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

Thierry Le Sergent† and Bernard Berthomieu‡

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 one of its assigned bounds. The upper and lower bounds on the load of each processor 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 can respond satisfactorily to fast load changes. The validity of these conditions are confirmed by simulation results.

Key words: Dynamic load balancing, distributed implementation of parallel languages.

1 Introduction

This paper discusses a technique for balancing process load between processors of a distributed computing system. The technique is suitable for systems in which the load of processors may vary in a wide range and at a high rate. Many papers have been written about load balancing, but our particular context requires a specific study.

The algorithm was designed for distributed implementations of the concurrent functional language LCS [BGG91, BL94] which provides medium grained parallelism, i.e. intermediate between instruction level parallelism and typical 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 [Le 93]. All these machines share a single virtual address space. It is assumed that each processor 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

*Extended version of paper accepted for CONPAR 94 / VAPP VI, Linz, Austria, September 1994.
†LFCS, Dept. of Comp. Science, University of Edinburgh, The King's Buildings, Edinburgh, EH9 3JZ, U.K.
E-mail: Thierry.Le.Sergent@dcs.ed.ac.uk, On leave from LAAS/CNRS Toulouse.
‡LAAS/CNRS, 7, avenue du Colonel Roche, 31077 Toulouse Cedex, France.
E-mail: Bernard.Berthomieu@laas.fr

widely depending on the processor on which the request is processed. We will thus use a "load balancing" technique [BS85], rather than a simpler "load sharing" [ELZ86] 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 [BF81, BS85] of processes. The designation of a process by a single reference, and the single address space assumption for the the virtual machines, make migrations of LCS processes cheap. The migration policy answers the questions "when" (decision), "where" (location) and "which" (identification) processes [CLZ92] should be migrated.

The identification aspects will not be discussed here, they may involve specific properties of processes and processors. The 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 reduces the cost of that information, but the balance would imply a greater number of migrations [CLZ92]. 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 initiated by a processor will be triggered asynchronously, on occurrence of some local events, rather than periodically [NXG85, Zho88]. With our fast load changes assumption, the latter solution would require a very short period and 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 [ELZ86, Zho88, SC89, CLZ92, XH93].

There are two classical decision policies [WM85]: 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). 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.

A problem of stability may appear when communication latency is high compared to the speed of variation of the load [ELZ86]. Unnecessary migrations should be avoided. [Sta85] defines a distributed scheduling algorithm to be stable if "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.

In the next section, we describe the algorithm and analyze its properties in the ideal case where communications are instantaneous; we also discuss the effects of its parameters, the functions which calculate the dynamic bounds. 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. The last two sections relate our approach to existing work and discuss further investigations, respectively.

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 functions $b()$ and $B()$, the band $]b(e_i), B(e_i)[$. These functions will be called the band control functions. 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 having asked all the 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 incremented (resp. decremented), determining a new band for its load. If its load is still out of its new band, the processor restarts the algorithm. The processors that accepted some request also adjust their level if migration made their load leave their band.

The algorithm is described 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 the cooperation between two processors.

2.2 Properties

Let us assume that the time for transferring a message between two processors, and the time for processing it is negligible; the effects of removing this assumption will be discussed in section 3. With this assumption, the execution of the algorithm is instantaneous, so the load of each processor is maintained within its band, i.e. the following property is satisfied:

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

When the load varies very quickly, a situation our algorithm is intended to handle, trying to enforce a bound on a difference of loads between processors may lead to instability. We weaken this control by enforcing instead a bound on the difference of levels between the processors. We require that there must not be a difference of level of more than one between any two processors, i.e.:

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

Together with the property $\mathcal{P}_1$, an algorithm that has the property $\mathcal{P}_2$ realizes a balance of the load, parameterized by the functions which link the levels and the bounds.

```
case n_i ≤ b_i :
    P_i : if n_i ≤ b_i
          then Algo_Server(m_i);
               if n_i ≤ 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: amount of
               if n_j > M_i                    % load that P_i would like to receive
               then Agree_Server(min(n_j − M_i, D_i), P_i);
                    if n_j ≤ b_j then Decrease(e_j);
               else Disagree_Server(P_i);
case n_i ≥ B_i :
    P_i : if n_i ≥ B_i
          then Algo_Source(m_i);
               if n_i ≥ 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: amount of
               if n_j < M_i                    % load that P_i would like to give
               then Agree_Source(min(M_i − n_j, D_i), P_i);
                    if n_j ≥ B_j then Increase(e_j);
               else Disagree_Source(P_i);
```

Figure 1: Distributed load balancing algorithm.

The set of functions that can be used is restricted to those which satisfied certain conditions, intuitively it is necessary that at most two bands can have a non empty intersection. It is proved in [Le 93] that conditions $\mathcal{C}_1$ and $\mathcal{C}_2$ below on these functions are necessary and sufficient to have $\mathcal{P}_2$. 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 the functions which define the bounds of bands from the levels. 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 give three different sets of such 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 $\mathcal{C}_3$.

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

Now, 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 the weakening of the load balancing constraint when the load becomes high. This is desirable since a given difference of load between processors is more significant when few processes are running than where there are many of them.

The first set of band control functions determines constant width bands:

$e \geq 0$ integer; $a > 0$ real, the size of a half band

$b(e) = ea$
$m(e) = (e+1)a$
$B(e) = (e+2)a$

We give two examples of bands with varying width. In the first one the width of the bands increases linearly with the level:

$e \geq 0$ integer; $a > 0$ real, increment of a half band

$b(e) = ae(e-1)/2$
$m(e) = ae(e+1)/2$
$B(e) = ae(e+3)/2$

In the second one, the width of the bands increases geometrically:

$e \geq 0$ integer; $a > 1$ real, increasing factor of bands

$b(e) = a^e$
$m(e) = a^{e+1}$
$B(e) = (2a-1)a^e$

The bands produced by these sets of functions are shown in figure 2.

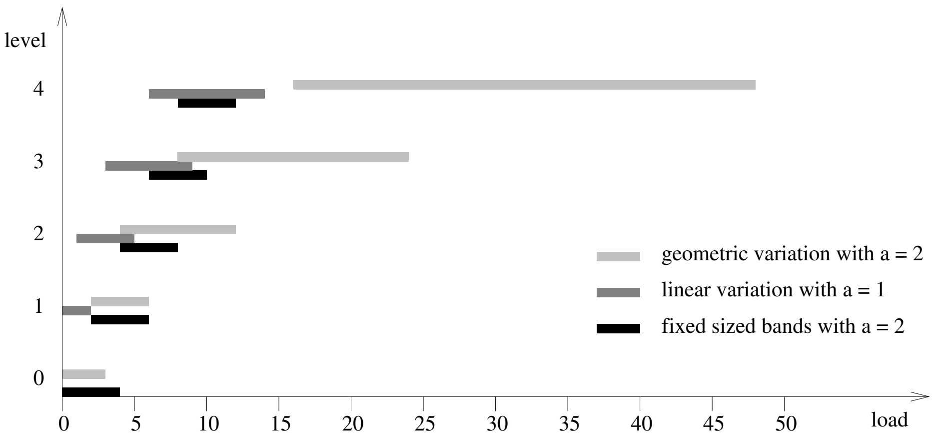


Figure 2: Bands for the first five levels.

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

- With functions determining constant size bands, the maximum difference D of load between any pair of processors is bounded by a constant. With properties $\mathcal{P}_1$ 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 bound on the maximum difference of load between any processors depends on their 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, D also depends on e, but 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

Some processors in the system may become idle, i.e. their load reach zero. The case should only arise when the total load of the system is so small that it cannot be distributed between all processors.

Let us call ms the lowest level. A very precise balance can be obtained by using:

$$\begin{aligned} &b(ms) = 0 \\ &m(ms) = (b(ms) + B(ms))/2 \\ &B(ms) = c \end{aligned}$$

c is the smallest load that can be divided. For example, if the load is measured by the number of processes, c can be chosen such that $c = 2$ (the example band control functions we gave in section 2.3 have to be slightly modified to match the constraint we have 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 on level ms. As soon as one of them has divisible work, its load will cross its upper bound ($B(ms) = c$), and it will start a source algorithm which will wake up the idle processor and send it some work.

This property is also very interesting for the initialization of the system: the level of all the processors is ms. If some of them have a load greater than $c = B(ms)$, they start a source algorithm, and the load will be automatically balanced without requiring any startup procedure by the processors which have an initial load of 0.

2.5 Automatic synthesis of band control functions

In addition to the interesting given band control functions, it would be interesting to synthesize new functions from some required balancing properties. This section discusses a technique to build automatically a set of functions that satisfy the necessary conditions.

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 $x()$, the function $b()$ can be built by solving the equation:

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

A solution is:

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

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

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

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 $b(1) = m(0) = B(0)/2 = c/2$, 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 the description given 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 depends on the nature of the underlying network; it can be based on a centralized protocol, use broadcast or point to point communication, etc. We shall give an example of a simple implementation in 3.2.

In real implementations, communications between the processors are not instantaneous. We first study this crucial problem.

3.1 The communication delay problem

The procedures of the load balancing algorithm are executed asynchronously; a processor continues its work while waiting for messages relevant to the load balancing algorithm.

If the speed of the variation of the load is faster than the speed of performing load balancing (because the communications take too much time), 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 go out of its band. Since the execution delay of the algorithm is constant, increasing the band widths allows the algorithm to maintain the expected result for longer periods

Two features 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 below two operating conditions ensuring a correct behavior of the algorithm.

3.1.1 Stability

We must ensure that a transfer of load resulting from the balancing protocol does not make the load of the initiating processor leave its band, leading to processor thrashing [ELZ86].

Let us call Δt_{proc} the maximum time of the execution of a procedure proc. 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 maximum received load is $q \leq m_i - n_i(t)$, $n_i(t)$ is the value of 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 $n_i(t + \Delta t_{\textsf{Protocol\_Server}}) + q < B_i$.
So the condition for a correct execution of the algorithm in this case is:
$n_i(t + \Delta t_{\textsf{Protocol\_Server}}) - n_i(t) < B_i - m_i = (B_i - b_i)/2$

For P_j **:** The maximum load transferred is $q \leq n_j(t') - m_i$.
And $n_j(t' + \Delta t_{\textsf{Agree\_Server}}) - q \leq b_i$ should be true
(If $n_j(t' + \Delta t_{\textsf{Agree\_Server}}) \leq b_j$, then P_j decrease its level).
So the condition for a correct execution of the algorithm in this case is:
$|n_j(t' + \Delta t_{\textsf{Agree\_Server}}) - n_j(t')| < m_i - b_i = (B_i - b_i)/2$

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

Symmetrically, the conditions for P_i and P_j are:

$|n_i(t + \Delta t_{\textsf{Protocol\_Source}}) - n_i(t)| < m_i - b_i = (B_i - b_i)/2$

$n_j(t' + \Delta t_{\textsf{Agree\_Source}}) - n_j(t') < B_i - m_i = (B_i - b_i)/2$

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

3.1.2 Explosions and implosions of load

We say that the algorithm handles explosion or implosion of load on a processor if the delay for changing its level when its load crosses its bounds is such that, when the new bounds are established, its load is not already out of its new band.

By contrast with the problem of oscillation of load which involves only the cooperation between two processors, explosions or implosions involve all processors since there is negotiation with all of them before changing of level.

Case $n_i \leq b_i$ **:** The time required by P_i to decrease its level e when the server algorithm fails is $\Delta t_{\textsf{Algo\_Server}}$. Since $e-1$ is its new level, P_i's bounds become then $b(e-1)$ and $B(e-1)$. If, at that time, its load is already less than $b(e-1)$ (there was an implosion of the load), then the algorithm did not manage to fill the gap of the variation of the load (it is re-executed with the new level). To avoid this situation, the delay of the variation of the load of $b(e) - b(e-1) = m(e-1) - b(e-1) = (B(e-1) - b(e-1))/2$ on P_i, must 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 verify the validity of these conditions with the simulations given in section 4.

3.1.3 Summary

Studying the limits under which a load balancing algorithm performs properly is not easy. The problem has been split into three cases:

- Small oscillations (smaller than the width of a half band) of the load on each processor cause no problem, whatever their speed, because the processors always reach a level for which these oscillations are within the associated 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 of the load. In other words, 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, and this value depends directly on the band control functions. The choice of these functions in actual implementations should follow from a tradeoff between the precision required for load balancing and the ability to handle fast load variations, taking into account the latency of the underlying communication medium between processors.

3.2 An example of implementation

The interface to the communication network is a procedure <u>send</u> for sending a message to a particular processor, and a corresponding procedure <u>receive</u> called on reception of a message. We assume the network is reliable and provides direct point to point connections between all pairs of processors.

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.

Procedures **Algo_*** activate the corresponding procedure **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 it will move its level as specified in the algorithm figure 1).

The procedures are given in figure 3 in a syntax that should be self explanatory. Procedure **TakeLoad** return the processes (i.e. the load) subtracted from the local queue, and procedure **AddLoad** adds them to the local 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);
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_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) ;
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_Server(M, D, P_i) : receive SERVER(M, D) from P_i
Notice_Source(M, D, P_i) : receive SOURCE(M, D) from P_i
Disagree_*(P_i) : send NO to P_i;
Agree_Server(D_j, P_i) : x_j:=TakeLoad(D_j); send LOAD(x_j) to P_i;
Agree_Source(D_j, P_i) :
    send OK(D_j) to P_i;
    when receive LOAD(x) from P_j
        AddLoad(x);
```

Figure 3: An implementation with point to point communications.

We call *tpm* the maximum delay in transferring a message between two processors, and processing it. In our system the migration of processes is done by a single message transfert, so the cost of the migrations is also represented by *tpm*. With N, number of processors in the system, the execution times of the procedures involved in the conditions are at most:

$$\Delta t_{\mathsf{Algo\_*}} \leq 2(N-1)tpm$$
$$\Delta t_{\mathsf{Protocol\_*}} \leq 2tpm$$
$$\Delta t_{\mathsf{Agree\_*}} \leq 2tpm$$

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 given in figures 1 and 3, together with a stochastic simulator, have been programmed in the language LCS [BGG91, BL94], which is convenient for prototyping parallel algorithms. We present and discuss here the results we obtained.

The model used for each processor is, classically ([NXG85, ELZ86, SC89] etc.) a waiting file $M/M/1$. The Poisson arrival rate, representing the creation of the processes, is $\lambda = 1/dt$ on each processor, and the process service time is exponentially distributed with rate μ.

In addition to the average time dt between two creations of processes on a processor and a relation between λ and μ, our simulator accepts as parameters the number of processors N, the delay of transmission of a message between two processors tpm, and the band control functions.

4.1 Framework

In [WM85, ELZ86, Zho88, SC89], 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 observe its ability to handle load changes. Observing stability is more difficult because the effects of instability are only visible in some circumstances which are not easy to simulate. We will therefore only consider load explosion here.

Since we want to observe the behavior of our algorithm in the worst possible conditions, as in [NXG85], we run examples 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 null 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; we compute this value 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

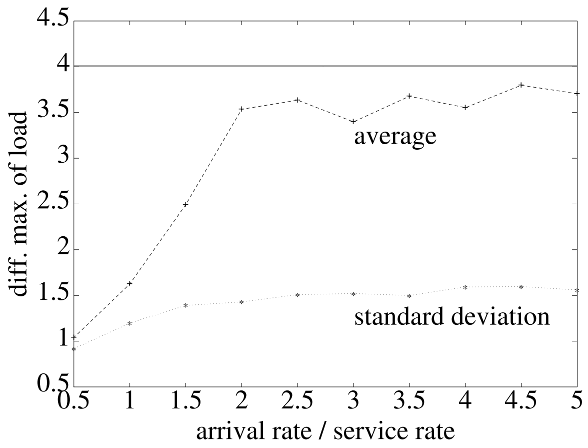
The curves in figure 4 show the result of the load balancing algorithm in a quite difficult situation: half of the processors have a null arrival rate, and on the others, the average time dt between two creations of processes is only four times greater than the time for transmitting a message tpm ($dt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

We simulated a system with four processors ($N = 4$), and band control functions giving constant width bands, with parameter $a = 2$. The theoretical result of the algorithm 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 $(B_i - b_i)/2 = 2a/2 = a = 2$ processes 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 sections 3.1.1 and 3.1.2).

Figure 4a 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[1]. Since the standard deviation of the sequences are low, the curve of their average represents faithfully the maximum difference of load for each simulation.

The abscissa in figure 4 represents the ratio between λ and μ for one of these processors. So the ratio between the total arrival rate and the total service rate is half of the numbers written. As the ratio of arrival rate to service rate increases above 2 (the load of the whole system explodes), figure 4a 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.



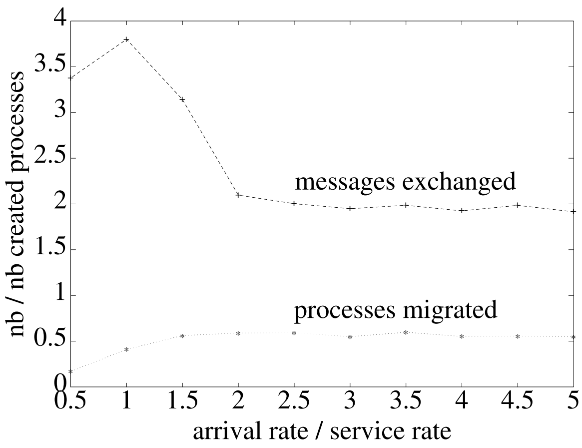


Figure 4: a) Influence of the load on the balancing. b) Cost of the algorithm.

The cost of the algorithm in term of the number of processes migrated and the number of messages exchanged, are shown in figure 4b. When the arrival rate on one processor is greater than its service rate, approximately one process in two is migrated. No unnecessary migrations are done since only half of the processors have a null arrival rate.

For a total arrival rate greater than the total service rate, 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.

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

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 $em = 0$, 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

The delay for transmitting and processing a message is an important parameter of the stability and LVH conditions. In order to study the influence of this parameter, we start from one of the previous simulations (with $\lambda/\mu = 4$), and we increase the delay for transmitting and processing the messages tpm. The new curve of the average of the differences in load between processors for different values of tpm/dt is shown in figure 5a.

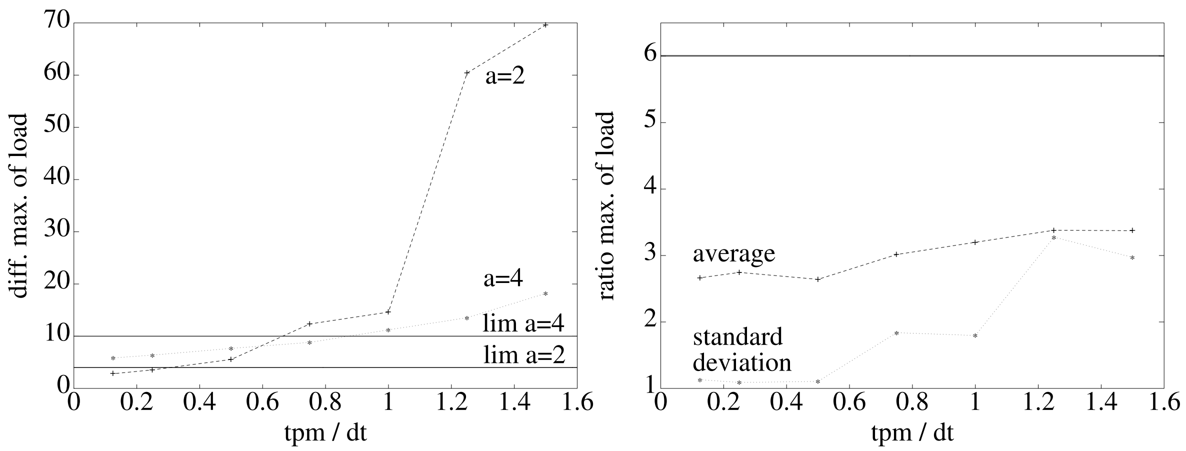


Figure 5: a) Constant width bands. b) Geometric variation of the band widths.

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 performance 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 5a. 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 is $2(N-1)tpm = 6tpm < adt = 4dt$, i.e. $tpm/dt < 0.666$. The limit found by simulation is greater than that; this is explained by the fact that it is rare to have $a = 4$ creations of processes on a processor without any completed service when $\lambda = 4\mu$.

When a is not significantly smaller than λ/μ the delay for an increase of a processes should

depend on the average rate of increase, i.e. $\lambda - \mu$. So the time for an increase of a processes is in this case $\dfrac{a}{\lambda - \mu} = \dfrac{a}{\lambda - \lambda/4} = 4adt/3$.
And the LVH condition is $6tpm < 4adt/3$, i.e. $tpm/dt < 0.888$. This value is confirmed by the simulation.

Another way to weaken the balancing constraints is to use varying width bands. Figure 5b 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 maximum ratios during a simulation 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, there is a tradeoff between precise balancing when the load is small and the ability to handle fast load changes.

5 Related work

There is a large number of papers on load sharing; the survey paper [BSS91] contains many references. But as pointed out in the introduction, our requirements make classical algorithms, often based on static thresholds on the load of each processors, unsuitable. We discuss here three papers that are closer to our work.

[CLZ92] uses two thresholds defined around the average load (so they are dynamic) and processors may perform either a sender- or a receiver-initiative migration policy. The same algorithm as [BS85] (using a timer) is used to exchange information to compute that average. Their conclusion is that in their context, a non-directly connected set of processors, local strategies (each node communicates only with its neighbors) are better than global ones in terms of cost of communications and scalability. But they conclude also that global strategies lead to fewer migrations and to a better balance.

[XH93] is interesting in two ways; the thresholds used by their sender-initiated algorithm are updated regularly to handle the load of the system, and the intervals between the exchange of information to make this update vary dynamically according to the speed of the variation of the load. In contrast to our system, the collection of information is centralized by an Information Collector located on the front-end host of a multicomputer, and their algorithm is not preemptive.

[MVZ93] studies processor allocation strategies on shared-memory parallel computers for two-level schedulers, i.e. where the operating system kernel allocates processors to multi-threaded applications, and the applications themselves schedule their threads on those processors by a co-operating set of kernel-schedulable processes, the virtual processors. Their conclusion is that these two levels should cooperate. We are in a context of non-shared memory, so the cooperation between the virtual processors to schedule the threads of an application is a problem by itself, and this is exactly what our algorithm treats, but we keep [MVZ93] in mind for further investigations.

6 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 aim 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 has its desirable properties. The simulation confirms the expectations about the behavior of 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 us to find the best possible balance of the load, taking into account the speed of the variation of the load and the latency of the 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 notion, 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 of the algorithm it is necessary to design the protocols in a way which take into account 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 can be partitioned into 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 [CLZ92].

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

[BF81] R. M. Bryant and R. A. Finkel. A stable distributed scheduling algorithm. In *2nd Int. Conf. Distributed Comput. Syst.*, pages 314–323, 1981.

[BGG91] Bernard Berthomieu, Didier Giralt, and Jean Paul Gouyon. LCS users manual. Rapport de Recherche 91226, CNRS-LAAS, Septembre 1991.

[BL94] 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.

[BS85] A. Barak and A. Shiloh. A distributed load-balancing policy for a multicomputer. *Software Practice and Experience*, 15(9):901–913, September 1985.

[BSS91] Guy Bernard, Dominique Stève, and Michel Simatic. Placement et migration de processus dans les systèmes répartis faiblement couplés. *TSI*, 10(5):375–392, 1991.

[CLZ92] Antonio Corradi, Letizia Leonardi, and Franco Zambonelli. Load balancing strategies for massively parallel architectures. *Parallel Processing Letters*, 2(2 & 3):139–148, 1992.

[ELZ86] 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.

[Le 93] 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.

[MVZ93] 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.

[NXG85] 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.

[SC89] 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.

[Sta85] John A. Stankovic. Stability and distributed scheduling algorithms. *IEEE Transactions on Sftware Engineering*, 11(10):1141–1152, October 1985.

[WM85] Yung-Terng Wang and Robert J. T. Morris. Load sharing in distributed systems. *IEEE Transactions on Computers*, C-34(3):204–217, March 1985.

[XH93] J. Xu and K. Hwang. Heuristic methods for dynamic load balancing in a message-passing multicomputer. *Journal of Parallel and Distributed Computing*, 18(1):1–13, May 1993.

[Zho88] Songnian Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering*, 14(9):1327–1341, September 1988.