Distributed synchronization of parallel programms:why and how?

Patrice Quinton, Jean-Pierre Verjus

IRIA

CENTRE DE RENNES
IRISA

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tel.: (1) 39 63 55 11

Rapports de Recherche

N° 526

DISTRIBUTED SYNCHRONIZATION OF PARALLEL PROGRAMS: WHY AND HOW?

Patrice QUINTON
Jean-Pierre VERJUS

Mai 1986

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Distributed Synchronization of Parallel Programs: Why and How ?

Publication Interne n° 292 - Avril 86 - 16 pages

International Workshop on Parallel Algorithms and Architectures,
Luminy, Marseille, France, 14 – 18 April, 1986

Patrice QUINTON
Jean – Pierre VERJUS

IRISA, Campus de Beaulieu,
35042 RENNES Cedex
FRANCE

Résumé : Au fur et à mesure que les architectures évoluent vers le parallélisme massif, les délais de communication entre processeurs deviennent prédominants. Par conséquent, la synchronisation de programmes parallèles ne peut plus être réalisée de façon centralisée. L'alternative est de faire appel à une synchronisation distribuée. Dans cet article, quelques techniques de base pour le contrôle distribué sont introduites et illustrées. L'article est orgenisé de la façon suivante. On commence par présenter un algorithme pour le calcul des valeurs propres d'une matrice tridiagonale, pour lequel on donne une première version parallèle. Les problèmes posés par cette version sont mis en évidence. Dans une seconde partie, on présente des techniques générales pour mettre en oeuvre la synchronisation distribuée, en les illustrant sur l'exemple de la gestion d'un parking. Ensuite, nous revenons à l'algorithme de départ, et nous montrons comment ces techniques peuvent être utilisées pour le mettre en oeuvre de façon distribuée.

Abstract : As computer architectures evolve towards massive parallelism, communication delays between co – operating processing units become predominant. As a consequence, the synchronization of parallel programs can no more be implemented efficiently by centralized control monitors. An alternative is to use distributed synchronization. In this paper, some basic techniques to distributed control are introduced and illustrated. The paper is organized as follows. We start from the example of the calculation of the eigenvalues of a tridiagonal matrix, for which we give a first parallel implementation. Problems arising in this implementation are shown. In a second part, general techniques for implementing distributed synchronization are presented and illustrated by the example of a parking lot management. Finally, we return to the numerical example, and we show how these techniques can be applied to obtain a distributed implementation of the algorithm.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATI
EN INFORMATIQU
(LABORAT

PAPIER RÉCUPÉRÉ ET RECYCLÉ

1 Introduction

One of the main events of these recent years concerning computer architecture is the now general acceptance that parallel architectures will predominate in the future. In this respect it is very significant that some supercomputers companies (CRAY for example) are now offering multiprocessor configurations. Moreover, there is a large number of new parallel architectures entering the market that aim at filling the gap between extremely high performance supercomputers and conventional computers. Hypercubes being offered or announced by such companies as Intel, N-Cube, Ametek or other kind of parallel architectures such as BBN Butterfly are representative of this new evolution (See Dongarra et al. (1985) for an up-to-date description of available supercomputers).

As the tendency goes towards more and more parallelism, programming becomes a central issue. Synchronizing processes that co-operate for the completion of a given task is one of the most difficult problem. When dealing with tightly coupled parallel architectures, the difficulties can be overcome by implementing a centralized control, for example by using a fast acces shared memory when available, and the synchronization tools needed (semaphores for example) are well known. However this favorable situation is not likely to occur when one wants to use a massively parallel architecture. Indeed the time necessary to access a central resource (memory, or processor) becomes non negligible, as communication between remote units takes time. Moreover, a centralized control introduces a bottleneck, as processors compete for the acces of a single resource. Finally, when the number of processors is large, reliability issues are much better addressed by distributed than by centralized resource management.

Some algorithms have properties that allow central control to be naturally avoided, for example when computations can be executed by regularly and locally connected processes. Systolic arrays or SIMD architectures can then be of advantage. However, this is far from being the general situation, and it may be useful to implement distributed control mechanisms, in order to try to overcome the problems just cited.

In this paper, we shall anticipate the evolution of architecture towards massive parallelism (e.g. hundreds or thousands of processing elements) and introduce some basic techniques underlying distributed synchronization. These techniques have been known for some time since they were first studied in the context of distributed network protocols. A good introduction to these problems is provided by André et al. (1985).

The paper is organized as follows. In section 2, we describe a very simple parallel program in order to illustrate some of the numerous problems that arise when dealing with the parallel execution of a program. Basic techniques to solve these problems are introduced in section 3 and

4 using the toy example of the management of a parking lot. In particular, we examine in section 4 the three main approaches for implementing distributed control, i.e. distributing, splitting, or duplicating the variables associated with the constraints of synchronization. In section 5, we return to the example of section 2 and propose a solution to the control of this program.

2 A (very) simple example

The following example is inspired by the paper of Lo et al. (1986). Consider the task of finding in parallel the eigenvalues of a tridiagonal matrix within an interval $[a,b]$. This problem can be solved by multisection by using the Sturm sequence property (see Golub and Van Loan, 1983, pp. 306 – 308). Basically, it consists in finding the roots of the characteristic polynomial, and the Sturm sequence property allows the number of roots within an interval to be known before solving the polynomial.

Let us consider a pool of processes, called *find_root* (see figure 1), accessing a common queue. We assume a procedure solve_root that finds the root inside an interval by bisection, and two procedures pick_interval and put_interval that access and manage the common queue.

```
process find_root;
  while not finished do begin
  pick_interval (a,b,N); {N is assumed to be non null}
    if N = 1 then
      solve_root (a,b); {find the root by bisection}
    else begin
      for k:= 0 to N – 1 do begin
            { split the interval }
        x1:=a + k(b – a)/N; x2:=a + (k + 1)(b – a)/N;
        p:=number_of_roots_of (x1,x2);
        if p ≥ 1 then put_interval (x1,x2,p)
      end {for}
    end {if N = 1}
  end {while}
end {find_root}
```

Figure 1. Process find_root.

Each process does the following job:

- pick an interval from the queue;

- if there is one root, find it by bisection;
- if there are more than one root say N, split the interval in N equal subintervals;
- for each subinterval, evaluate the number of roots p within the interval, and put it in the queue whenever $p \geqq 1$

This example may serve as an illustration of the concepts of synchronization. Basically, four questions must be asked.

Question 1: How can we be sure that the queue will be managed consistently, even if we consider that there is no limit on the storage resource ? This problem is the classical issue of access conflict management.

Question 2: How can we handle the (more realistic) case when there is a limited amount of resource storage for the queue ? To show that this problem is far from being trivial, notice that it is perfectly possible to find the roots using a sequential program that will use a memory space that is independent of the number of roots N (for example, by a bisection that tries the leftmost subinterval first). On the other hand, the method we have just described uses an amount of memory that is related to the number of roots N within $[a,b]$. In the case when N is larger than the amount of memory available, a deadlock situation may occur, as all the processes may find the queue full when trying to put a new interval (i.e. they will all be blocked on the procedure call *put_interval*). It is clear that it should be possible to regulate nicely the production of intervals by having the processes recognize this situation, and take together the appropriate action.

Question 3: Suppose that the number of processes is very large, and that communications between the processes are slow compared to the calculation time. How would it be possible to distribute the queue among clusters of processors co-operating locally, or, at least, to distribute the management of a unique queue, in order to avoid message congestion in the queue monitor ?

Question 4: When is the job terminated ? This is a difficult question, as one can see that it can only be answered by gathering information from all the processors. Note for example that a processor that finds the queue empty cannot conclude that the algorithm is finished, as another processor could add later an interval to the queue.

It is the purpose of the following sections to describe general methods that help solving these problems. In order to make our presentation easy to follow, we shall base it on the toy example of the management of a parking lot. This example is representative of the problems arising in producer-consumer schemes.

3 Basic definitions and example

In a distributed information system, there are two principal reasons for providing synchronization tools : concurrency and co-operation between processes. These interactions require the exchange of data which is regulated by means of mutual exclusion to shared variables in a centralized system, but is made through communication channels in a distributed system. In this presentation, we introduce the synchronization problem by means of a resource allocation example.

A distributed system is a set of separate sites. Each site has its own memory and there is no common memory among the sites. The sites are interconnected by communications channels. Consider a system of processes to be synchronized. The processes are considered to proceed in discrete steps, each step producing an event. The event can be local to the process and imperceptible to the rest of the system, or to the contrary it may involve the problem of synchronization just posed. The latter is termed an **observable event** or a **synchronization point**. In consideration of such events, the following definition is offered:

Synchronization is the regulation of the evolution of concurrent processes, and subsequently of the occurence of observable events, as a function of the history of events in the system of processes.

Four our abstract model, we only consider those systems in which the duration between two events can be ignored. Each process is represented by a succession of events. The logical synchronization (as opposed to real-time synchronization) of a set of processes, each of which has reached some synchronization point, consists of bringing the processes into concordance with the rules governing the behaviour of the particular system. The role of the synchronization monitor is to order the set of events produced by the concurrent processes, that is to schedule and regulate the steps of the different processes.

For example, consider the following system based on the action at a parking lot. The cars are the processes that compete to occupy parking spaces. Each of these processes can be represented by the following sequence of events (event trace):

- e – Entry. A car gains entry into the lot.
- p – Parking. A car parks in an empty space.
- d – Departure. A car leaves the lot.

Only e and d are externally observable events. For a parking lot with N spaces, a legal event trace is a sequence of e's and d's such that in any prefix of the sequence:

1. the number of *e*'s is necessarily greater than or equal to the number of *d*'s, by virtue of the model, and
2. the number of *e*'s must be kept less than or equal to the number of *d*'s plus *N*, in order not to overfill the lot.

For example, allowing $N = 3$, the trace "*e e d e e d*" is legal, but "*e e e e d*" is illegal.

3.1 The synchronization monitor

Let us consider an actual monitor which implements the abstract expressions of synchronization constraints, such as those posed in the parking lot example. The monitor overseas the occurrence of observable events, and is empowered to stop and restart processes when they reach synchronization points, so as to guarantee that at any time the effective event trace remains legal. Thus in the above example, if the monitor has produced the legal event trace "*e e e*" then it must block any process which attemps to cross synchronization point *e*, since there is no space left in the lot. It continues blocking this synchronization request until one of the three processes which has crossed point *e* requests to cross synchronization point *d*, i.e. until a space becomes available.

The synchronization monitor distinguishes among three classes of actions:

1. Request for synchronization;
2. Authorization;
3. Bookkeeping.

Thus it is the authorizations which will be represented in the event trace, and which will delimit the occurrences of observable events.

In our example, the monitor corresponds in reality to the attendant who guards the sole outlet of the lot (or perhaps to a micro-processor controlled barrier). The attendant keeps a record of all arrivals and departures, and can thereby effectively implement the synchronization of these events in accordance of the system.

3.2 Problem of perceiving and scheduling events

The perception of requests by the monitor, and the perception of authorizations by the processes, can be delayed from the actual occurrence of the observable event. This can result from propagation delays inherent in distributed systems.

In the simple example above, this delay can be illustrated as follows: even if there is only one outlet and one attendant, the latter has only partial knowledge of the state of the system. He can think that the lot is full and refuse entry to waiting cars, when in reality there may be some cars which have left their spaces on the way out, but have not reached the outlet. There is thus a time lag between the actual state of the events in the system and the perception of the state.

If the monitor is centralized on a single processor, it is possible to construct a definitive event trace of authorizations. On the contrary, if the monitor is distributed over several processors without a common clock, one must prove that the combination of event traces on the individual processors will appear to the outside observer as identical to his perception of the single overall event trace.

Thus if there are several outlets and attendants for the parking lot, each attendant becomes aware of the actions of the others only after some delay. This introduces a second kind of uncertainty due to the multiplicity of decision-making centres. An attendant can think that no more spaces remain when another attendant has just logged the departure of a car but has not yet notified the other attendants of the availability of the space. Similarly, several attendants might allocate the same space to cars waiting at their respective outlets if they have not properly co-ordinated the exchange of information among themselves.[1]

Two important conclusions are drawn from the above:

- (C1) At a given time, a process occurring at one site can have only approximate knowledge of the state of any other site.
- (C2) Any two observable events in a system can be perceived as occurring in a different order by different sites. Referring to figure 2, at site 1 the event trace *a2 a1* represents the perceived order of observable events *e1 e2*, whereas the perception at site 2 produces the event trace *b1 b2*. Without an adequate tool which can guarantee a consistent ordering among the sites, it will be difficult to reconstruct the true and legal trace of events in a distributed system.

---

[1] In the latter case, one can imagine a solution in which the attendants are linked by a telephone system. But this does not conform to the characteristics of the communication channels in a distributed system. Indeed, the delay in the transmission of a message in a distributed system is greater than the time which separates two observable events of the same process.

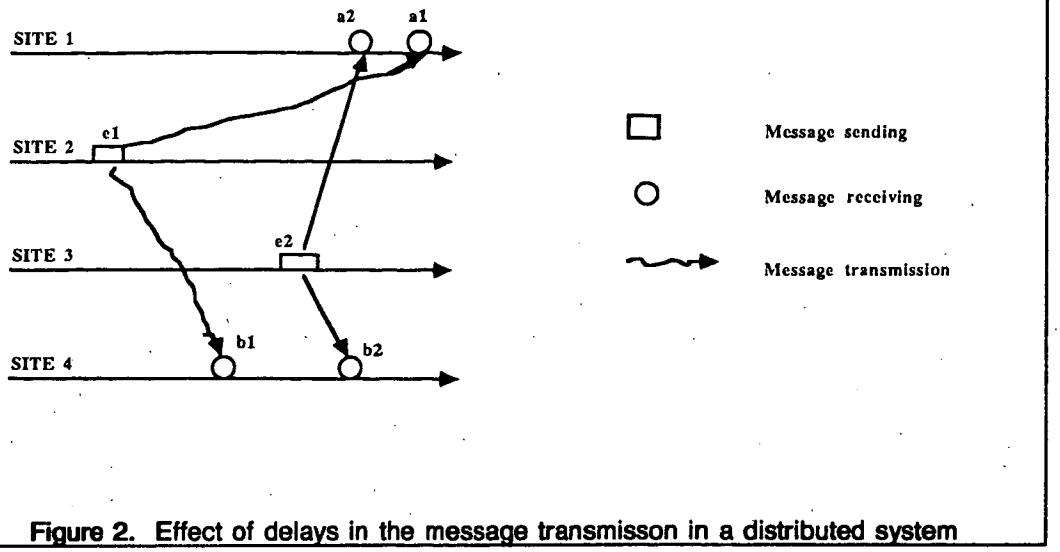

Figure 2. Effect of delays in the message transmisson in a distributed system

3.3 Consistency, Deadlock, Fairness and Priority

Finally, in order to resolve the problems of synchronization among processes, very often two complementary objectives must be guaranteed:

1. that the system will function consistently (e.g. legal usage of resources: in the example, no car is allowed into the lot if there is no space available), and that the system will prevent deadlock (e.g. two attendants will not be in the situation of waiting for each other).
2. that the system will either treat processes with equality, i.e. fairness (not to allow a car to wait for entrance while more recent arrivals are granted entry), or on the other hand that the system will establish priviledges, i.e. priority (to allow a higher priority vehicle to enter ahead of other vehicles, regardless of the order of arrival at the inlets).

4 Expression of a problem of resource allocation

If we denote by $\#e$ and $\#d$ the number of $e-$events (entries) and $d-$event (departures), respectively, that have occurred, the abstract expression for the synchronization constraints is

$$\#e - \#d \leq N \qquad (1)$$

where N is the total number of parking spaces available. Note that the condition $\#e \geqq \#d$ is satisfied by definition and requires no specific control.

To implement expression (1) we use the variables E and D to represent the values of the counters $\#e$ and $\#d$ respectively. The controller must cause the following actions to be taken at the synchronization point e and d:

- Ae: wait until $E - D < N$; $E := E + 1$
- Ad: $D := D + 1$

If several processes arrive simultaneously at a synchronization point, e.g. at e, the controller must ensure that the actions Ae are mutually consistent. This can be done by causing these actions to be performed in a mutually exclusive manner, i.e. serially; and similarly for point d.

If the system is implemented in a centralized way, with a single lane serving both an entry and an exit, there is no difficulty in implementing the control, as events e and d are observable on the same site. But if the system is distributed, this may not be the case, and abstract expression (1) can be implemented in either of two ways, as follows.

Centralized control : this is a trivial solution, consisting in bringing together onto a single site, all the variables and statements needed for synchronization. A single control process is imposed, to which all other processes at the various sites send messages. The situation is now exactly that of a centralized system, with little parallelism and reliability depending on the single privileged site, etc.

Distributed control : distributing control among p sites is equivalent to having p control processes. In the following section we consider three types of solution to this problem:

- Distributing the variables in the abstract expressions among the controllers.
- Partitioning these variables into p components.
- Replicating the variables, while maintening consistency.

4.1 Distributed variables

Suppose there are two lanes to the car park, one for entry and the other for exit. This is equivalent to stating that one controller (or attendant) records the number of entries $E = \#e$ at site 1, and another records the departures $D = \#d$ at site 2. An *a priori* condition for a car to be allowed to enter the car park is $E - D < N$. The attendant at site 1, who must impose this

conditions, knows the value of E exactly, whilst the value of D is known exactly to the attendant at site 2, who must send this value to his colleague. Because of delays of transmission, attendant 1 will at any time have only a delayed image D' of D, with $D' < D$; if therefore he checks that $E - D' < N$ is satisfied, he is certain that invariant (1) is satisfied. This implementation is a good illustration of the method of distributing monotonically-increasing counters described in (André et al., 1985).

Note: the growth of the numbers E and D can be limited by recording these modulo some sufficiently large number K. In this example, $K > N$ is suitable, giving, for the expression of the condition $0 \leqq E - D < N$ *or* $N < D - E$ where E and D are taken modulo K.

4.2 Partitioned variables

Suppose that the car park has p lanes, each allowing both entry and exit; if E_i and D_i are the total numbers of cars entering and leaving respectively, recorded by the attendant at lane i, we have:

$$E = E_0 + E_1 + \ \ldots \ + E_{p-1}$$
$$D = D_0 + D_1 + \ \ldots \ + D_{p-1} \qquad (2)$$

The method developed in section 4.1 can be applied to the processing of these $2p$ values, but it results in a very complex algorithm and a need for the transmission of very large number of messages. Alternatively, we can distribute the parking spaces among the attendants, giving each a "credit" of Y_i available spaces. The initial distribution is modified in the course of events: some attendants may find that they have to refuse entry although there are free spaces (controlled by other attendants), a drawback which can be remedied by a periodic redistribution of the credits.

When a car wishes to enter the car park through lane i, the attendant allows it to do so if $Y_i > 0$; otherwise, he waits until either a car leaves by his lane, or a message of redistribution of credits arrives from his neighbour.

4.3 Maintenance of the consistency of a shared variable

A possible way in which a cooperation between the attendants is achieved is to ensure that only one of them has access to variable $Y = N - E + D$, giving the number of spaces available, at any one time. A simple way to do this is to arrange the attendants in a virtual ring, with each linked to a predecessor from which it receives messages and to a successor to which it sends messages.

A first attempt to a solution uses the p attendants, one at each lane, and a runner. The latter goes round the ring from one attendant to the next, carrying the value Y. If, when he arrives at a lane where there is a car waiting to enter, this value is positive, the car can be allowed to enter and the value is reduced by 1. Strictly, no car should be allowed to leave by any lane unless the runner is there, so that the value of E can be increased appropriately. In fact, however, the attendant there can allow cars to leave, noting the number that do so and giving this number to the runner when he arrives. With this arrangement only one lane can provide entry at a time and the rate at which entries can be made is determined by the speed with which the runner goes round the ring. It might therefore be more efficient to designate just one particular lane as an entry point as long, of course, as it is available. Any study of a solution must therefore be made in terms of the conditions imposed by the need to strike a balance between effectiveness and reliability.

It should be pointed out that the solution just given does not go without problem. First, it does not guarantee that a waiting car will be allowed to enter, for it could happen that the runner always arrives at this lane with a zero value for Y: so there is the possibility of an **infinite wait** or **starvation**. Solutions exist to overcome this problem, but they are beyond the scope of this paper.

Secondly, the order in which cars are allowed to enter the car park is set by the runner: it is not necessarily that in which they arrive at entry points. In that sense, one can say that this solution is unfair. More generally, inconsistencies may arise due to the fact that some order in the events has to be respected. Here also, solutions exist (see André et al., 1985).

5 Solving the parallel eigenvalue algorithm

We now return to the example of section 2 and give possible answers to the questions that were posed. Figure 3 gives a new version of the program of figure 1; the only purpose of modifying our first version is to ensure that the program will never enter a deadlock, when the amount of space available for the queue is not large enough to contain N intervals, where N is the number of eigenvalues. The control of the program is done by the boolean functions *pick_interval*, *put_interval*, and the procedure *signal_process_free*. Function *put_interval* returns *true* to the process if there exists an empty space in the interval queue. Let Q be the size of this queue. Using two variables IP and IC to record respectively the number of intervals produced and consumed, the action of function *put_interval* will be:

```
function put_interval (a,b,N);
   if IP − IC = Q then return false else
   begin IP:=IP + 1; put a,b,N in the queue; return true end
```

```
process find_root;
  while pick_interval (a,b,N) do begin
    {From now on, the process is handling an interval, and
     can potentially produce new intervals.
     We shall say that the process is busy}
    x:=a + (b-a)/N;
    repeat begin
      p:=root_number_of (a,x);
      case p in
        0: {discard [a,x], keep [x,b] and do not change N}
           begin a:=x; x:=a+(b-a)/N end
        1: {resolve [a,x] by bisection, then keep [x,b] and change N}
           begin solve_root(a,x); N:=N-1;
           a:=x; x:=a+(b-a)/N end
        otherwise: {try to put [a,x] in the queue. If it is possible,
           keep [x,b] and change N; otherwise, split [a,x]}
           if put_interval(a,x,p) then begin
             N:=N-p; a:=x; x:=a+(b-a)/N
               end else x:=a+(x-a)/p;
      end {case}
    until N=0;
    signal_process_free; {Now we say that the process is free }
  end {while}
end {find_root}
```

Figure 3. New version of the Program find_root. The boolean functions pick_interval and put_interval and the procedure signal_process_free are monitors that are explained in the text.

Note that a process is never stopped when it calls *put_interval*, in order to avoid the possibility of a deadlock. Instead, the process goes on by splitting the interval. Eventually, it will find an interval containing a single root, and find it by bisection.

The action of function *pick_interval* and procedure *signal_process_free* is a little more subtle, as it must provide for both a termination condition and the access synchronization of the interval queue. When calling *pick_interval*, a process must wait until either there is an interval in the queue, or the algorithm is terminated. Of course, in the particular case of this algorithm, completion is achieved when exactly N eigenvalues are found. However, we shall use another more general condition, which does not depend on this particular knowledge. The algorithm is terminated when the following conditions are satisfied:

1. there are no more interval in the queue, and

2. none of the process is handling an interval

Therefore, if one is able to keep track of the processes that actually have an interval, it will be possible to implement a termination condition. In fact, the number of processes that have an interval is *IC*. If we denote as *D* the number of processes that become free, i.e. the number of calls to procedure *signal_process_free*, then the program is terminated when $IC - D = 0$ and $IP - IC = 0$, i.e., as $IP - D = 0$. This condition is illustrated by figure 4. Using a guarded command[2], an implementation of the functions could be:

```
boolean function pick_interval (var a,b,N) ;
begin
   do
      {guarded command: the program waits for one of the conditions
      to become true, and executes the corresponding action. Note
      that these conditions here are exclusive.}
      IP - D = 0 - > return false; {termination}
      IP - IC > 0 - >
         { there is an interval }
         begin IC: = IC + 1; pick a,b,N; return true end
   od
end

procedure signal_process_free;
begin D: = D + 1 end;
```

All methods that were described in section 4 can be used for our algorithm. They serve as a guideline for implementing any particular solution that appears desirable from the point of view of efficiency. As an example, let us only consider the case when the interval queue is handled by a single site, but controlled by distributed synchronization. Our goal is to obtain processes as asynchronous as possible. A first solution is to maintain the consistency of shared variables, using the method of subsection 4.3. A message containing these variables would then circulate round the processes, and a process would be able to change or test its value only if it has the message. This solution has the main drawback that it would introduce a large overhead if the number of processes is large.

---

2 A guarded command is a language construct introduced in order to describe the behaviour of an non deterministic program (see Hoare, 1978).

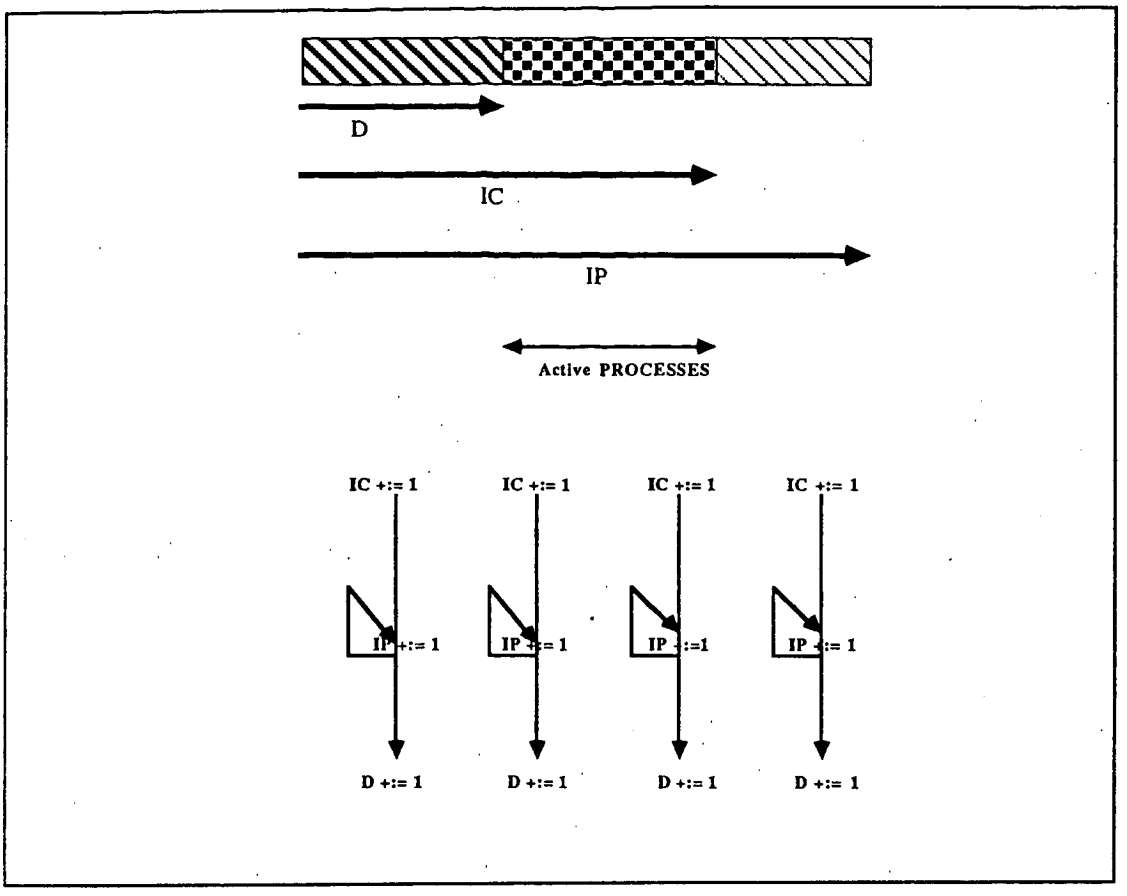

Figure 4. Illustration of the behaviour of the interval queue. At a given instant, *IP – IC* intervals are in the queue, and *IC – D* processes are active, i.e. they can still produce new intervals. The program is terminated when *IP = IC = D*

Consider first the case of function *put_interval*. We assume that there are *P* processes. The situation is very similar to that of the parking lot with *P* entries, and we can use the partitioned variables approach. Initially, each process has a credit of *Q/P* spaces in the queue, and it keeps in a variable *Y* the number of spaces available. *Y* is local to the process. Whenever the process wants to put an interval in the queue, it checks that *Y* is positive, and resuces it by 1. Then, the interval is sent to the interval queue. The procedure *put_interval* is local to the process, and is :

```
function put_interval  (a,b,N);
   if Y = 0 then return false else
   begin
      Y:=Y - 1;
      send a,b,N to the queue manager;
      return true
   end
```

Note that the process never waits when calling this procedure.

The case of *signal_process_free* and *pick_interval* is a little more tricky, as the termination condition $IP - D = 0$ is global to the system and must be perceived by the processes as it would be by an observer of the system. Our choice here is to have variables D, IC and IP, handled by the interval queue site. The procedures *signal_process_free* and *pick_interval* will therefore be executed by this site, and remotely called by messages sent by the processes. When the queue site receives a message from *put_interval*, (i.e., an interval), it increases IP by 1. When it receives a message corresponding to a call to *pick_interval*, it checks the value of IP, IC, and D, and acts in consequence by returning an interval and increasing IC by one, or by signaling the end of the program. However, the values D and IP seen by the queue site are delayed version of their actual values, due to communication delays. Let IP' and D' denote respectively the values of IP and D as they are perceived by the queue site. In order for the termination condition to be properly implemented, it is necessary that $IP' - D' = 0$ only if $IP - D = 0$. A careful examination (see figure 4) shows that this is true under the (realistic) hypothesis that messages exchanged between two sites are sent and received in the same order. If we make this assumption, IP' is always increased by a process before D', because the message sent by a process when calling *pick_interval* is guaranteed to reach the queue site before the message sent when calling *signal_process_free*. Note that the proof of this last property is far from obvious. It can be obtained by examining the properties of the language describing the sequence of events of the system.

In order to complete the description of this implementation, it remains to examine the mechanism by which processes are alloted new free spaces in the queue. This can be done either independently from the queue site, for example by having each process increasing its own Y when receiving a new interval. One can also imagine that credits are redistributed among the processes, in order to regulate the charge of the system.

To summarize, variables IP, IC, and D are handled on the interval queue site. Processes put asynchronously new intervals in the queue by checking that their free space credit is non null. They also increase D asynchronously. They may only be blocked when calling the function *pick_interval*, either for receiving a new interval, or to receive the termination signal.

6 Conclusion

Starting from an algorithm for the eigenvalue calculation, we have shown various techniques for implementing distributed synchronization of parallel programs. We have shown how these techniques can be used in the case of the eigenvalue calculation algorithm. With the advent of

highly parallel message-passing architectures, we believe that distributed synchronization will at some point become mandatory, in order to reduce the amount of messages circulated among the processors and consequently reduce the overhead due to synchronization. As we have seen, the main difficulty is to be sure, by construction, that the implementation which results is correct, especially because of the communication delays. Another issue that we did not discussed here, is the efficiency of the implementation. We believe that this issue can be better addressed if a large choice of synchronization techniques (either centralized, or distributed) are available, among which the designer will be able to choose according to efficiency criteria. Clearly, this paper is only a first attempt towards this direction.

7 References

(André, 1985) F. André, D. Herman, J.P. Verjus, *Synchronization of Parallel Programs*, The MIT Press, Cambridge, 1985.

(Bochman, 1983) G.V. Bochman, *Concepts for Distributed System Design*, Springer Verlag, 1983.

(Cornafion, 1985) Cornafion, *Distributed Computer Systems: Communication Cooperation and Consistency*, Elsevier Pub. Company, 1985.

(Dongarra, 1985) J. J. Dongarra, I. S. Duff, Advanced Architecture Computers, Technical Memo. No. 57, Argonne National Laboratory, Oct. 1985.

(Hoare, 1978) C.A.R. Hoare, "Communicating Sequential Processes," *CACM*, Vol. 21, no. 8, April 1978, pp. 666-667.

(Lo, 1986) S. S. Lo, B. Philippe, A. Sameh, A Multiprocessor Algorithm for the Symmetric Tridiagonal Eigenvalue Problem, Cedar Document, CSRD #513, Univ. of Illinois at Urbana-Champaign, 1986.

(Paker, 1983) Y. Paker, J.P. Verjus (eds), *Distributed Computing Systems: Synchronization, Control and Communication*, Academic Press, 1983.

P 284 **An overview of the GOTHIC Distributed Operating System**
Jean-Pierre Banatre, Michel Banatre, Florimond Ployette – 24 pages ; Janvier 86.

PI 285 **Optimal sensor location for detecting changes in dynamical behavior**
Michèle Basseville, Albert Benveniste, Georges Moustakides, Anne Rougée – 32 pages ; Février 86.

PI 286 **La tolérance aux fautes dans un système temps-réel à contraintes strictes**
Maryline Silly – 32 pages ; Février 86.

PI 287 **A new statistical approach for the automatic segmentation of continuous speech signals**
Régine André-Obrecht – 38 pages ; Mars 86.

PI 288 **Synthèse sur les réseaux locaux temps-réel**
Philippe Belmans – 40 pages ; Mars 86.

PI 289 **Calcul distribué d'un extrémum et du routage associé dans un réseau quelconque**
Jean-Michel Hélary, Aomar Maddi, Michel Raynal – 36 pages ; Mars 86.

PI 290 **A new matrix multiplication systolic array**
Patrice Quinton, Brigitte Joinnault, Pierrick Gachet – 12 pages ; Avril 86.

PI 291 **Une introduction à quelques techniques du contrôle distribué à travers un exemple**
Noël Plouzeau, Michel Raynal, Jean-Pierre Verjus – 22 pages ; Avril 86.

PI 292 **Distributed Synchronization of Parallel Programs : why and how ?**
Patrice Quinton, Jean-Pierre Verjus – 16 pages ; Avril 86.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique