, where r is typically chosen as $r = 32$ or $r = 64$ [2]. At each step, the processor that owns the pivot block factors it and broadcasts it to all the processors, which update their remaining column blocks. At the next step, the next block of 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 $r = 64$.

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[2]. 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 r columns. If the matrix size is $n = M \times r$, there are $M - 1$ chunks. We can use Algorithm 2.1 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 $M - 2$. If we 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 columns blocks to processors that will remain the same throughout the computations, as the elimination progresses. We aim at balancing the updates of all steps with the same allocation. As illustrated in Figure 2, 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 M are updated at step i. Hence our objective is to find a distribution such that for each $i \in \{2, \ldots, M\}$, the amount of blocks in $\{i, \ldots, M\}$ owned by a given processor is approximately inversely proportional

[2]See Section 4.4 for more refined strategies.

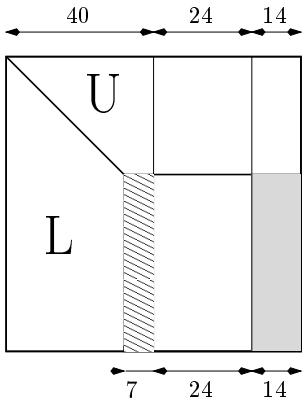


Figure 2: 33^{rd} step of LU decomposition (indices are block numbers): with the former distribution, the computation becomes less balanced. Here, after factoring block 33, processor 1 has 7 updates and works for $7 \times 3 = 21$ units of time, while processor 2 works $24 \times 5 = 120$ units of time.

to its speed. To derive such a distribution, we use a dynamic programming algorithm which 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 2: Running the dynamic programming algorithm with 3 processors: $t_1 = 3$, $t_2 = 5$, and $t_3 = 8$.

A dynamic programming algorithm In Table 2, 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: the execution time, for an allocation $\mathcal{C} = (c_1, c_2, \ldots, c_p)$ is $\max_{1 \leq i \leq p} \; c_i t_i$ (the maximum is taken over all processor execution times), so that the average cost to execute one chunk is

$$cost(\mathcal{C}) = \frac{\max_{1 \leq i \leq p} c_i t_i}{\sum_{i=1}^{p} c_i}$$

For instance at step 4, i.e. to allocate a fourth chunk, we start from the solution for three 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 3: Static allocation for $B = 10$ 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 $(c_1, c_2, c_3) = (2, 1, 1)$.

Of course, if we are to allocate 10 chunks, we can use Algorithm 2.1 and find that 5 chunks should be given to processor 3 to P_2 and 2 to P_3. But the dynamic programming algorithm returns the optimal solution for allocating any number of chunks, from 1 chunk up to B chunks:

Proposition 2 *(see [3]) The dynamic programming algorithm returns the optimal allocation for any number of chunks up to B.*

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.

Application to LU decomposition For LU decomposition we allocate slices of B blocks to processors, as illustrated in Figure 3. B is a parameter that will be discussed below. For a matrix of size $n = m \times r$, we can simply let $B = m$, i.e. define a single slice.

Within each slice, we use the dynamic programming algorithm for $s = 0$ to $s = B$ in a "reverse" order. 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 3. The allocation of chunks to processors is obtained by reading the second line of Table 3 from right to left: $(3, 2, 1, 1, 2, 1, 3, 1, 2, 1)$ (see Figure 4 for the detailed allocation within a slice). As illustrated in Figure 3, 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: at the first step when there are the initial 10 chunks (1 factor and 9 updates), 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 4 for the detailed allocation within a slice, together with the cost of the updates.

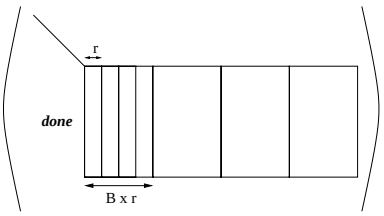


Figure 3: Allocating slices of B chunks.

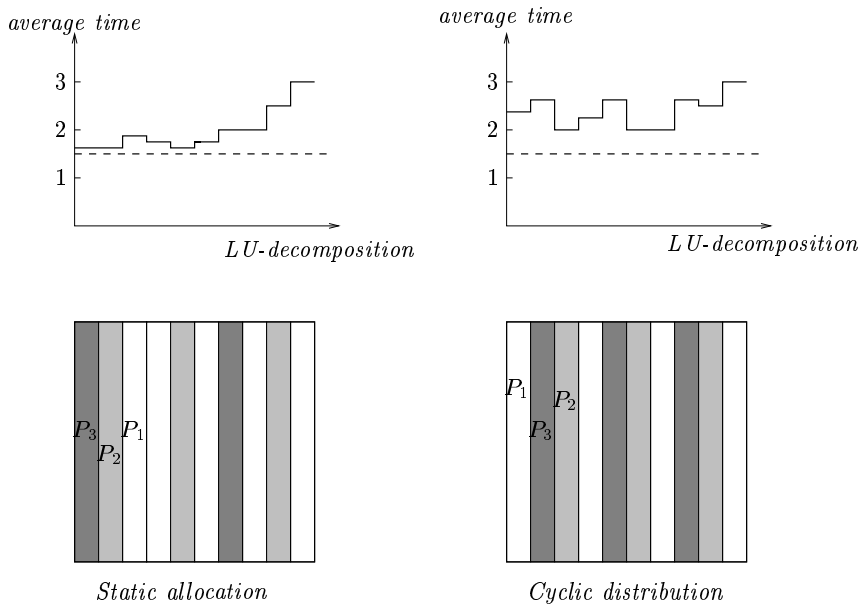


Figure 4: Comparison of two different distributions for the LU-decomposition algorithm on a heterogeneous platform made of 3 processors of relative speed 3, 5 and 8. The first distribution is the one given by our algorithm, the second one is the cyclic distribution. The total number of chunks is $B = 10$.

2.3 A proposal for a cluster ScaLAPACK

We are ready to propose a first solution for a heterogeneous cluster ScaLAPACK library devoted to dense linear solvers such as LU or QR factorizations. It turns out that all these solvers share the same computation unit, namely the processing of a block of r columns at a given step. They all exhibit the same control graph: the computation processes by steps; at each step the pivot block is processed, and then it is broadcast to update the remaining blocks (see Section 4.1 for a more precise statement of this property).

The proposed solution is fully static: at the beginning of the computation, we distribute slices of the matrix to processors in a cyclic fashion. Each slice is composed of B chunks (blocks of r columns) and is allocated according to the previous discussion. The value of B is defined by the user and can be chosen as M if $n = m \times r$, i.e. we define a single slice for the whole matrix. But we can also choose a value independent of the matrix size: we may look for a fixed value, chosen from the relative processor speeds, to ensure a good load-balancing. We come back on the usefulness of this parameter B when discussing extensions to this proposal (Section 4.5).

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 $CYCLIC(r)$ distribution for the matrix data, and we define p PVM processes. In the heterogeneous case, we still use a $CYCLIC(r)$ distribution for the data, but we define B PVM processes which we allocate to the p physical processors according to our load-balancing strategy. The experiments reported in the next section fully demonstrate that this approach is quite satisfactory in practice.

3 PVM Experiments

In this section we report several PVM experiments that fully demonstrate the usefulness of the static approach explained in Section 2.2 .

Description In this section, we report experiments on two different heterogeneous NOWs presented in Tables 4 and 5. The first network is made of 6 different workstations of the LIP laboratory, interconnected with Ethernet/IP (we call this network *lip*). The second network is made of 3 processors, interconnected with Myrinet/IP (we call this network *lhpc*). We compare the ScaLAPACK implementation of the standard purely cyclic allocation ($CYCLIC(r)$ to be precise) with a PVM implementation of our static distribution (with $B = 9$). We use two different matrix decomposition algorithms, namely LU and QR . Matrices are of size $n \times n$, and they are divided into blocks of $r = 32$ columns.

As already pointed out, these experiments have been obtained with little programming effort, because we did not modify anything in the ScaLAPACK routines. We only declared several PVM processes per machine. We did pay a high overhead and memory increase for managing these processes. This is a limitation in the use of our program as such: we chose the value $B = 9$ instead of $B = \frac{n}{r}$, which would have led to a better allocation. A refined implementation (maybe using MPI kernels) would probably yield better results.

We mentioned in Section 2.1 that processor cycle-times are to be computed via small-scale experiments (repeated several times, with an averaging of the results). Hence, for each algorithm, we have measured the computation time on different matrix sizes. The measure of processor speeds for LU decomposition on the *lhpc* network is reported in Figure 5.

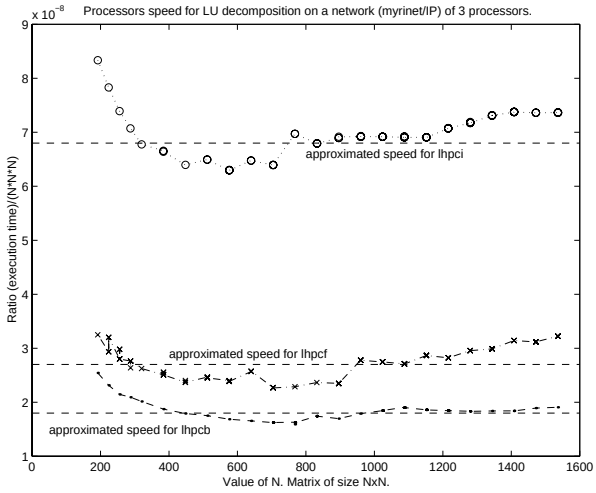


Figure 5: For each application (LU, QR), we have evaluated the different processor speeds by running these applications on different problems sizes (*lhpc* network). We take $t_{lhpci} = 353$, $t_{lhpcf} = 143$ and $t_{lhpcb} = 100$.

Now, we investigate the value of the "reasonable" speedups that should be expected. We use the example of LU decomposition on the *lip* network. Processor speeds are described in Table 4. One can say that on a heterogeneous NOW, asymptotically, the computation time for LU decomposition with cyclic distribution is imposed by the slowest processor. Hence, in the example, a *cyclic*(6)

LU on lip		Average time per column block
farot	Ultra 1	100
arnica	Sparc 5	326
smirnoff	Sparc 5	303
loop	Sparc 5	297
arquebuse	Sparc 20	161
xeres	ELC	284
B=1	farot	100
B=2	arquebuse	80
B=3	farot	66
B=4	xeres	71
B=5	loop	59
B=6	farot	50
B=7	smirnoff	43
B=8	arquebuse	40
B=9	arnica	36
B=∞	$\frac{1}{\frac{1}{100}+\frac{1}{326}+\frac{1}{303}+\frac{1}{297}+\frac{1}{161}+\frac{1}{284}}$	34
Cyclic(6)	$\frac{326}{6}$	54

Table 4: Processor speeds of these 6 workstations (*lip*) have been measured by computing the same LU decomposition on each machine separately. The speed of farot has been set to the value 100 for reference. The time for the purely cyclic distribution is obtained by noting that the computation time is imposed by the slowest processor.

LU on lhpc		Average time per column block
lhpcb	Pentium BiPro	100
lhpcf	Pentium	143
lhpci	Pentium	353
B=1	lhpcb	100
B=2	lhpcf	71
B=3	lhpcb	66
B=4	lhpcf	71
B=5	lhpcb	60
B=6	lhpci	59
B=7	lhpcb	57
B=8	lhpcf	54
B=9	lhpcb	55
B=∞	$\frac{1}{\frac{1}{100}+\frac{1}{143}+\frac{1}{353}}$	50
Cyclic(3)	$\frac{353}{3}$	118

Table 5: Processor speeds of these 3 workstations (*lhpc*) have been measured by computing the same LU decomposition on each machine separately. The speed of lhpcb has been set to the value 100 for reference. The time for the purely cyclic distribution is obtained by noting that the computation time is imposed by the slowest processor.

distribution of column blocks to processors will lead to the same execution time as if computed with 6 identical processors of speed 101. Let q be the slowest processor, i.e. suppose that $\forall i$, $t_q \geq t_i$. The best computation time that can be achieved on the heterogeneous NOW would be $t_{optimal} \approx \frac{1}{\sum_{i=1}^{p} \frac{1}{t_i}} \times \frac{p}{t_q} \times t_{cyclic(p)}$. Hence, in our example, $\frac{t_{optimal}}{t_{cyclic(6)}} \approx \frac{16.8}{10.5}$.
One also can approximate the computation time of our distribution by

$$t_{hetero(B)} \approx \frac{\max_{i=1}^{p} c_i t_i}{B} \times \frac{p}{t_q} \times t_{cyclic(p)} = t_{theoretical}.$$

Experiments

For each algorithm (LU, QR), we represent on the same graph the execution time for:

- **cyclic distribution**: it is the ScaLAPACK/PVM implementation.
- **our distribution**: 9 processes are created on different processors according to the distribution given by the dynamic programming algorithm. Then we use the ScaLAPACK/PVM implementation with a $CYCLIC(32)$ distribution onto the 9 processes.
- **optimal algorithm**: it is a theoretical computation time, proportional to the *cyclic distribution* computation time. The ratio is calculated as previously explained.

Discussion As one can see, we get a better speedup on the *lhpc* (Myrinet) network than on the *lip* (Ethernet) network. It is easily explained by the fact that the "optimal" speedup has been

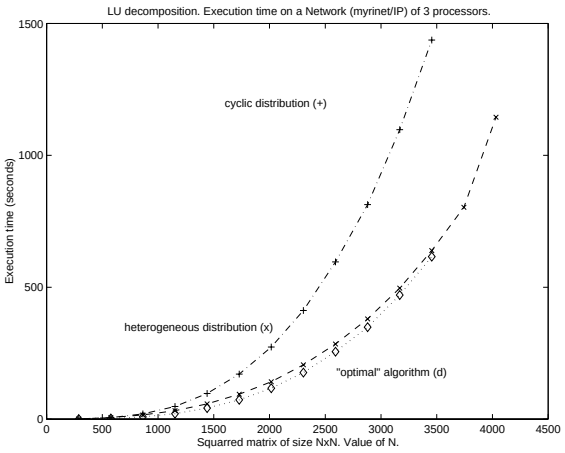


Figure 6: For LU decomposition on *lhpc*, the relative speeds of the processors have been found to be: (lhpci:1060, lhpcf:430, lhpcb:300). Hence, the distribution for hetero(9) is cyclic(lhpc-b,f,b,i,b,f,b,f,b). The ratio $t_{cyclic(3)}$ over $t_{optimal}$ is 2.3 and $\frac{t_{cyclic(3)}}{t_{theoretical}} = 2.3$.

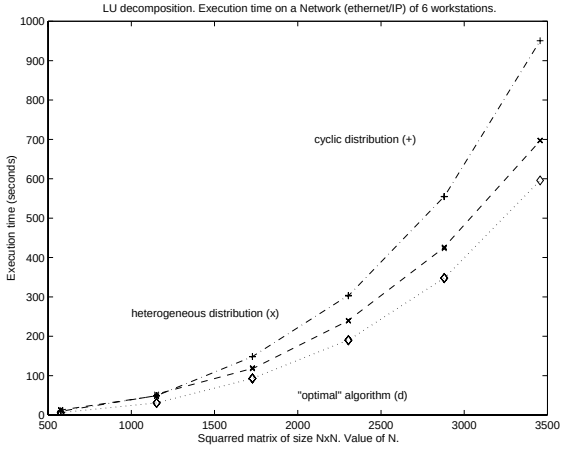


Figure 7: For LU decomposition on *lip*, the relative speeds of the processors has been found to be: (arnica:101, smirnoff: 94, xeres:90, loop:92, arquebuse:50, farot:31). Hence, the distribution for hetero(9) is cyclic(arnica, arquebuse, smirnoff, farot, loop, xeres, farot, arquebuse, farot). The ratio $t_{cyclic(6)}$ over $t_{optimal}$ is 1.6 and $\frac{t_{cyclic(6)}}{t_{theoretical}} = 1.5$.

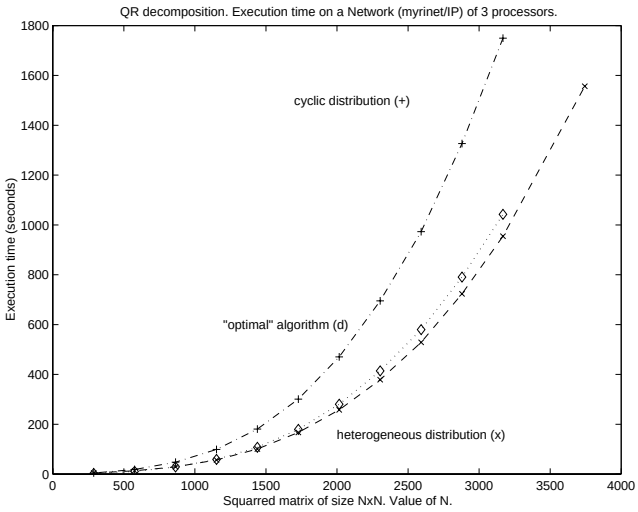


Figure 8: For QR decomposition on *lhpc*, the relative speeds of the processors has been found to be: (lhpci:160, lhpcf:121, lhpcb:59). Hence, the distribution for hetero(9) is cyclic(lhpc-b,i,b,f,b,b,i,f,b). The ratio $t_{cyclic(6)}$ over $t_{optimal}$ is 1.7 and $\frac{t_{cyclic(3)}}{t_{theoretical}} = 1.4$.

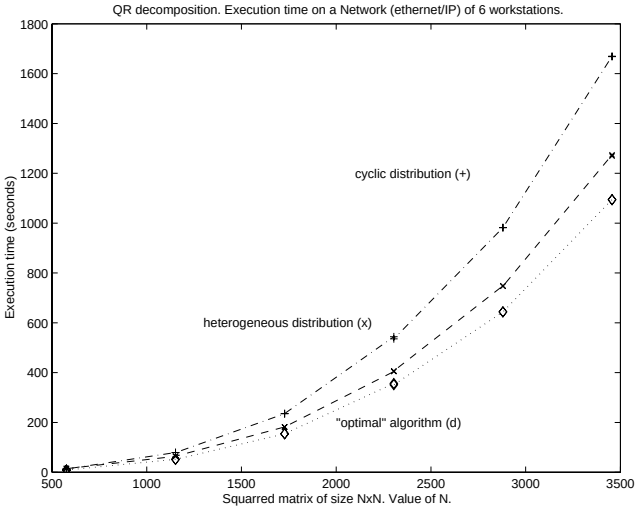


Figure 9: For QR decomposition on *lip*, the relative speeds of the processors has been found to be: (arnica:200, smirnoff:190, xeres:165, loop:161, arquebuse:107, farot:72). Hence, the distribution for hetero(9) is cyclic(farot,arquebuse,arnica,smirnoff,xeres,loop,farot,arquebuse,farot). The ratio $t_{cyclic(6)}$ over $t_{optimal}$ is 1.5 and $\frac{t_{cyclic(6)}}{t_{theoretical}} = 1.4$.

calculated without considering the communications. Hence, since in both cases, communications are not overlapped with computations, the impact of communications is much more important on *lip* than on *lhpc*. The effects of cache size, and of the pivoting computations make our algorithm faster than the theoretical approximation does predict. It means that we should take such effects into account to improve our algorithm (see Section 4.4).

4 Dynamic scheduling

Dynamic solutions are more general than static ones. In this section, we investigate several dynamic allocation strategies for implementing dense linear solvers on a heterogeneous NOW. Our goal is twofold: we aim at comparing the static and dynamic approaches, but we also have the objective to present refined strategies based upon a mixture of static and dynamic allocation.

4.1 Task graph

The task graph of blocked algorithms for dense linear solvers has been studied by several authors, see [8, 6] among others. We represent this task graph in Figure 4.1. A task represents the processing of a column block at a given step. Factor tasks (denoted as F) represent the processing of the current column block (the pivot block in LU). The new value of this block is used to update the remaining blocks (updates are denoted as U tasks). In Figure 4.1 arrows represent dependences. There are broadcast dependences between a factor task and all the following update tasks, and vertical dependences between two consecutive accesses to the same column blocks. QR decomposition obeys the same control graph as LU decomposition. The only difference resides in the task durations [5].

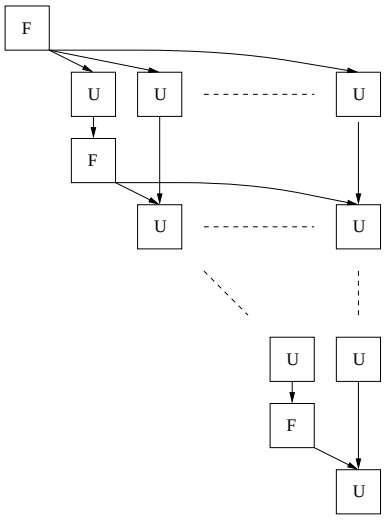


Figure 10: The task graph of LU and QR decompositions

Tasks will be allocated to processors according to a scheduling and allocation strategy. Communications will occur each time that a given task and one of its predecessors are not allocated to the same processor. Next, we define a level in a task graph as a set of tasks which are independent, i.e. there are no data dependences between the tasks of the same level. In Figure 4.1 we check that a factor task is the single element of a level, and that all the update tasks following the same factor task are in the same level.

4.2 Master-slave paradigms

A first approach is to implement a master-slave solution. Such a solution can be organized by level: the master computes the the factor task and then dispatches all update tasks of the level to the slaves. There are two different possibilities according to the memory management policy:

Centralized solution A first possibility is to allocate all column blocks in the memory of the master. The master computes the factor task and informs the slaves that there are ready tasks to compute. The slave processors pick up new update tasks just as they become idle. But for computing each task, they need two column blocks, the one that they update and the pivot, hence a lot of communication overhead. This overhead can be reduced by broadcasting the pivot, so that it is communicated only once to each processor. Still, all updates imply a communication from and to the master. Moreover, with this solution, the size of the problem that can be solved is limited by the memory of the master.

Decentralized solution The second possibility to allocate the column blocks is to distribute them cyclically (or randomly) at the beginning to balance the load. Then a column block belongs to the processor which has been the last to update it. This implies that the master must keep tracks of block owners. The algorithm is the same: the master computes the pivot and informs the slaves when update tasks are available. The pivot block is broadcast. A free processor picks up a new task, but he needs to know the owner of the block to update. Contrary to the first solution, we are not limited by the memory of the master, but we have to carefully manage the memory of all the processors, because we have to know at every time who is the owner of every block. The implementation would be very difficult. As many communications as in the first solution would take place.

4.3 Scheduling heuristic by levels

In this section we describe the heuristic of Maheswaran and Siegel [9] to map a task graph on a heterogeneous NOW. We suppose that we already have an initial mapping of the tasks on the cluster of machines. We consider a set of p machines, where m_j be the j-th machine. The tasks are denoted as s_j, $1 \leq j \leq T$. The estimated (expected) computation time of subtask s_i on machine m_j is given by $e_{i,j}$. The earliest time at which machine m_j is available is stored in $A[j]$.

First we compute the critical path of each task using a bottom-up traversal of the graph. We examine the tasks from the last level up to the first level. The method to compute the critical path is the following: consider a task s_i assigned to a machine m_x by the initial mapping. Let $iss(s_i)$ be the immediate successor set of task s_i. If s_j is an immediate successor of s_i, $c_{i,j}$ is the data transfer time for s_j to get all the relevant data items from s_i. The value of $c_{i,j}$ depends on the machines assigned to tasks s_i and s_j. On a heterogeneous NOW we can let $c_{i,j}$ be zero if the two machines are the same, and be proportional to the amount of data to be communicated otherwise. The critical path is given by:

$$cp(s_i) = e_{i,x} + \max_{s_j \in iss(s_i)} (c_{i,j} + cp(s_j))$$

Recursively, we compute the critical path for each task in the graph.

During the execution of the heuristic, we want to change the initial mapping dynamically. The remapper starts examining the level k of the graph while the task belonging to the previous level $(k-1)$ begin their execution. The remapping algorithm is the following: within a level, the tasks

are ordered on a priority based on the critical path. We examine the tasks from the highest priority to the lowest priority. We want for each task to minimize the partial completion time. Let $pct(s_i, x)$ denote the partial completion time of task s_i on machine m_x, $dr(s_i)$ be the time at which the last data item required by s_i to begin its execution arrives at m_x, and let $ips(s_i)$ be the immediate predecessor set of task s_i.

For any task s_i in level 0, $pct(s_i, x) = e_{i,x}$. For any task s_j not in level 0, where $s_j \in ips(s_i)$ and s_j is mapped onto machine m_y, we have

$$dr(s_i) = \max_{s_j \in ips(s_i)} (c_{j,i} + pct(s_j, y))$$

$$pct(s_i, x) = e_{i,x} + \max(A[x], dr(s_i))$$

The task s_i is remapped onto the machine m_x that gives the minimum $pct(s_i, x)$ and $A[x]$ is updated using $pct(s_i, x)$. Then the next task from the list is considered for remapping.

We use again the toy-example of Section 2.2: we have 3 machines of respective cycle-times 3, 5 and 8. We report in Figure 11 the output of the heuristic for LU decomposition of a matrix of size 20. For the sake of comparison, we also report the static allocation of Section 2.2, and another optimized allocation that is described below.

As for the heuristic, we point out that there are many partial redistributions of column blocks between levels, so that an actual implementation would be very difficult: we have to keep track of the owners of the column blocks, because many blocks are moved dynamically during the execution from one machine to another. Allocating and managing memory will be challenging.

To compare the performance of the heuristic with that of the static approach, we have run simulations using matrix sizes ranging from 1000 to 2000. The cycle-times of the three machines are 3, 5 and 8. The cost for a task is proportional to its number of arithmetic computations, while the cost of a communication between any two processors is proportional to the size of the message we have to send. We simply took the same value for the elemental computation and elemental communication times. We sum up all the costs needed to finish the program for every machine and we keep the maximum time of these costs as the estimated computation time. Results are reported in Figure 12.

The simulation shows that there is little benefit to expect from the dynamic approach. Its performances are quite similar to that of our static distribution. Furthermore, in an actual implementation of the heuristic, we would pay an additional price for the memory management ad the control overhead. In a word, the dynamic approach offers extra flexibility but cannot compete with the static solution, due to the numerous dependence constraints in the task graph.

4.4 Optimized distribution

We briefly discuss an optimized version of the static distribution. Contrarily to the previous solution, all factor tasks will be executed by the fastest processor. To minimize communications, we request that the fastest processor also executes each update task that is the immediate predecessor task of a factor task. To keep an optimal allocation, we have to modify our dynamic programming algorithm to compute the distribution.

Assume that the fastest processor is the first one. Instead of beginning with an allocation $\mathcal{C} = (0, 0, \cdots, 0)$ we start with $\mathcal{C} = (2, 0, \cdots, 0)$. This corresponds to allocating the first two tasks on each level to the faster processor. The rest of the algorithm is unchanged. The effect of this

Figure 11: Allocation found by the heuristic (left), by the static approach of Section 2.2, and by the optimized allocation.

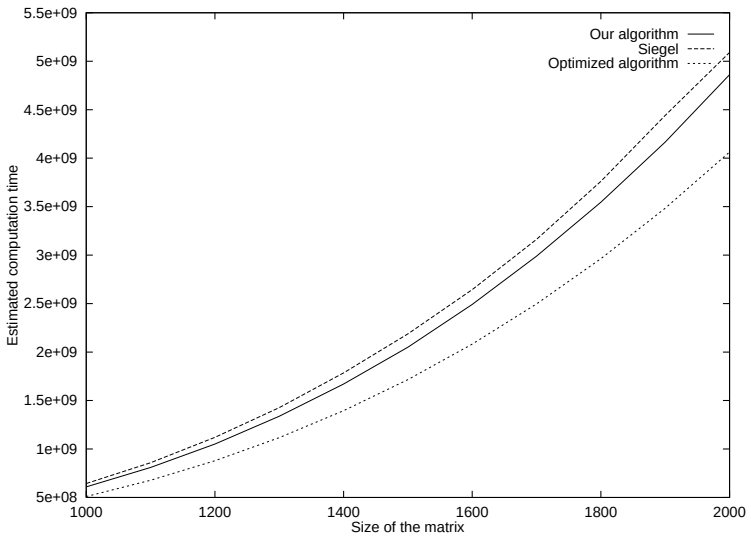


Figure 12: Comparison of the (simulated) execution times of the three schedules

strategy is to balance all the update tasks of a given level together with the factor task of the next level.

This new allocation strategy is illustrated in Figure 11. Note that the amount of communications generated by the program has been increased. In the previous version, we had to broadcast the pivot block at each level. Now, we still need a broadcast at each level, but there are additional communications. Each time that we have to compute an update and its following factor task, this computation is done by the fastest processor. Hence the the processor which originally owns the corresponding column block has to send it to the fastest processor. When the computations are done, the fastest processor sends the column block back to the original owner. So, for each level, we have two additional communications.

4.5 A semi-static strategy

Towards a "Computational Grid" ScaLAPACK When discussing the optimized algorithm, we have pointed out that this algorithm implies several redistributions of column block on the fly, as the elimination progresses. Implementing this algorithm would require a major rewriting of the ScaLAPACK library. Basically, we would need to define two basic kernels:

- one kernel aimed at factoring a given column block
- one kernel aimed at updating a given column block with the input of the factored block

The tools to write these kernels are there: these are nothing but BLAS3 operations. The tools to organize the communications between kernels are there too, namely the BLACS subroutines. What seems unavoidable is a change in the philosophy: we would access pointers to local arrays rather than addressing a shared-memory global matrix as in the current ScaLAPACK distribution. Anyway, we believe that such a major change is a *sine-qua-non* to tackle the implementation of ScaLAPACK on the computational grid [7]: it does not seem reasonable to emulate a global addressing on a collection of heterogeneous NOWs or parallel servers that are scattered all around the world. We believe that a hierarchical approach based on the ideas of our optimized algorithm will prove very efficient to solve this challenging problem.

Variations in the machine loads and speeds When discussing all the static distributions presented in this paper, we make the implicit hypothesis that the (estimated) speeds of the processors will remain the same throughout the computation. As discussed in Section 2.1, the only possible strategy to take speed variations (maybe due to variations in the machine loads) into account is to split the work into phases and to use the actual processor speeds computed in a given phase to load-balance the work for the next phase.

Translated to dense linear solvers, this implies a redistribution of the data at each phase. We would redistribute the remaining column blocks to account for a new estimation of the processor speeds. Owing to the parameter B that bounds the number of chunks that are allocated (as defined in Section 2.3), we can redistribute on the fly without any information on the problem size: only the estimations of the cycle-times t_i are needed to recompute the allocation of B consecutive column blocks. So-to-speak, we would end up with a semi-static strategy: the allocation is static within each phase but is recomputed between phases (and an inter-phase redistribution may well take place).

5 Conclusion

In this paper, we have discussed static allocation strategies to implement dense linear system solvers on heterogeneous computing platforms. Such platforms are likely to play an important role in the future. We have shown both theoretically and experimentally (through PVM experiments) that our data and computation distribution algorithms were quite satisfactory.

We also have compared the static approach with dynamic solutions, and we have show that we could reach comparable (or even better) performances, while retaining the simplicity of the implementation.

The next project would be to target a collection of heterogeneous NOWs rather than a single one. Implementing linear algebra kernels on several collections of workstations or parallel servers, scattered all around the world and connected through fast but non-dedicated links, would give rise to a "Computational Grid ScaLAPACK". Our results constitute a first step towards achieving this ambitious goal.

References

[1] 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.

[2] 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.

[3] P. Boulet, J. Dongarra, F. Rastello, Y. Robert, and F. 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`.

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

[5] J. Demmel, J. Dongarra, R. van de Geijn, and D. Walker. ScaLPACK: Linear algebra software for distributed memory architectures. In T. L. Casavant, P. Tvrdik, and F. Plasil, editors, *Parallel Computers: Theory and Practice*, pages 267–282. IEEE Computer Society Press, 1996.

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

[7] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1998.

[8] Apostolos Gerasoulis and Tao Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Trans. Parallel and Distributed Systems*, 4(6):686–701, 1993.

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