Linear Algebra Algorithms in a Heterogeneous Cluster of Personal Computers

J. Barbosa*, J. Tavares† and A.J. Padilha
FEUP-INEB
Grupo de Arquitecturas e Sistemas
Praça Coronel Pacheco, 1, 4050 Porto (P)
{jbarbosa,tavares,padilha}@fe.up.pt

Abstract

Cluster computing is presently a major research area, mostly for high performance computing. The work herein presented refers to the application of cluster computing in a small scale where a virtual machine is composed by a small number of off-the-shelf personal computers connected by a low cost network. A methodology to determine the optimal number of processors to be used in a computation is presented as well as the speedup results obtained for the matrix-matrix multiplication and for the symmetric QR algorithm for eigenvector computation which are significant building blocks for applications in the target image processing and analysis domain. The load balancing strategy is also addressed.

1. Introduction

Several personal computer or workstation based cluster systems have been developed, from commercial off-the-shelf processors to high performance ones such as SMP architectures [3] and using high performance networks like Myrinet [2, 19]. Most of the work is devoted to the high performance computing aiming to achieve the performance of a specific supercomputer at a lower cost.

Our aim is not to build a cluster of personal computers for parallel processing but to do parallel processing on already existing group clusters, where each node is a desktop computer running the Windows operating system. These clusters are characterized by having a low cost network, such as a 10 Mbits/s Ethernet, connecting different types of processors, of variable processing capacity and amount of memory, thus forming a heterogeneous parallel virtual computer. Due to network restrictions, which do not allow simultaneous communication among several nodes, the application domain is restricted to one or two dozens of processors.

The need for a methodology to determine the ideal number of processors comes also due to network restrictions, since as the number of processors increases the network acts as a communication bottleneck and the time spent in data exchange can overcome the benefits of more processing power. This is not usually referred in the high performance clusters literature, due to the usually huge problem size, however, in [17] a scheduling policy is studied for multiprocessor systems based on that some applications cannot exploit the computational power available, due to hardware and software constraints. In [4] a performance model for heterogeneous processing was proposed but not in the context of processor co-operation to solve a task.

Our motivation for a parallel implementation of linear algebra algorithms comes from image and image sequence analysis needs, posed by various application domains, which are becoming increasingly more demanding in terms of the detail and variety of the expected analytic results, requiring the use of more sophisticated image and object models (e.g., physically-based deformable models), and of more complex algorithms, while the timing constraints are kept very stringent.

A promising approach to deal with the above requirements consists in developing parallel software to be executed, in a distributed manner, by the machines available in an existing computer network, taking advantage of the well-known fact that many of the computers are often idle for long periods of time. It is quite common in many organizations that a standard network connects several general purpose workstations and personal computers, accumulating a very substantial computing power that, through the use of appropriate managing software, could be put at the service of the more computationally demanding applications.

Existing software, such as the Windows Parallel Virtual Machine (WPVM) [1], allows building parallel virtual computers by integrating in a common processing environment a set of distinct machines (nodes) connected to the network.

* PhD grant BD/2850/94 from PRAXIS XXI
† PhD grant BD/3243/94 from PRAXIS XXI

Although the parallel virtual computer nodes and the underlying communication network were not designed for optimized parallel operation, very significant performance gains can be attained if the parallel application software is conceived for that specific environment.

This paper addresses the problem [22] of determining, from a pool of available nodes, which ones should be selected for building a parallel virtual computer that achieves the fastest application response time, and it also discusses the issue of computational load distribution; the study considers that the nodes available prior to running the application may differ from time to time, as different users and machines are active. At every program initiation phase, the highest performance computers from the available set are selected, in a number that is computed for optimizing the processing time.

The test cases presented, a parallel matrix multiplication algorithm and the QR algorithm, while pertinent to many advanced image analysis methods, are also a common module in many other fields, such as in simulation problems. In a previously reported work [5], the step edge operator proposed by Shen and Castan [20] was also tested.

2. Computational model

Several computational models [23, 7, 14] were presented in order to estimate the processing time of a parallel program. Although they could be adapted for the cluster of personal computers, a specific and simplified model is presented below. The target machine is composed by nodes with different processing capacities, resulting from different amounts of available memory and from various processor types and versions, connected by a standard interconnection network, such as the Ethernet. Each node of the machine is characterized by the processor capacity S, measured in Mflops. The network is characterized by the number of messages that are allowed simultaneously, the bandwidth LB measured in Mbits/s, and by the existence or not of broadcasting capacity.

The computational model for the virtual machine, describing the behavior for a given algorithm, is obtained by summing the time spent in sequential operations T_S and the time spent in parallel operations T_P. Sequential operations include communications, data input/output and other processing that cannot occur in parallel due to each particular algorithm characteristics. Parallel operations are those that the time spent by one processor can be divided by p if p processors are used. The total processing time, as a function of the number of processors p and the problem size n is given by equation 1.

$$T_T(n,p) = T_S(n,p) + T_P(n,p) \tag{1}$$

The interconnection network is modeled by a temporal expression, T_C, representing the time required to transmit a message of nb bits between two network nodes, assuming a distance 1 network.

$$T_C = T_L + nb(\frac{1}{LB} + T_E) \tag{2}$$

The latency time T_L represents the time gap between the processor order to transmit and the beginning of transmission and T_E the packing time. The logical topology of an Ethernet provides a single channel, or bus, that carries Ethernet signals to all stations, allowing broadcast communications. There is only one signal channel delivering packets over the network to all stations. Each message is divided into packets of length 46 to 1500 bytes of data ($packetsize$), to be sent sequentially and individually onto the shared channel. For each packet the computer has to gain access to the channel [21]. This division of a message into packets leads to a latency time for each message that is proportional to the number of packets (K) into which it is split, resulting equation 3.

$$T_{Comm} = KT_L + nb(\frac{1}{LB} + T_E) \tag{3}$$

The value of K is given by equation 4. A typical value for $packetsize$ is 1024 bytes.

$$K = \lceil \frac{nb/8}{packetsize} \rceil \tag{4}$$

For a heterogeneous virtual machine T_L and T_E depend on the processor speed S. Several experiments were conducted in order to measure these parameters, for the network referred to in the results section, which is composed by processors as illustrated in table 1. The values were measured for the matrix multiplication algorithm over different matrix sizes, resulting the average values of table 1.

$S(Mflops)$	244	161	60	50	49
$T_L(\mu s/byte)$	70	130	180	180	180
$T_E(\mu s/byte)$	0.05	0.07	0.13	0.13	0.13

Table 1. Processors parameters

Although the Ethernet physically allows broadcasting the WPVM converts a broadcast in a p processor machine to $p-1$ messages [1]. Therefore, to model correctly a broadcast the time spent in one message has to be multiplied by $p-1$.

Independent communications over rows or columns, either for 1-D or 2-D grids, can originate network collisions. Examples are all slave processes trying to send results to the master process at the same time; or for the matrices multiplication algorithm, in each step, the distribution of the matrices are independent over rows, for one matrix, and

over columns for the other matrix. To avoid collisions a set of communication routines using a ring communication pattern, as shown in figure 1, were developed. They allow processes to synchronize by establishing the order of communications according to the processes position on the grid.

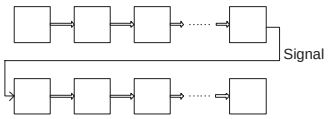

Figure 1. Communication pattern for two independent row broadcasts

The parallel component T_P of the computational model, equation 5, represents the operations that can be divided over a set of p processors obtaining a speedup of p, i.e. operations without any sequential part.

$$T_P(n,p) = \frac{\psi(n)}{\sum_{i=1}^{p} S_i} \tag{5}$$

The numerator $\psi(n)$ is the cost function of the algorithm measured in floating point operations $(flops)$ as a function of the problem size n. For example, to multiply square matrices of size n, the cost is $\psi(n) = 2n^3$ [8]. This operation count does not include memory operations resulting, therefore, a higher complexity. To obtain a correct operation count one should consider the memory references made and have an estimation of the memory access time. The nodes of the virtual machine have different levels of memory (cache, main memory, and disk) with different access times, and one cannot predict how many accesses are made to each one.

Figure 2 shows the processing capacity achieved by an 161 Mflop peak performance processor for the matrix multiplication algorithm. The computational cost is $\psi(n) = 21.5n^3 flops$. Figure 2 also shows that a non block oriented algorithm cannot assure a constant coefficient of $\psi(n)$, which is a requirement in order to be able to estimate the time the processors will take to execute the algorithm. From this point on the coefficient of $\psi(n)$ will be referred to as the algorithm constant β. The value β does not depend on the processor but rather is a characteristic of the algorithm.

The denominator of equation 5 is the processing capacity used which is obtained by summing the individual processing capacities of the machines. For this equation to be valid each machine should not take more than T_P seconds to process its part. This assumes a perfect load balancing in the heterogeneous machine.

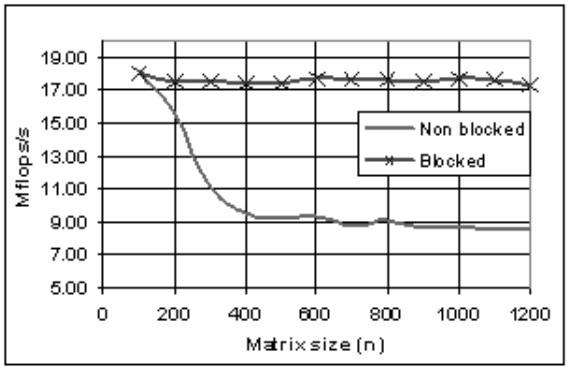

Figure 2. Performance of the matrix multiplication algorithm on a 161 Mflop peak performance processor as a function of matrix size

3. Load balancing strategy

In this section a static load distribution algorithm is presented and issues related to the optimization of processing time in a heterogeneous environment are discussed.

3.1. Data distribution

To avoid the slowest processors to determine the parallel processing time, the load should be distributed proportionally to the capacity of each processor. The aim is to assign the same amount of processing time which may not correspond to the same amount of data.

The matrices are organized in square blocks of data which are assigned to the processor grid. To achieve a balanced distribution in the heterogeneous machine the number of blocks assigned to each processor should be proportional to its processing capacity compared to the entire machine:

$$l_i = \frac{S_i}{\sum_{k=1}^{p} S_k} \tag{6}$$

The load index l_i although theoretically correct, is not fully applicable in practice since the number of blocks assigned has to be an integer value. As an example, for a machine composed by 6 processors of capacity $\{244, 244, 161, 161, 60, 50\}$ Mflops, l_i would be $\{0.265, 0.265, 0.175, 0.175, 0.065, 0.054\}$. To distribute a matrix of size 1800 over a (1,6) processor grid the assignment would be 1800 rows by $\{477, 477, 315, 315, 118, 98\}$ columns respectively.

The strategy implemented is to compute the number of blocks to assign to each processor rounding the real value obtained down to the nearest integer, so that some blocks are left to be assigned. Then, to obtain an optimal solution the remaining blocks are assigned one at a time to the grid

of processors, choosing the one that will take less time to finish the job.

For the test case presented, consider a block size of 25 elements, which lead to an assignment in terms of number of blocks, of {19, 19, 12, 12, 4, 3} summing 69 in a total of 72 blocks, leaving 3 blocks unassigned. Using the time complexity analysis presented in the next section for the matrix multiplication algorithm, $T_P = 21.5n^3/S$, the estimated computational time per processor is {135.6, 135.6, 129.8, 129.8, 116.1, 104.5} seconds. Each block will take {7.14, 7.14, 10.82, 10.82, 29.03, 34.83} seconds in each processor respectively. This block processing time is summed to the total time each remaining block being assigned to the processor that would finish first. The first block is assigned to processor 6 and the last two blocks to processors 3 and 4, resulting an estimated processing time of {135.6, 135.6, 140.6, 140.6, 116.1, 139.3} seconds. A perfect load balancing cannot be achieved, however for this block size it is the optimal assignment, i.e. the assignment that leads to the minimum processing time. Figure 9 shows the processing time measured for each processor.

Another issue in data distribution for a heterogeneous machine is to keep the load balance in the whole algorithm. For some algorithms, such as tridiagonal reduction and LU factorization, in each iteration part of the matrix is fully computed and not visited again, the working matrix being smaller from step to step. This can lead to an imbalance load if the distribution is not cyclic. For the example above, if contiguous blocks are assigned to each processor, one of the fastest processors would be idle after computing 19 blocks of the matrix , remaining 53 blocks to be processed.

To overcome this load imbalance, blocks are organized in balanced groups. Being l_i the load index of processor i, one define group block G_B as:

$$G_B = \frac{1}{min(l_i)} \tag{7}$$

If $G_B/Q < 2$ then $G_B = 2/min(l_i)$, where Q is the number of column processors. For a (P, Q) grid the algorithm is applied to columns and rows independently, considering the processing capacity by column and row respectively, as shown in table 2.

For the example given above $G_B = 1/0.054 = 18$ blocks, giving a group block of {5, 5, 3, 3, 1, 1}.

With this strategy it is guaranteed that from the beginning to the end of the algorithm all processors are involved in proportion of their load indices l_i, allowing an effective load balancing. When the last group block is being processed, the last 8 blocks would be computed by the slowest processors; it is reasonable that in cases where some processors cannot participate due to the lack of data, it should be the fastest ones doing the computation. Therefore, the cyclic distribution is used inside each group block.

3.2. Data redistribution

In order to exploit the computational capacity of the target machine, the algorithms must be implemented in order to increase the computation to communication ratio, mainly due to the slow network. Therefore, data redistribution is allowed in order to switch to the optimal grid computed for each algorithm. Data distribution is represented by system independent objects, allowing the system to switch between two unrelated processor grids.

The cost of redistribution is estimated by the communication of n^2 elements for a matrix of size n, which is the worst case, i.e. every element being allocated to a different processor. The redistribution algorithm starts from the first processor (1,1) to the last, changing data synchronously with the remaining processors.

For related grids, e.g. switching from (1,6) to (1,7), the system evaluates if the gain in time due to the addition of one processor is overcome by the data redistribution time. In that case the grid change does not occur.

3.3. Block size

The block size should be chosen according to the following conditions: first, it should maximize the individual processing capacity, that as shown in figure 2 degrades for a block size 1, and second, to allow the implementation of a load balancing distribution. For the machines tested a block size in the range 15 to 40 ensure an almost constant processing capacity.

For a sequence of parallel algorithms, e.g. for eigenvector computation where different grids are used, the block size should satisfy all grids in terms of load balancing since, although there is data redistribution, this parameter remains unchanged.

3.4. Processor selection policy

The system keeps a record of the computers enrolled in the parallel virtual machine ordered by decreasing computational capacity. If only part of the machine is needed to execute the algorithm the computers are selected from the fastest to the slowest one.

A computer is considered available for parallel processing if there is no user activity for at least half the processing time of the last parallel algorithm. If a user starts using his/her computer during a parallel execution, the system does not transfer the work to another computer; it completes the current job and then marks the computer as unavailable. For the problem size addressed, whose processing time is expected to be of a few minutes, this policy is satisfactory.

4. Optimization of the processing time

A parallel algorithm may have two aims: to obtain a better accuracy of results by using a more detailed domain which could not be possible in a single processor, usually due to memory limitations, or, for a given accuracy, to obtain a reduction in the processing time. The time gain obtained with the parallel algorithm is usually called *Speedup* and is defined as the quotient of the serial algorithm time (T_1) over the parallel algorithm time (T_T).

$$Speedup = \frac{T_1}{T_T} \tag{8}$$

Depending on how the serial processing time is measured one can have different definitions of *Speedup*. *Relative Speedup* is obtained if the serial time is the processing time of the parallel algorithm in a single node of the parallel computer. *Real Speedup* is obtained if the serial time is the processing time of the most efficient sequential algorithm in a single node of the parallel computer. *Absolute Speedup* is defined when the serial time is obtained for the fastest sequential algorithm executed in the fastest sequential computer available [18]. In the context of the envisaged applications of the parallel virtual machine, we define *Speedup* as the ratio between the processing time of the serial version in the computer that controls the parallel execution (master), over the processing time of the parallel program. This is the effective gain as seen by the user, who has a choice between his/her own single machine (master) or the parallel virtual machine; the definition is also globally fair when the master computer is one of the fastest available, which is the case in the test cases presented below. In a parallel virtual machine it is quite common that each node of the computer network is not fully available for the user that is running a parallel application. The application should not schedule work for nodes that are in use by other users, and therefore it should have a record of the ones that are free. The aim in scheduling work for distributed processing is to obtain a processing time that is as small as it can be obtained for that particular network, even if some of the nodes are left in the idle state. Therefore, the relevant parameter to be considered is the Speedup in detriment of the Efficiency, which is often used in other contexts.

Given the above definitions, one can state the goal of the work herein reported as the determination of the optimum number of processors using a criterion of minimum processing time. The optimal number of processors p, which minimizes $T_T(n,p)$, is the one for which an increase on the serial component, due to the addition of one more processor, will be balanced by the gain obtained on the processing time of the parallel component.

4.1. Application to a homogeneous machine

For a given algorithm, characterized by the constant β, and for size n matrices, p can be obtained by solving equation 9 in order to p [5].

$$\frac{\partial T_T}{\partial p} = 0 \tag{9}$$

For a homogeneous machine equation 5 simplifies to $T_P(n,p) = \frac{\psi(n)}{pS}$ and the communication parameters T_L and T_E assume the same value for all machines, allowing a straightforward solution.

4.2. Application to a heterogeneous machine

For a heterogeneous machine another degree of complexity is added to equation 9: first, processors have different computational capacities (S) and second, the communication parameters T_L and T_E also vary with S, as shown in table 1.

To tackle this problem one first orders the nodes by decreasing value of S_i (the capacity of node i), and then schedules the work from the fastest to the slowest free node, resulting the denominator of equation 5: $S_T(p) = \sum_{i=1}^{p} S_i$. To compute the first derivative of T_T in order to p it is required to find the sum $S_T(p)$, which cannot be computed beforehand since one does not know how many processors will be used. The function $S_T(p)$ increases monotonically with p, having a growth rate that decreases with increasing p, as shown in figure 3 for a machine composed by processors of capacities {244, 244, 161, 161, 60, 50, 49} Mflops in decreasing order.

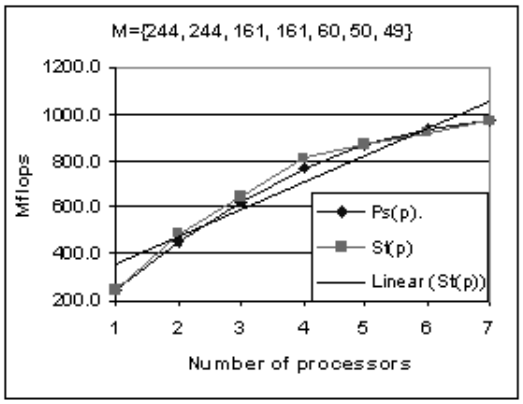

Figure 3. Processing capacity of the heterogeneous machine as a function of the processors used

The aim is to approximate $S_T(p)$ by a polynomial function in p in order to be able to solve equation 9. A first

order polynomial function as used for a homogeneous machine is not adequate here. The ideal polynomial function would be one that passes in all points of $S_T(p)$; however, its computation time may be significant for a large number of processors. The solution adopted was an iterative quadratic approximation. The first function is defined by zero and the extreme points of $S_T(p)$. The iterative process allows the reevaluation of the cost function $T_T(n,p)$ in the neighborhood of the solution computed. In each iteration only half of the processors used in the last iteration are considered being the polynomial function defined by: if P is the total number of processors, $p^{(i-1)}$ the solution for iteration $(i-1)$ then in iteration i the function is defined by the three points of equation 10.

$$S_T(p^{i-1} \pm P/2^{i+1}) \quad and \quad S_T(p^{(i-1)}) \quad i = 1, 2, ... \tag{10}$$

The second degree polynomial function has the same behavior as $S_T(p)$ and is written as:

$$P_S(p) = ap^2 + bp + d \tag{11}$$

resulting the first derivative of $T_P(n,p)$ in order to p in:

$$\frac{\partial T_P(n,p)}{\partial p} = \frac{\partial}{\partial p}\left(\frac{\psi(n)}{ap^2 + bp + d}\right) = 0 \tag{12}$$

which must be solved in order to obtain the number of processors p that minimizes the total processing time.

If the logical grid of processors affects the processing time, then changing to a 2D grid (e.g. (r,c) grid) or 3D (e.g. hypercube), one or two dimensions are added to the problem respectively. For the 2D grid the quadratic approximation with $p = rc$ becomes:

$$P_S(r,c) = a(rc)^2 + b(rc) + d \tag{13}$$

The communication parameters T_L and T_E also need to be modeled by a function of p in order to solve $\partial T_S(n,p)/\partial p$. To transmit a message from computer A to B the latency and packing time depend on the speed of processor A. If one can predict the amount of data each processor will be responsible to transmit, one can estimate the time spent in communications by the whole machine. According to the data distribution algorithm to each processor is allocated an amount of data proportional to its relative speed in the heterogeneous machine: $l_i = S_i / \sum_{k=1}^{p} S_k$. Therefore, functions to model these parameters are defined by equations 14 and 15, corresponding to an weighted mean of these values for each possibility of p processors. The values of $(T_L)_i$ and $(T_E)_i$ are shown in table 1.

$$T_{TL}(p) = \sum_{i=1}^{p} l_i \times (T_L)_i \tag{14}$$

$$T_{TE}(p) = \sum_{i=1}^{p} l_i \times (T_E)_i \tag{15}$$

For the machine considered (figure 3), the functions $T_{TL}(p)$ and $T_{TE}(p)$ are shown in figures 4 and 5 respectively. In those figures it is also shown a first degree polynomial approximation to be included in $T_S(n,r,c)$.

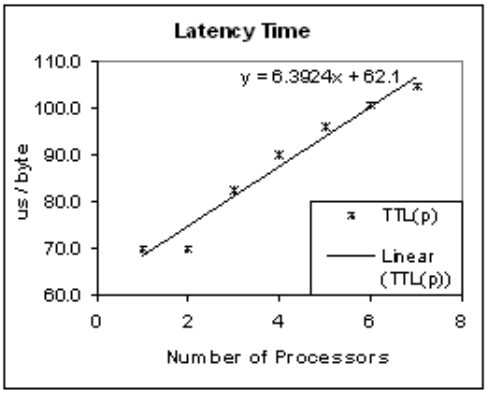

Figure 4. Approximation for $T_{TL}(p)$ per byte

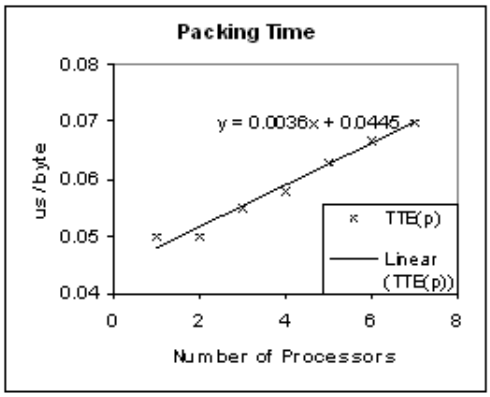

Figure 5. Approximation for $T_{TE}(p)$ per byte

The (r,c) configuration that minimizes the processing time is obtained by $\nabla T_T(n,r,c) = 0$. Since one wants to compute the ideal grid (r,c) for a given problem size n, the first derivative of $T_T(n,r,c)$ in order to n is zero. Thus, the optimal configuration is obtained by solving the system of equations 16.

$$\begin{cases} \frac{\partial T_T(n,r,c)}{\partial r} = 0 \\ \frac{\partial T_T(n,r,c)}{\partial c} = 0 \end{cases} \tag{16}$$

4.3. Applying the methodology to a matrix multiplication algorithm

The methodology presented above will be tested with an improved implementation of the matrix multiplication operations [11]. Figure 6 shows an hypothetical data assignment for a $(2,3)$ processor grid. For simplicity, the blocks displayed are formed by contiguous data, although the block cyclic data distribution is used [10].

To compute the matrix product $C = A \times B$, in each iteration of the algorithm each processor multiplies one column block of A by one row block of B, updating the correspondent block of C. The shadowed area in matrix C represents the block that processor $(0,0)$ has to update in each iteration.

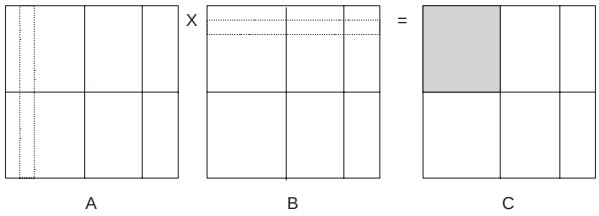

Figure 6. Matrix multiplication operations

Considering a grid (r,c) of processors, the matrices $A = (m,k)$, $B = (k,l)$ and $C = (m,l)$ the amount of data required to broadcast matrix A over the rows of processors is:

$$\frac{m}{r}\frac{k}{c}(c-1)rc = mk(c-1) \tag{17}$$

Note that $(c-1)$ appears because the broadcast is in fact performed by sequential communications. To broadcast matrix B over the column of processors it is required to transmit:

$$\frac{k}{r}\frac{l}{c}(r-1)cr = kl(r-1) \tag{18}$$

The time required to compute the inner loop products is given by:

$$T_P = \beta\frac{mlk}{S_T(r,c)} \tag{19}$$

where $S_T(r,c)$ is the processing capacity of the heterogeneous machine when rc processors are used. The value β for the matrix multiplication is 21.5, as given in section 2. The total estimated processing time, assuming square matrices of size n, is expressed as:

$$\begin{aligned} T_T(n,r,c) &= (\frac{n^2(r+c-2)}{packetsize})T_{TL}(r,c) \\ &\quad +(n^2(r+c-2))(LB^{-1}+T_{TE}(r,c)) \\ &\quad +\beta\frac{n^3}{P_S(r,c)} \end{aligned} \tag{20}$$

Depending on the data types used (float or double) the correspondent communication factors have to represent the amount of data in bytes. LB is the bandwidth per byte.

For the machine of figure 3 the quadratic approximation, equation 11, becomes $P_S(r,c) = -17.595(rc)^2 + 261.6rc$. This approximation is close to the real curve $S_T(r,c)$ for values $0 \leq rc \leq 7$. Outside this domain the polynomial function may introduce false minima in the processing time function. Therefore, the minimization must be restricted to the allowed domain by the number of processors available. This can be accomplished by introducing the Lagrange multipliers [15] in the system of equations 16. An additional function to restrict the domain is included:

$$\begin{cases} \frac{\partial T_S(n,r,c)}{\partial r} + \frac{\partial T_P(n,r,c)}{\partial r} = -\lambda c \\ \frac{\partial T_S(n,r,c)}{\partial c} + \frac{\partial T_P(n,r,c)}{\partial c} = -\lambda r \\ \lambda(rc-7) = 0 \end{cases} \tag{21}$$

The following figures, 7 and 8, present results for matrices of size 1800. Figure 7 displays the communication estimated (Est.) and measured (Meas.) time for one and two rows of processors, limited to 7 processors. And figure 8 displays the total processing time $T_T(n,r,c)$ obtained by estimation with the quadratic approximation for machine processing capacity (Tot. E), by estimation using the exact processing capacity (Tot. R), and the measured time (Tot. M). The communication times are modeled correctly, existing only a slight difference for some grids. The total estimated processing time differs from the measured one due to the quadratic approximation which underestimates the processing capacity in some cases and overestimates in others, although the behavior is similar to the measured curve and it does not introduce false minima in the processing time function. The curve obtained with the real processing capacity of the heterogeneous machine shows that the overall model is correct and that the processing time can be accurately estimated.

Solving the system of equations 21, the values of $r = c = 2.65$ are obtained for $n = 1800$ and $LB = 100Mbits/s$. Since one wants an integer solution, it can be assumed $c = 3$ which implies $r = 2$, since $rc \leq 7$. The grid (3,2) would be equivalent. Figure 8 shows that the minimum is obtained for grid (2,3), confirming the system solution, although there is an increase in the processing time compared to the estimation. This is the consequence of an imbalance grid which cannot be overcome for that machine. Table 2 shows the processor layout for grid (2,3). The first two columns of processors are equilibrated what does not happen for column 3, in which either processor (1,3) will

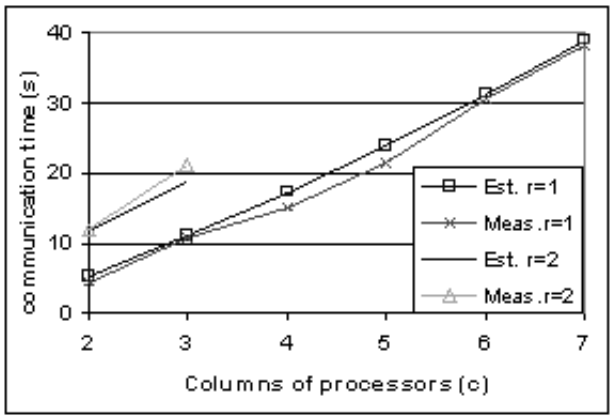

Figure 7. Communications for the matrix multiplication algorithm (matrix size 1800)

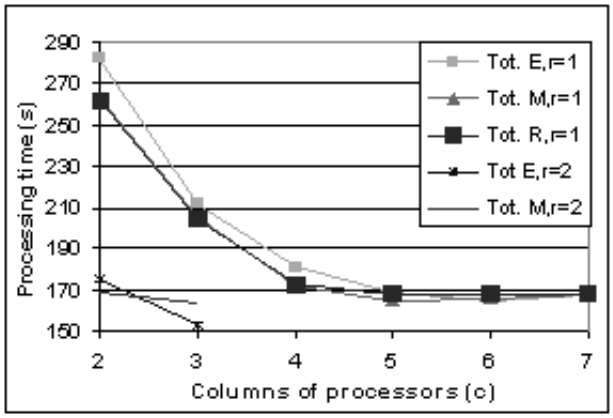

Figure 8. Processing time for the matrix multiplication algorithm (matrix size 1800)

be underloaded or processor (2,3) will be overloaded, delaying all other processors as they will be always waiting to communicate.

Figure 9 shows the processing time for all processors, where it can be seen that processor 6 is delaying the process for grid (2,3). Grid (1,6) is better balanced but the ideal load balance is not achieved due to the data blocks indivisibility. For this network, due to processor relation in processing speed, a balanced load can only be achieved with small blocks of data. The squared block size used was 25. A smaller block size, e.g. 10, while improving the load balance, would decrease the individual performance of processors due to a sub-utilization of the processors cache memory.

Note that although grid (2,3) is less balanced and there is one processor that takes more time, it makes a better solution than grid (1,6) due to the fact that this grid requires more communication time, as it can be seen in figure 7.

244	161	60	=465
244	161	50	=455
=488	=322	=110	

Table 2. Processor layout for grid (2,3)

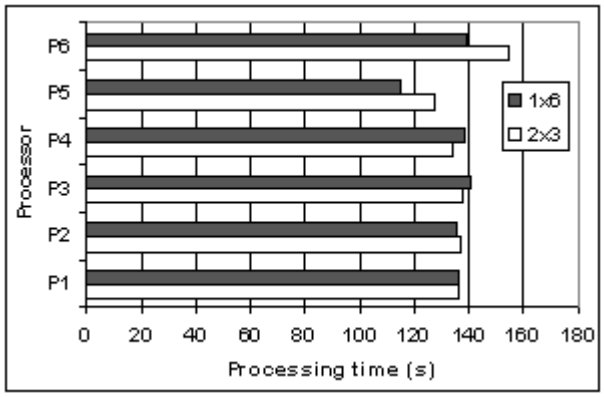

Figure 9. Matrix multiplication processing time

5. Results

In this section results for tridiagonal reduction (TRD), LU and QR factorization algorithms in the heterogeneous machine represented in figure 3 for an Ethernet network at 100 Mbits/s are presented. Figure 10 shows the performance of each algorithm in a single processor. The QR performance is divided by 2 for displaying purposes. As shown before for the matrix multiplication algorithm, the processor performance is kept almost constant for the block versions of these algorithms, for matrices greater than 400 elements. The correspondent β value considered for each algorithm is the average in that domain. The square block size used in all cases varies from 15 to 40. There is some variation in the processor performance for a given matrix size, mainly due to the operating system (Windows NT) which stochastically has some activity; however, this represents a variation in the processing time below 1%.

The estimated values presented below are obtained by applying the system of equations 21 using the time function of each algorithm respectively.

5.1. LU factorization algorithm

The LU factorization algorithm is applied in order to solve directly a system of equations. The implementation

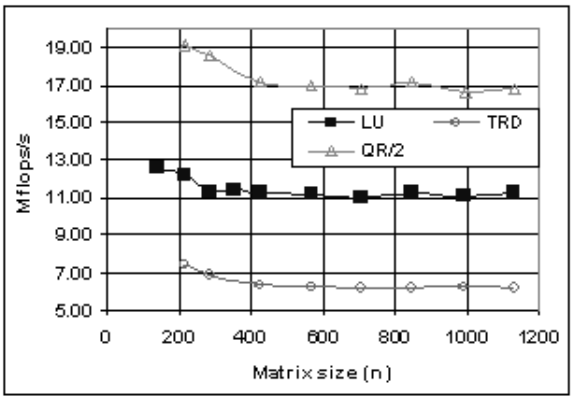

Figure 10. Performance of LU, QR and TRD block algorithms on a 161 Mflop peak performance processor as a function of matrix size

is the right-looking variant where algorithm details can be found in [9]. For a (r, c) grid of processors, the amount of data (double/float) transmitted in the parallel matrix update is:

$$(r + c - 2)\frac{n^2}{2} \tag{22}$$

and the parallel processing time is:

$$T_P(n, r, c) = \beta\frac{n^3}{S_T(r, c)} + \Theta(n^2) \tag{23}$$

The β for LU is 7.5. There is a component of complexity n^2 correspondent to the computations made by the pivot processor. Figure 11 shows the processing time estimated and measured for a matrix of size 1800. Although it is hardly perceptible in the figure, the optimum value estimated for (r, c) is (1,5). In practice the optimum is grid (1,4), which outperforms grid (1,5) by only 0.5 seconds. In this case the difference is due to the quadratic approximation for machine processing capacity. If the real values are used the estimated optimum is (1,4).

Figure 12 shows the estimated (E) and measured (M) communication times for matrices of size 1200 and 1800. In general the communications are well modeled. The differences observed are less than 3 seconds. This can lead to a grid selection that is not the optimal one; however, since the processing times obtained for grids (1,4), (1,5) and (1,6) are 69.1, 69.6 and 70.0, the main drawback would be to have unnecessary processors allocated.

Figure 13 shows the load distribution for the matrix of size 1800. For up to 5 processors a good load balancing is achieved, with processors taking almost the same time to process the data allocated to them. The block size from processor (1,1) to (1,5) is 1800 rows by 500, 500, 340, 340

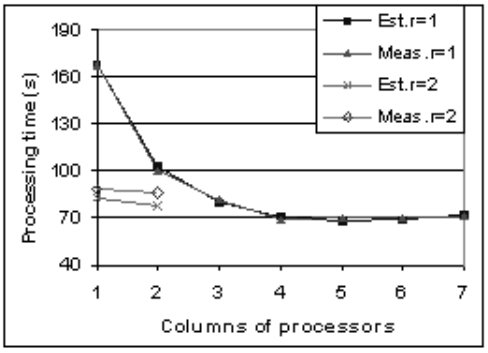

Figure 11. LU processing time for a matrix of size 1800

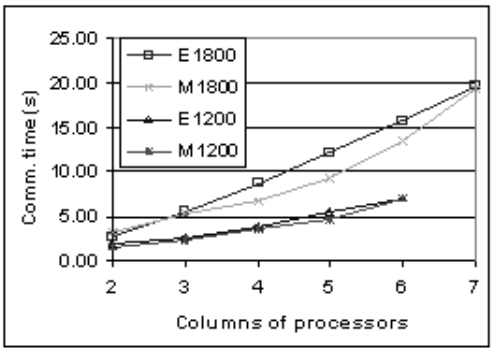

Figure 12. Estimated (E) and measured (M) communications for LU algorithm

and 120 columns respectively. Ideally they should receive 504, 504, 333, 333 and 124 columns.

5.2. Tridiagonal reduction algorithm

The tridiagonal reduction algorithm (TRD) is a step in the computation of the eigenvalues and eigenvectors of a symmetric matrix. Details of the algorithm can be found in [6]. For a (r, c) grid of processors, the amount of data to transmit is:

$$2n(r - 1) + 4n^2(rc - 1) \tag{24}$$

for computation and broadcast of Householder vectors, parallel matrix update and matrix vector products. The parallel processing time is:

$$T_P(n, r, c) = \beta\frac{n^3}{S_T(r, c)} + \Theta(n^2) \tag{25}$$

The β for TRD is 28 and there is also a negligible term in n^2. Figure 14 shows the processing time for a matrix

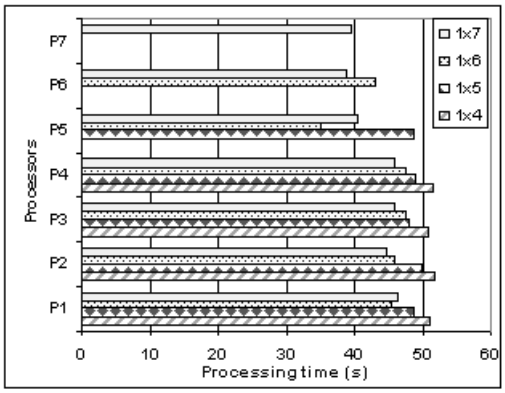

Figure 13. LU load distribution for a matrix of size 1800

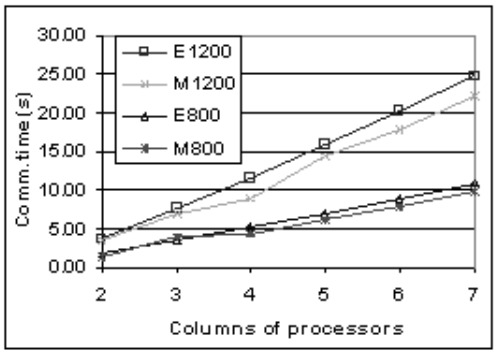

Figure 15. Estimated (E) and measured (M) communications for Tridiagonal reduction algorithm

of size 1200. For grids (1,1) to (1,4) the estimated time is higher than the measured one; the maximum error occurs for grid (1,4) which coincides with the maximum error in the quadratic approximation of computational capacity. The minimum is correctly determined as grid (1,4). Again if grid (1,3) was chosen the total time would be marginally higher: 104.7 s instead of 100.0 s. To guarantee the selection of the best grid the scheduler can operate with real values of processing capacity for estimating the processing time in the neighborhood of the solution obtained by the system of equations 21.

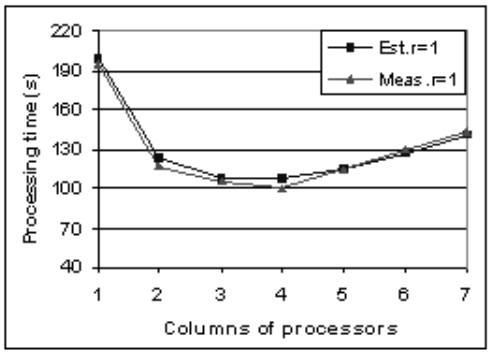

Figure 14. Tridiagonal reduction processing time for a matrix of size 1200

Figure 15 shows the estimated (E) and measured (M) communication times for matrices of size 800 and 1200. The more significant differences are for matrix of size 1200 where communications are overestimated. In all cases the difference is below 1.1 second.

Figure 16 shows the load distribution for the matrix of size 1800. For grid (1,4) a good load balancing is achieved. For grid (1,5) one process takes 3 seconds less than the others because it was assigned one block less of size 20. The data allocated to each processor was 1200 rows by 340, 320, 220, 220 and 100 columns respectively; ideally it should be 1200 by 336, 336, 222, 222, 83. Grids (1,6) and (1,7) are not well balanced also due to block indivisibility.

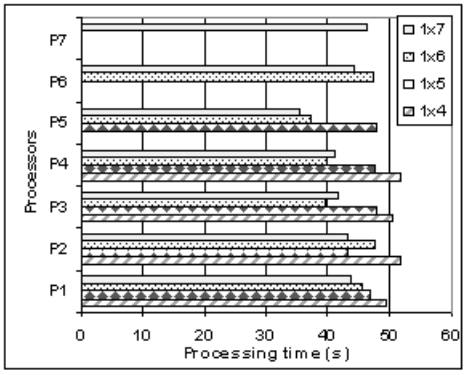

Figure 16. Tridiagonal reduction load distribution for a matrix of size 1200

5.3. QR iteration algorithm

The QR iteration is the last step in the eigenvector computation sequence, preceded by the tridiagonal reduction of a symmetric matrix and orthogonal matrix computation.

Synthetically, the procedure is to compute Givens rotations in order to reduce the tridiagonal matrix into a diagonal one whose elements are the eigenvalues. Eigenvectors are computed by updating the orthogonal matrix, resulting from the tridiagonal operation, with the rotations. Each rotation affects only two columns of the orthogonal matrix; a detailed explanation is given in [12].

The parallelization implemented takes advantages of the fact that one rotation updates only two columns without inter-row dependencies. For the tridiagonal reduction a column oriented distribution is more favorable; however, that data allocation will imply communications between boundary columns, with the additional drawback of using cyclic distributions, which increase the boundary columns drastically. A column oriented algorithm applying the technique of considering multiple bulges [13] was implemented, but only a marginal speedup, below 1.5, was obtained due to the fact that multiple bulges increase the number of iterations required which, associated to boundary communications, is not suited for the slow bus network of the target machine.

Alternatively, it was given the possibility of data redistribution in order to match the ideal processor grid for each algorithm. In this case, QR iteration was a row oriented strategy.

The QR iteration has two computational tasks: one, to do the bulge chase of order n^2, and the other to update the orthogonal matrix of order n^3:

$$T_P = \beta \frac{n^3}{T_T(r,c)} + \Theta(n^2) \tag{26}$$

The β for QR is 43. The time to compute the chases is in fact negligible compared to the $\Theta(n^3)$ term (e.g., for the matrix of size 1600 used it takes 2.1 seconds to compute the chases and 721 seconds to update the matrix in a 244 Mflop computer). Therefore, the solution adopted was to do the chases in one computer (1,1), the fastest one, which at the end of a chase transmits the correspondent rotations to the remaining processors. Then, all processors update the part of the orthogonal matrix allocated to them without requiring any data exchange, i.e. true parallelism.

Figure 17 shows the estimated and measured processing time for a matrix of size 1000. The difference for grid (1,4) is mainly due to error of the quadratic approximation which is maximum for 4 processors. The estimated minimum is 6 processors; in practice it is 7 processors. This is due to a load imbalance occurring for 6 processors, in which there is a processor that takes 2 seconds more than the others, as shown in figure 18.

The communications involved are only to distribute the Givens rotations, estimated assuming a convergence rate of γ, as:

$$\gamma n^2 (r-1) \tag{27}$$

This is an estimation because the number of chases depends on the rate of convergence of the QR iteration. This rate is expected to be less than 2 [8]. The estimated values of figure 19 were obtained with $\gamma = 0.9$ obtained experimentally with the matrix used. In this algorithm the communication parameters T_E and T_L refer to the machine that computes the Givens rotations, since it is the only emitter in the QR iteration.

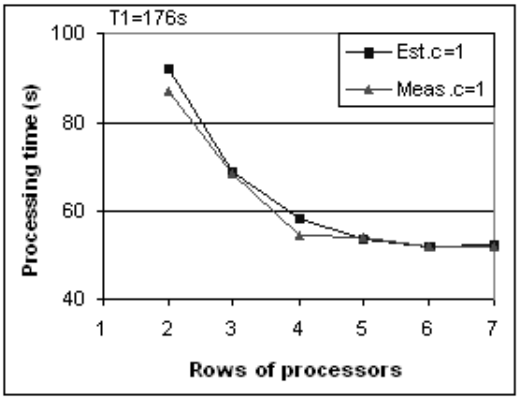

Figure 17. QR iteration processing time for a matrix of size 1000

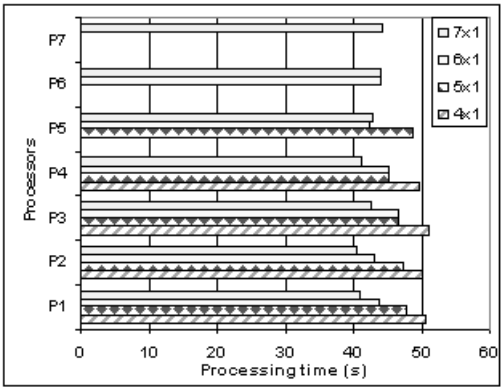

Figure 18. QR iteration load distribution for a matrix of size 1000

5.4. Symmetric eigenvector computation

In this subsection the whole algorithm for eigenvector computation executed in the heterogeneous machine is compared to a serial version [16] when executed in the fastest node.

The performance metrics used to evaluate the parallel application is, first, the runtime, and second the speedup achieved. To have a fair comparison in terms of speedup, one defines the Equivalent Machine Number $(EMN(p))$ which considers the power available instead of the number of machines that, for a heterogeneous environment, is an ambiguous information. Equation 28 defines $EMN(p)$ for p processors used, and S_1 is the computational capacity of the processor that executed the serial code, also called the

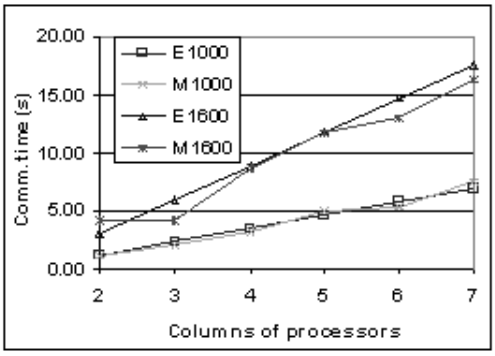

Figure 19. Estimated (E) and measured (M) communications for QR iteration

master processor.

$$EMN(p) = \frac{\sum_{i=1}^{p} S_i}{S_1} \tag{28}$$

For the machine presented in figure 3 $EMN(6) = 3.77$ and $EMN(7) = 3.97$, i.e. using 6 processors of the heterogeneous machine is equivalent to 3.77 processors identical to the master processor and to 3.97 if 7 processors are used.

Figure 20 and table 3 compare the virtual machine to the fastest node of the machine used to run the sequential code. Different grid configurations are used for the different algorithms, according to the optimal grid computed by equation 21.

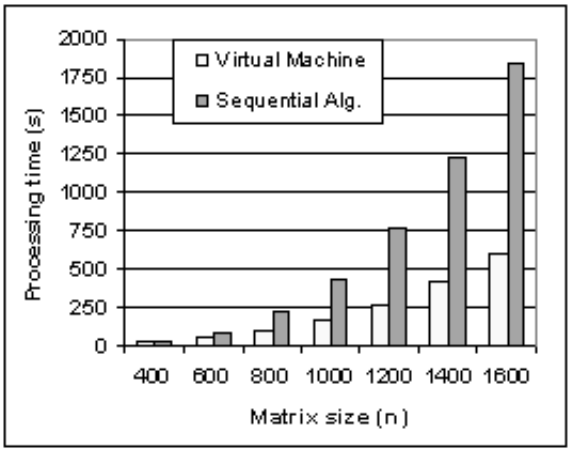

Figure 20. Eigenvector computation in a 7 processor heterogeneous machine compared to the sequential algorithm executed in the fastest node

Stage	Number of processors used							GRID
(n)	400	600	800	1000	1200	1400	1600	(pxq)
TRD	1	2	3	3	4	4	4	1xq
Q (Orth.)	5	6	6	6	7	7	7	1xq
QR it	5	6	6	6	6	6	6	px1
Speedup	1.0	1.7	2.3	2.6	2.9	3.0	3.1	
EMN	3.6	3.8	3.8	3.8	4.0	4.0	4.0	
Efficiency	0.3	0.5	0.6	0.7	0.7	0.8	0.8	

Table 3. Processors used in each stage of the eigenvector computation

6. Conclusions

Briefly stated, the methodology presented in this paper was designed to address problems arising in the context of using image processing and analysis algorithms for interactively extracting important data and information from images of a specific application domain, e.g. medical imaging.

Currently, this activity is often conducted by exploring the functionality (hardware and software) of general purpose systems, which usually trade off algorithm sophistication and user comfort; this means that more advanced image tools may be absent in these systems due to practical considerations.

The main goal of the work herein presented was to take advantage of the existence of a network of computers (this is a very frequent situation in many user organizations) to try and move the aforementioned trade-off in the direction of allowing the provision of more advanced and sophisticated algorithms without sacrificing user comfort.

The results presented show that, for the important linear algebra building blocks of many advanced image analysis methods, the stated goal may be accomplished; an improvement has been achieved in the execution time, by a factor of about 3, which may bring more image analysis tools into the feasible condition for new general-purpose software.

A collection of machines with a wide range of processing capacities, from 244 to 49 Mflops in the case presented, can cooperate and achieve a considerable speedup in linear algebra algorithms. The load balancing strategy proved to be a determinant condition for the quality of the results.

A methodology to determine in a computer network the number of active processors that minimizes the total processing time for a specific parallelized algorithm was presented. The main objective is that the user of a computationally demanding application may benefit from the computational power distributed over the network, while keeping other active users undisturbed.

This goal can be achieved in a transparent manner for the user, once the modules of his/her application are correctly

parallelized for the target network and the performance of the machines in the network is known. The application, before initiating a parallel module, determines the best available computer composition for a parallel virtual computer to execute it, and then launches the module, achieving the best response time possible in the actual network conditions.

Practical tests of the methodology were conducted both on homogeneous and heterogeneous networks, using basic algorithms from linear algebra; in both cases, the theoretical values computed were confirmed by the measured performance. It was shown that a good load balancing could be achieved even for a heterogeneous environment, by using an appropriate processor layout. Other generic modules will be parallelized and tested, so that an ever increasing number of image analysis methods may be assembled from them. Application domains other than image analysis may also benefit from the proposed methodology.

References

[1] A. Alves, L. Silva, J. Carreira, and J. Silva. Wpvm: Parallel computing for the people. In *HPCN'95 High Performance Computing and Network Conference*, Milan (http://dsg.dei.uc.pt/wpvm), 1995. Springer-Verlag.

[2] T. Anderson, D. Culler, D. Patterson, and T. N. Team. A case for now (network of workstations). *IEEE Micro*, February 1995.

[3] M. Baker, R. Buyya, and D. Hyde. Cluster computing: A high-performance contender. *IEEE Computer*, 32(7):79–83, July 1999.

[4] S. Balsamo, L. Donatiello, and N. V. Dijk. Bound performance models of heterogeneous parallel processing systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(10), October 1998.

[5] J. Barbosa and A. Padilha. Algorithm-dependent method to determine the optimal number of computers in parallel virtual machines. In *VECPAR'98, 3rd International Meeting on Vector and Parallel Processing (Systems and Applications)*, volume 1573, Porto, 1998. Springer-Verlag.

[6] J. Choi, J. Dongarra, and D. Walker. The design of parallel dense linear software library: Reduction to hessenberg, tridiagonal and bidiagonal form. Technical Report LAPACK Working Note 92, University of Tennessee, Knoxville, January 1995.

[7] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: Towards a realistic model of parallel computation. In *4 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, 1993.

[8] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[9] J. Dongarra, S. Hammarling, and D. W. Walker. Key concepts for parallel out-of-core lu factorization. Technical Report CS-96-324, LAPACK Working Note 110, University of Tennessee Computer Science, Knoxville, April 1996.

[10] J. Dongarra and D. Walker. The design of linear algebra libraries for high performance computers. Technical Report LAPACK Working Note 58, University of Tennessee, Knoxville, June 1993.

[11] R. Geijn and J. Watts. Summa: Scalable universal matrix multiplication algorithm. Technical Report CS-95-286, University of Tennessee, Knoxville, 1995.

[12] G. Golub. *Matrix Computations*. The Johns Hopkins University Press, 1996.

[13] G. Henry, D. Watkins, and J. Dongarra. A parallel implementation of the nonsymmetric qr algorithm for distributed memory architectures. Technical Report Technical Report CS-97-352 and LAPACK Working Note 121, University of Tennessee, March 1997.

[14] J. JáJá and K. Ryu. The block distributed memory model. Technical Report CS-TR-3207, University of Maryland, January 1994.

[15] J. E. Marsden and A. J. Tromba. *Vector Calculus*. W. H. Freeman and Company, 1981.

[16] W. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1997.

[17] E. Rosti, E. Smirni, L. Dowdy, G. Serazzi, and K. Sevcik. Processor saving scheduling policies for multiprocessor systems. *IEEE Transactions on Computers*, 47(2), February 1998.

[18] S. Sahni and V. Thanvantri. Performance metrics: Keeping the focus on runtime. *IEEE Parallel & Distributed Technology*, pages 43–56, Spring 1996.

[19] C. Seitz. Myrinet - a gigabit per second local-area network. *IEEE Micro*, February 1995.

[20] J. Shen and S. Castan. An optimal linear operator for step edge detection. *CVGIP: Graphical Models and Image Processing*, 54(2):112–133, 1992.

[21] C. Spurgeon. *Ethernet Configuration Guidelines*. Peer-to-Peer Communications, Inc, 1996.

[22] A. Steen. Methodology, metrics and presentation of results. In *Tutorial in VECPAR'98, 3rd International Meeting on Vector and Parallel Processing (Systems and Applications)*, Porto, 1998.

[23] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

Biographies

Jorge Barbosa got a diploma in Electrical Engineering from FEUP (Faculdade de Engenharia do Porto), a MSc. in Digital Systems from UMIST, UK, and he is currently a PhD. student at FEUP, researching the application of parallel computing in image processing. João Tavares got a diploma in Mechanical Engineering and a MSc. in Electrical Engineering from FEUP, and he is currently a PhD. student at FEUP researching defornable object models in image processing. Armando Padilha is associate professor at FEUP and GROUP research leader at INEB (Biomedical Engineering Institute, http://ineb.fe.up.pt).