HAL
open science

Address synchronized multiprocessor architecture

André Seznec, Yvon Jégou

▶ To cite this version:

André Seznec, Yvon Jégou. Address synchronized multiprocessor architecture. [Research Report] RR-0527, INRIA. 1986. inria-00076027

HAL Id: inria-00076027
https://inria.hal.science/inria-00076027v1

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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

Tél.: (1) 39 63 55 11

Rapports de Recherche

N° 527

ADDRESS SYNCHRONIZED MULTIPROCESSOR ARCHITECTURE

André SEZNEC
Yvon JEGOU

Juillet 1986

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 - RENNES CÉDEX
FRANCE
Tél. : (99) 36.20.00
Télex : UNIRISA 95 0473 F

PUBLICATION INTERNE N° 301

Juin 1986

40 Pages

Version Préliminaire

Address Synchronized Multiprocessor Architecture

André SEZNEC

Yvon JEGOU

IRISA, Campus de Beaulieu

35042 RENNES CEDEX

FRANCE

ABSTRACT

To satisfy the growing need for computing power, a high degree of parallelism will be necessary in future supercomputers. Up to the late 70s, supercomputers were either multiprocessors (SIMD-MIMD) or pipelined monoprocessors. Future industrial realizations should combine these two levels of parallelism.

In [Je86], a new model of pipeline architecture the Data Synchronized Pipeline Architecture was introduced; the behavior of this architecture in multiprocessor environment is very good. In this paper, we define a new model of tightly coupled multiprocessor : the Address Synchronized Multiprocessor Architecture. Three modes of computing exist : slice mode, free mode and iteration mode. These three modes allow the reaching of very good performances on a very large spectrum of algorithms —a large subset of

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

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

this family is considered as sequential codes for today's available supercomputers.

To allow a realistic implementation of an ASMA computer, we also introduce a new interconnection network for tightly coupled multiprocessors : the GREEDY network. This network allows a cost-effective implementation of the three modes of computing in an ASMA computer.

RESUME

Un haut degré de parallélisme est nécessaire pour satisfaire les besoins grandissants en puissance de calcul. Jusqu'à la fin des années 70, les super-calculateurs étaient soit des multiprocesseurs, soit des pipelines monoprocesseurs. Les réalisations futures intégreront ces deux niveaux de parallélisme.

Dans (Je86), nous avons introduit un nouveau modèle de calculateurs pipelines, le comportement de ces calculateurs reste bon dans un environnement multiprocesseur. Dans ce papier, nous introduisons un nouveau modèle de multiprocesseur fortement couplé. Trois modèles de calcul existent : mode tranche, mode itération et mode libre. Ces trois modes permettent d'atteindre de très bonnes performances sur un large spectre d'algorithmes - une importante sous-famille de ces algorithmes est traitée en scalaire sur les super-calculateurs d'aujourd'hui.

L'implémentation réaliste d'un multiprocesseur de ce modèle est rendue possible grâce à la définition d'un nouveau réseau d'interconnexion : le réseau GLOUTON (GREEDY).

PAPIER RECUPERE ET RECYCLE

Introduction

Need for computing power seems unlimited in various scientific applications. During the last ten years, tremendous progress has been made in the domain of component integration. But todays' supercomputer clocks are of the same order of magnitude as those of supercomputers ten years ago. E.g. the clock of the Cray2 (1985) is only three times faster than the one of the Cray1 (1976).

Requirements for performance have lead manufacturers to the design of parallel structures. The first industrial parallel supercomputers were pipeline processors (Cray1, CDC Cyber 205, ..). Today, these pipeline computers can be considered as the state of the art in monoprocessor architecture. Since the late 1970's, a lot of multiprocessor projects have been initiated [Ga83][Go83][Si81][Ho85].

Multiprocessors can be classified in two large families : the loosely coupled multiprocessors (hypercube family [Sei85], CEDAR [Ga83], ..) and the tightly coupled multiprocessors.

In a loosely coupled multiprocessor, each processor has its own local memory and executes a task on data stored in this memory, transfers of data between the processors are performed through an interconnection network directly between the processors (fig.1)(e.g. in hypercube computers) or through a global shared memory (fig.2) (e.g. CEDAR). Transfers between the processors are very expensive : near one hundred cycles are necessary to transfer an element between two nodes in the Floating Point Systems hypercube TESSERACT. Transfers of data must be very limited in loosely coupled multiprocessors if one wants to reach correct performance.

In tightly coupled multiprocessors, all the data are stored in a global shared memory (fig.3). They must be accessed through an interconnection network by the processors. The throughput of the interconnection network must be of the same order of magnitude as the throughput of the memory. This constraint limits the size of tightly coupled multiprocessors. This explains why the more ambitious projects are loosely coupled multiprocessors. Nevertheless, tightly coupled multiprocessors must be studied : they may be used as basic nodes in loosely coupled multiprocessors.

Tomorrow supercomputers will combine the three previous levels of parallelisms.

- One cannot imagine supercomputing without pipelining
- One can imagine a cluster of processors tightly coupled
- A set of clusters may be organized as a loosely coupled architeture

E.g. these three levels of parallelism are foreseen in the architecture of the CEDAR project [Ga83]. Great attention must be taken in the design of each of these three levels of parallel architecture. When designing a level of this hierarchy, one has to think of its integration in the superior level. In [Je86], we have presented an original model of pipeline architecture the Data Synchronized Pipelined Architecture (DSPA) which is very efficient on a very large family of non-regular code and whose structure allows a realistic integration in a tightly coupled multiprocessor. In this paper, we introduce a new model of tightly coupled multiprocessors the Address Synchronized Multiprocessor Architecture (ASMA)

—with the idea that the basic processor is a DSPA processor. Three modes of computing are available :

- slice mode : it allows vector processing
- iteration mode : it allows parallelism on non-vector loops
- free mode : it allows independant processing by the processors elements

These three modes of computing allows a good efficiency on a very large spectrum of algorithms. The feasability of ASMA computers is also shown by the definition of a new interconnection network for tightly coupled multiprocessors : the GREEDY network.

II Existent models of tightly coupled architecture

1 SIMD computers

During the 70's, supercomputing was essentially considered as vector computing, two families of vector computers have been studied the pipelined monoprocessors and the SIMD computers. The SIMD (Single Instruction stream, Multiple Data stream) computers are the simplest multiprocessors : N processors execute the same instruction on N distinct flows of data. The N elements who are treated in parallel can be considered as a slice of N consecutive elements of a vector; SIMD computers are then generally refered as vector processors.

The first SIMD computer was the ILLIAC IV [Ba68]. Each processor had its own memory and can communicate with only its four neighbours (fig.4). Transfers of data on this computer were too expensive; performances on many algorithms were too bad. This structure of interconnection network has been withdrawn for the design of general SIMD computers but is always used

in the design of special purpose machines -MPP[Gi83], systolic arrays [Kung78].

The Burroughs Scientific Processor (BSP) is a SIMD computer with a global shared memory [Ku82]. The interconnection network of the BSP is a crossbar network. Technological problems limits the size of crossbar networks : N*M switches are required to connect N inputs to M outputs. To allow a simple control of the processing elements and of the interconnection network, the designers of the BSP chosed a memory to memory instructions set. When the machine was designed, the behavior of a restricted set of vector instructions was optimized. To limit the number of cases to be studied, a constant increment definition of a vector and a prime number -17- of memory banks was chosen. Only two cases are possible : 17 consecutive elements of a vector can be accessed in parallel or the whole vector is stored in the same memory bank [La75][La82]. The BSP can reach good performances on vector instructions, but its definition of a vector was too restrictive; the following loop 1 is treated in scalar mode.

Loop 1 :

```
    DO 1 I=1,N
  A(I)= B(P(I))*C(I)
```

This loop may be considered as a vector loop in a SIMD machine, but conflicts may arise during the access to vector B, these conflicts have to be treated at the level of the address network : two addresses for the same memory bank cannot flow out from this network at the same time -it is obvious that to guarantee the signification of a vector access, the decisions taken for the routing of the address network and for the routing

of the data network must be the same. When priority rules are used in this treatment, the loop 2 may also treated as a vector loop :

Loop 2 :

```
      DO 2 I=1,N
   A(P(I))= B(Q(I))*C(R(I))
```

Unfortunately, the general treatment of conflicts in a crossbar network is very expensive : its complexity increases with the square of the number of memory banks of the computer and this treatment cannot be pipelined because the rejected requests must be treated at the following cycle.

Other interconnection networks have been proposed to increase the size of the SIMD computers [La75][Le78][Wu80] [Se84][Se86]; but the problem of indirections treatment -as in loop 1 or in loop 2- has not been solved , even for small sizes of multistage networks.

2.MIMD computers

Performances of the SIMD computers on non-vector code are very bad; other forms of parallelism may exist in scientific programs. The family of MIMD computers (Multiple Instruction stream, Multiple Data stream) can take advantages of these other forms of parallelism - independent tasks.

As recall in the introduction, we can classify the MIMD computers in two subfamilies the loosely coupled MIMD computers and the tightly coupled MIMD computers. The two families correspond to two different approaches of the programmation of parallel computers.

When programming loosely coupled MIMD computers, one must divide its program in large tasks where the amount of computation is large versus the amount of data transfers. Classical programs cannot be automatically adapted to this structure of computations. New programs must be written for this family of computers.

On the other hand, it is often possible to write relatively short tasks in programs. On a loosely coupled architecture, the time spent in data transfers would be too long in regard to the amount of computation. When using a tightly coupled MIMD computer, all the data are accessed in the global memory; transfers of data between two tasks does not really exist : the different processors have the same rights to access all the memory. The major problem when designing a tightly coupled MIMD computer is the interconnection network; in a SIMD computer like the BSP hardware treatment of conflicts on the network is avoided by the restrictive definition of the vectors, in a MIMD computer this treatment must exist : there are no relations between the accesses done by two distinct processors. This explains why tightly coupled MIMD computers have generally less than 16 processors.

Automatic detection of parallelism in classical programs may be used to generate code for tightly coupled MIMD computers because automatic detection generally generates very short tasks.

III The goals

1.Some examples of "parallel" loops

On a few examples of loops, we try to point out the limitations of existent tightly coupled architecture -K processors.

1.1 A pure vector loop

Let us consider the following loop :

Loop 4

```
      DO 4 I=1,N
    A(I)=B(I)+C*D(I)
```

This loop is treated as a vector loop by all the existent multiprocessors : there are no dependencies on the distinct iterations of this loop, no problems exist for the parallel treatment of this loop neither on SIMD computers nor on MIMD computers.

▷In SIMD computers, slices of K consecutive elements can be simultaneously accessed in memory.

▷In MIMD computers, a Kth of the N iterations is assigned to each processor.

1.2 Vector loop with a gather access

Let us consider the loop 1 ; vector is read by indirection through vector P.

▷In a SIMD computer, slices of K consecutive elements of vectors P, C and A are simultaneously accessed in the memory. A problem exists to access vector B : may be there are some conflicts on the memory; e.g B(P(1)) may be stored in the same memory bank as B(P(2)), this problem may be solved at

the level of the address network, only one address to each memory bank is routed in a cycle —the remaining addresses are routed at the following cycles; then all the processing elements are waiting for the arrival of the last element of the slice of B to perform the multiplication.

▷As in the previous example, in a MIMD computer, a Kth of the iterations is assigned to each processor.

1.3 A vector loop with a scatter access

Let us consider the loop 2;

▷As in the previous example, in a SIMD computer the accesses to vector B and C can be done in parallel with a treatment of conflicts on the address network; accesses to vector A cannot be performed in an arbitrary order : when P(J)= P(K) and J<K, the write of A(P(J)) must be performed before the write of A(P(K)). To always respect this condition, it is sufficient to impose static priority rules of the address network : an address computed by processor i has the priority on an address computed by the processor j if i<j. With this condition, loop 2 can be executed as a vector loop on a SIMD computer.

▷For MIMD computers, in the two first examples no synchronization primitives have to be used; but for loop 2 synchronization primitives must be used and then will strongly damage performances.

1.4 A loop of independent tasks

Let us consider the following loop 5:

Loop 5:

```
      DO 5 I=1,N
      DO 6 J=1,1000
```

```
      A(I,J)= B(I,J)+1
      IF A(I,J)>0 THEN GOTO 5
    6 CONTINUE
    5 CONTINUE
```

▷Nor the internal loop 6 neither the external loop 5 can be considered as vector loops : this loop must be considered as a scalar loop for a SIMD computer.

▷The iterations of the external loop 5 are independent; they can be distributed on the processors of a MIMD computer, no synchronization primitives have to be executed during the iterations.

1.5 A vector loop with possible dependencies between successive iterations

Loop 7:

```
      DO 7 I=1,N
    7 A(P(I))=A(Q(I))
```

Neither on SIMD computers nor on MIMD computers, the parallel execution of this loop is possible; dependencies may exist between two iterations of this loop, three forms of dependencies are possible :

▷Q(J)=P(K) with J>K :the write of A(P(K)) must be performed before the read of A(Q(J)) -Read After Write (RAW).

▷P(J)=P(K) with J>K :the write of A(P(K)) must be performed before the write of A(P(J)) -Write After Write (WAW).

▷P(J)=Q(K) with J>K :the read of A(Q(K)) must be performed before the write of A(P(J)) -Write after Read (WAR).

Cytron [Cy84] proposed a mechanism to treat in parallel this family of loops; its proposition consists in inserting delays between the beginning of the successive iterations to ensure the respect of the dependencies. This is not very efficient : in loop 7, there are only four accesses to the memory and only the accesses to P and Q can be done before the total end of the previous iteration. If vector A is large, real dependencies are very rare; this will be very interesting to be able to treat in parallel the iterations where no dependencies occur at execution.

2. The challenge

In the previous paragraph, we have pointed out that existent tightly multiprocessors are not efficient on a large family of algorithms -SIMD computers are not efficient on non-vector code, MIMD computers are not efficient on code where there are possible dependencies.

It is unrealistic to hope to distribute the more internal loops of a program -regular or non-regular- on several distinct clusters of processors, e.g. loop 1 and loop 2 must be executed by only one cluster : the whole vector B must be accessed in a minimum delay.

Designing a cluster which is powerful on non-regular code is very important :

- As SIMD computer with a very large number of processing elements can be designed [Se84], performances of general multiprocessors will be judged on non-vector algorithms.
- If performances of the clusters on non-regular code are sufficient, programming a supercomputer which includes a level of loosely coupled parallelism will be easier, one will have only to divide its programs

in large tasks where the amount of computations is large versus the amount of data transfers between the tasks.

Our goal is then to design a tightly coupled multiprocessor which is actually efficient in parallel on a very large spectrum of algorithms without rewriting programs.

This spectrum must include all classical vector loops, vector loops with scatter and gather, loops which can be divided in independent tasks and also the loops where possible dependencies may occur.

This last family of loops is very important; indirections create possible dependencies in a lot of codes :

Loop 8 :

```
      DO 8 I=1,N
      A(P(I))= ..
         .
         .
         .
         .
 8    ..= A(Q(I)) + ..
```

Many loops have this structure, they are generally considered as scalar loops. But in general cases when A is a large vector there are no dependencies between several iterations of the loop, it would be very interesting to be able to compute the successive iterations of this kind of loops in parallel when no real dependencies occur at execution.

IV Address Synchronized Multiprocessor Architecture : principles

1.Principles

In a program, the order of the accesses to the same location in the memory is a very significant order; the three forms of dependencies we have presented in III.1.5 must be respected. Respecting the WAR dependencies and the WAW dependencies is not a difficulty on a monoprocessor. It seems natural that a read instruction can immediatly be executed by the memory as a write instruction must wait for the data to be written; it is also natural to impose to the results to flow out from the processor in the good order. These two constraints do not greatly affect the performance of a computer. On the other hand in [Je86], we showed that it is critical to be able to pass the write by the reads in a pipeline processor and that this passing the writes by the reads must be treated by an hardware mechanism of treatment of RAW hazards and not by software tools.

As on a monoprocessor, on a tightly coupled multiprocessor the WAR and the WAW dependencies may be treated by imposing a write to be performed after all the reads and writes on the same memory bank which have been received before it.

In [Je86], we also showed how to respect the signification of a sequence of memory accesses when using a monoprocessor and a single flow of memory requests ; when using a multiprocessor, the accesses to the memory are performed by several processors at the same time. Two or three processors may want to access the same memory bank at the same time; the hardware has to give a signification to any sequence of accesses -which processor has the priority ?-; it must also garantee a unique signification and avoid deadlocks.

This has lead us to the definition of the Address Synchronized Multiprocessor Architecture (ASMA).

In an ASMA computer, the processors are synchronized through the memory by the addresses :

- the internal sequencing of the processors are independent.
- the only relations between the processors are performed at the level of the requests on the memory. The hardware gives a signification to RAW hazards induced by multiprocessing.
- there may be different modes of computing. We will defined three modes. Others may be (?) imagined.

2.Slice mode

We have seen in the previous sections that a very large set of algorithms may be treated as "vector" loops for a SIMD computer when one garantees static priority rules on the request interconnection network - Loop 1, 2, and 3 may be treated as vector loops.

The slice mode is introduced to treat this family of algorithms :

Definition :When processing in the slice mode, the requests are routed by slice of K requests addresses to the memory -K is the number of processors-, each processor must produced a request : the requests are routed when the slice is completed -some requests may be null requests introduced to complete the slice, e.g. for the treatment of the uncomplete slice of a vector instruction. When conflicts occur -i.e. more than one address concern a single bank memory- static priority rules on the network ensure that the address coming from processor i does be treated -i.e.

performed for a read, or stored in the RAW mechanism— by the memory bank before the address coming from processor j if i>j.

The slice mode supposes the existence of an hardware tool to detect if all the processors have produced a request.
But it allows to treat in slice mode all the "vector" loops previously considered e.g. loops 1 and 2. The main difference with the vector execution in a SIMD computer is that we have not imposed the data accessed in memory to be routed in slice mode. This allows to treat the following loop as a vector loop at the condition to dispose a read-modify instruction access on the memory :

Loop 9

```
      DO 9 I=1,N
    9 A(P(I))= A(P(I))+V(I)*C(Q(I))
```

There are possible dependencies between two successive iterations of this loop, parallel treatment of this loop is impossible on a SIMD computer because in a SIMD computer all the processing elements must perform the same instruction on different data, e.g. when P(5)=P(1), A(P(5)) must not be read before A(P(1)) has been written, but on a eight processors SIMD computer the new value ofA(P(1)) cannot be computed and written before the read of A(P(5)), the loop 9 must then be sequentially treated.

The slice mode allows a parallel execution of this loop, this is very interesting because if vector A is large versus the number K of processing elements the real dependencies which occur at execution are very rare.

3.Free mode

The free mode is introduced to treat independent tasks on distinct processors of the same multiprocessor; from our point of view, synchronization sections have to be very rare and the programmer has to accept an important cost for these sections. Our basic goal was to allow the parallel treatment of loops of the form :

Loop 10 :

```
      DOALL 10 I=1,N
   10 CALL TASK(I)
```

where only local variables are modified by TASK(I) -i.e. if a variable is modified by TASK(I), it is not read or modified by TASK(J) for J#I.

Definition : Assuming that a request is either routed to the memory or refused and resubmitted at the following cycle, in the free mode, there are no synchronizations between the memory requests of the different processing elements.

This definition avoids hardware gulty deadlocks between tasks executed by distinct processors :

Proof :

One can give a date to each request : the cycle when it is arrived on the memory bank, if we consider the whole set of requests which are present on the memory at a moment, it exists a request which date is minimum, unless software errors this request will be executed -if the request is a read, it may be performed at the moment, if the request is a write, as the operands necessary to compute the data to be written

have been read because these read requests are older than the considered write –the resubmission of the same request by the processor ensures that successive requests flowing from the same processor have successive dates–, this data will be available in a finite delay. Then no hardware gulty deadlocks are possible.

Q.E.D.

This datation is very important : when there is no mean to date the requests, deadlock may arise on memory as in the following example :

Example 11:

Processor 1	Processor 2
READ A ▷ R1	READ B ▷ R1
WRITE R1 ▷ B	WRITE R1 ▷ A

A is located in bank 0, B is located in bank 1; if there are no datation of the requests coming from a same processor –i.e if the requests coming from the same processor may enter the memory banks in a distinct order from their departure from the processor (this may occur in multistage networks with redundant pathes and memorisation of the rejected request)–, the following situation may occur : the two write requests enter the memory banks at the cycle t and then the two read requests enter the memory banks at the cycle t+1; the two memory banks are thenwaiting for data which cannot arrive.

One can remark that we can use the read-modify request on memory to implement synchronization sections in programs, and then the machine can be programmed as a classical tightly coupled MIMD computer; but in our mind,

tasks where a lot of synchronization sections are necessary have to be avoided.

4.Iteration mode

We have pointed out that in many loops possible but not frequent dependencies exist. We have also seen that performance of existent machines on this family of loops is very disappointing : very poor parallelism is exploited in these loops.

The main idea to allow parallel execution of loops where possible dependencies exist is to disconnect the production of the addresses for the requests on the memory and their consumation by the memory banks -i.e. their treatment by the RAW mechanism.

When we consider the loop 7, vectors of indirections P and Q may be read in slice mode -more than one slice of vector P and Q may be read-, then the whole set of processors commutes in iteration mode and computes in parallel the addresses of A(Q(I) and A(P(I)); these addresses are routed the memory banks, but at the level of the memory banks the addresses enter a FIFO queue associated with the processor origin of the requests. Requests for the Ith iteration are treated by the memory bank when it has treated all the requests of the previous iterations i.e all the reads for the previous iterations have been initiated, all the writes have been treated by the RAW detection mechanism, and no requests concerning the previous iterations can arrive later -a processor must announce to all the memory banks that it has ended the send of the addresses of an iteration.

This mode is particulary interesting, it allows parallel execution of a lot of loops which were previously considered as sequential loops; the efficiency of a vectorizer becomes less critical on machines which have the

iteration mode of execution because performance is only damaged by the real dependencies and not by the possible dependencies. We hope that the existence of the iteration mode will decrease the ratio of algorthms with tasks communicating by synchronisation sections.

The existence of the iteration mode seems to be very expensive in hardware : in each a FIFO queue of requests associated with each processor, a mechanism to select the good FIFO queue of requests, .. . Efficiency of the iteration mode can also be discussed when using a classical crossbar network : we can imagine that all the memory banks are reading data for the same processor, the crossbar network can only accept one data for this processor; this example proves us that the real throughput of a crossbar network in iteration mode may be very small. In the next section, we present an original interconnection network we have imagined for an ASMA computer, this network may accept a datum by input even when several of them have the same destination. The amount of hardware required in the design of this network is not very large besides the functions it executes.

V The GREEDY network : an interconnection network for an ASMA computer

1.Some limitations of the crossbar network

The major limitations which have been pointed out for the crossbar network in the past generally concern the limited size of the crossbar networks that can be actually built. This size seems to be limited to around twenty inputs and twenty outputs. We does not think that the real limitation of the crossbar network is due to its limited size; the real

limitation of a crossbar is due to the necessity to treat destination conflicts : informations -i.e the destinations of all the elements to be routed- must be centralized and the major problem is the impossibility to pipeline the hardware treatment of conflicts -a rejected request must be submitted again on the following cycle.

Another problem which appears on a crossbar network when using in a tightly coupled MIMD computer, is the problem of the real throughput of the network. For example, when submitting 16 requests arbitrary distributed on a 16*16 crossbar network, only an average of 10 requests are accepted, the others must be submitted again on the following cycles; this limits automatically the speed-up when computing on independent flows of data arbitrary distributed to a factor of around 10 for a MIMD computer with 16 processors. Moreover we have pointed out at the end of the previous section that the decrease of performance on iteration mode due to a crossbar network may be huge. So we have imagined an interconnection network the GREEDY network which accepts a request on each input even when several destinations are the same.

2.The GREEDY network

2.1.Origin and definition

In [Je86], we have developped an original model of pipeline architecture the Data Synchronized Pipeline Architecture (DSPA); our goal was to conceive a pipeline processor which is efficient on the same spectrum of algorithms we have defined in the section II. The model ASMA is developped to take part of this pipeline processor. In our mind, each basic processor of an ASMA computer is a DSPA processor, even if the theoretical model can

support other basic processors. From our point of view, it is very important to support the three modes of computing defined in the previous section : DSPA increases efficiency of a monoprocessor on a lot of non-regular algorithms by a factor of three or four, we think that if iteration mode can well perform on a 16 processors ASMA computer, a new factor around ten may be won on the performance on this family of non-regular algorithms.

In DSPA, the distinct functional units (FUs) communicate the data through FIFO queues, and their sequencing are totally independent; a FIFO queue is associated to each path between an output of a FU and an input of a FU —may be the second FU is the same as the first one; the operands are read on the origin FIFO queues, the results are written in the destination FIFO queue; synchronization is done by the data which enter the FIFO queues —when an origin FIFO queue is empty, the FU does not work until a datum enter the FIFO queue. The interconnection scheme of a DSPA processor has lead us to imagine an original interconnection network to link together memory banks and the processors in an ASMA computer.

Definition (fig.5):

A N*M GREEDY network is a N inputs M outputs crossbar network where each crosspoint has been replaced by a FIFO queue.

First we can remark that a GREEDY network can accept a datum from each input at each cycle and a datum may flow out from the network on each output at each cycle. When a FIFO queue is full, the GREEDY network must refuse a datum, but one must note that this information is local : it is

the state of one FIFO queue, the other FIFO queues may continue to work independently.

If the definition of the GREEDY network is very simple, it does not solve the problem of the control of the network; the three modes defined in section IV corresponds in fact to three different modes of controlling the GREEDY network used as address network.

2.2 Slice mode

In the slice mode, the requests are routed by slices of K elements (K is the number of processors). When computing in slice mode, the requests does not enter the GREEDY address network before the slice is completed. When the slice is completed, the requests enter the GREEDY network and are stored in the FIFO queues associated with the path between the processor and the memory bank. In each memory bank, there is a memory sequencer unit (MSU), this MSU receives from each processor a one bit bus : on this bus, the information "the request concerns this memory bank" is coded. The whole vector of informations is stored in a FIFO queue (width K bits) in the MSU; an immediate optimization consists in storing this vector only if there is some request concerning the memory bank.

The treatment of the requests in a memory bank M is then very simple :

1. Extraction of a vector V of K bits in the FIFO queue,
   if the FIFO queue is empty then GOTO 1.
2. If there is no nonnull element in V then GOTO 1,
   X <-- number of the first nonnull bit
   V(X)=0

3.Extraction of a request in the GREEDY network from the FIFO queue associated to the pair (X,M) and immediate treatment by the RAW mechanism, GOTO 2

It is obvious that the sequence we have described respects the order of the accesses to the memory which have been induced by the slice mode definition : treatments of requests on a memory bank are done slice after slice and in the same slice the priority to the request coming from the minimum numbered processor is respected. The hardware required in the MSU to perform this treatment is very simple : a FIFO queue and a priority encoder.

2.3.Free mode

The difference between the slice mode and the free mode is only the absence of synchronization at the entry of the address network, so at each cycle in free mode a request -when it has been produced- enters the GREEDY address network in the FIFO queue associated with the processor and the destination memory bank. As in slice mode, on each cycle the MSU of each memory bank receives the vector of bits describing "the request of processor i concerns this memory bank". The MSU sequencing is exactly the same as in slice mode.

One can easily verify that no hardware gulty deadlock is possible in free mode.

2.4. Iteration mode

As in free mode, the requests may enter without synchronization at the entry of the address network; the difference with the other modes is at the level of the consumation of the requests by the MSUs of the different functional units.

For the slice and the free mode, the MSU considers vectors of bits; for iteration mode the consumation of the requests must be done producer after producer.

Several solutions may be imagined to implement the iteration mode; for example, the following sequence can be repeated :

```
1 REPEAT
      treatment of requests of processor i
  UNTIL Request=End of iteration;
  i=i+1;
GOTO 1
```

We have here supposed that the information "End of iteration" is diffused from the processor to the whole set of memory banks through the GREEDY network. This solution may be considered as a very cheap solution to implement the iteration mode because it requires no special developments of hardware for this specific mode. But we think that this is not the good solution : the MSU —and then the memory bank— is automatically busied during a cycle by each iteration —by the instruction "End of iteration". Efficiency of the iteration would be bad on very short loops —e.g. loop 7.

We think that introduction of the iteration mode in the machine may justify some cost of hardware development. Then we propose to treat the iteration mode at the MSU level.

Principles :

The MSU countains a one-bit large FIFO queue associated with each processor.

When a requests from processor i concerns the memory bank a 1 is stored in the associated FIFO queue, the request "End of iteration" is diffused to all the memory banks and then a 0 is stored in the associated FIFO queue.

The requests are treated as following :

The bank treats the requests coming from processor i until the head of the FIFO queue is a 0, then it jumps to the treatment of the requests of the next processor for which the associated FIFO queue -in the MSU- is empty or begins by a 1 -all the FIFO queues which have been ignored because of their heading by a 0 have to advance.

Such a hardware mechanism allows to reach a good throughput of the memory bank in iteration mode even on loops where only a very little number of accesses are done on the memory -e.g. in loop 7.

2.5.Some examples of efficiency of the use of the GREEDY network

The very simple treatment of conflicts on a GREEDY network allows to hope the reaching of the asymptotic throughput of max(N,M) data by cycles even when the requests presented to the network on a cycle concern the same output. We illustrated this on a few examples

When the indirect accesses are uniformly distributed on the memory banks, loop 1 and 2 run at full speed in slice mode —i.e. each memory bank really performs an access by cycle. If no real dependencies occurs loop 9 also runs at full speed when the indirect accesses are uniformly distributed.

Let us consider the following loop :

Loop 12:

```
      DO 12 I=1,N
      DO 13 J=1,M
   13 A(I)=A(I)+ B(I,J)
   12 CONTINUE
```

When executing this loop, it is natural to consider the external loop 12 as a vector loop to minimize the number of reads and writes on the memory, addresses of K elements of a line of B are then produced in parallel. When B is stored in FORTRAN i.e. columnwise, problems will appear in the cases where the number of columns in B and the number of the memory banks are not relatively prime : on each slice of line, conflicts appear on the address network. Using a GREEDY network, these conflicts are solved by the network, and after a start-up delay the computer will run at full speed.

Another example of the efficiency of the GREEDY network is its efficiency to perform the perfect-shuffle :

Generally, on general-purpose supercomputers the address unit cannot produce at full speed the addresses to perform in parallel a perfect-shuffle, the simpler algorithm to perform the perfect-shuffle in parallel is to execute the following loop :

Loop 14

```
      DO 14 I=1,N
      B(2*I)= A(I)
   14 B(2*I+1)=A(I+N)
```

When using a GREEDY network, this loop will run at full speed in slice mode.

3.About the GREEDY network in an ASMA computer

3.1.The read network and the write network

In the previous section, we have essentially presented the routing of the GREEDY address network. In fact, three GREEDY networks have to be used in a ASMA computer -the address network, the read network and the write network. Controlling the read network and the write network is very simple; we detail here the control of the GREEDY read network.

In a processor, the acquisition of the data from the GREEDY read network by the Data Acquisition Unit (DAU). When routing a read address in the GREEDY address network, the number of the memory bank destination is stored in a FIFO queue in the DAU. To obtain the data coming from the memory in the same order their reads addresses have been computed, the DAU reads its FIFO queue and uses this value to read the origin FIFO queue of the datum -the DAU waits until the designed FIFO queue is not empty.

One can remark that the DAU is very simple and has to be compared with the hardware mechanism which would be necessary to reorder the data flowing from a crossbar network; these data would have to be stored in an

intermediate support because if it is easy to ensure that data coming from the same memory bank are in the good order, in a MIMD computer it is nearly impossible to guarantee this condition for data flowing from distinct memory banks.

Another remark must be done; when using crossbar networks as address, read and write networks a lot of informations must be passed through the networks with the data to allow the association of the data and the requests : number of the origin processor for the address and the write networks, number of the origin memory bank for the read network.

3.2 Extensions towards outside of the processing unit

We have described the GREEDY network between a set of processing elements which execute numerical algorithms and a memory divided in logical memory banks; all the processors were considered to have the same status but one can hope to have other processors to perform exchange with other ASMA computers to allow the third level of parallelism or a unit of secondary memory -may be disk units or global memory shared with others ASMA computers. Entries on the GREEDY network may be affected to these processors

3.3 The feasability of the GREEDY network

Our project consists in the design of a 16 processors ASMA computer in which the basic processor is a DSPA processor. The efficiency of this computer will greatly depend on the availability of a GREEDY network. The feasability of the GREEDY has been proved [Co86]. The basic chip of the GREEDY network will be a 4*4 GREEDY network of 12 bits width FIFO queues and about 32 words by FIFO queues. The development of such a cell appears

to be very important, three 20*16 (or 16* 20) GREEDY networks are necessary -inputs on the write and address networks are foreseen for external communications, outputs on the read networks also- and the internal interconnection scheme of the DSPA processors will also use this cell as basic cell : about 600 cells of this type will be used in the design of one 16 processors ASMA computer.

VI Conclusion

We have presented a new model of tightly coupled architecture; this new approach of synchronization of the processors at the level of the address network allows a very good efficiency on a large spectrum of algorithms which includes the classical vector algorithms, the generalized vector algorithms with scatter and gather accesses to the memory, the whole family of problems that can be divided in independent tasks and a large family of loops which were generally considered as scalar loops.

The introduction of the iteration mode seems to be a major step in the advance of tightly coupled architecture : all the loops where there is no external jumps can be treated in this mode, the efficiency of vectorizing techniques [C182] becomes less critical.

The implementation of the three modes of computing is really possible because of the introduction of the GREEDY network. The control of this network is very original for use in tightly coupled multiprocessor : the producer can always send its data in a FIFO queue, but the consumer must explicitly read the data on FIFO queues. This has allowed to implement the three modes of computing at the level of the address network. The average throughput of the GREEDY is also greatly increased besides the average

throughput of a crossbar network. The cost of the GREEDY network seems to be reasonable besides the cost of a crossbar network for the same size of network : the hardware treatment of conflicts is very expensive for a crossbar network and a lot of hardware is saved in the processors and in the memory banks because of the possible memorization in the GREEDY network. The GREEDY network will be easilier extended than the crossbar network and external communications may be through this network at a reasonable cost : inputs or outputs of the networks may be assigned to these communications.

Bibli]ography

[Ba80] J.L.Baer, *Computer Systems Architecture,* Computer Science Press, 1980

[Ba68] G.H. Barnes & al. "The Illiac IV Computer" I.E.E.E. Transactions on Computers, vol C-17, pp.746-757, Aug.1968.

[Ch81] A.E.Charlesworth, "An approach to scientific array processing : the architectural design of the AP120B/FPS 164 Family" Computer, september 1981

[Cl82] N.A.Clifford " Performance evaluation of three automatic vectorizing packages" International Conference on Parallel Processing pp235-242 1982

[Co86] C.Courtel, DESS microelectronique report June 1986 University of Rennes.

[Cr79] Cray-1 Computer Systems, *Hardware Reference Manual,* Cray Research Inc., Chippewa Falls, WI 1979

[Cy84] R.G.Cytron "Compile-time scheduling and optimization for asynchronous machines " Dpt of Computer Science, University of Illinois 1984

[Do85] J.J.Dongarra, "Performance of various computers using standard linear equations software in a FORTRAN environment", Computer Architecture News, pp3-11, march 1985

[Ga83] D.Gajski, D.Kuck, D.Lawrie, A.Sameh, "Cedar : a large scale multiprocessor", International Conference on Parallel Processing 1983, pp524-529

[Gi83] P.A.Gilmore "The Massively Parallel processor (MPP): a large scale SIMD processor" Proceedings of SPIE, Real Time Signal Processing VI, Vol.431, pp166-174 1983

[Go83] A.Gottlieb & al., "The NYU Ultracomputer - Designing an MIMD shared memory parallel computer" IEEE Transactions on Computers, Vol. C-32, pp175-189, feb.1983

[Ho81] R.W.Hockney, C.R.Jesshope, *Parallel computers : architecture, programming and algorithms,* Adams Hilger, Bristol 1981

[Ho84] R.W.Hockney, "MIMD computing in the USA - 1984", Parallel Computing, 1985, pp119-136

[Hw84] K.Hwang, F.A.Briggs, *Computer architecture and parallel processing,* Mac Graw Hill 1984

[Je86] Y.Jegou, A.Seznec, "Data Synchronized Pipeline Architecture : Pipelining in Multiprocessor Environment" to appear in Proceedings of the 1986 International Conference on Parallel Processing and in the Journal of Parallel and Distributed Computing

[Ko81] P.M.Kogge, *The architecture of pipelined processors,* Mac Graw Hill 1981

[Ku82] D.J.Kuck, R.A.Stokes, "The Burroughs Scientific Processor (BSP)", IEEE Transactions on Computers, vol C-31, pp. 363-376, May 1982.

[Kung82] H.T. Kung "Why systolic architectures" Computer, Janvier 1982 p37-46

[Kung78] H.T. Kung and C.E. Leiserson: "Systolic arrays for VLSI" Introduction to VLSI, C.A Mead and L.A Conway ed., Addison Wesley, 271-287

[La75] D.H.Lawrie,"Access and alignment of data in an array computer", IEEE Transactions on Computers, vol C-24, pp.1145-1155, dec.1975.

[La82] D.H.Lawrie, C.R.Vora, "The prime memory system for array access", IEEE transactions on Computers, vol C-31, pp. 435-442, May 1982.

[Le78] J.Lenfant, "Parallel permutations of data : A Benes network control algorithm for frequently used permutations" IEEE Transactions on Computers, vol C-27, pp.637-647, july 1978.

[Ra77] C.V.Ramamoorthy, H.F.Li, "Pipeline Architecture", Computing Surveys, Mars 1977

[Sei85] C.L.Seitz, "The Cosmic Cube", Communications of the ACM, Vol.28, pp.22-33, 1985

[Se84] A.Seznec, "A new interconnection network for SIMD computers : the Sigma network" Submitted to IEEE Transactions on Computers.

[Se86] A.Seznec, "An efficient routing control unit for the RSI4" Proceedings of the 13th International Symposium on Computer Architecture June 1986

[Si81] H.J.Siegel & al., "PASM : a partitionable SIMD/MIMD system for image processing and pattern recognition", IEEE Transactions on Computers, Vol C-30, pp934-947, 1981

[Sm78] B.J.Smith, "A pipelined shared resource MIMD computer ", IEEE Proceedings 1978 International Conference on Parallel Processing, pp6-8

[Ta85] H.Tamura, Y.Shinkai, F.Isobe, "The supercomputer FACOM VP system", Fujitsu Sc. Tech. J. March 1985

[To67] R.M.Tomasulo, "An efficient algorithm for exploiting multiple arihmetic units", IBM J., Vol. 11, Jan. 1967

[We84] S.Weiss, J.E.Smith, "Instructions issue logic in pipelined supercomputers", Transactions on Computers, pp1013-1022, Nov. 1984

[Wu80] C.Wu, T.Feng, "The reverse-exchange network" IEEE Transactions on Computers, vol C-29, pp. 801-811, Sept. 1980.

[Wulf72] W.A.Wulf, C.G.Bell "C.mmp: A multi-mini-processor" Proceedings AFIPS Fall Joint Computer Conference, 41 (2), pp765-777, 1972

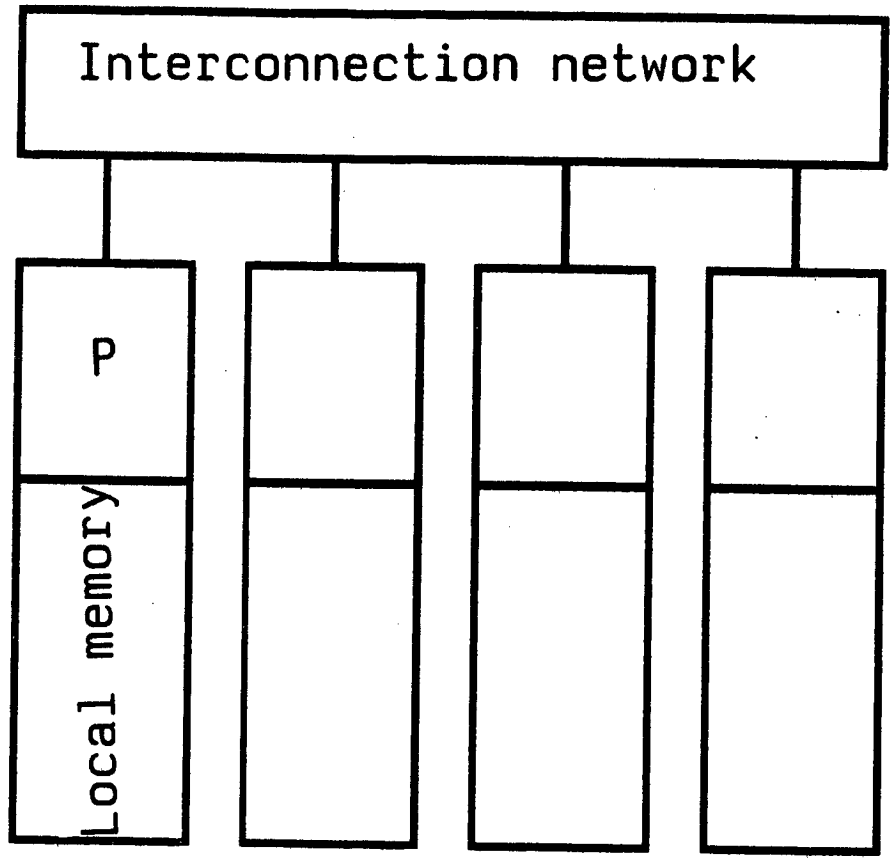

Fig.1 : example of loosely coupled organization

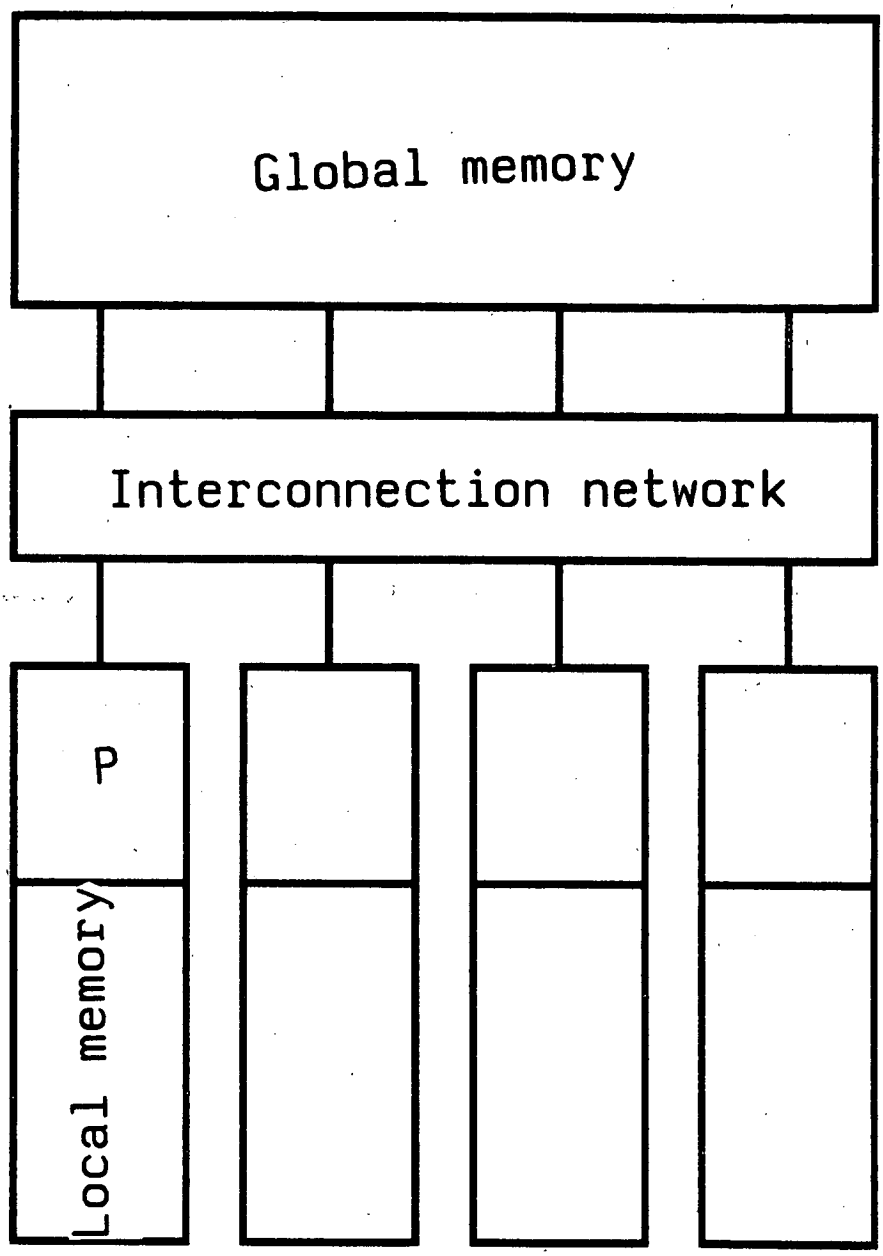

Fig.2 : example of loosely coupled organization

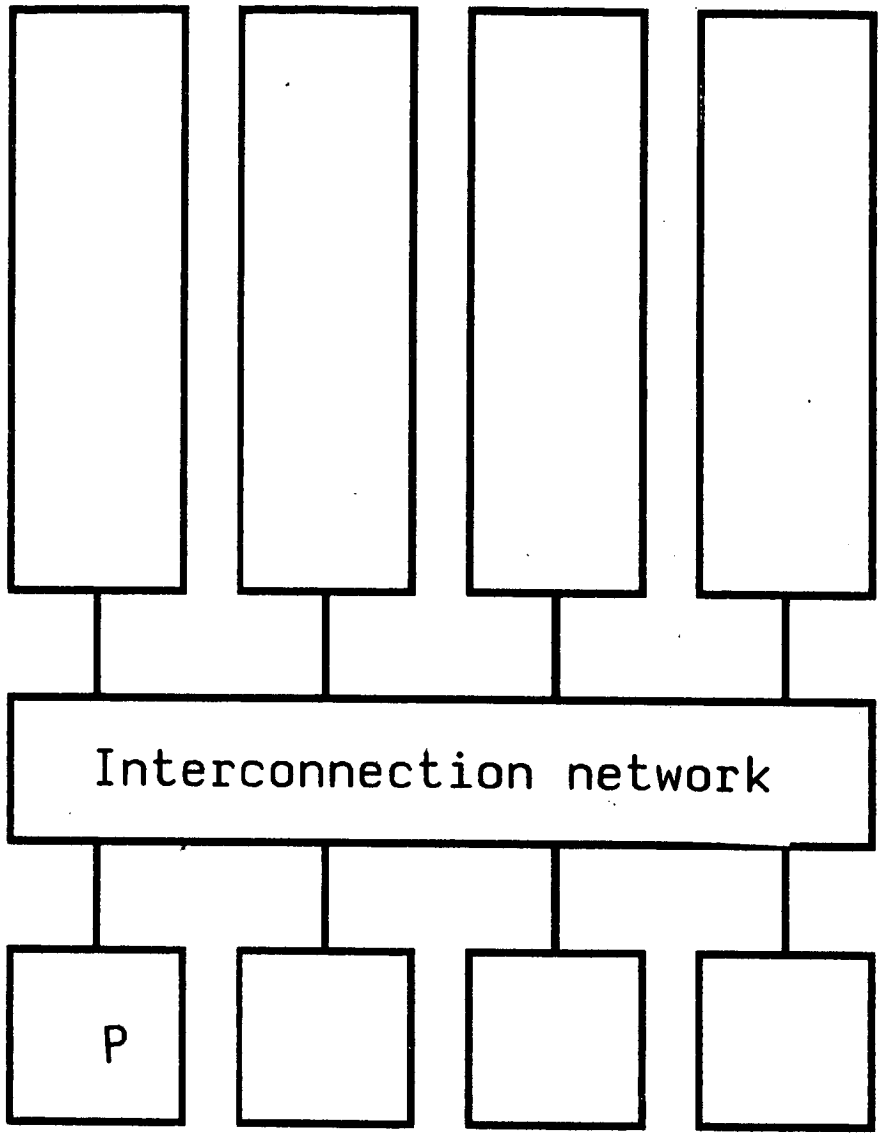

Fig.3 : structure of a tightly coupled multiprocessor

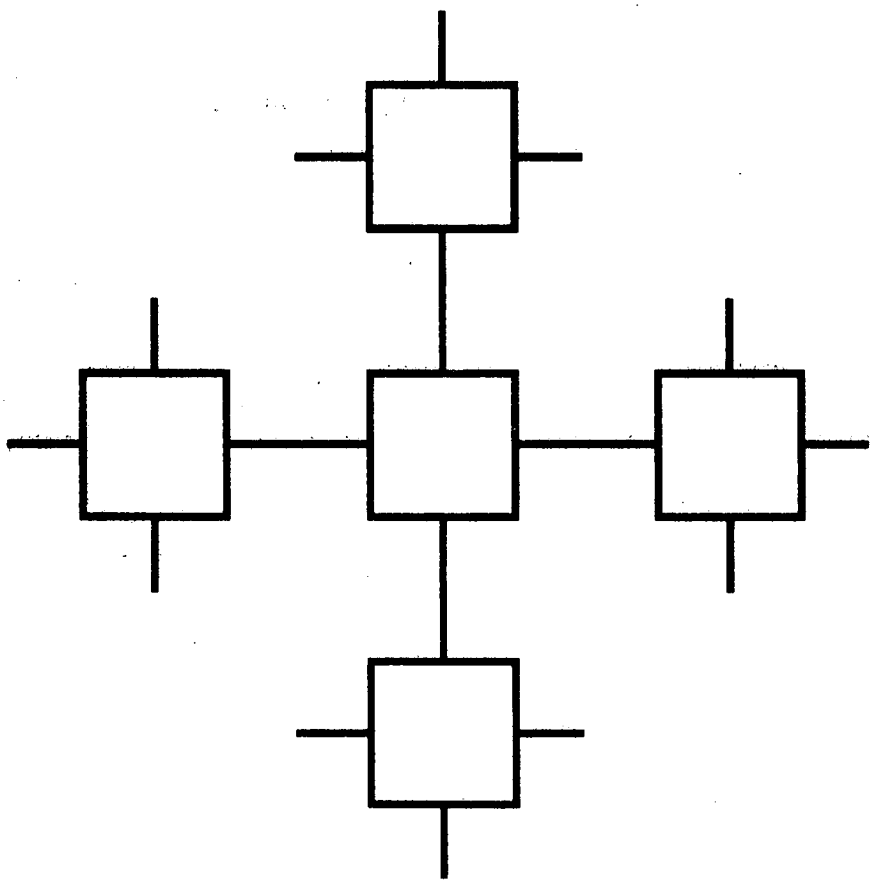

Fig. 4 : a node of the interconnection network of the interconnection network of the ILLIAC IV

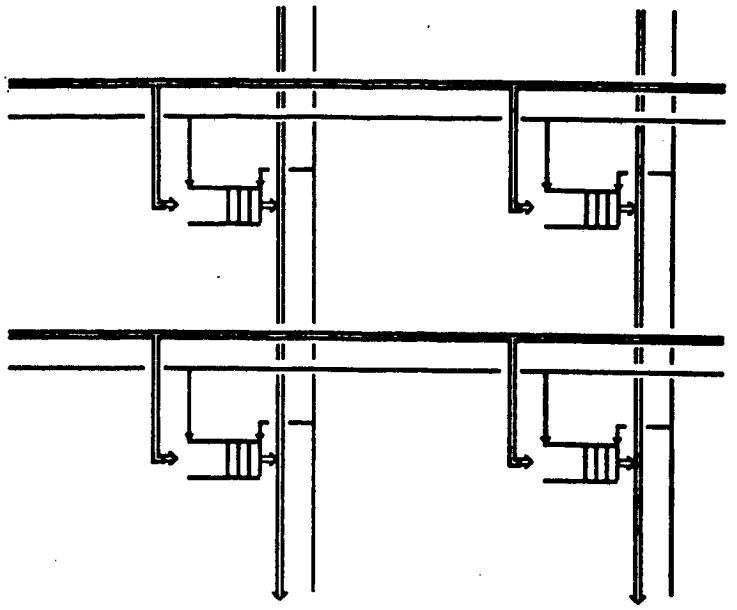

Fig.5 : a 2*2 GREEDY network

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