Balancing load under large and fast load changes in distributed computing systems - A case study

Thierry Le Sergent[1] and Bernard Berthomieu[2]

[1] LFCS, University of Edinburgh, Edinburgh, EH9 3JZ, U.K.
E-mail: Thierry.Le.Sergent@dcs.ed.ac.uk (On leave from LAAS/CNRS Toulouse)
[2] LAAS/CNRS, 7, avenue du Colonel Roche, 31077 Toulouse Cedex, France
E-mail: Bernard.Berthomieu@laas.fr

Abstract. This paper discusses a load balancing technique for distributed processing systems in which the load may vary over a wide range and at a high rate. Each processor performs a source or server algorithm for migrating processes when its load crosses some assigned upper or lower bound; these bounds are dynamically adjusted. Taking into account the speed at which loads vary and the latency of the underlying network, we specify conditions under which the algorithm is stable and responds satisfactorily to fast load changes. Simulation confirms the validity of these conditions.

1 Introduction

This paper discusses a technique for balancing process load between processors of a distributed computing system. The technique is intended to handle large and fast load changes. Many papers have been written about load sharing or load balancing (see e.g. [2] for references), but our particular context requires a specific study.

The algorithm was designed for distributed implementations of the concurrent functional language LCS [4, 3] which provides medium grained parallelism, i.e. intermediate between instruction level and task level parallelism. In a short time, an LCS application may create thousand of micro-processes, most having a short life. LCS processes are implemented as linked structures and all stand in a single address space; a process is identified by a single reference.

The distributed implementations of LCS are constituted of a small fixed number of virtual machines cooperating by exchange of messages, each bound to a physical processor, and each managing a possibly large number of LCS processes [8]. All these machines share a single virtual address space. It is assumed that each of them manages its own queue of ready processes and that the communications between any pair of processors have the same cost. The load of the processor is measured by the number of LCS processes it manages.

The property we would like to ensure is that all the processes have a similar service rate. For interactive applications, this guarantees that the response time to users requests will not vary widely depending on the processor on which the request is processed. We will thus use a "load balancing" technique [1], rather than a simpler "load sharing" [7] technique.

Load transfer can be implemented by moving processes at the time they are created (placement), but this method would not handle load implosions. Instead, we will use preemptive migrations of processes, as in [5, 1]. The designation of a process by a single reference, and the single address space assumption for the virtual machines, make migrations of LCS processes cheap.

A migration policy determines "which" (identification), "when" (decision) and "where" (location) processes should be migrated [6]. We shall not discuss here the identification aspects, which typically involve specific properties of processes and processors. Our decision and location policies are based on exchange of information. Using only local information, i.e. having each processor communicating only with a subset of all processors, would reduce the cost of that information, but the balance would imply a greater number of migrations [6]. We will use a global strategy instead, though it could be used in a hierarchical setting to suit systems with a very large number of processors.

The exchange of information by processors will be triggered asynchronously, on occurrence of some local events, rather than periodically as in [10, 15]. With our fast load changes assumption, the latter solution would require a very short period and thus much more messages exchanged. The relevant events are those which modify the load of the processors, i.e. the creation or termination of processes. However, exchanging information at each of these events would be too expensive; a better solution is to issue an event when the load of some processor crosses some assigned lower or upper bound [7, 15, 11, 6, 14].

There are two classical decision policies [13]: The "source initiative" method in which processors ask for help when their load overflows an upper bound, and the "server initiative" method in which processors ask for work when their load underflows a lower bound. Most of the published algorithms use one of these policies with a single constant bound. A drawback of constant bounds is that the load will not be balanced when all processors have overflowed their upper bound (or underflowed their lower bound). Like [6] and [14], our algorithm is adaptive with the system load; it uses both source and server policies and is based upon dynamic lower and upper bounds on the load of each processor. On the contrary to [6] the information policy does not use a timer, and amongst the differences with [14], it is not centralised.

Stability of the algorithm becomes an issue when the communication latency is high compared to the speed of variation of the load [7]; unnecessary migrations should be avoided. [12] defines stability by "the response to any –reasonable– excitation does not exceed some bound". These excitations may include, for example, the case of an arrival rate temporarily faster than the total service rate. We shall study precisely the conditions under which our algorithm is stable.

The algorithm is described in the next section, where its properties are analyzed in the ideal case of instantaneous communications. In section 3, we take into account the communication latency between processors and derive the practical operating conditions under which our algorithm balances the load correctly. Section 4 presents simulation results which confirm the properties of the algorithm. Some further investigations are discussed in the concluding section.

2 The algorithm

2.1 Principles

The load of each processor P_i is allowed to fluctuate between a lower bound b_i and an upper bound B_i. The interval $]b_i, B_i[$ is called the band of the processor; each time its load goes out of its band, the processor starts the balancing algorithm.

The bands are dynamic; they are defined from the notion of level, a positive integer. The level e_i of each processor determines, through two band control functions $b()$ and $B()$, the band $]b(e_i), B(e_i)[$. In addition to the bounds $b_i = b(e_i)$ and $B_i = B(e_i)$, we shall need the middle value m_i, defined by $m_i = (b_i + B_i)/2 = m(e_i)$. The algorithm proceeds as follows:

- While its load n_i overflows B_i (resp. underflows b_i), processor P_i asks the other processors if they would like to accept (resp. give) some processes in order to make its load n_i nearer the middle m_i of its band. As soon as its load is back in its band, the algorithm running on P_i stops.

- A processor P_j accepts the request if an exchange of processes with P_i makes its load n_j nearer m_i (without crossing it), otherwise it rejects the offer.

- If, after asking all other processors and performed migrations whenever this was possible, the load n_i of P_i is still out of its band, its load level e_i is adjusted, determining a new band. If its load is still out of its new band, the processor restarts the algorithm. The processors that accepted some request also adjust their levels if migration made their load leave their band.

The algorithm is shown in figure 1. An implementation of the various Server and Source procedures are described in section 3. The procedures Algo_Server and Algo_Source involve all the processors. Depending on their implementation, they may or may not use some procedures Protocol_Server and Protocol_Source that implement cooperation between two processors.

case $n_i \leq b_i$:
P_i: if $n_i \leq b_i$
  then Algo_Server(m_i);
    if $n_i \leq b_i$ then Decrease(e_i);
P_j:
when Notice_Server(M_i, D_i, P_i);
% M_i: middle of P_i. D_i: load needed
if $n_j > M_i$
then Agree_Server($\min(n_j - M_i, D_i), P_i$);
  if $n_j \leq b_j$ then Decrease(e_j);
else Disagree_Server(P_i);

case $n_i \geq B_i$:
P_i: if $n_i \geq B_i$
  then Algo_Source(m_i);
    if $n_i \geq B_i$ then Increase(e_i);
P_j:
when Notice_Source(M_i, D_i, P_i);
% M_i: middle of P_i. D_i: load to give
if $n_j < M_i$
then Agree_Source($\min(M_i - n_j, D_i), P_i$);
  if $n_j \geq B_j$ then Increase(e_j);
else Disagree_Source(P_i);

Fig. 1. Distributed load balancing algorithm.

2.2 Properties

Let us assume for the moment that the time for transferring a message between two processors, and the time for processing it, are negligible; this assumption will be discussed in section 3. With this assumption, execution of the algorithm takes no time, so the load of each processor is maintained within its band, as expressed by the following property:

$\mathcal{P}_1 : \forall P_i.\ b_i < n_i < B_i$

When the load varies very quickly, trying to enforce a bound on a difference of loads between processors may lead to instability. We shall enforce instead a bound of one on the difference of levels between the processors, i.e.:

$\mathcal{P}_2 : \forall P_i . \forall P_j .\ |e_i - e_j| \leq 1$

The algorithm is parameterized by the functions $b()$ and $B()$ which determine the bounds from the levels. These functions must satisfy certain conditions. Intuitively

it is necessary that at most two bands can have a non empty intersection. It is proved in [8] that conditions $\mathcal{C}_1$ and $\mathcal{C}_2$ below on these functions are necessary and sufficient to have property $\mathcal{P}_2$. Condition $\mathcal{C}_3$ forbids two contiguous bands to have a non empty intersection.

$$\left.\begin{array}{l}\mathcal{C}_1 : \forall e \geq 0.\ m(e+1) \geq B(e) \\ \mathcal{C}_2 : \forall e \geq 0.\ b(e+1) \geq m(e)\end{array}\right\} \Rightarrow b(e+2) \geq m(e+1) \geq B(e)$$
$$\mathcal{C}_3 : \forall e \geq 0.\ B(e) > b(e+1)$$

2.3 The band control functions

The quality of balance clearly depends on functions $b()$ and $B()$. For example, if $b(0) = -1$ and $B(0) = \infty$, there is no balance at all because the bounds are never reached. At the opposite end, if the load is measured by integers and the functions are defined by $b_i = n_i - \epsilon$ and $B_i = n_i + \epsilon$ where $\epsilon < 1$, the algorithm tries to make a perfect balance.

We shall give below three different pairs of band control functions that yield interesting balancing properties. They all conform with conditions $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$. Further, in order to make the algorithm as stable as possible, we choose to have the largest possible intersection between contiguous bands. This is done by functions that satisfy $\mathcal{C}'_3$ below. Note that $\mathcal{C}'_3$ implies $\mathcal{C}_2$ and $\mathcal{C}_3$.

$$\mathcal{C}'_3 : \forall e \geq 0.\ m(e) = b(e+1)$$

The band width $B(e) - b(e)$ may be chosen constant, or it may be chosen to vary with e. In the latter case, it must increase with the level e in order to satisfy $\mathcal{C}_1$. A width increasing with the level, i.e. with the load, allows to weaken the balancing constraint when the load becomes high. This is reasonnable since a given difference of load between processors is more significant when few processes are running than when there are many of them.

The first set of band control functions (1) determines constant width bands, the other two determine varying width bands, with a linear increase (2) and a geometrical increase (3).

(1) $a > 0$ real,	(2) $a > 0$ real,	(3) $a > 1$ real,
$b(e) = ea$	$b(e) = ae(e-1)/2$	$b(e) = a^e$
$B(e) = (e+2)a$	$B(e) = ae(e+3)/2$	$B(e) = (2a-1)a^e$

Very different balancing properties are obtained by these three sets of functions:

- With constant width bands, the maximum difference D of load between any pair of processors is bounded by a constant. With $\mathcal{P}_1$ and $\mathcal{P}_2$, we have:
  $\forall n_i > n_j.\ D < B_i - b_j$ with $e_i \leq e_j + 1$, thus:
  $\forall e \geq 0.\ D < B(e+1) - b(e) = (e+1+2)a - ea = 3a$
- With a linear increase of band width, the difference D depends on the level e; it is here the ratio R of the difference of load to the level which is bounded:
  $$\forall e \geq 0.\ R = \frac{B(e+1) - b(e)}{e+1} = 3a - \frac{a}{e+1} < 3a.$$
- Finally, if bands increase geometrically, it is the ratio X of the load of the most loaded processor to the load of the least loaded processor which is bounded:
  $$\forall e \geq 0.\ X < \frac{B(e+1)}{b(e)} = \frac{(2a-1)a^{e+1}}{a^e} = (2a-1)a$$

2.4 Idle processors

A processor becomes idle when its load reaches zero. The case would only arise when the total load of the system is so small that it cannot be distributed among all processors.

Let us call ms the lowest level. A very precise balance can be obtained by using $b(ms) = 0$ and $B(ms) = c$ where c is the smallest load that can be divided. For example, if the load is measured by the number of processes, one can take $c = 2$ (the example band control functions given in section 2.3 have to be slightly modified to match the constraint added here).

When $n_i = b(ms) = 0$, P_i asks for work with a middle m_i less than the smallest divisible load. According to the algorithm, if there exists a processor that has a divisible load, a migration will occur, and P_i will no longer be idle. Otherwise, P_i stops executing the algorithm and remains idle until another processor requests some help. When this case arises, the properties of the algorithm ensure that all the processors are at level ms. As soon as one of them has a divisible work, its load will cross its upper bound ($B(ms) = c$), and it will start a source algorithm which will send some work to some idle processor.

This property is also interesting for initialization of the system. Taking ms as the initial level of all processors, the initial load will be automatically balanced without requiring any startup procedure by idle processors.

2.5 Automatic synthesis of band control functions

This section discusses a technique to build automatically band control functions implying some required balancing properties. From conditions $\mathcal{C}_1$ and $\mathcal{C}'_3$ and the definition of middle $m(e)$, we obtain the following system:

$$\begin{cases} \mathcal{C} : \forall e \geq 0.\ b(e+2) - 2b(e+1) + b(e) \geq 0 \\ \forall e \geq 0.\ m(e) = b(e+1) \\ \forall e \geq 0.\ B(e) = 2m(e) - b(e) \end{cases}$$

The series of the differences of function $b()$ is defined by:
$\forall e \geq 0.\ \Delta b(e) = b(e+1) - b(e)$, and the series of the second difference by:
$\forall e \geq 0.\ \Delta^2 b(e) = \Delta b(e+1) - \Delta b(e)$. We have: $\mathcal{C} \Leftrightarrow \Delta^2 b(e) \geq 0$.

The series of the second differences Δ^2 is the analogue for series of the second order derivative for real functions. The predicate $\mathcal{C}$ means that the function $b()$ must be monotonically increasing.

Given a positive integer function $x()$, function $b()$ can be built by solving:

$\forall e \geq 0.\ b(e+2) - 2b(e+1) + b(e) = x(e)$ with $\forall e \geq 0.\ x(e) \geq 0$

A solution is:

$\forall e \geq 0.\ b(e+2) = (e+2)b(1) - (e+1)b(0) + (e+1)\sum_{i=0}^{e} x(i) - \sum_{i=0}^{e} i x(i)$

Where $b(0)$ and $b(1)$ are the initial conditions of the system.

As an example, the following values allow to derive the functions given in 2.3:

constant bands :	$b(0) = 0$	$b(1) = a$	$x(i) = 0$	$b(e) = ea$
linear variation :	$b(0) = 0$	$b(1) = 0$	$x(i) = a$	$b(e) = a(e-1)e/2$
geometric variation :	$b(0) = 1$	$b(1) = a$	$x(i) = a^i(a-1)^2$	$b(e) = a^e$

To obtain new functions, with the property discussed in 2.4, it is sufficient to set $b(0) = 0$, and $b(1) = B(0)/2 = c/2$, and to choose any positive integer function $x()$. $x()$ can be regarded as the rate of increase of the bounds with the level.

3 Implementation

An implementation of the algorithm consists in defining the procedures used in figure 1, particularly Algo_Server and Algo_Source, which themselves may use Protocol_Server and Protocol_Source, and Agree_Server and Agree_Source. Of course an efficient implementation of these procedures would depend on the nature of the underlying network; we shall give an example of a simple implementation in 3.2. We will first discuss the crucial problem of the latency of communications.

3.1 The communication delay problem

The procedures of the load balancing algorithm are executed asynchronously; a processor continues to work while waiting for load balancing messages. If the speed of the variation of the load is faster than the speed of performing load balancing (because of slow communications), the properties $\mathcal{P}_1$ and $\mathcal{P}_2$ that the algorithm tries to enforce may well be never satisfied. The time during which these properties hold depends directly on the choice of the band control functions. The wider the bands are, the longer it will take for the load of a processor to leave its band. Since the time to execute the algorithm is constant, increasing the band widths allows to maintain the expected result for longer periods

Two scenarios must be studied: On one hand, the load of a processor may oscillate so rapidly that the algorithm becomes unstable. On the other hand, the load of a processor may explode or implode so rapidly that the processor cannot adjust its level quickly enough. For these two cases, we derive operating conditions that ensure a correct behavior of the algorithm.

Stability

No transfer of load should make the load of the initiator leave its band, leading to processor thrashing [7]. For some procedure proc, let Δt_{proc} be the maximum time taken by its execution. P_i asks for a migration of processes when either $n_i \leq b_i$ or $n_i \geq B_i$. Let P_j be a processor which accepts the request:

Case $n_i \leq b_i$ **:**

For P_i **:** The load received is $q \leq m_i - n_i(t)$, $n_i(t)$ is the load of P_i at the instant t of beginning of execution of procedure Protocol_Server. After the migration of processes, we must have $n_i(t + \Delta t_{\mathsf{Protocol_Server}}) + q < B_i$. So the condition for a correct execution of the algorithm is in this case:
$n_i(t + \Delta t_{\mathsf{Protocol_Server}}) - n_i(t) < B_i - m_i = (B_i - b_i)/2$

For P_j **:** The load transferred is $q \leq n_j(t') - m_i$.
And $n_j(t' + \Delta t_{\mathsf{Agree_Server}}) - q \leq b_i$ should be true (P_j decrease its level if $n_j(t' + \Delta t_{\mathsf{Agree_Server}}) \leq b_j$). So the condition is in this case:
$|n_j(t' + \Delta t_{\mathsf{Agree_Server}}) - n_j(t')| < m_i - b_i = (B_i - b_i)/2$

Case $n_i \geq B_i$ **:**

Symmetrically, the conditions for P_i and P_j are:
$|n_i(t + \Delta t_{\mathsf{Protocol_Source}}) - n_i(t)| < m_i - b_i = (B_i - b_i)/2$
$n_j(t' + \Delta t_{\mathsf{Agree_Source}}) - n_j(t') < B_i - m_i = (B_i - b_i)/2$

Finally, the algorithm can handle oscillation of load provided the load on any processor does not vary by more than $(B_i - b_i)/2$ within the time required to execute Protocol_* and Agree_*. These conditions are called stability conditions.

Explosions and implosions of load

We shall say that the algorithm handles explosion or implosion of load on a processor if, after having changed its level following a load overflow (resp. underflow), its load does not overflow (resp. underflow) its new band. By contrast with the problem of oscillations which involves two processors only, explosions or implosions involve all processors since level changes result from a global negotiation.

Case $n_i \leq b_i$ **:** The time required by P_i to decrease its level e when the server algorithm fails is $\Delta t_{\mathsf{Algo_Server}}$. Since $e-1$ is its new level, P_i's bounds become then $b(e-1)$ and $B(e-1)$. At that time, its load should not be smaller than $b(e-1)$, and thus the delay for a load variation of $b(e) - b(e-1) = m(e-1) - b(e-1) = (B(e-1) - b(e-1))/2$ on P_i should be greater than $\Delta t_{\mathsf{Algo_Server}}$.

Case $n_i \geq B_i$ **:** Symmetrically, the algorithm is able to handle load explosions only if the delay of a load variation of $B(e+1) - B(e) \geq (B(e+1) - b(e+1))/2$ on P_i is greater than $\Delta t_{\mathsf{Algo_Source}}$.

These two conditions are called large variation handling conditions (or LVH conditions for short). We will check their validity by simulation in section 4.

Summary

Three cases have been considered:

- Small load oscillations (smaller than a half band) on each processor cause no problem, whatever their speed. The processors always reach a level at which these oscillations are within the load band, and so no messages are exchanged.

- Larger oscillations are handled properly if the stability conditions are fulfilled. Otherwise, the algorithm may perform migrations which may worsen the balance. The algorithm fails because it is unable to maintain properties $\mathcal{P}_1$ and $\mathcal{P}_2$.

- Large variations of load can be handled if the LVH conditions are met. These conditions involve the full set of processors. If they are not met, then the algorithm will be late in absorbing explosions and implosions of the load on a processor.

All these conditions depend on the time it takes for the load to vary by a half band, which time depends directly on the band control functions. The choice of these functions in an actual implementation should follow from a tradeoff between the precision of the balance of load and the ability to handle fast load variations, taking into account the latency of the underlying communication medium.

3.2 An example of implementation

We shall assume that the network linking the processors is reliable and provides direct point to point connections between all pairs of processors through two, send and receive, procedures. The cooperation realized between two processors P_i and P_j (i.e. initiated by procedure Protocol_*) when the load on P_i leaves its band is:

Case $n_i \leq b_i$ **:** 2 messages are exchanged: the request from P_i, and the answer from P_j which, if positive, contains the migrating processes.

Case $n_i \geq B_i$ **:** 2 or 3 messages are exchanged because we do not want to try to migrate processes before the decision is taken: the request of P_i, the answer of P_j if positive leads to the transfer of processes from P_i to P_j.

The procedures are given in figure 2. Procedures Algo_* activate the corresponding Protocol_* for all the processors in the system, one after each other, until either the load of the initiator is back into its band or it has talked to all the processors (in this case its level will be adjusted). Procedure TakeLoad returns the processes removed from the local queue, and AddLoad adds them to the queue.

```
Algo_Server(M_i) :
  for all P_j and as long as n_i ≤ b_i do
        Protocol_Server(M_i, m_i − n_i, P_j);
Protocol_Server(M_i, D_i, P_j) :
  send SERVER(M_i, D_i) to P_j;
  when receive msg from P_j
       case msg of NO → ;
                 | LOAD(x) →
                   AddLoad(x) ;

Notice_Server(M, D, P_i) :
  receive SERVER(M, D) from P_i
Agree_Server(D_j, P_i) :
  x_j:=TakeLoad(D_j);
  send LOAD(x_j) to P_i;

Disagree_Server(P_i) : send NO to P_i;
```

```
Algo_Source(M_i) :
  for all P_j and as long as n_i ≥ B_i do
        Protocol_Source(M_i, B_i − m_i, P_j);
Protocol_Source(M_i, D_i, P_j) :
  send SOURCE(M_i, D_i) to P_j;
  when receive msg from P_j
       case msg of NO → ;
                 | OK(D) →
                   (x_i:=TakeLoad(D);
                   send LOAD(x_i) to P_j);

Notice_Source(M, D, P_i) :
  receive SOURCE(M, D) from P_i
Agree_Source(D_j, P_i) :
  send OK(D_j) to P_i;
  when receive LOAD(x) from P_j
       AddLoad(x);

Disagree_Source(P_i) : send NO to P_i;
```

Fig. 2. An implementation with point to point communications.

Let tpm be the maximum time for transferring a message between two processors, and processing it. The migration of processes is done here by a single message transfert, so tpm also represents the cost of migrations. With N processors in the system, the execution times of the procedures in the conditions satisfy:

$\Delta t_{\mathsf{Algo_*}} \leq 2(N-1)tpm \qquad \Delta t_{\mathsf{Protocol_*}} \leq 2tpm \qquad \Delta t_{\mathsf{Agree_*}} \leq 2tpm$

The operating conditions of this implementation can now be made precise:

Stability Condition : the time for a variation of load of $(B_i - b_i)/2$ on a processor P_i must be greater than $\Delta t_{\mathsf{Protocol_*}} \leq 2tpm$.

LVH Condition : the time for a variation of load of $(B_i - b_i)/2$ on a processor P_i must be greater than $\Delta t_{\mathsf{Algo_*}} \leq 2(N-1)tpm$.

4 Simulation

The algorithm in figures 1 and 2, together with a stochastic simulator, have been programmed in LCS [3]. The processor model is classically [10, 7, 11] a waiting file $M/M/1$. The Poisson arrival rate, representing the creation of processes, is $\lambda = 1/dt$ on each processor, and the process service time is exponentially distributed with rate μ. Another parameter is the delay of transmission of a message tpm.

4.1 Framework

In [13, 7, 15, 11], the authors studied only the stationary behavior of the system when λ and μ are constant, with $\lambda/\mu < 1$. We want instead to test our algorithm when the arrival rate is temporarily higher than the service rate, to analyze its ability to handle load explosion.

As in [10], examples are run in the worst possible conditions, i.e. in which all processors have the same service rate μ, but only half of them have a non null arrival rate λ. Without load balancing, half of the processors would have a zero load while the load on the others would explode (case $\lambda \geq \mu$).

The direct result from the simulations is the number of processes running on each processor at each step (quantum of time). We are interested in the load of the processors with the highest and lowest loads. Depending on the band control functions used, either the difference of these loads or their ratio is significant; this value is computed at each step.

If the algorithm works properly, then these sequences should remain almost constant, so we can summarize them with the value of their average to show with a single curve the influence of one of the parameters of the simulation. To show that the average is a good representation of each sequence, we also draw the curve of their standard deviation. When the algorithm does not work properly, the sequence increases. Since the simulation is run a long enough time, the average is in this case greater than the theoretical bound.

4.2 Influence of the load, and cost of the algorithm

The curves in figure 3 show the result of the balancing algorithm in a quite difficult situation: half of the processors have a null arrival rate, and on the others, the average time dt between two creations of processes is only four times greater than the time for transmitting a message tpm ($dt = 4tpm$).

We simulated a system with four processors ($N = 4$), and constant width bands with parameter $a = 2$. The theoretical result is a maximum difference of load of $3a - 2 = 4$ processes between any processors (see section 2.3).

The time it takes for a variation of load of $(B_i - b_i)/2 = a = 2$ on a processor is at least the time for creating these 2 processes on the processors which have a non null arrival rate $\lambda = 1/dt$, i.e. $2dt$. Since this time is greater than $\Delta t_{Algo_*} = 2(N-1)tpm = 6tpm = 3/2dt$, the stability and LVH conditions are met (see section 3.1).

Figure 3a shows the result of the simulations, i.e. the average of the sequence of the maximum difference of load between any processors during each simulation. Several simulations were done for various loads[3]. Since the standard deviation of the sequences are low, the curve of their average faithfully represents the maximum difference of load for each simulation.

As the ratio of arrival rate per processor to service rate increases above 2 (the load of the whole system explodes), figure 3a shows that the curve of the averages of differences of load remains constant, with a value below the theoretical limit. So, the algorithm works adequately for any load.

The cost of the algorithm in term of the number of processes migrated and the number of messages exchanged, are shown in figure 3b. When the arrival rate on one processor is greater than its service rate, approximately one process out of two is migrated. No unnecessary migrations are done since half of the processors have a null arrival rate.

[3] Since $dt = 1/\lambda$ and tpm are already linked by $dt = 4tpm$, we vary the load by varying the service rate μ.

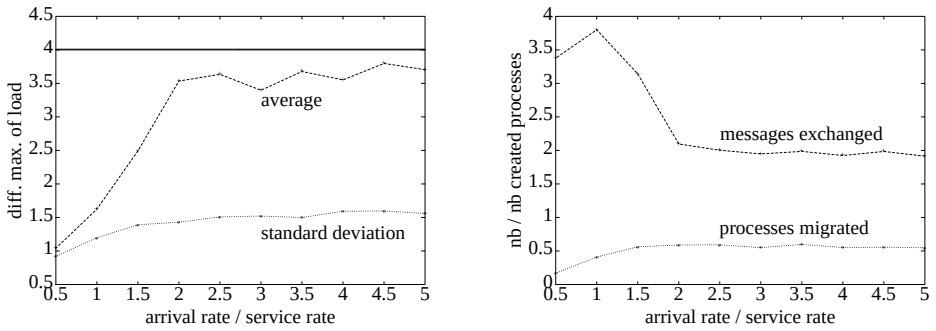

Fig. 3. a) Influence of the load on the balancing. b) Cost of the algorithm.

For a total arrival rate greater than the total service rate (abscissa 2), the number of messages exchanged per process created is stable at two; only four messages are exchanged per migrated process. Our algorithm is cheap in exchanging messages because several processes may be migrated together. For a total arrival rate below the total service rate, the number of messages is higher because the load of the processors is low, so most of them are at the lowest level, and, at this level, only one process at a time may be migrated.

So the algorithm successfully handles explosions of load when the operating conditions determined in section 3 are met. We now study the limit under which it runs properly, i.e. check the validity of the LVH condition.

4.3 Influence of the delay of message transmission

In order to study the influence of the delay for transmitting and processing the messages tpm, we start from one of the previous simulations (with $\lambda/\mu = 4$), and we increase its value. The new curve of the average of the differences in load between processors for different values of tpm/dt is shown in figure 4a.

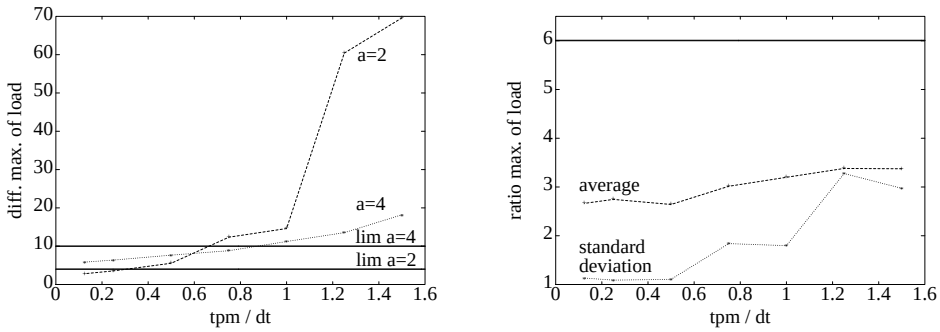

Fig. 4. a) Constant width bands. b) Geometric variation of the band widths.

As before, the theoretical limit (i.e. if $tpm = 0$) of the maximum difference of load is 4. The curve for a=2 shows that when $tpm/dt > 0.3$ this limit is exceeded. This value found by simulation corresponds closely with the theoretical value: The LVH condition is $2(N-1)tpm = 6tpm < adt = 2dt$, i.e. $tpm/dt < 0.333$.

Weakening the balancing constraints by a change of the band control functions improves the behavior of the algorithm. Two examples are detailed in the sequel.

We first keep constant width bands, but we increase their width by setting parameter a to 4, instead of 2. The curve for a=4 is also shown on graph 4a. The theoretical limit of the difference of load is now $3a - 2 = 10$; the algorithm does not attempt to balance the load as precisely as before. The simulations with a small ratio tpm/dt, i.e. when the LVH condition is met, confirm that. But now, the theoretical limit is exceeded only when $tpm/dt > 0.9$, and beyond that the difference of load does not increase as quickly as before.

The LVH condition relies on the delay for an increase of a processes on any processor. When a is not significantly smaller than λ/μ this delay relies on the average rate of increase, so is in this case $a/(\lambda - \mu) = 4adt/3$. The LVH condition is $6tpm < 4adt/3$, i.e. $tpm/dt < 0.88$, value confirmed by the simulation.

Another way to weaken the balancing constraints is to use varying width bands. Figure 4b shows the results of the same simulations as before, but with bands increasing geometrically with a parameter $a = 2$ (see 2.3). This time, it is the maximum ratio of the load between any processors that is bounded. The curve shows that, even for $tpm/dt > 1$, the averages of ratios are well below the theoretical limit $(2a - 1)a = 6$.

For $tpm/dt > 0.5$, these averages, and above all the standard deviation of the sequences, increase. This follows from the fact that the sequences of ratios are stable only after the load has reached a high enough level, i.e. a level which corresponds to a band width large enough to make the LVH condition satisfied.

Finally, for dynamically varying bands, a tradeoff must be done between precise balancing when the load is small and the ability to handle fast load changes.

5 Conclusion and further work

The load balancing algorithm presented in this paper was designed to handle a low level of load as well as a very high level of load. This is achieved by using dynamic lower and upper bounds on the load of each processor. The algorithm may operate in situations where load changes quickly compared to the speed of communications between the processors. We have studied precisely the operating conditions under which our algorithm holds its desirable properties. The simulation confirms the expectations about the algorithm when these conditions are met, and confirms the validity of these conditions.

The parameterization of the algorithm by functions which determine a load interval from a level characterizing the load allows one to tune the algorithm and find the best possible balance of load, taking into account the speed of the variation of the load and the latency of communications.

The algorithm is efficient in terms of messages exchanged because message are exchanged only when the load of a processor crosses its bounds and because several processes may be migrated in a single step.

To go further with the adaptive aspects, we can imagine the band control functions also taking into account the current speed of the variation of the load and resizing the bands accordingly. If it is observed that the variation of the load is slower than expected, then reducing the band widths will improve the balance. We have not yet investigated the issue but it is surely worthwhile.

The algorithm does not rely on a particular implementation. It is described by using simple procedures. For an efficient implementation, the protocols should

match at best the properties of the underlying network.

One could question the scalability of the algorithm on the grounds that a processor might have to communicate with all others. When there are many processors, a possible solution would be to use the algorithm hierarchically. The set of processors would be partitioned into several groups; a first layer of load balancing would balance load within each group and a second layer, possibly with different load balancing parameters and using a different communication protocol, would balance load between the groups. In the context of a non-directly connected set of processors, the partition of the processors should match the graph of the network, thus recovering the idea of neighborhood [6].

Acknowledgments

Many thanks to Dr. Dave Matthews of the Laboratory for Foundations of Computer Science, for proof reading the paper and the many discussions we have had.

References

1. A. Barak and A. Shiloh. A distributed load-balancing policy for a multicomputer. *Software Practice and Experience*, 15(9):901–913, September 1985.
2. Guy Bernard, Dominique Stève, and Michel Simatic. Placement et migration de processus dans les systèmes répartis faiblement couplés. *TSI*, 10(5):375–392, 1991.
3. B. Berthomieu and T. Le Sergent. Programming with behaviors in an ML framework: the syntax and semantics of LCS. In *European Symposium On Programming*, April 1994. Edinburgh, Scotland.
4. Bernard Berthomieu, Didier Giralt, and Jean Paul Gouyon. LCS users manual. Rapport de Recherche 91226, CNRS-LAAS, Septembre 1991.
5. R. M. Bryant and R. A. Finkel. A stable distributed scheduling algorithm. In *2nd Int. Conf. Distributed Comput. Syst.*, pages 314–323, 1981.
6. A. Corradi, L. Leonardi, and F. Zambonelli. Load balancing strategies for massively parallel architectures. *Parallel Processing Letters*, 2(2 & 3):139–148, 1992.
7. D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptative load sharing in homogenous distributed systems. *IEEE Trans. on Soft. Engineering*, SE-12(5):662–675, May 1986.
8. Thierry Le Sergent. *Méthodes d'exécution, et machines virtuelles parallèles pour l'implantation distribuée du langage de programmation parallèle LCS.* Thèse de doctorat de l'Université Paul Sabatier, Toulouse, Février 1993.
9. Cathy Mccann, Raj Vaswani, and John Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, May 1993.
10. L. M. Ni, S. Xu, and T. Gendreau. A distributed drafting algorithm for load balancing. *IEEE Transactions on Software Engineering*, SE-11(10):1153–1161, October 1985.
11. K. G. Shin and Y. Chang. Load sharing in distributed real-time systems with state-change broadcasts. *IEEE Transactions on Computers*, 38(8):1124–1142, August 1989.
12. John A. Stankovic. Stability and distributed scheduling algorithms. *IEEE Transactions on Sftware Engineering*, 11(10):1141–1152, October 1985.
13. Yung-Terng Wang and Robert J. T. Morris. Load sharing in distributed systems. *IEEE Transactions on Computers*, C-34(3):204–217, March 1985.
14. J. Xu and K. Hwang. Heuristic methods for dynamic load balancing in a message-passing multicomputer. *Journal of Par. and Dist. Computing*, 18(1):1–13, May 1993.
15. Songnian Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering*, 14(9):1327–1341, September 1988.