POMP* or How to design a massively parallel machine with small developments†

Philippe Hoogvorst   Ronan Keryell   Philippe Matherat
Nicolas Paris

—

Laboratoire d'Informatique de l'Ecole Normale Supérieure
URA 1327 CNRS
45 rue d'Ulm, 75005 PARIS
Tel: (+ 33 1) 44.32.20.32, fax: (+ 33 1) 44.32.20.80.
E.mail: ...@dmi.ens.fr
...@frulm63.bitnet

—

Technical Report LIENS-91-5

Abstract

The design of a SIMD machine is usually complex because it leads to developping an efficient Processing Element and to writing all the softwares required by the chip and the control of the machine.

We propose a different approach by using an efficient 32-bit off-the-shelf processor with its software environment (compiler and assembler) and a programmable gate array for the network. It limits the development to the minimum and leads to a rather general SIMD cluster built with off-the-shelf chips which can be considered as a SIMD transputer.

Résumé

La conception d'un ordinateur SIMD est souvent complexe car il faut concevoir un Processeur Elémentaire SIMD efficace avec tout l'environnement de programmation nécessaire à l'utilisation de la machine.

C'est pourquoi nous proposons l'utilisation originale d'un processeur 32 bits courant, avec son environnement logiciel standard, associé à un réseau de portes programmable pour la gestion des communications entre processeurs et des problèmes de synchronisation. Cela limite les développements à effectuer au minimum tout en conservant une puissance de calcul importante.

* *un Petit Ordinateur Massivement Parallèle*: a small massively parallel computer. Project supported by the French Ministry of Research and Technology, in collaboration with Thomson Digital Image, and the *Programme de Recherches Coordonnées en Architectures Nouvelles de Machines informatiques* (CNRS-MRT).

† This paper was prepared for the proceedings of the *Conference On Parallel Architectures And Languages Europe*, Eindhoven, The Netherlands, June 10-13, 1991. It is published by Springer-Verlag in PARLE'91, Lecture Notes in Computer Science, Volume 505(1), pages 83–100. It is available by *anonymous ftp* on the machine `spi.ens.fr` (`129.199.104.3`), `pub/reports/liens/liens-91-5.A4.ps.Z`.

1 Motivation

In this article we propose a methodology for the development of a SIMD machine. The philosophy of the development consists in minimizing the development effort. The excessive complexity of parallel machines is probably the major cause of failure in academic projects. The first quality of a machine is its existence at the end of a project. In this article, we show that it is possible to develop a coarse-grain SIMD machine that offers good performance with very little effort on both hardware and software aspects.

Even if the specificity of this article is to show development reduction, it is important to explain why we have decided to develop this kind of machine. Our main field of interest is image synthesis.

2 Why choose SIMD for image synthesis?

Commercial machines for image synthesis are often very specialized with dedicated hardware to speed up the computation of a single algorithm [HH80, AJ88]. This specialization is the major drawback of this approach. Machines become rapidly obsolete because new rendering algorithms require ever hardware. Only large companies are able to invest large amounts of money and man-power to develop custom machines that will be obsolete in a few months' time. We propose POMP as a non-specialized architecture (with no hardware dedicated to any special algorithm), which is a step beyond the other alternatives (partially non-specialized) proposed in [KV90, FPE+89].

We have to balance the loss of power due to this non-specialization by a massively parallel approach (up to 256 32-bit processors, in fact 8,192 bits of data-paths). This massively parallel organization prohibits the organization in a multiprocessor with shared memory. Each processor has its own local memory and an interconnection network enables data interchange.

This class of architecture contains 2 major subclasses:

- The MIMD machines. Each processor runs its own program on its own data.
- The SIMD machines. Each processor executes the same instruction at the same time on its own data. We do not need a program memory for each processor.

In the graphic pipeline, the last stage is rasterization, which requires most of the computation. A SIMD structure offers the best performance on these computation. [FP81] introduces the concept of *smart memory* which are a set of SIMD memory-PE[1] clusters.

[1] Processing Element.

The POMP project tries to generalize this *smart memory* concept to all the algorithms of the whole pipeline. We need for each cluster a general purpose PE, which is able to handle 32-bit integer numbers, floating point numbers, pointer data types, etc.

We also prefer the SIMD structure because a lot of synchronization problems are avoided [BCJ89] and a high MFlop/dm^3 ratio can be reached [BDW85]. Furthermore we can build a very simple programming model which enables to develop debugging environments.

3 The basis of the machine: the programming model

The efficiency and the programmability are the final targets of computer designing. The relationship between hardware and software is the main problem. Most of our choices for the architecture of POMP are consequences of the programming model.

3.1 The programming model

Variables belong to two classes:

- scalar variables (for standard calculation and flow control),
- parallel variables (also called vectors).

An n-PE SIMD machine is able to simultaneously perform the same operation on a vector of size n. Some SIMD programming environments try to hide the number of processors behind the concept of virtual processors (for instance the Connection Machine). The size of massively parallel variables is assumed to be larger than the number of PEs. Each physical processor emulates one or more virtual processors (*vp*). Vectors are not broken into individual elements but into smaller arrays equally distributed over the PEs.

In a typical massively parallel application, vectors of different sizes are required and need to interact. The vectors must be partitioned into classes called *vp-set* for the CM and *collection* for POMPC. Each *collection* corresponds to one set of virtual processors.

The size is the first attribute shared by the vectors of a same *collection*. The other attributes of the *collection* are:

- the activity. This vector of boolean elements (also called *context*) is the mask which indicates which elements of the vectors of the collection are active.
- the topologic organization. These information describes the topologic relative organization of the virtual processors and the mapping of these virtual processors on the PEs.

3.2 The POMPC language

A detailed description of this language can be found in [Par90]. This model has led to designing of a programming language which is called POMPC. POMPC must be considered as a symbolic macroassembler for SIMD machine as is C for general computers. This language is the direct translation of a programming model and emphasizes the SIMD aspect of the machine: we do not provide an autovectorizing language which hides the structure of the machine from the programmer. This kind of higher level languages can be implemented over POMPC.

Most of the SIMD machines provide this kind of basic language and a taxonomy of many SIMD languages and machines can be found in [Tuc90]. POMPC has been inspired by the previous version of C* [Thi87, pages 35–41] and is rather similar to the new version of this language. MPL [Chr90] and MultiC [Wav90] are also alike (without the collection mechanism).

To implement this model, we must define at any time what the different processors (the scalar one and the different virtual SIMD machines) are doing. As only the scalar processor has the control over the program flow and as the PEs are slaves, the best way to express this dependency is to include the instruction for the PE into the sequential program of the scalar processor (it leads to the definition of a very simple and very convenient controller explained in the next section).

As we expect to write the addition of 2 vectors like the addition of 2 scalars, the major problem is to determine from the source file the location of each calculation. Typechecking on expressions and statements provides these informations.

POMPC is an extension of the Kernighan & Ritchie C [KR78]. The extensions are as follows:

- It is possible to define collections.
- Each variable can be either a scalar (like in C), a vector of the particular `processor` collection (one datum per PE) or a vector belonging to another collection. Each vector is declared as a member of a collection. Thus it is possible to associate a collection with each vectorial statement or each vectorial expression.
- The `where/elsewhere` operators allow to change the activity of a collection, during the execution of a block. This activity is modified according to the value of a boolean vector of the collection. The `where` statement is the equivalent of the `if` statement except that the block is always executed (it mays contains scalar statements or statements concerning other collections). Every other flow-control statements (even `break`, `continue` and `return` but not `goto`) has been translated for a parallel usage.

```
/***********************************************************************
  Mapping a picture on a scrambled surface :
     The view of an underwater chessboard under a dripping tap
***********************************************************************/
#include "pompc.h"                    /* pompc standard include file */
#include "pc_math.h"                  /* pompc math include file */
collection [256,256] pixel;           /* pixel is a 2D 256 x 256 collection */

pixel chessboard()                    /* returns a chessboard picture */
{
    pixel x,y;                        /* x,y : the local coordinates */
    x = pc_coord(0);y = pc_coord(1);
    where((x & 16) ^ (y & 16)) return 255;
    elsewhere return 0;
}

main()
{
    pixel char color,picture;         /* two pictures */
    pixel int x0,y0;                  /* local coordinates */
    pixel int u,v;                    /* mapping coordinates */
    int time,screen;                  /* current time and screen number */

    screen = gr_open_graphic();       /* gets a window where to display the movie */
    gr_set_cmap(screen,0,0,0,-1);     /*     sets a standard color map table... */
    color = chessboard();             /* gets the picture of a chessboard */
    x0 = pc_coord(0) - 128;           /* x0,y0 : coordinate system from */
    y0 = pc_coord(1) - 128;           /*     the center of the chessboard */
    for(time=0;;time++) {             /* and let's go forever... */
    {
        pixel double X,Y,d,d1,phi;
        X = x0;Y = y0;
        d = pc_sqrt(X*X+Y*Y);         /* d : distance from the origin */
        phi = time - d/16.0;          /* phi : phase delay */
        where(phi < 0) phi = 0;       /* drop touches the surface at phi=0 */
        d1 = 1 + 8*pc_sin(phi)/d;     /* d1 : new distance from center */
        u = X * d1 + 128;             /* coordinates where to get the */
        v = Y * d1 + 128;             /*     color of the local pixel */
    }
    picture <- [u,v]color;            /* global indirection */
    gr_flash(screen,&picture,0,0,1,1); /* displays the result */
    }
}
```

Table 1: Example of a POMPC program.

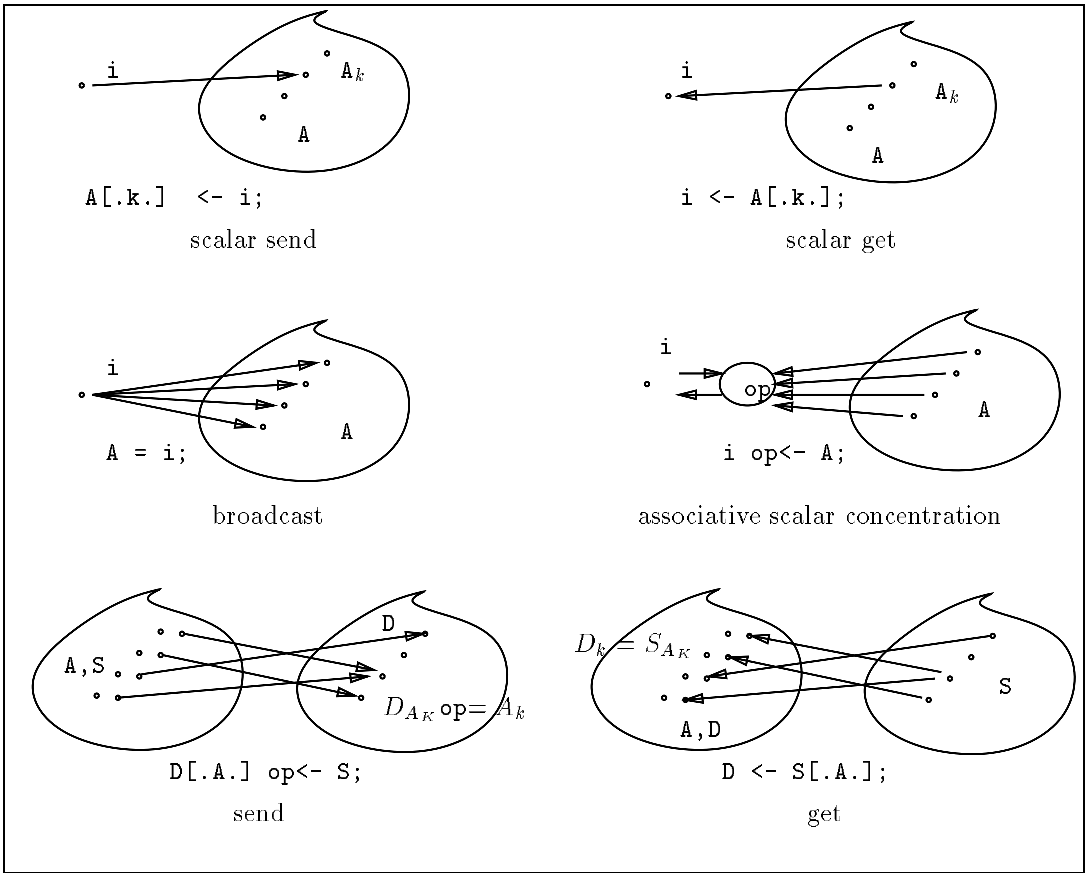

Figure 1: The communications used by POMPC.

- Communications are required to perform non-local interactions. Most of the communications are expressed in the syntax of the language because they require only standard network specificities, the rest being carried out by library functions. Figure 1 summarizes the different syntactical constructions for POMPC communications. The first 4 types of communications are interactions between scalar variables and vectorials ones. The 2 last types are interactions between collections. The `[. .]` operator specifies transformations on the rank of the elements. When a communication may send more than one datum on a given element, a accumulative operator can be specified to accumulate different data in the resulting element. Accumulative operators are addition, subtraction, multiplication, bit-wise and, bit-wise or, exclusive-or, minimum and maximum.

Table 1 shows an example of the POMPC language.

This program computes a chessboard picture (the `chessboard` function) and distorts it (as the deformation of a water surface under a water drip) according to a mapping achieved by a *get*.

4 Architecture of POMP

4.1 Processor designing: a necessary evil?

The first choice during the design of a SIMD machine is the size of the PEs. In fact, this is the first choice because everyone considers that PEs are necessarily custom-made and that we can freely choose the width of the datapath of the PEs.

For some very special applications (mostly image processing), it is interesting to choose 1-bit PE because of the size of the data (from 1 to 8 bits). In order to be efficient these machines require full-custom processors [NCR84]: classical sequential processors are not adapted for this computation, because of the inadequation of the width of the 32-bit processors to 1-bit and 8-bit data.

In the other fields of application for the SIMD (like ours), the required data sizes are more conventional (`int`, `float`, `double`) [Hor82, AB86] and it seems easier and more efficient to use a powerfull processor than to interface a floating-point coprocessor with 1-bit processor, as in the Connection Machine 2. In this last case, 1-bit PEs are no longer used for scientifical computations...

Unfortunately, no commercial 32-bit SIMD PE exists such as the GAPP [NCR84] for 1-bit SIMD machines or the Transputer (an MIMD PE [INM89]).

We consider that PEs for non dedicated SIMD machines must have the same qualities as classical processors. An intermediate choice could be to design a rather small PE with all the necessary hardware required to micro-code efficiently the floating-point operations (like the MasPar machine [Bla90a]).

The consequences of this coarse-grain choice are important because this seems to suppose the development of a very complex PE. We need very broad competences to be able to design competitive 32-bit PE with floating-point and only large semiconductors companies can cope with such developments. The problem is not limited to chip design but also to the development of all the software environment. Developing the PE is not the good solution.

Let us summarize the requirements for the PE:

- a lot of MIPS: an efficient integer ALU,
- a lot of MFLOPS: an efficient floating-point ALU,
- indirect access to local memory: a data address generator,

- local flow control: a local enable mechanism,
- communications: an efficient network and a routing mechanism.

In fact it is very similar to a classical processor.

4.2 Why not use an off-the-shelf processor ?

Such an approach had already be done for the PASM computer [SSNJDK84]. The advantages of using a commercial processor are clear:

- we need not develop a PE, it is cheaper and less time-consuming,
- a C compiler is available for our PE,
- we can benefit from every improvement of the processor (this is very important since the speed of RISC processors is regularly doubled inside a common architecture).
- if we remain rather independent of the processor (particularly in the software domain), it is possible to change processors when it appears that a more suitable architecture has been introduced on the market.

We can drastically limit our developments and provide an easy evolution for our machine. The general concept is to choose the best processor at any given time.

Four sensitive points have to be coped with:

- We have to broadcast an instruction to every processor. This is easier if the chosen PE has a Harvard architecture[2].
- We have to keep every PE synchronous. Each instruction must take the same time independently of the data processed. This is mostly the case in RISC processors (as opposed to microcoded processors), provided that all accesses to the memory last the same time. This prohibits the use of PEs with caches.
- We have to independently freeze every PE to process the `where` statement. It is possible if an "instruction not ready" mechanism is implemented on the code bus. This is the case every RISC processor with off-chip cache.
- We have to provide each PE with an access to the network and some facilities to communicate with the scalar processor. It requires special hardware and which will be discussed in the section 5.

In 1991, there exists one processor presenting the required characteristics: the Motorola 88100 [MOT88]. We chose it as the PE of POMP.

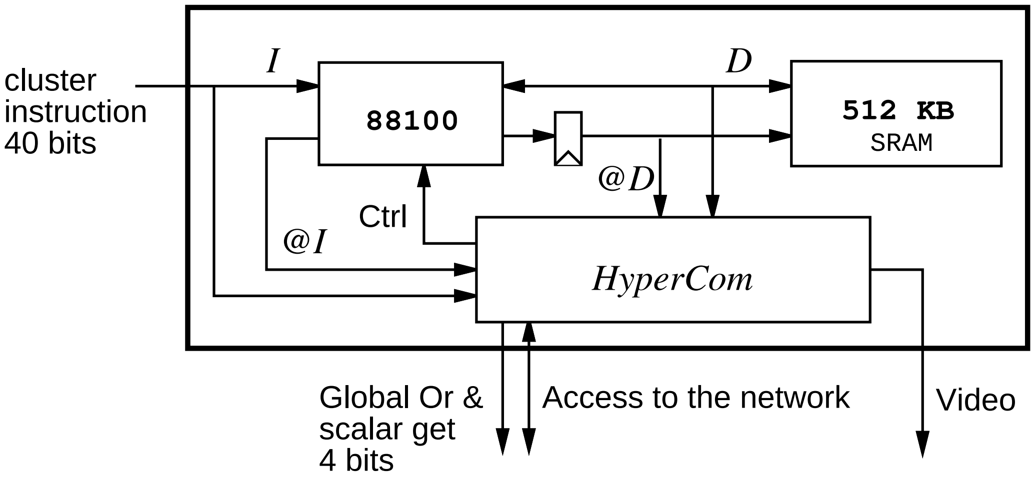

Figure 2: The basic cluster.

4.3 The Processing Element

We can now present the basic cluster for the PE (figure 2). It contains mainly:

- the 88100 at 20MHz (17 MIPS and 7 MFLOPS);
- 128K $\times$ 32-bit static RAM with a 35ns access time in 4 chips;
- *Hypercom*, a chip to customize the CPU to its SIMD environment.

The SIMD approach permits us to use only 9 integrated circuits per PE.

A 40-bit instruction bus broadcasts the instructions and the control of the *Hypercom* chip to the cluster. The *Hypercom* provides the following mechanisms:

- the activity management. Depending on the current activity loaded in the *Hypercom*, The control bus defines for each instruction if it is executed. The eventuality of n nested `where` seems to require an n-depth stack to save the current activity. In fact, it can be implemented with a counter [Ker89, Lev90] which is convenient to cope with the complicated activity handling required when using `break`, `continue`, `return`, `case` and `default`.
- the network access. It consists in limited routing capabilities and shift registers. The section 5 is dedicated to the network and will give some

[2] a special input bus for the instructions.

indications on the hardware required in each *Hypercom* chip for the access to the network.

- the hardware required for the communications between the PEs and the scalar processor: a 4-bit open-collector bus to get the *global or* of a distributed variable. This also allows to send a vector element to the scalar processor (useful for the *scalar get* and for the final stage of the associative scalar concentration) by nibbles of 4 bits as in [Bla90a], which is a good compromise.

- the hardware required to correctly recover from an exception or an interruption.

4.4 The controller and the scalar processor

Some SIMD machines use an independent sequencer to run scalar code or to expand microinstructions generated by an host computer [Thi87]. The use of the host computer as scalar processor facilitates the software development but requires high input/output bandwidth for the broadcast of the code from the host to the PEs. It is possible only if an intermediate sequencer expands some high-level instructions into microcode (typically 32-bit instructions expanded into 1-bit microinstructions, when 1-bit PEs are used). We cannot use such a structure because we need a 20 MHz 40-bit instruction rate. The scalar processor must be directly located in the SIMD machine. This choice has been made in [Bla90a, AB86].

As we use a commercial processor for the PE, it is natural to use the same processor as scalar processor for code orthogonality and for easier synchronization between PEs and the scalar processor. It simultaneously fetches its own 32-bit instruction and the 40-bit instruction broadcasted to the clusters. The whole machine is driven by a 72-bit Long Instruction Word (*LIW*). The figure 3 presents the global architecture of POMP.

An history of the cluster instructions is saved in a FIFO: when an exception occurs, the PEs can correctly resume execution. The scalar processor can access a register to override some fields of vectorial instructions, enabling scalar broadcasts of values.

Since the most important argument claimed by the SIMD defenders is the removal of synchronization issues, we think that the implicit LIW is a way to go further in the synchronization of scalar code with the parallel code, allowing a more global code optimization.

5 The interconnection network

Choosing a network consists in choosing the best trade-off between performance and cost for a given class of applications.

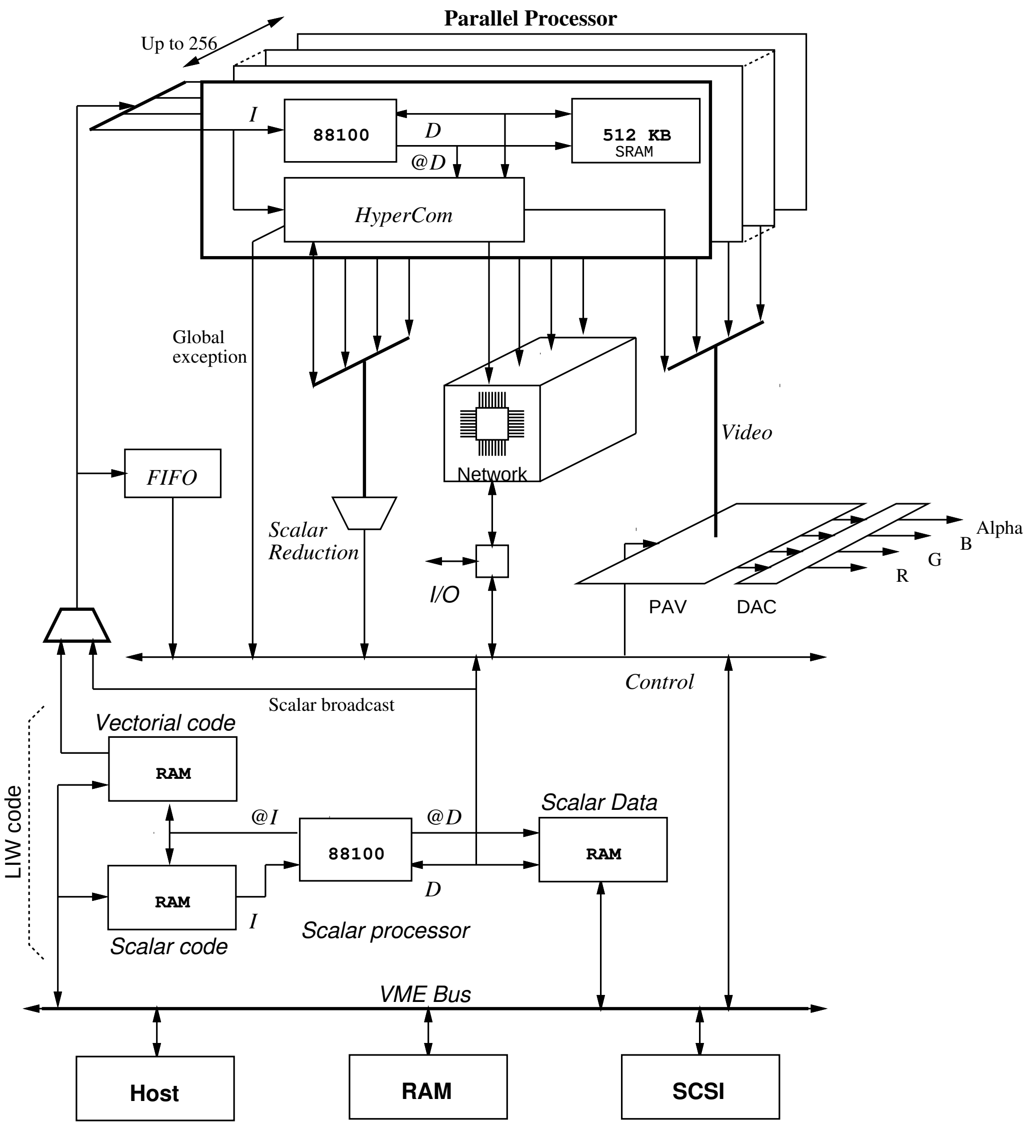

Figure 3: The global architecture of POMP.

5.1 Measuring the performance

Applications may require different communication types:

- random access,

- 1-neighbour access,
- all neighbours according to a multidimensional mesh,

and different object granularities for the network:

- size of the packets (1 bit to 1 Kbit),
- number of physical processors,
- number of virtual processors per physical processor (*vp-ratio*).

To measure the needs of a target application, we have to evaluate the occurrence of every combination above. A unit system is required for such measurements to compare the performance of the network with the performance of the PEs. We have decided to speak in terms of:

$$\frac{\text{time required for the communication of 32 bits for each virtual processor}}{\text{time required for the addition of 32 bits for each virtual processor}}$$

5.2 Measuring the cost

The global cost of the machine depends on the cost of the PEs and the cost of the network. The latter is not easy to evaluate because it is not a linear function of the performance. This cost grows by step when the implementation must move from one technology to another at different hierarchical levels [FWT82]:

- the number of transistors required for the network by each PE,
- the number of pins required by each PE,
- the density on each motherboard (the number of routing levels on motherboards),
- the number of connections between motherboards.

5.3 Choosing the Network

Many network designs have been described in the literature and can be classified according to some criteria such as operation mode, control strategy, switching method and network topology [Fen81, Gil86, Kot87].

In our case, the network is synchronous (SIMD machine) and the control is distributed (for scalability and simplicity). The choice of the switching method is not obvious:

- packet switching requires local storage and more complex hardware,
- circuit switching needs to establish a connection through several physical links.

The network topology is probably the major issue in parallel computer designing because it depends on the applications and on almost all the machine parameters.

5.4 Implementation of a hybrid interconnection network

We propose a network for applications which require mostly random accesses (required for image synthesis with distributed data-base) but also simultaneous accesses to all neighbours on a multidimensional mesh.

These aspects seem incompatible and would require respectively a dynamic (switched) and a static network. Existing machines demonstrate it:

static network CM-2 [Thi87], ILLIAC IV [Hor82], MPP [Bat80],

dynamic network PASM [SSNJDK84], OPSILA [AB86],

static network and dynamic network MasPar [Bla90b].

A candidate for the static network is the hypercube network and a candidate for the dynamic network is the indirect binary cube MIN (multistage interconnection network) [Sch91].

But since a dynamic network is a spatial unfolding of a static network, it must be possible to use the physical wires between switches as a static network instead of using two separated networks like in MasPar.

Each stage of our hybrid MIN can be seen as a dimension of the hypercube. If such a MIN is built with $n \log_p n$ switches instead of $\frac{n}{p} \log_p n$ for a cube MIN with n PEs and $p \times p$-crossbars[3] (ie n lines of $\log_p n$ crossbars instead of $\frac{n}{p}$ lines), it is possible to partition the MIN into n similar subsets mapped on a hypercube, as seen on figure 4 for $n = 4$ and $p = 2$.

The classical design approach leads to the development of an ASIC (Application Specific Integrated Circuit) for the *Hypercom*. This is not convenient because we are quite obliged to redesign the ASIC if the number of processors or the network change.

In order to follow our minimalist philosophy, the *Hypercom* circuit can be implemented with some reprogrammable LCA (Logic Cell Arrays) such as the new 4000 family of Xilinx [XIL90], which offers the required performance, complexity and pin count. Each switch is reversible, offers broadcast capabilities and uses a destination tag algorithm to establish a connection.

For communications on a mesh, a control bit enable changing from the dynamic to the static network. Thus routing overheads are avoided.

[3] We consider a generalized hypercube pattern with p PEs totally interconnected on each dimension. $p = 2$ for the standard hypercube.

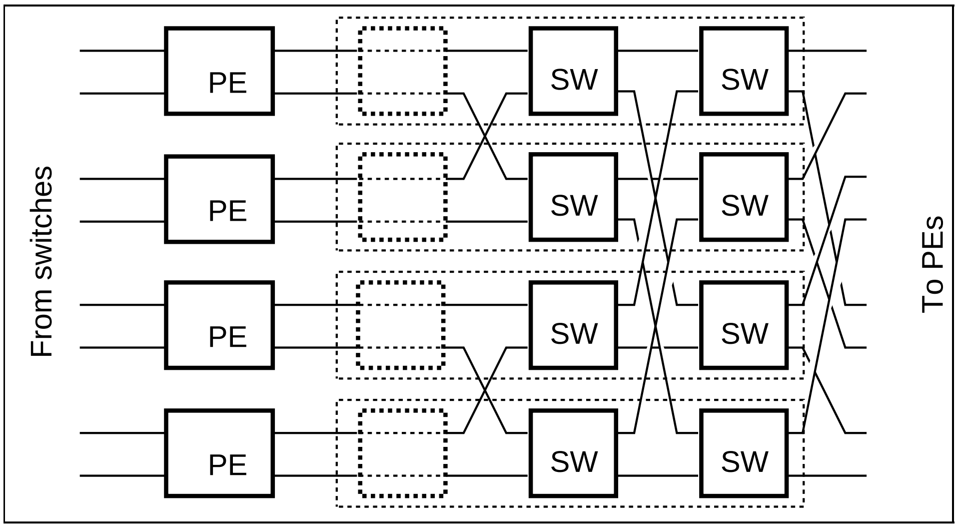

Figure 4: The network for $n = 4$ and $p = 2$.

5.5 Performance and cost

This study is illustrated for the case of 256 + 1 processors packaged as:

- 1 controller board,
- 16 motherboards of 16 PEs

in a 19" *Triple Europe* rack.

The performance evaluation of the network for random routing is complex, contrary to neighbourhood communications. We have simulated the random routing for a high *vp_ratio*, as shown in table 2, with 1-bit datapaths. Related costs are represented in table 3.

8-stage and 4-stage (figure 5) networks present a correct trade-off between performance for random routing and cost. They are both small enough to be implemented in the *Hypercom* with a reprogrammable LCA, even with 4-bit datapath, for the 8-stage MIN, which is then very performant.

6 The code generation for POMP

Figure 6 illustrates the code generation process for POMP. The final instruction is 72-bit wide and consists in the following fields:

- a 32-bit instruction for the scalar processor;

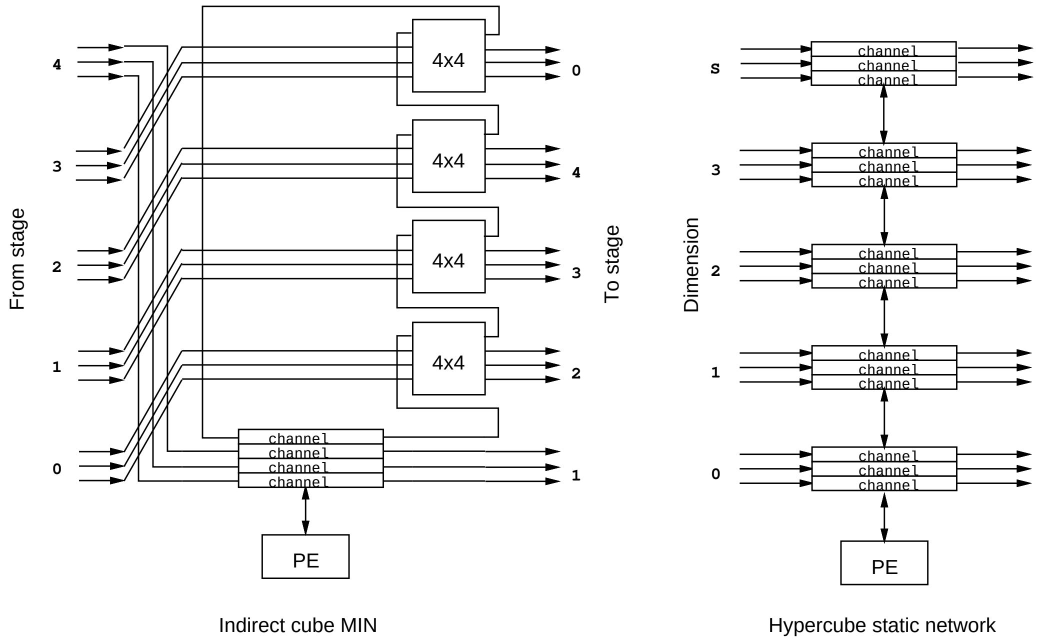

Figure 5: The two configurations of the network with $n = 256$ processors and $p = 4$.

Network type	*Average efficiency*	*Average #cycles per 32 bits*				*Peak throughput*[a]
		`int`	`double`	256 bits	∞	
8 stages, 2×2	0.30	66.8	63.4	60.8	60	1.3 GB/s
4 stages, 4×4	0.37	41.5	38.5	36.2	35.4	2.6 GB/s
2 stages, 16×16	0.48	21.4	15	10.2	8.6	10 GB/s
1 stages, 256×256	0.63	15.8	11.1	7.5	6.3	10 GB/s[b]

[a] *For regular routing, like matrix multiplication.*
[b] The throughput is limited by the PE data bus.

Table 2: Performance of some hybrid networks.

- a 32-bit instruction broadcasted to the PEs;
- an 8-bit instruction to control the *Hypercoms*.

The global idea of our code generation consists in using commercial compiler and assembler, which is coherent with our philosophy to develop as little software as possible.

Network type	*#Links /PE*	#Communication *pins/PE*	#switches */PE*	*#Wires between motherboards*
8 stages, 2×2	2	18	32	158
4 stages, 4×4	4	30	64	282
2 stages, 16×16	16	90	512	960
1 stages, 256×256	256	960	65536	15360

Table 3: Costs of some hybrid networks.

The most complex part for this generation is the splitting of the POMPC source file into two C files. This is the first phase of a compiler. It is necessary to develop a parser for POMPC. A typechecker identifies the collection of each expression and each statement. An instruction breaker cuts the different parts of expressions to separate instructions with scalar side effects (scalar assignment, scalar increments and decrements, function calls,...) from purely vectorial instructions which depend on an activity and are repeated as many times as necessary to handle the virtual processing management[4]. Consecutive instructions depending on the same collection are then associated to share the same virtual management loop. Vectorial local variables declared in block handled in a single virtual management loop are relegated to the *processor* collection: local arrays are transformed into single elements which are commonly compiled in registers (this confirms the usefulness of a RISC processor for the PEs). The final step of the program is the code generation. This part is simple because the generated code is C which is a symbolic language. Three kinds of generators are used:

- the generation of the code for POMP. The different files are generated with synchronization points declared in the C codes by dummy function calls to pseudo-functions (`synchro_1()`, `synchro_2()`,...)
- the generation of a C file for one or more processes simulation: it is in fact the same generation of code but in a single file. Each PE is simulated by a Unix process which also runs the scalar code. A shared memory segment allows to synchronize the processes and to simulate the communication. Thus it is possible to develop in POMPC the communication routines for POMP and for the simulation. It allows to measure the performance of the network. The number of processes is defined by an environment variable. The monoprocess simulation is a multiprocess simulation with only one PE.
- the generation of a CPaRIS code (the C Parallel Instruction Set of the

[4]This is the virtual management loop.

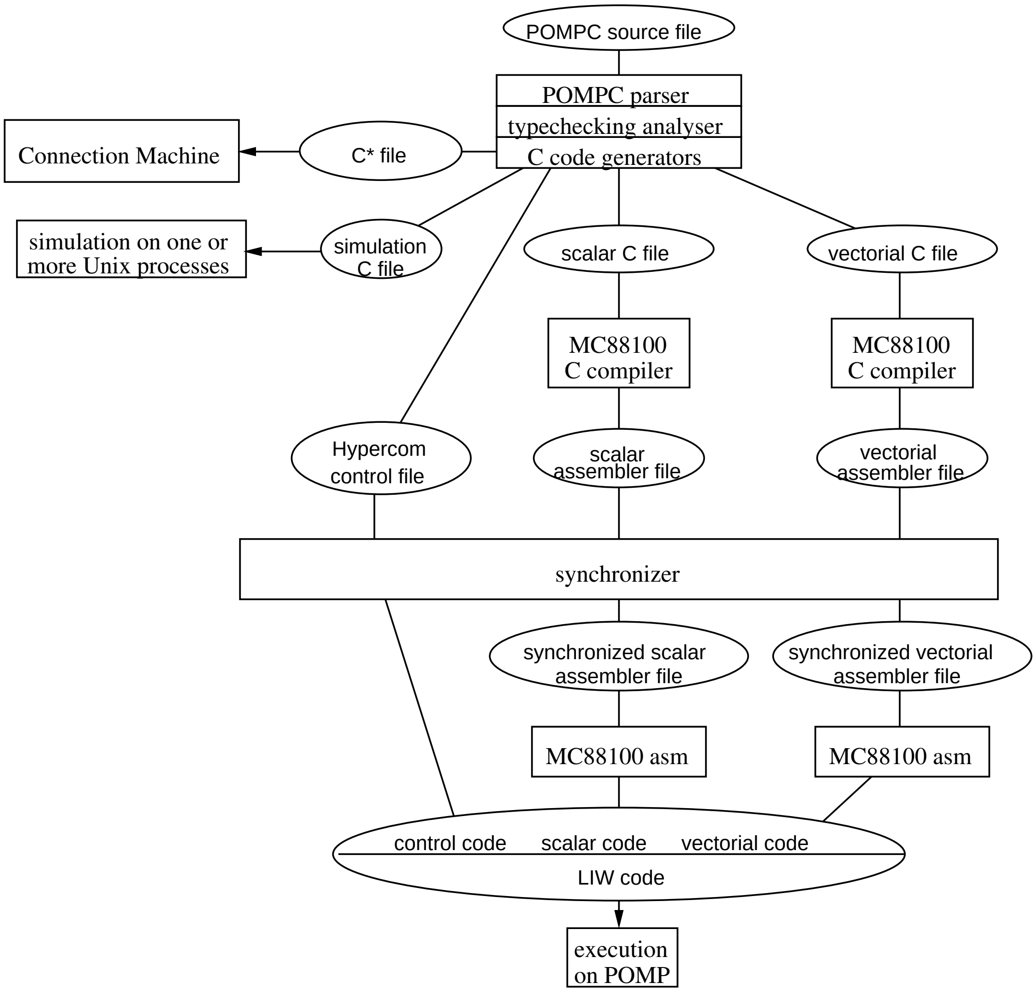

Figure 6: The code generation diagram.

Connection Machine [Thi87]) which allows to perform real-time simulations on the Connection Machine[5].

- other code generators can be thought of; for instance it could be interesting to write some for the Intel Hypercube or for the Sequent Machine and to study the interest of a SPMD language to program MIMD machines.

The second program to be developed is the synchronizer. It takes as inputs

[5] This language is used by people programming on the Connection Machine

the three files for each field and resynchronizes them. This program must understand the assembly code of the target processor (here the MC88100) in order to identify the synchronization pseudo-function calls. Synchronization is achieved by inserting `nop`s in the code to be delayed. This program must also take into account the pipeline structure of the controller and the internal pipeline of the MC88100. The internal scoreboarding of the chip must be managed at compile time to avoid the desynchronization of the whole machine at run time.

The third program is a simple loader with a parallel symbolic debugger.

Only the last two programs depend on the type of the choosen processor and must be rewritten if we choose another processor. This limits the complexity of the development for today and for tomorrow.

7 Conclusion

Choosing to limit the developments does not lead to poor performances.

As concerns the hardware, we have only to develop the controller board, which is easy thanks to the use of a commercial processor, the replicated module of the cluster (a very small board with 9 circuits) and the interconnections of the 16 mother boards.

Software developments are limited to the development of a POMPC preprocessor (20,000 lines of C) and the synchronizer (5,000 lines of C).

It is possible for programs requiring little networking (at most one global indirection every 50 instructions) to reach the full efficiency of the machine: 4000 MIPS and 1700 MFLOPS with a small machine ($\approx$ 1 kW).

8 Current Work

A 3-processor machine is now under development. It will demonstrate the feasibility of the controller and of the programming concepts. As soon as credits can be found (we need 1 MFF in commercial chips) a prototype with 257 processors will be built.

The POMPC compiler is written. Simulations on the Connection Machine 2 (located at the ETCA) and on Unix work. Small applications like a One-Step-Relaxation electrical simulator have been developed in POMPC and run on the Connection Machine and on the Unix simulators. Some aspects of the semantic of POMPC have been studied by Luc Bougé and Jean Luc Levaire [Bou90, Lev90]. A ray-tracer is under development using the spatial coherence of the rays with beam tracing techniques developed in [Thi90].

References

[AB86] M. Auguin and F. Boeri. The opsila computer. In INRIA, editor, *Parallel Algorithms & Architectures*, pages 143–153. North-Holland, 1986.

[AJ88] Kurt Akeley and Tom Jermoluk. High-performance polygon rendering . In *Computer Graphics (SIGGRAPH '88)*, volume 22(4), pages 239–246. ACM, Août 1988.

[Bat80] Kenneth E. Batcher. Architecture of a massively parallel processor. In *SIGARCH 80*, pages 168–173. The Institute of Electrical and Electronics Engineers, Inc., 1980.

[BCJ89] Edward C. Bronson, Thoms L. Casavant, and Leah H. Jamieson. Experimental application-driven architecture analysis of an simd/mimd parallel processing system. In *International Conference on Parallel Processing*, pages 59–67. The Institute of Electrical and Electronics Engineers, Inc., Academic Press, 1989.

[BDW85] John Beetem, Monty Denneau, and Don Weingarten. The gf11 supercomputer. In *SIGARCH 85*, pages 108–115. The Institute of Electrical and Electronics Engineers, Inc., 1985.

[Bla90a] Tom Blank. The design of the maspar mp-1, a cost-effective massively parallel computer. In IEEE, editor, *IEEE Compcon Spring 1990*, February 1990.

[Bla90b] Tom Blank. The maspar mp-1 architecture. In IEEE, editor, *IEEE Compcon Spring 1990*, February 1990.

[Bou90] Luc Bougé. On the semantics of languages for massively parallel simd architecture. Technical Report LIENS-90-13, Laboratoire d'Informatique de l'Ecole Normale Supérieure, Juin 1990.

[Chr90] Peter Christy. Software to support massively parallel computing on the maspar mp-1. In IEEE, editor, *IEEE Compcon Spring 1990*, February 1990.

[Fen81] Tse Yun Feng. A survey of interconnection networks. *Computer*, 14(12):12–27, Décembre 1981. The Institute of Electrical and Electronics Engineers, Inc.

[FP81] Henry Fuchs and John Poulton. Pixel-plane: a vlsi-oriented design for a raster graphics engine. *VLSI Design*, 2(3), 1981.

[FPE+89] Henry Fuchs, John Poulton, John Eyle, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. Pixel-plane 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. In *Computer Graphics (SIGGRAPH '89)*, volume 32(4), pages 79–88. ACM, Juillet 1989.

[FWT82] Mark A. Franklin, Donald F. Wann, and William J. Thomas. Pin limitation and partitionning of vlsi interconnection networks. *IEEE Transactions on Computers*, C-31(11):1109–1116, Novembre 1982.

[Gil86] Wolfgang K. Giloi. Interconnection networks for massively parallel computer systems. In *Future Parallel Computers*, volume 272, pages 321–348. Springer-Verlag, 1986.

[HH80] James H.Clark and Mark R. Hannah. Distributed processing in a high-performance smart image memory . *Lambda*, 1(4):369–374, 1980.

[Hor82] R. Michael Hord. *The ILLIAC IV, The First Supercomputer*. Computer Science Press, 1982.

[INM89] Inmos. *The Transputer Databook* , 1989.

[Ker89] Ronan Keryell. Pomp2 : D'un petit ordinateur massivement parallèle. Rapport de magistère, LIENS — Ecole Normale Supérieure, Octobre 1989.

[Kot87] S. C. Kothari. *Multistage Interconnection Networks fo Multiprocessor Systems*, volume 26, pages 155–199. Academic Press, 1987.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice-Hall, 1978.

[KV90] David Kirk and Douglas Voorhies. The rendering architecture of the dn10000vs. In *Computer Graphics (SIGGRAPH '90)*, pages 299–307. Association for Computing Machinery, Août 1990. Volume 24, Number 4.

[Lev90] Jean-Luc Levaire. *Deux sémantiques opérationnelles pour POMPC*. Diplôme d'etude approfondie, LIENS, Paris, Septembre 1990.

[MOT88] MOTOROLA. *MC88100 RISC processor user's manual* , 1988.

[NCR84] NCR. *Geometric arithmetic parallel processor NCR45CG72* , 1984.

[Par90] Nicolas Paris. Définition de pompc (version 1.5). Technical report, LIENS, Février 1990.

[Sch91] Isaac D. Scherson. Orthogonal graphs for the construction of a class of interconnection networks. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):3–19, Janvier 1991.

[SSNJDK84] Howard Jay Siegel, Thomas Schwederski, IV Nathaniel J. Davis, and James T. Kuehn. Pasm: A reconfigurable parallel system for image processing. *ACM SIGARCH Newsletter*, 12(4):7–19, Septembre 1984.

[Thi87] Thinking Machine Corporation. *Connection Machine Model CM-2 Technical Summary*, Avril 1987. HA87-4.

[Thi90] Jean-Philippe Thirion. Interval arithmetic for high resolution ray tracing. Technical Report LIENS-90-4, Laboratoire d'Informatique de l'Ecole Normale Supérieure, Février 1990.

[Tuc90] Russ Tuck. *Porta-SIMD: An Optimaly Portable SIMD Programming Language* . PhD thesis, University of North Carolina at Chapel Hill, Mai 1990.

[Wav90] Wavetracer Inc. *The multiC Programming Language: Extending C to Accomodate Data Parallel Processing*, 1990.

[XIL90] XILINX. *XC 4000 Logic Cell*^{TM} *Array Family*, 1990. Technical Data.

PUBLICATIONS DU LIENS

90 - 1	M.P. GASCUEL A. VERROUST C. PUECH	Animation with Collisions of Deformable Articulated Bodies
90 - 2	B. VIROT	Parallelization of the Simulated Annealing Algorithm Application to the Placement Problem
90 - 3	J.P.THIRION	Tries : Data Structures Based on Boolean Representation for Ray Tracing
90 - 4	J.P. THIRION	Interval Arithmetic for High Resolution Ray Tracing
90 - 5	F.P. PREPARATA J.S. VITTER M. YVINEC	Output-Sensitive Generation of the Perspective View of Isothetic Parallelepipeds
90 - 6	L. BOUGÉ P. GARDA	Towards a Semantic Approach to SIMD Architectures and their languages
90 - 7	L. PUEL A. SUÁREZ	Compiling Pattern Matching by Term Decomposition
90 - 8	D. DURE	Simulation Multi-Mode de Circuits VLSI (Thèse)
90 - 9	P.L. CURIEN	Substitution up to Isomorphism
90 -10	P.L. CURIEN G. GHELLI	Coherence of Subsumption
90 -11	F. FAGES	A New Fixpoint Semantics for General Logic Programs Compared with the Well-Founded and the Stable Model Semantics
90 -12	A. BOUVEROT	Pliage/Dépliage et Extraction de Programmes Logiques: Présentation Comparée
90 -13	L. BOUGÉ	On the Semantics of Languages for Massively Parallel SIMD Architectures

90 -14	K. BRUCE R. DI COSMO G. LONGO	Provable Isomorphisms of Types
90 -15	F. FAGES	Consistency of Clark's Completion and Existence of Stable Models
90 -16	J.D. BOISSONNAT O. DEVILLERS R. SCHOTT M. TEILLAUD M. YVINEC	Applications of Random Sampling to On-line Algorithms in Computational Geometry
90 -17	J.P. THIRION	Utilisation de la Cohérence des Rayons Lumineux pour le Lancer de Rayons (Thèse)
90 -18	M. POCCHIOLA E. KRANAKIS	Camera Placement in Integer Lattices
90 -19	R. M. AMADIO	Domains in a Realizability Framework
90 -20	G. LONGO	Notes on the Foundation of Mathematics and of Computer Science
90 -21	G. LONGO E. MOGGI	Constructive Natural Deduction and its "ω"-set Interpretation
90 -22	M.P. GASCUEL	Déformations de Surfaces Complexes : Techniques de Haut Niveau pour la Modélisation et l'Animation (Thèse)
90 -23	M. POCCHIOLA	Trois Thèmes sur la Visibilité : Enumération Optimisation et Graphique 2D (Thèse)
90 -24	Y. LAFONT A. PROUTÉ	Church-Rosser Property and Homology of Monoids
90 -25	P.H. CHEONG	Compiling Lazy Narrowing into Prolog
90 -26	P.H. CHEONG L. FRIBOURG	Efficient Integration of Simplification into Prolog
90 -27	A. VERROUST	Etude de Problèmes liés à la définition, la Visualisation et l'Animation d'Objets Complexes en Informatique Graphique (Thèse d'Etat)
90 -28	L. FRIBOURG	Generating Simplification Lemmas Using Extended Prolog Execution and Proof-Extraction
90-29	L. COLSON	Représentation Intentionnelle d'Algorithmes dans les Systèmes Fonctionnels : une Etude de Cas. (Thèse)
91 - 1	G. BERNOT	Testing Against Formal Specifiations : a Theoretical View
91 - 2	A. BOUVEROT	Comparaison entre la Transformation et l'Extraction de Programmes Logiques.(Thèse)
91 - 3	R. AMADIO	Bifinite Domains : Stable Case
91 - 4	A. BOUVEROT	Extracting and Transforming Logic Programs
91 - 5	P. HOOGVORST R. KERYELL P. MATHERAT N. PARIS	POMP or How to Design a Massively Parallel Machines with Small Developments

91 - 6	G. BERNOT M. BIDOIT T. KNAPIK	Observational Approaches in Algebraic Specifications: a Comparative Study
91 - 7	Y. LAFONT A. PROUTÉ	Church-Rosser Property and Homology of Monoids (revised version)
91 - 8	G. BERNOT M. BIDOIT	Proving the Correctness of Algebraically Specified Software: Modularity and Observability Issues
91 - 9	M. BIDOIT	Development of Modular Specifications by Stepwise Refinements Using the PLUSS Specification Language
91 - 10	R. Di COSMO	Invertibility of Terms and Valid Isomorphisms. A Proof Theoretic Study on Second Order λ-calculus with Surjective Pairing and Terminal Object
91 - 11	R. Di COSMO P.L. CURIEN	A Confluent Reduction for the λ-calculus with Surjective Pairing and Terminal Object and Terminal Object
91 - 12	P. CRÉGUT	Machines à Environnement pour la Réduction Symbolique et l' Evaluation Partielle (Thèse)
91 - 13	L. CHILLAN	Typing with Type Relations and ML-Polymorphism
91 - 14	P.COUSOT R. COUSOT	Inductive Definitions, Semantics and Abstract Interpretation
91 - 15	A. ASPERTI	A Linguistic Approach to Deadlock
91 - 16	P.L. CURIEN T. HARDIN A. RÍOS	Normalisation Forte du Calcul des Substitutions
91 - 17	N. PARIS	MOD2MAG User's Manual