# Concurrent programming and numerical applications

Jocelyne Erhel

INRIA

CENTRE DE ROCQUENCOURT

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 Techniques

N° 67

# CONCURRENT PROGRAMMING AND NUMERICAL APPLICATIONS

Jocelyne ERHEL

Mars 1986

# CONCURRENT PROGRAMMING AND NUMERICAL APPLICATIONS

Jocelyne ERHEL

## Abstract

This lecture gives some outlines about problems of concurrent programming. How to express parallelism and communication between processes require adequate languages or notations.

Primitives such as monitors are used to specify clearly synchronizations between tasks. Overhead due to the scheduling of concurrency must be limited to obtain good speed. Experiments on Cray-XMP show how to exhibit parallelism in certain numerical applications.

## Résumé

Ce cours a pour but d'exposer les problèmes relatifs à la programmation multitâches. L'expression du parallélisme et des communications entre processus nécessite un langage ou des notations adéquates.

Des primitives telles que les moniteurs permettent de spécifier clairement les synchronisations entre tâches. Il faut limiter les pertes de temps dues à la gestion du parallélisme pour obtenir une accélération satisfaisante. Des expériences sur Cray-XMP montrent comment paralléliser certaines applications numériques.

## INTRODUCTION

A lot of scientific fields make use of numerical simulations or computations. Aerodynamics, meteorology, physics and so on deal with a great amount of data objects and require large memories as well as fast execution.

Parallelism becomes an important key to design very fast computers. Pipeline was first greatly exploited and the scientific community is now used with the notions of vector programming. The next generation of super-computers seems to take advantage of multiprocessor configurations.

Some multi-vector processors are already used by scientific programmer, or announced for the next years. A lot of projects are studied in universities of research institutes. Section 1 will describe briefly three examples of multiprocessor systems.

Concurrent programming is no longer reserved to operating system designers but now concerns also numerical programmers. New notations are necessary to express parallel computations and required communications ; this is the object of section 2. Section 3 is devoted to the problems related to the specification of simultaneous execution. Synchronization mechanisms using shared variables are discussed in section 4, and some examples of high level synchronization tools are given ; whereas message passing primitives are briefly treated in section 5.

Since speed of execution is the main goal of paralllel computing, the measure of performances is crucial for the programmer. A well understanding of the factors to take into account is required to get maximal speed-ups. Some aspects of the question are discussed in section 6.

Last but not least, examples of numerical algorithms and results are described in section 7.

## 1 - Multiprocessor architectures

Schematically, a multiprocessor consists of several processors, each of them having arithmetic units and control unit, with may be program memory. Each processor is independant from the others and can proceed its own program at its own rate. Asynchronous and concurrent execution is the rule to keep in mind.

In most cases, a global memory is shared by the processors, which access it through an interconnection network. Local memories can be added to each processor. (Figure 1)

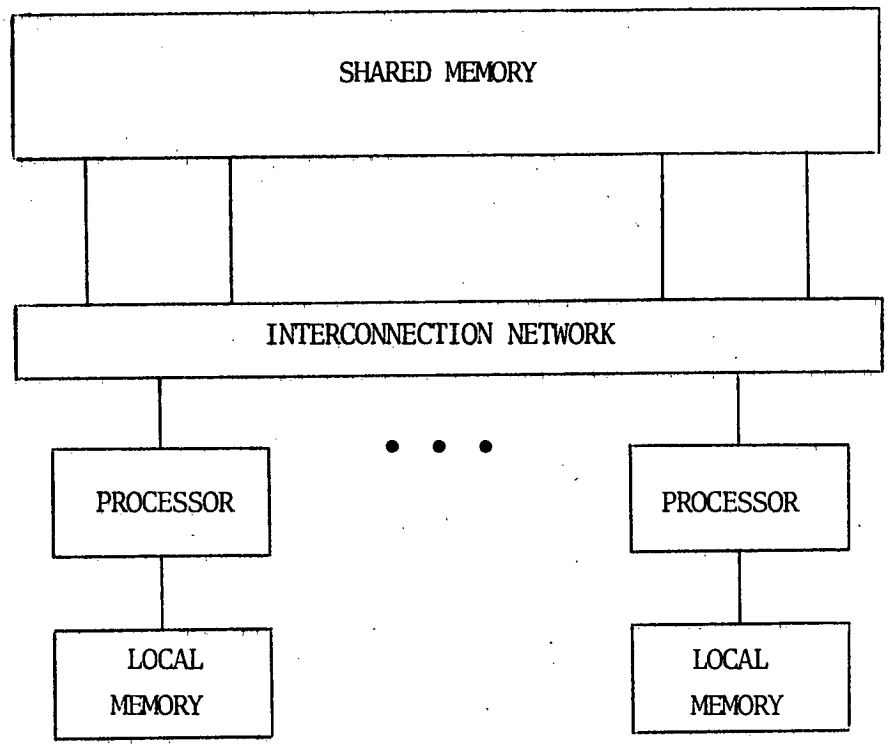



Figure 1

Communication between processors can be achieved through the shared memory if it exists or by means of message passing. The design of a parallel language depends of course on the type of communication.

The scheduling of the parallel components can be completely distributed among the processors or a supervisor can control the parallel flows. Once more, software development is heavily linked to the scheduling policy.

Existing multiprocessors make use of one or several of these concepts, in general in a hierarchical manner. We will describe briefly the Cray-XMP [1], the ETA$^{10}$ [2], and the Cedar Project [3].

The Cray-XMP computers consists of two or four vector processors with high-bandwith shared memory accessed by a one or two-level cross-bar network. Interprocessor communication tools are, in addition to shared memory, a shared real-time clock, shared semaphores, shared registers, shared I/O channels, and finally inter-CPU interrupt. (Figure 2)

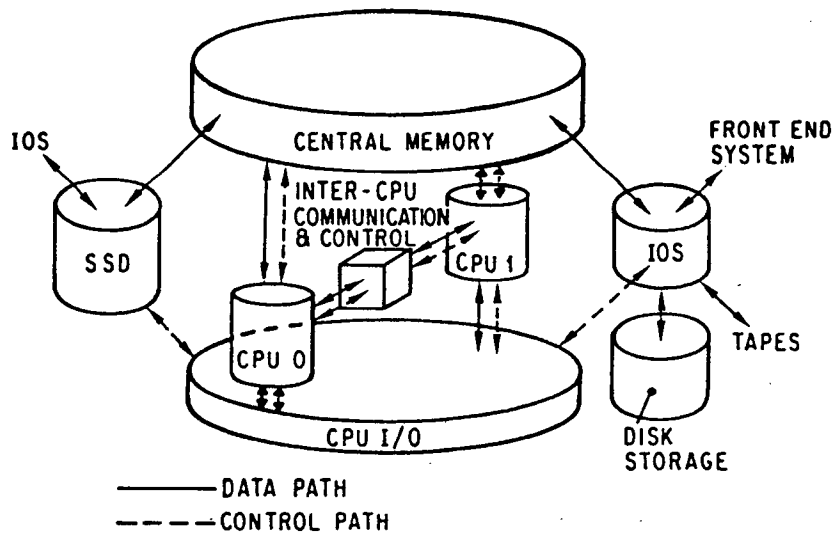



Figure 2 : Cray XMP-2 system organization

The Cray Operating System does not support a new parallel language but rather a multitasking library directly callable from Fortran codes [4]. It will be described in details in next section. The Cray-XMP is already available. Cray-2 is a slightly different four-processor system with shared memory, and Cray-3 will contain sixteen parallel processors.

. The ETA$^{10}$ computer, designed by ETA systems, is also a multiprocessor system with up to 8 processors, and has been announced for 1986. Each processor has a local high-bandwith memory, and a direct access to a very large shared memory. Each processor has a scalar unit and a vector unit. (Figure 3)

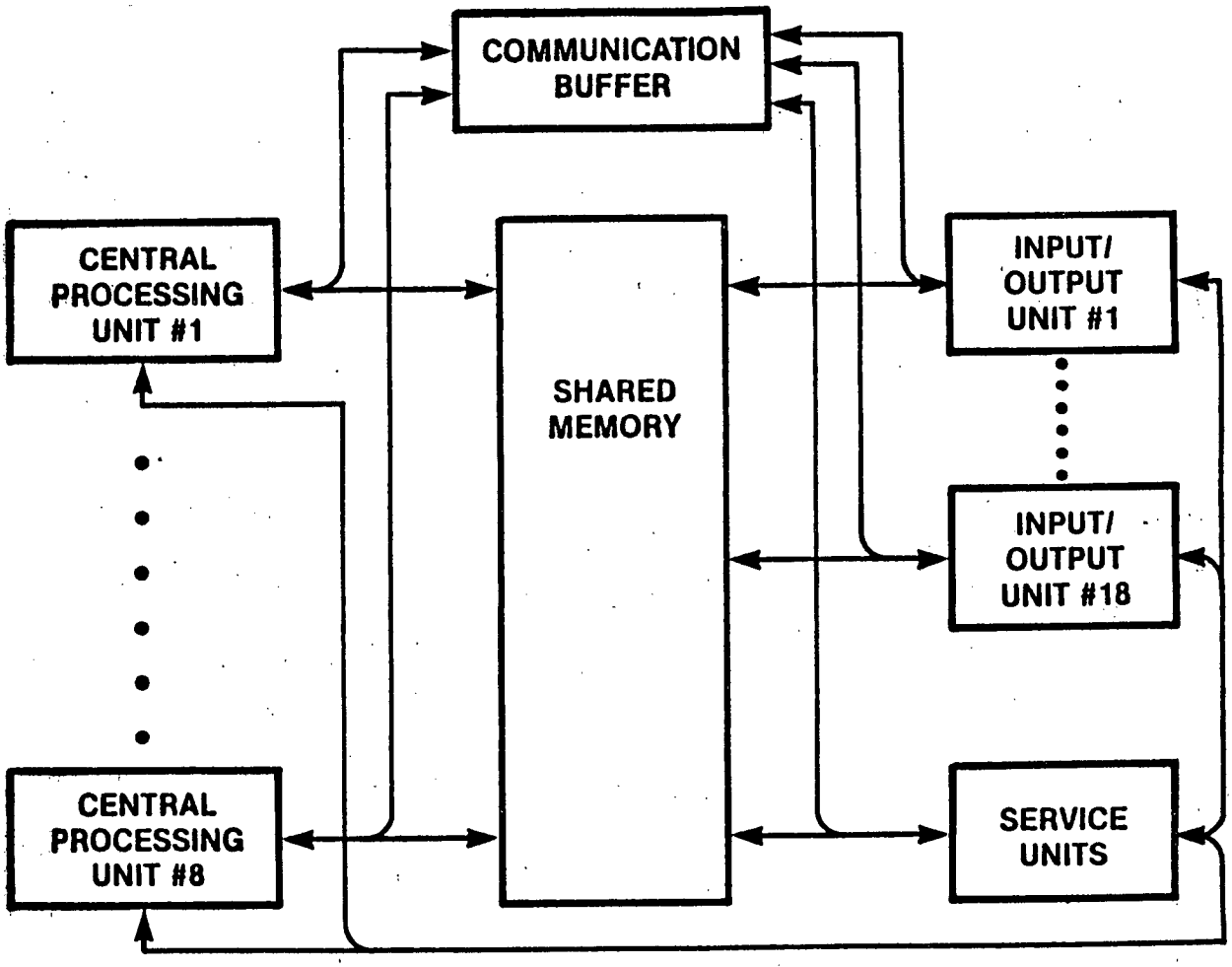



Figure 3

The Cedar project is developed at the University of Illinois at Urbana-Champaign. The architecture introduces the notion of a cluster of processors which is the "smallest execution unit in the Cedar machine". Each processor has its own control unit and a local memory. It can access other local memories of the cluster through a local network (crossbar switch) and the global shared memory through a global network (omega network). The global control unit controls the macro data flow. The program is viewed as a directed flow graph, the nodes of which are compound functions executed by processor clusters. (Figure 4)

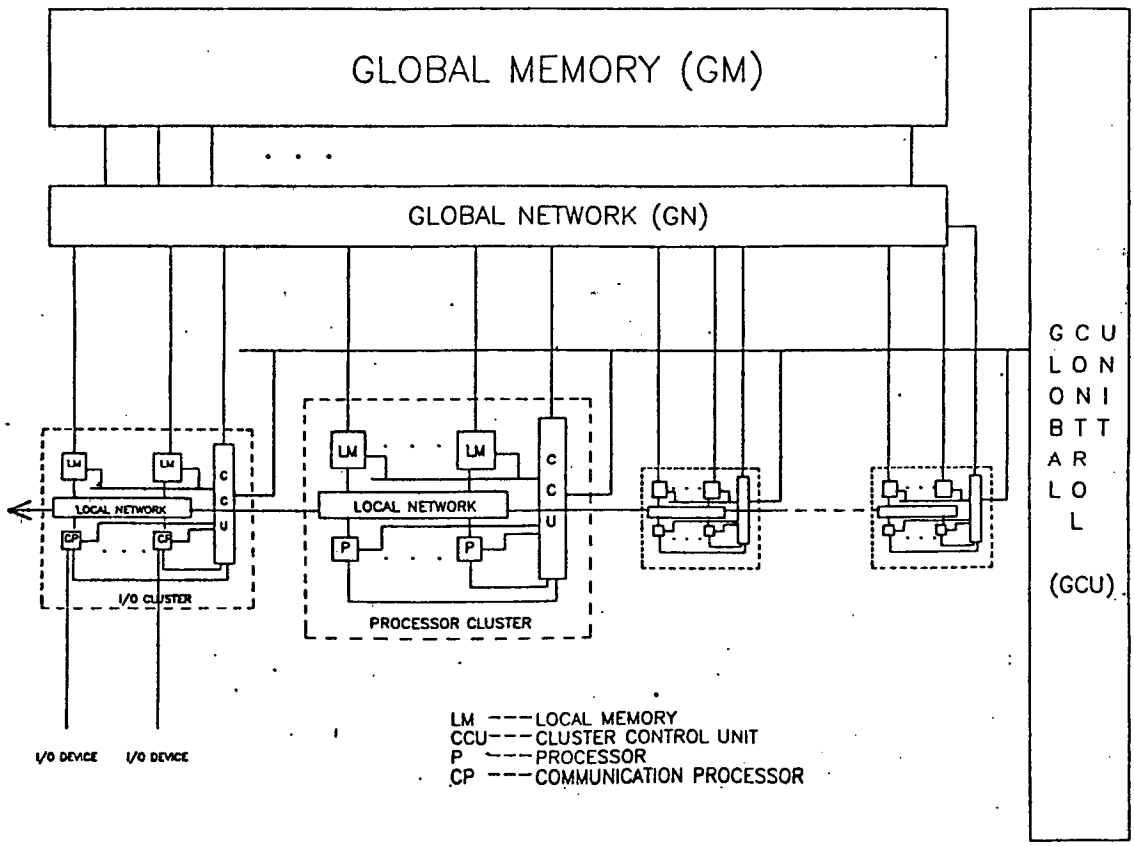



Figure 4 : Overall system diagram

## 2 - Parallel programming

New programming notations are necessary to express the parallelism of the applications to be run on multiprocessors. Concurrent programming is used in many areas such as design of operating systems, management of large database systems, control of real-time embedded systems. We will focus our attention to parallel scientific computations which may require some slightly different concepts.

The first point to design a concurrent program is to describe the parallel execution, by defining processes which are supposed to run simultaneously.

Those processes must communicate in order to exchange data or results and to cooperate to the same application. The kind of communication may depend on the architecture of the multiprocessor devoted to the execution. We will discuss first shared memory systems and primitives based on shared variables, and then briefly message passing primitives.

It is very important to keep in mind that the order of execution of the parallel processes is undeterministic with time. They are executed asynchronously at different speeds and no assumption must be made about duration or completion of a process.

But it is sometimes necessary to impose a precise order of some events. For example, a process updates a data used by another, which must wait until this data is made available. Synchronization mechanisms ensures some ordering to satisfy such constraints.

Because the discussion of such concepts need some notations, we will describe the multitasking library used on Cray Operating Systems. Hopefully the use of this language will not suppress the general interest of the examples. A survey of concurrent programming tools is reported in [5].

## 3 - Specification of parallelism

Some parallel languages propose a specific control language to express parallelism and synchronization. Processes are then viewed as black boxes without interferences with the exterior and all communications are achieved through the in- and out-data [6,7].

We prefer to define parallel processes which may cooperate at any point of execution by means of specialized primitives. This method seems to us more powerful and more flexible.

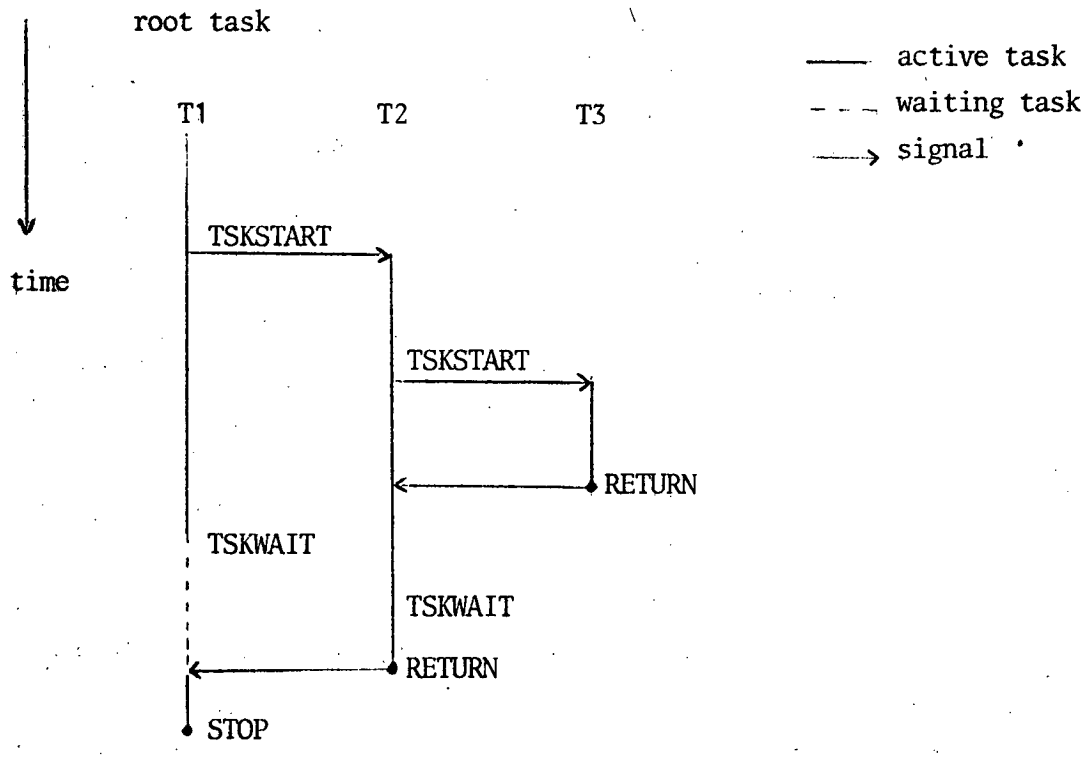
A sequential program contains statements executed in sequential order. A concurrent program contains processes executed independently and asynchronoulsy. The statements inside a process are executed in sequential order.

On Cray systems, processes are rather called tasks. An initial root task is created by the system when running a job. Then the user must express explicitly in the program (written always in Fortran) the activation of parallel tasks. A task is a subroutine, but when it is invoked, it may proceed simultaneously with the calling task. Another routine is provided to wait for the completion of a task. Those primitives are respectively TSKSTART and TSKWAIT.

This construction is similar to the FORK and JOIN statements [8], and statements used in UNIX operating system [9].

It is very powerful and permits to express a lot of parallel graphs. It is also well suited to dynamic task creation or multiple activations of the same code. Numerical applications make an extensive use of this mechanism, to do concurrently the same computation on different data sets.

On the other side, these routines must be used in a disciplined manner to understand the concurrent program execution. In figure 5 are shown some parallel graphs easily described by routines TSKSTART and TSKWAIT.


root task
active task
waiting task
signal
T1
T2
T3
time
TSKSTART
TSKSTART
RETURN
TSKWAIT
TSKWAIT
RETURN
STOP

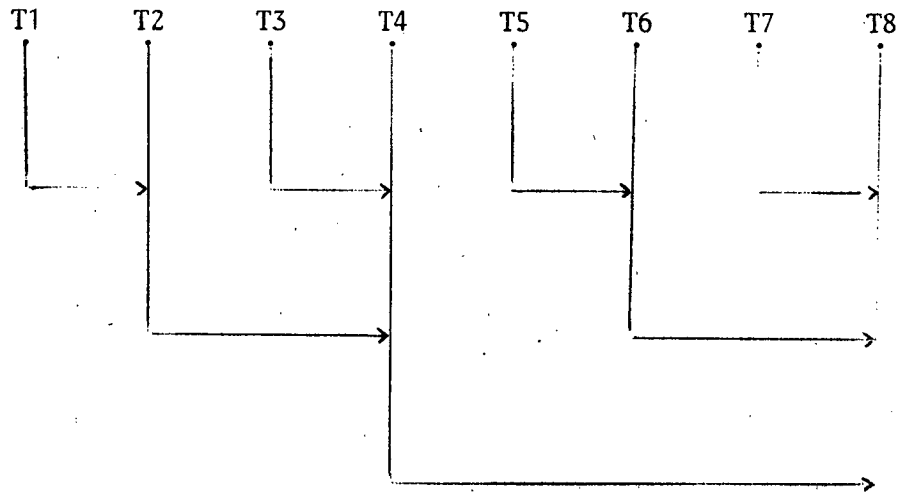

T1
T2
T3
T4
T5
T6
T7
T8

Figure 5

## 4 - Synchronization with shared variables

### 4.1 - Scope of variables

On Cray multiprocessor systems, processors share a common memory, and concurrent tasks communicate by means of shared variables.

The scope of a variable is the region of a program in which it is defined and can be referenced. In a parallel environment, two kinds of variables must be carefully distinguished :

- the local variables of a task are accessible only by that task, and are guaranteed only for the lifetime of this task.
- a special case of local variables has been introduced by Cray to define variables common to the subroutines of a task but still local to that task. This is called a TASK COMMON.
- the global variables are stored in COMMON blocks and may be accessible by several tasks. They are guaranteed for the lifetime of the entire program.

The variables worked on by more than one task or used for communication between tasks must be included in common data.

### 4.2 - Mutual exclusion problem

Let us give first an exemple to illustrate the problem. Suppose x is a shared variable common to tasks T1 et T2, and initialized to 0. If both tasks update this variable, which will be the final value of x ?

| T1 | T2 | result |
|---|---|---|
| x:=x+1 | x:=x+2 | x=1/2/3 ? |

Figure 6

This underterministic result comes from interleaving of the sequences of atomic actions generated by the tasks (for example, (i) load the value of x from the memory ; (ii) perform the addition in registers ; (iii) store the new value of x back to memory).

To avoid this problem, the assignments statements must be made indivisible, by controlling the ordering of the atomic actions. Synchronization must enforce restrictions on possible interleavings.

More generally, operations performed on a shared data object must be treated as indivisible, in order to get a meaningful result.

A sequence of statements which must be executed as an indivisible operation is called a CRITICAL SECTION. Critical sections are executed in MUTUAL EXCLUSION : at any time, only one process may execute a critical section, while other processes must wait for gaining access to their corresponding critical section.

Figure 7 shows how to ensure a proper updating of the shared variable x by including critical sections. The order of execution of the two statements is not known but they are indivisible and executed in mutual exclusion, one after the other.

| T1 | T2 | Result |
|---|---|---|
| Enter Critical Section | Enter Critical Section | |
| x:=x+1 | x:=x+2 | x=3 |
| Exit Critical Section | Exit Critical Section | |

Figure 7

The Cray multitasking library provides a mechanism to enforce Critical Sections. Special variables, called LOCKS, have two states : ON and OFF. They are represented by integer variables on which perform several primitives.

A critical section is defined by associating a lock to it. To enter the critical section, the lock must be set ON. If the lock is already ON, the task waits until it becomes OFF and then sets the lock ON. To exit the critical section simply means to set the lock OFF. Because the lock remains ON during execution of the critical section, mutual exclusion is guaranteed. Of course, hardware mechanisms provide indivisibility of lock primitives !

The precedent example can be now expressed by using locks, as shown on figure 8 .

```
          T1                              T2
   COMMON/DATA/x,LKX               COMMON/DATA/x,LKX
C enter critical section        C enter critical section
   CALL LOCKON(LKX)                CALL LOCKON(LKX)
        x:=x+1                          x:=x+2
C  exit critical section         C  exit critical section
   CALL LOCKOFF(LKX)               CALL LOCKOFF(LKX)
```

Figure 8

The lock variables may be viewed as binary semaphores, and the LOCKON and LOCKOFF primitives correspond respectively to the P and V primitives [10].

The synchronization is implemented on Cray by using queues. If a lock is already ON, the task is placed on a waiting list, and its execution is suspended. The LOCKOFF primitive removes the first task waiting for that lock which then can resume execution. This mechanism avoids starvation as could happen when using busy-waiting ; any task will be eventually reactivated after a finite time ; no task will wait for ever.

### 4.3 - Condition synchronization

In some situations, a shared data must be updated by a task before another task can operate on it (Figure 9 ).

```
Task T1          Task T2
x:= ...          y:= f(x)
```

Figure 9

Task T2 must be delayed until task T1 has performed the assignment statement [x:= ... ].

Condition synchronization refers to this type of synchronization, where a task must wait for a special event to occur before resuming execution ; this event should be eventually signalled by another task. The order of execution of the tasks is controlled and is partially imposed by introducing condition synchronization points.

Wait and Signal operations are included in task codes to ensure synchronization (Figure 10).

```
        T1                  T2
x:= ...             WAIT condition
SIGNAL condition    y:=f(x)
```

Figure 10

The Cray multitasking library provides EVENT variables and primitives operating on them to express condition synchronization. An event variable has two states : POSTED and CLEARED. A task can POST an event, CLEAR it, or WAIT for an event to be posted.

At any condition is associated an event. The condition is signalled when the event is posted. Figure 11 illustrates a possible use of an event.

```
          T1                        T2
COMMON/DATA/x,IEVX        COMMON/DATA/x,IEVX
x= ...                    C wait
C signal                    CALL EVWAIT(IEVX)
  CALL EVPOST(IEVX)         CALL EVCLEAR(IEVX)
                            y=f(x)
```

Figure 11

An event remains posted unless it is explicitely cleared. It is in general safe to clear an event immediately after having waited for it. The event mechanism is very powerful but requires a disciplined coding to use it correctly. In many occasions, two events must be used, one for signalling a condition, and the other for acknowledging the

reception of the signal. We will come back later to this problem.

If an event is not yet posted, a task executing the primitive evwait is put on a waiting-list. The evpost routine removes all tasks waiting for that event. No starvation occur since no task remains in the queue.

4.4 - Deadlock problem

Synchronization tools are necessary to ensure exclusive access to shared data objects and to enforce some constraints on the order of execution. But they must be used carefully and in a proper manner, else unintented situations would arise.

A deadlock situation occur when all participating tasks are waiting for some reason. Because no task is still active, no task can be awakened. They are all blocked for ever. Of course, a safe code must prevent from this problem ; it must be proved that at any time at least one task is active.

Some deadlock situations are easily recognized and avoided.

First of all, every call to LOCKON must be followed by a call to LOCKOFF, at least in most cases. Similarly, a call to EVWAIT must be balanced by a call to EVPOST in another task. An event must remain posted until all tasks waiting on it have resumed execution, although it must be cleared for a new use (in a loop for example).

Secondly, the use of nested Critical Sections must be carefully organized, as illustrated by figure 12 :

| T1 | T2 |
|---|---|
| enter C.S. 1 | enter C.S. 2 |
| enter C.S.2 | enter C.S. 1 |
| <code> | <code> |
| exit C.S. 2 | exit C.S. 1 |
| exit C.S. 1 | exit C.S. 2 |

Figure 12

It may happen that both tasks enter simultaneously Critical Sections 1 and 2, and then wait -always simultaneously !- for entering Critical Sections 2 and 1. That is unfortunately a deadlock situation.

A simple rule, to get free of deadlock, is to nest the Critical Sections everywhere in the same order.

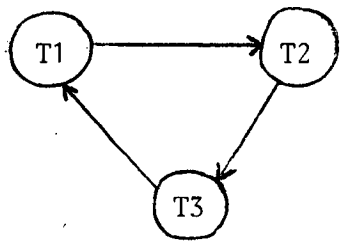
Thirdly, it seems very dangerous -although it may be correct- to wait for a condition inside a critical section ; in many cases, the condition is signalled after or inside the same critical section, as illustrated by figure 13 :

| T1 | T2 |
|---|---|
| enter CS | enter CS |
| signal | wait |
| exit CS | exit CS |

Figure 13

Both tasks are blocked if waiting on Critical Section access and on signalling of condition.

Deadlock situations are even more common and more difficult to avoid with more than two tasks. Any cycle in the synchronizations is a source of deadlock.




The arrows indicate a condition synchronization.

Figure 14

The Cray Operating System recognizes deadlock situations only when all tasks are blocked. But it may happen that a partial deadlock occured first between a subset of the tasks, finally degenerating in an overall stop of all tasks. Debugging of such problems is not easy,

all the more that they are not reproducible since they depend on the order of execution.

Last but not least, the synchronization mechanism is not always correctly implemented, even if some tests seems to work well.

It is much more difficult to debug a parallel code than a sequential one because different runs may give different results and some errors may be masked for a long time before appearing in a new environment. Also the source of an error is not well defined, it may be located several statements before it is discovered or even come from another task. Therefore high-level constructs and modularity are recommended.

### 4.5 - Monitors

A monitor is a high-level programming tool useful to deal with shared resources. It facilitates prove of correctness and debug.

A monitor encapsulates both a resource definition and operations that manipulate it [11,12]. It can be viewed as a module and allow to ignore the details of its implementation when using it.

A monitor consists of global shared variables, procedures which implement operations on them, and some initialization code executed once before any monitor invocation.

Any procedure in a monitor is executed in mutual exclusion, in order to protect access to its global variables.

Wait and signal operations realize condition synchronization in monitors. The wait operation causes the invoker to relinquish its control of monitor if it is blocked ; the signal invoker is also suspended if another process is blocked on the condition.

Monitors are implemented on Cray using locks and events, respectively to ensure mutual exclusion and condition synchronization. The variables of the monitor are stored in a common block, as well as lock and event variables used inside the monitor.

Each procedure begins with a call to LOCKON and ends with a call to LOCKOFF. Any call to EVWAIT is preceeded by a call to LOCKOFF, and in general is followed by a call to EVCLEAR, and any call to EVPOST is followed by a call to LOCKOFF. In this way, a monitor can be proved to be correct and deadlock free. The modularity and the debug are greatly enhanced.

We will give now some examples of monitors, which have been implemented on Cray-XMP.

4.6 - Fork and Join monitor

In many numerical applications, a parallel program contains a loop, the body of which begins by concurrent execution of several tasks (FORK) and ends with a global synchronization of all tasks (JOIN).

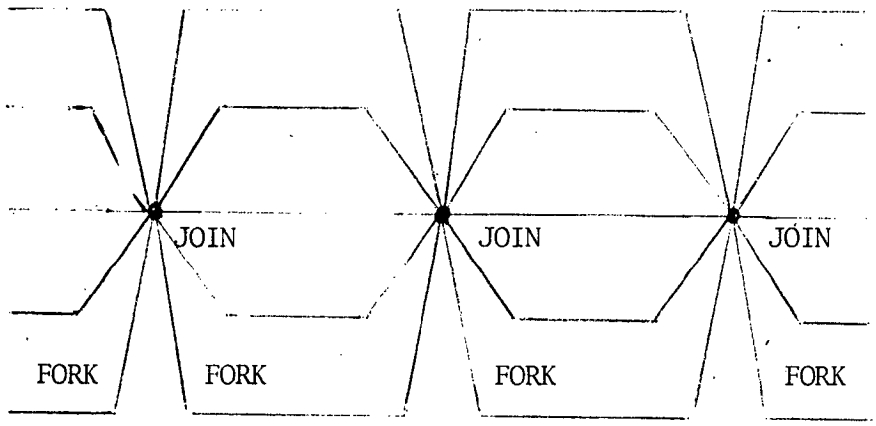



Figure 15

It could be implemented by means of TSKSTART and TSKWAIT routines ; but this solution would be very expensive in terms of time of execution. It is much better to implement the monitor described in figure 16 .

The trick here is the use of two events in flip/flop. It is necessary to keep the event associated to the present join posted until the next join and ensure the event associated to the next event is cleared.

```
c   Monitor fork and join

c   monitor variables

c   idlock  = lock of the monitor
c   idevent = events of the monitor
c   fliop   = flip/flop on the events
c   nwait   = number of waiting tasks
c   nproc   = number of tasks to be joined

c   monitor procedure

        subroutine forkjoin
        common/mlbx/idlock,idevent(2),fliop,nwait,nproc
        integer fliop
c   enter the monitor
                call lockon (idlock)
                nwait = nwait + 1
                if (nwait.lt.nproc)then
C   exit the monitor and wait for the last task
                locflp = fliop
                call lockoff (idlock)
                call evwait (idevent (locflp))
                else
c   last task : wake up the others, change the flip/flop
c   and exit the monitor
                nwait = 0
                call evpost (idevent (fliop))
                fliop = 3 - fliop
                call evclear (idevent (fliop))
                call lockoff (idlock)
                endif
c   end
        return
        end

c   monitor intialization

        subroutine mlbxinit (ntask)
        common/mlbx/idlock,idevent(2),fliop,nwait,nproc
        integer fliop

        call lockasgn (idlock)
        call evasgn (idevent (1))
        call evasgn (idevent (2))
        nwait = 0
        fliop = 1
        nproc = ntask
        return
        end
```

Figure 16

## 4.7 - Rendez-vous monitor

The rendez-vous concept has been introduced in concurrent languages such as CSP [ 13 ] and Ada [ 14 ]. Two tasks which execute a rendez-vous know exactly the status of the other. They meet for some time at a precise point of execution and then go their separate way.

In general, one task is waiting for the other to awaken it, and after some computations, signals back the end of the rendez-vous.

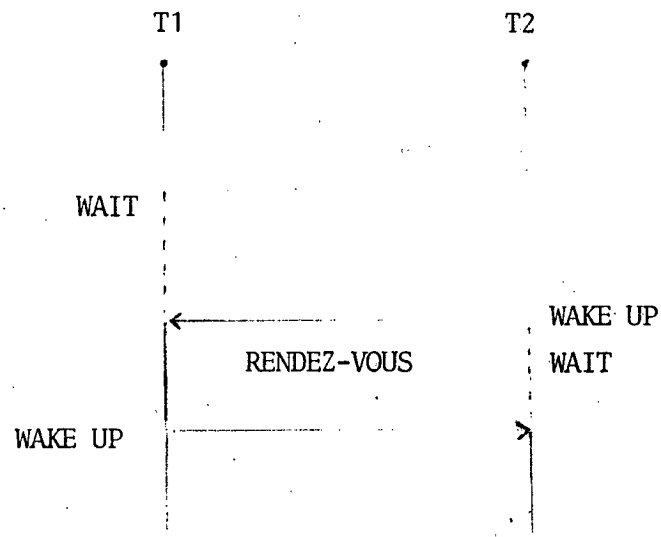



Figure 17

The monitor implemented on Cray to realize a rendez-vous is special in the sense it uses only event variables. One event is used to wakeup and the event for the back. This va-et-vient and the calls to evclear allows safe repeated rendez-vous. (Figure 18)

```
c   rendez-vous monitor

c   monitor variables

c   idev(1:2)    = events for rendez-vous
c   idev(1)      = posted by sender and waited by receiver
c   idev(2)      = posted back by receiver and waited by sender

c   monitor procedure wake up

        subroutine wake up
        common/mbrv/idev(2)

c   post the send
        call evpost (idev(1))
c   wait for the back
        call evwait (idev(2))
        call evclear (idev(2))
c   end
        return
        end

c   monitor procedure wait

        subroutine wait
        common/mbrv/idev(2)

c   wait for the send
        call evwait (idev(1))
        call evclear (idev(1))
c   post the back
        call evpost (idev(2))
c   end
        return
        end

c   monitor intialization

        subroutine mbrvinit
        common/mbrv/idev(2)

        call evasgn (idev(1))
        call evasgn (idev(2))
        return
        end
```

Figure 18

## 4.8 - DOALL monitor

A doall means that all iterations of a loop are independent, hence can proceed simultaneously. Even if this loop is not vectorizable, it can be multitasked.

Of course, the program could activate one task per iteration. But this solution is too costly and the resulted overhead would decrease dramatically the speed-up. (cf. next section).

It is better to define a priori a fixed small number of tasks (for example the number of processors) and to assign them a group of iterations. If the workload of the iterations is almost constant, then static partition is enough. But if it is unknown, dynamic partition will give a better speed-up.

The monitor implemented on Cray is performed by each task to look for a new index of iteration (figure 19). If the overhead is too high compared with the effective time of computation, the monitor would allocate small groups of iterations instead of only one.

```
c   monitor doall

c   monitor variables

c   lkiter = lock of the monitor
c   iter   = current number of iteration

c   monitor procedure

        subroutine getiter (myiter)
        common/doal/lkiter,iter
c   enter the monitor

        call lockon (lkiter)
c   copy iter and increment it
                myiter = iter
                iter : iter + 1
c   exit the monitor
        call lockoff (lkiter)
c   end
        return
        end

c   monitor initialization

        subroutine doalinit
        common/doal/lkiter,iter

        call lockasgn (lkiter)
        iter = 1
        return
        end
```

Figure 19

## 5 - Message passing primitives

On distributed systems without shared data, processors communicate through messages. Languages such as CSP [13] and Ada [14] are based on message passing.

Communication is accomplished throughout the contents of messages. A process receives values sent by another one. Synchronization is achieved by the order imposed upon the sending and the reception of the same message.

General commands are defined to send and receive a message. The semantics of the primitives must specify how to adress the source and the destination and how to synchronize the message passing.

### 5.1 - Channels of communication

A communication channel consists of a source and a destination.

Direct naming used in CSP is the simplest mechanism : source and destination are names of processes. But it is not always powerful enough and other schemes have been defined to express complicated interactions.

Instead of adressing directly processes, primitives use maibloxes to send and receive messages. The implementation is more tricky but this mechanism is well suited for producer/consumer interactions.

Static or dynamic naming is also an important issue of the design of the language.

### 5.2 - Synchronization

In general, the reception of a message is blocking because the process is delayed until a message has effectively been sent. But some primitives can be added to test without blocking if a message is received.

If send is never blocking, message passing is said asynchronous. At the other extremety, synchronous message passing occurs when send is blocking until a corresponding receive is executed. Both processes are then synchronized and this is a kind of rendez-vous. Intermediate solution is buffered message passing where only a finite number of messages can be sent without blocking.

## 6 - Performances of parallel programs

A crucial point of many numerical applications is their time of execution. Whereas weather forecasting is an evident illustration of this rule, it is not the only domain where such constraints are imposed on the time of execution. Furthermore, the concurrent development of very large memories allows large scale simulations which require a large amount of operations, hence fast computing capabilities.

The use of parallel computers can improve greatly the performances,

although several factors slow down the execution. It is very important to get an idea of the improvements achievable with parallelism.

In general, the user compares times of execution of a sequential program and its parallel version. The notion of speed-up has been defined to measure performances of parallel programs. The speed-up is the quotient of the sequential time of execution over the parallel one. Well, that is fine, but what is the sequential time ? To be fair, it would be the "best" sequential time, but what does it mean ? So, say that it is the best measured sequential time. This precision is important because parallel versions may be quite different from the initial one and work worse in a sequential environment, sothat it is not fair to compare sequential and parallel execution of the same code.

The speed-up divided by the number of processors is called the efficiency and measures the actual use of the processors. It is often expressed in percentage and the aim is of course to get as near as possible from the 100 % barrier.

The first factor influencing the speed-up is the so-called "load balancing". It is evident that an equal amount of work should be done by each processor, in order to reduce inactive time by keeping all processors active. The figure 20 illustrates this remark.

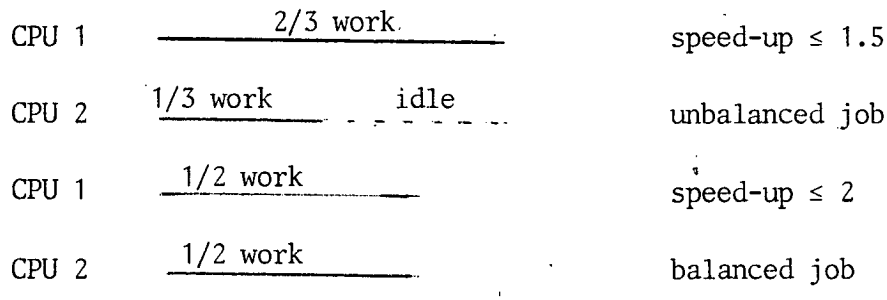



Figure 20

But what is easy to remark is not so easy to realize ! In some cases, it is possible to define a static partition of the work. Concurrent tasks are defined with similar workloads, when the time of execution can be known a priori, and parallelism can be statically expressed. Load balancing is guaranteed before execution by a partition into tasks of same duration.

But in general pieces of a program have unknown workloads. A dynamic partition is necessary to keep the tasks busy, by looking for and executing the next piece of work. The workload of each piece of work must be great enough to compensate the overhead due to the management of the dynamic partition.

This guides us to the second factor influencing the time of execution, namely the so-called "task granularity". We have seen that first concurrent tasks must be activated, then they must communicate and synchronize them. So the time of execution of a task must be divided into the effective computing time and the overhead due to task management and synchronization (waiting time on a lock or an event for example).

Even if the work is well balanced, we have the following relation :

$$Tp > T1/p + 0$$

where Tp is the parallel time on p processors,
T1 is the sequential time,
and 0 is the overhead

so that $$Sp < p/(1 + 0 * p/T1)$$

where Sp is the speed-up.

In the figure 21, we have drawn the speed-up Sp as a function of T1/0, in the case of 4 processors (p = 4).

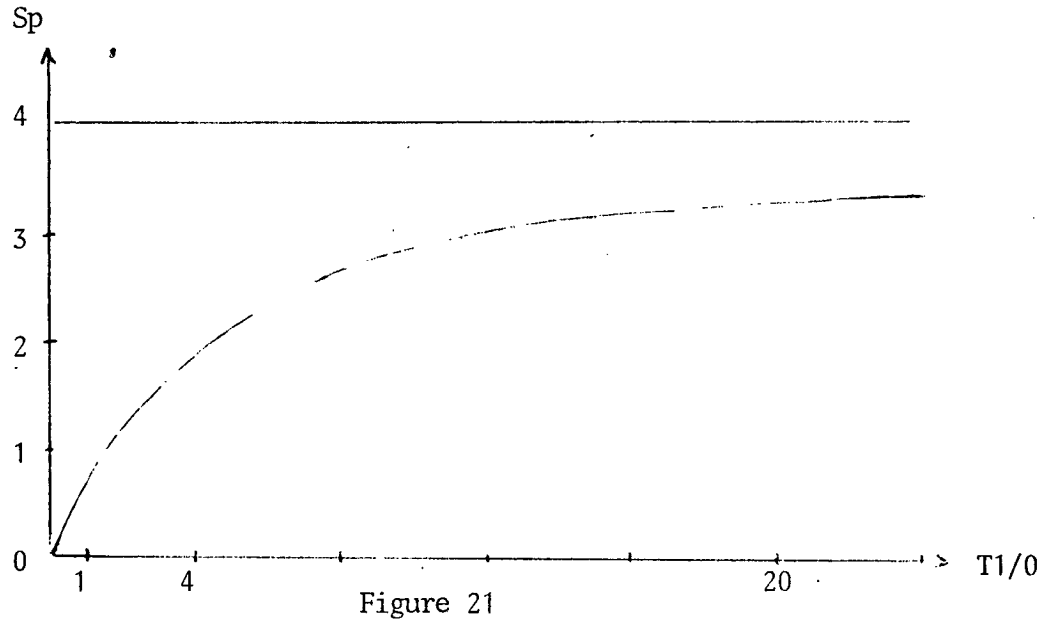



Figure 21

The third point which may decrease the speed-up is the potential memory bank conflicts. Some results obtained on Cray-XMP seem to show a non negligeable effect of memory conflicts on the performances [15]. Tests and simulations study in more details memory conflicts on Cray-XMP [16].

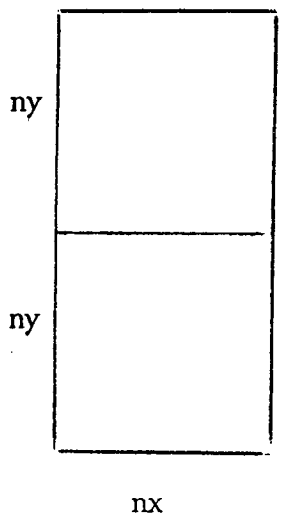
Numerical algorithms were parallelized and measured on the multiprocessor simulator MUPI [17]. They have shown that the overhead become preponderant when increasing the number of processors, if the size of the problem is not great enough to get sufficiant amount of effective work. It is therefore necessary to find a tradeoff between the amount of possible parallelism and the overhead inherent to it.

## 7 - Numerical experiments

An important part of numerical computations consists of the resolution of a large and sparse matrix. Vectorization can be handled if the studied domain is regular enough. Parallelism is extracted by using techniques of subdomains [18]. Each subdomain can be treated concurrently and separators impose some synchronizations. This mechanism is well suited to not regular matrix structures issued from finite element methods.

Experiments have been conducted using Incomplete Choleski Conjugate Gradient, which is inherently fast and vectorizable, except the strongly coupled resolution of the preconditioner. Results are given concerning the resolution of a Laplacien over a domain divided into two rectangles (figure 22) ; results will soon appear with four subdomains [16].

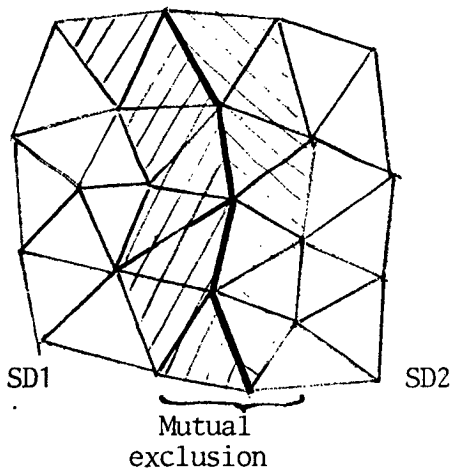
The assembly of a finite element method has also been implemented in a parallel environment. The main difficulty comes from the concurrent update of the shared matrix structure. A subdomain technique has also been tested : parallelism is guarenteed on subdomains, while elements of separators must be assembled in mutual exclusion. Results are given for the assembly of the Laplacien on two domains meshed by P1 Finite Elements. (Figure 23 ) It should be noted that vectorization would be better with another operator or element (more computations to do) and using vectors with hardware indirect adressing.




Cray-XMP CFT X.11

| Unknowns (nx* ny) | ELAPSED T. 1 CPU Mflops | ELAPSED T. 2 CPUs Mflops | SPEEDUP |
|---|---|---|---|
| 14400<br>120*60 | 0.91<br>51 | 0.64<br>71 | 1.4 |
| 20000<br>100*100 | 0.93<br>69 | 0.66<br>97 | 1.4 |
| 40000<br>100*200 | 1.41<br>90 | 0.94<br>129 | 1.50 |
| 80000<br>200*200 | 2.7<br>95 | 1.8<br>142 | 1.49 |

Figure 22




| DOM | ELEMENTS | ELAPSED T. 1 CPU | ELAPSED T. 2 CPU | SPEED UP |
|---|---|---|---|---|
| A | 1174 | 0.043 | 0.023 | 1.83 |
| B | 3234 | 0.118 | 0.062 | 1.89 |

Figure 23

REFERENCES

[1] Cray-XMP reference manual, Cray Research.

[2] Eta$^{10}$ supercomputer, ETA systems.

[3] Cedar "a large scale multiprocessor",
January 1983, Dept. of Comp. Sc. University of Illinois at Urbana Champaign.

[4] Cray Multitasking User Guide, Cray Research, 1984.

[5] Gregory R. Andrews & Fred B. Schneider
"Concepts and notations for concurrent programming",
ACM computing surveys, Vol. 15, N° 1 (March 1983), 3-43.

[6] LC 2 reference manual, Sintra Alcatel (Jan 1983).

[7] Smir M.
"A parallel computer prototype",
Comp. Sc. Dept. University of Jerusalem, Israël (April 1985).

[8] Dennis J.B. & Van Horn E.C.
"Programming semantics for multiprogrammed computations",
Comm. ACM 9, 3 (March 1966), 143-375.

[9] Ritchie D.M. & Thompson K.
"The Unix timesharing system",
Comm ACM 17, 7 (July 1974), 365-375.

[10] Dijkstra E.W.
"The structure of 'THE' multiprogramming system",
Comm. ACM 11, 5 (May 1968), 341-346.

[11] Brinch Hansen P.
"Concurrent programming concepts",
ACM Comput. Surveys 5, 4 (Dec 1973), 223-245.

[12] Hoare C.A.R.
"Monitors : an operating system structuring concept",
Comm. ACM 17, 10 (Oct 1974), 549-557.

[13] Hoare C.A.R.
"Communicating sequential processes",
Comm. ACM 21, 8 (Aug 1978), 666-677.

[14] U.S. Department of Defense
"Programming language Ada",
reference manual, Vol. 106 Lecture Notes in Computer Science,
Springer Verlag, New York, 1981.

[15] Meurant G.
Workshop GMD-INRIA (Jan 1985).

[16] Butel R.
"Experiments on Cray-XMP 4",
Rapp. Rech. INRIA to appear.

[17] Erhel J.
"Parallèlisation d'algorithmes numériques",
Thèse 3ème cycle, Mars 1982.

[18] Erhel J., Jalby W., Thomasset F., Lichnewsky A.
"Quelques progrès en calcul parallèle et vectoriel",
Proceedings of the sixth international symposium on Computing methods in applied sciences and engineering Versailles,
France, (Dec 1984).

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