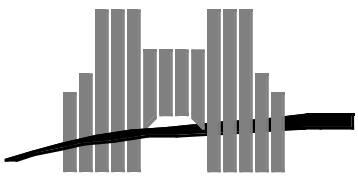
Laboratoire de l'Informatique du Parallélisme
Ecole Normale Supérieure de Lyon
Institut IMAG
Unité de Recherche Associée au CNRS n° 1398

Evaluation of the MasPar MP-1 Performances

Cyril GAVOILLE February 1992

Technical Report N° 92-01



Ecole Normale Supérieure de Lyon
46, Allée de l'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33 72 72 80 00; Télécopieur :+33 72 72 80 80
Adresses électroniques :
lip@frensl61.bitnet; lip@lip.ens-lyon.fr (uucp)

Evaluation of the MasPar MP-1 Performances

Cyril Gavoille

February 1992

Abstract

The goal of this study is to create a model of the various calculation and transmission operations on a massively parallel computer, the MasPar*. This machine holds over a thousand processors capable of processing information. These two problems, processing and transmission, are achieved by an architecture of the SIMD type (Single Instruction Multiple Data). We present by thorough studies of the instruction set made available to the user, and this ends up in the evaluation of performances in these two areas, thanks to the programming of benchmarks.

Keywords: MasPar MP-1, Performances, Benchmarks, communications

Résumé

Le but de cette étude est de modéliser les différentes opérations de calculs et de communications sur un ordinateur massivement parallèle, la MasPar*. Cette machine comporte plus d'un millier de processeurs capables de traiter et de communiquer de l'information. Ces deux problèmes, traitement et communication, sont réalisés par une architecture de type SIMD (Single Instruction Multiple Data). La modélisation est obtenue par des études expérimentales approfondies du jeu d'instructions mis à la disposition de l'utilisateur, et aboutit à l'évaluation de performances dans ces deux domaines, grâce à la programmation de bancs d'essais.

Mots-clés: MasPar MP-1, Performances, Benchmarks, communications

Evaluation of the MasPar Performances

Cyril GAVOILLE

Training Course for a Masters in Computer Science

Claude Bernard University

- 1991 -

Produced within the LIP

Table of Contents

1. Introduction……3
2. Architecture of the MasPar……4
2.1. The frontal or FE (Front End)……4
2.2. The sequencer or ACU (Array Control Unit)……4
2.3. The processors grid……4
2.4. Processors or PE (Processor Element)……4
2.5. Clusters ……5
2.6. DPU (Data Parallel Unit)……5
3. Elementary arithmetic operations……8
3.1. General characteristics of programming language ……8
3.2. Performance drives ……8
3.3. The different variables of simple types……9
3.4. Presentation of the experimental measures results ……11
3.5. Elementary arithmetic operations and basic operations……12
3.6. Table analysis ……14
3.7. Maximal performances obtained on the MasPar……15
4. Communications procedures……17
4.1. Generalities ……17
4.2. Assignments……17
4.3. Proc communication……18
4.4. X-Net communication……20
4.5. Router communication ……32
4.6. Summary of the communication procedures……38
5. Typical communication pattern ……39
5.1. One to all……40
5.2. Transposition……41
5.3. Personalized one to all……44
6. Data processing with communication……47
6.1. Scan, reduce and sendwith functions……47
6.2. Sorting and ranking functions……48
6.3. Cosine, square root, arctangent and absolute value functions……48
7. Conclusion……49
8. Appendix……50
8.1. Xnet, xnetp and xnetc communication table……50
8.2. Xsend and xfetch communication table ……51
8.3. Router communication table……55
8.4. Rsend and rfetch communication table……56
8.5. TOOLS.H header file……60
9. References ……67

1. Introduction

Since the beginnings of Parallel in Computer Science, it has appeared that the strength of parallel machines, that is, the capacity to quickly solve problems, is not only measured by the intrinsic power of each processor. It depends indeed on the global organization of processors and the way they exchange information. To care for more and more complex and growing problems, we will only be able to determine the architecture of the machine in use if we precisely know its performances in the processing and communication operations.

The goal of this study is to create a model for the various calculation and communication operations on a massively parallel computer, the MasPar. This machine holds over a thousand processors capable of processing information. These two problems, processing and communication, are achieved by an architecture of the SIMD type (Single Instruction Multiple Data). The demonstration is obtained by thorough studies of the instruction game made available for the user, and this ends up in the evaluation of performances in these two areas, thanks to the programing of benchmarks.

2. Architecture of the MasPar

The MasPar (Massively Parallel) at ENS Lyon is a massively parallel SIMD machine (Single Instruction Multiple Data) whose processors are linked by a two dimensional toric connection network. We distinguish several elements on this computer: the frontal, the sequencer, the grid, the processors and the clusters.

2.1. The frontal or FE (Front End)

It is a 3520 VAX STATION operating under the ULTRIX V4.0 operating system where the compilation of the source program takes place. Starting from the DEC compiler, we generate a program in the machine language particular to the Mas Par, which is globally sent on the sequencer.

2.2. The sequencer or ACU (Array Control Unit)

It is a 4 bits processor that has its own registers (32 of the 32 bits registers), its own memory for instructions (1 RAM MBytes) and data (120 KBytes). It simultaneously receives the executable code and is charged, at the time of execution, to send the instructions as well as data to all the grid processors. This is achieved by a specialized bus. It has 5 of the 32 bit registers which can be used by the programmer.

2.3. The processors grid

It is a two dimensional toric grid with 32 x 32=1024 processors. This grid defines the network connection, that it is the means of communication between processors. Each processor can communicate with its 8 neighbors forming a star around it. This physical connection is good for all the processors and this is done in a cyclical fashion (the extreme processors are linked to each other). We notice the communication times from whichever PE to one of its neighbors to be equal, no matter which direction it may take. The grid processors are identified by a number starting from 0 to 1023. They are numbered just as a matrix (line, column). That way #0 is physically connected to #1, 33, 32, 62, 31, 1023, 992 and #993 (see Figure 2.2)

2.4. Processors or PE (Processor Element)

Their role is to execute the instructions in a synchronous way, at a 12.5MHz clockrate, on their own data or on a memory zone. They have a 16 Kbytes private memory and 192 bytes register space whose access are 10 times faster than those of the PE's RAM. 32 bytes are reserved for the instruction's micro-code and the remaining 160 can be used by the programmer. The operations on the register can be done on 1, 8, 16, 32 or 64 bits. However PE's don't have much calculation strength as they are 4-bit processors. Each PE contains an arithmetic and logic unit, which performs standard operations on the registers. Besides, a memory unit execute the memory load and store operations. Theses two units run concurrently

on the PE's. On each PE there are two control units, one controlling the register calculations, the other the memory access, which is capable of operating parallely.

2.5. Clusters

Another organization of PE's exists in the grid itself: clusters. They regroup 16 PE's at the grid, organized in a 4x4 sub-grid (see figure 2.1). Then, there is a total of 64 clusters. This regrouping occurs at the level of the global communications. Indeed, all communications between clusters are parallel to each other (simultaneously) but only one of the 16 PE's of the cluster comes in action. If several PE's of a same cluster need to transmit in a non-uniform way (with another cluster or with itself) then this global communication will be done sequentially (see the `router` communication).

2.6. DPU (Data Parallel Unit)

This concerns all that is composed of the ACU and the PE's grid.

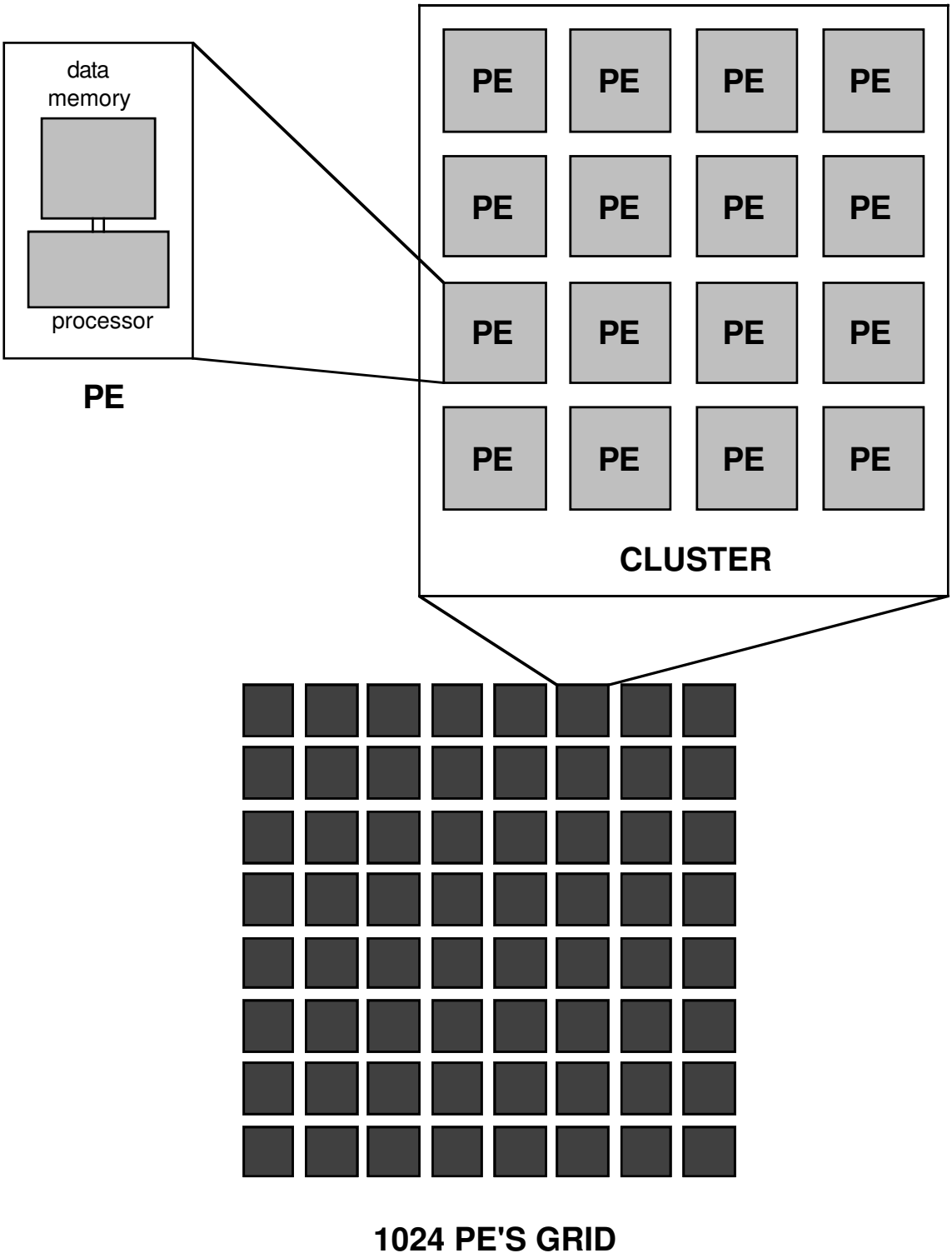


Figure 2.1: Processors grid regrouped in clusters

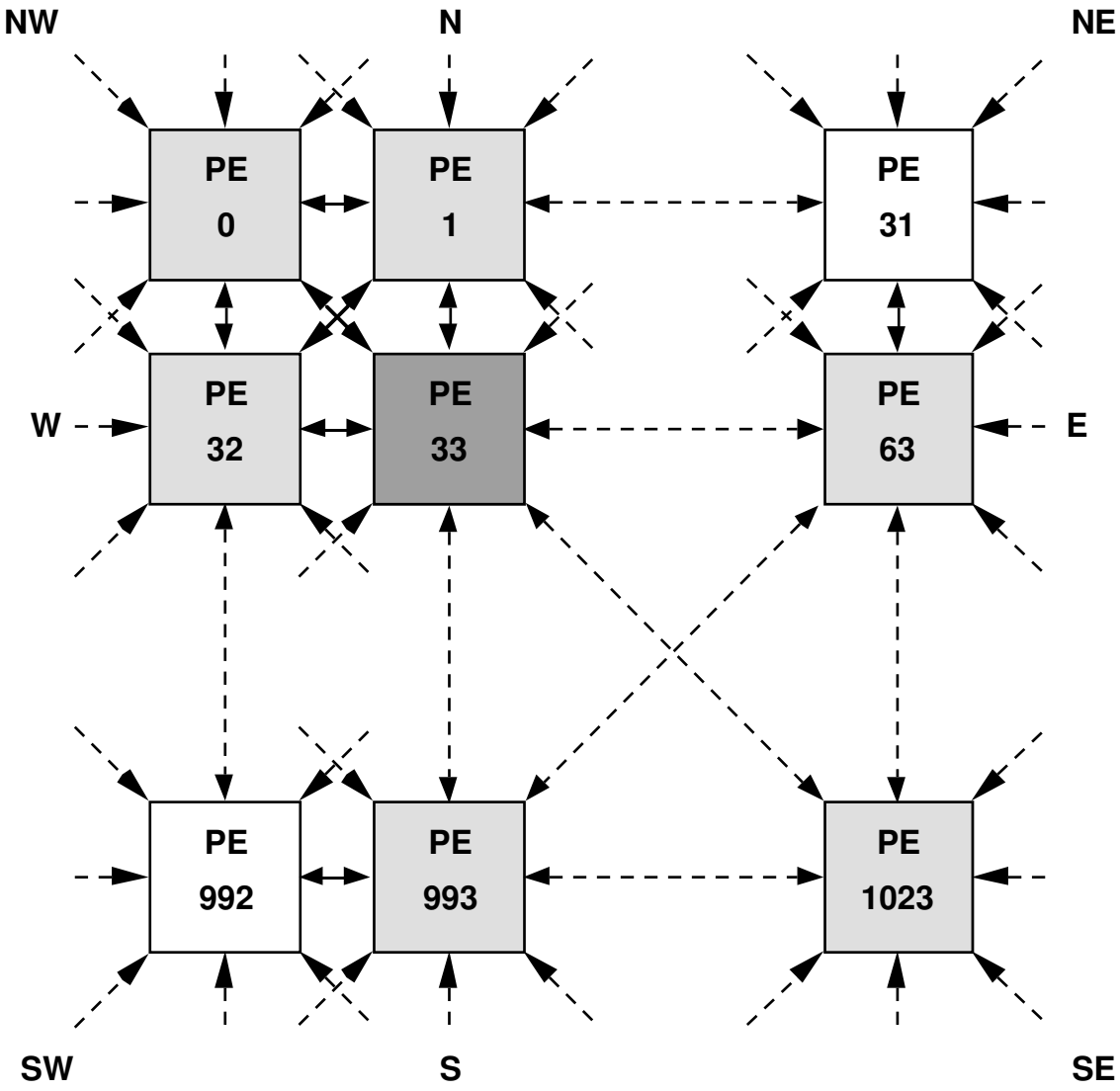


Figure 2.2: The physical connections between PE's and the 8 directions of the X-Net communications

3. Elementary arithmetic operations

This paragraph describes the elementary operations of the ACU (sequential calculation) as well as at the PE's operations (parallel calculation).

3.1. General characteristics of the programming language

The programming language used is the C language, standard to [K&R], extented by a MPL module (MasPar Parallel Application Language) to express the parallel instructions. We must note that standard C instruction (in fact micro-instructions), sent by the ACU on the grid, is parallely executed by all PE's. There is a possibility for instance to select some of the PE's of the grid to keep the PE's from modifying their data all in the same way. We then use the conditional instructions of the K&R C on a local variable, for instance the PE's number (each contains an `iproc` that has the PE's number). If a processor is not selected it goes into an inactive state, that is, it executes the instructions while protecting its memory zone from the writing mode. Because the ACU has its own memory, we will be able to refer to two types of variables in memory: those of the ACU (sequential) and those of the PE's (parallels, differentiated by the key-word, `plural`, placed as a prefix in the standard definition of the C variables).

3.2. Performance drives

All the recorded times are truly observed times, that is that they are obtained from a program written in C on the DPU. So the times take into account the specific instruction times as well as the memory allocation, the types of variables, the errors, all of these managed by the compiler. All of the times are obtained thanks to a macro giving the number of the clock cycles for whichever instruction block:

Syntax: `CLOCK(`*`instruction`*`,n,t)`

where *`instruction`* must be a simple instruction or a function call, or else a block of instructions. `n` is the number of repetitions of the measure and `t` the variable name (which must be a global `unsigned int`) receiving the total number of cycles of the repetition of instructions. This macro initializes then reads a register of the MasPar to give the number of cycles requested. The function was able to be modelized into a linear function of the number of `n` repetition, as well as its error. We can consider the average error as constant for the macro `CLOCK()`. It is worth 20 cycles[1]. Of course, as soon as `n` grows, the error dicreases, and we can consider that for `n`>50 the absolute error is less than one cycle and is therefore negligible. Another macro gives more useful data from the same arguments:

[1]All you need to do to convert the number of cycles in times is to divide by the frequency, that is: 12.5MHz: 20 cycles=20/12.5 MHz=1.6 µs.

Syntax: `CYCLES(`*`instruction`*`,n,t)`

which makes two constants[2] deducted by the modelization of the `CLOCK():FORI` and `STARTUP` come into action. It gives the mean number used for executing `instruction` (in cycles). An average is done on two series of measures which each have `n`/2 repetitions (in this macro we need `n`>1). If the two measures are not close (enough limit is arbitrarily fixed to 5%) we chose the smallest one as the final measure of *`instruction`*. We use this strategy to greatly decrease the random modifications of the register permitting the measure. Indeed, if we repeat (independently from `n`) the macro `CLOCK()` many times, we notice that the most significant 16 bits, of the `t` variable which is 32 bits long, are random (often we measure small times inferior to 2^{16}/12.5MHz=0"052488, that is why we lean towards the smallest measure in case there is a problem). These problems occur with a frequency of about 1%.

3.3. The different variables of the simple type

All the simple variables can be preceded in their declaration by the modifiers of the `signed`, `unsigned` or `register`. The `int`, `long`, and `long int` types are identical, as well as the `float` and `long float` types. We present in the table below the sizes of the different types (in bytes), for the ACU as well as PE's. The two columns represent the different variables whose content is located on the ACU or the PE's (see figure 3.1 for more details on pointers).

[2]The various macros, variables and constants described above are all defined in a header file named TOOLS. H (see Appendix).

Variables on the ACU	(bytes)	Variables on the PE's	(bytes)
char	1	plural char	1
short	2	plural short	2
int	4	plural int	4
long long	8	plural long long	8
float	4	plural float	4
double	8	plural double	8
void*	4	void* plural	4
char*	4	char* plural	4
short*	4	short* plural	4
int*	4	int* plural	4
long long*	4	long long* plural	4
float*	4	float* plural	4
double*	4	double* plural	4
plural void*	4	plural void* plural	4
plural short*	4	plural char* plural	4
plural char*	4	plural short* plural	4
plural int*	4	plural int* plural	4
plural long long*	4	plural long long* plural	4
plural float*	4	plural float* plural	4
plural double*	4	plural double* plural	4

These variables are lined up in memory following their memory size. That means they are stocked on an address multiple of their size. In fact, except for the `char` types (alignment on a random address) and `short` types (alignment on an even address), all the other types have a memory alignment of 4 bytes. The structural and union elements are obviously aligned up following the same rules as their respective types. We must notice that with the `int* plural` pointer type, it is possible to have writing conflicts on the ACU. If we call `pp` such a pointer, then the result will be indefined for all the multiple writes in the ACU by the `*pp=value` indirection (where `value` is or is not a parallel expression), as soon as at least two `pp` addresses of two PE's are equal. On the other hand, if all the addresses contained in `pp` are different, the writing on the ACU is possible (the operation is sequentialized into 1024 assignments).

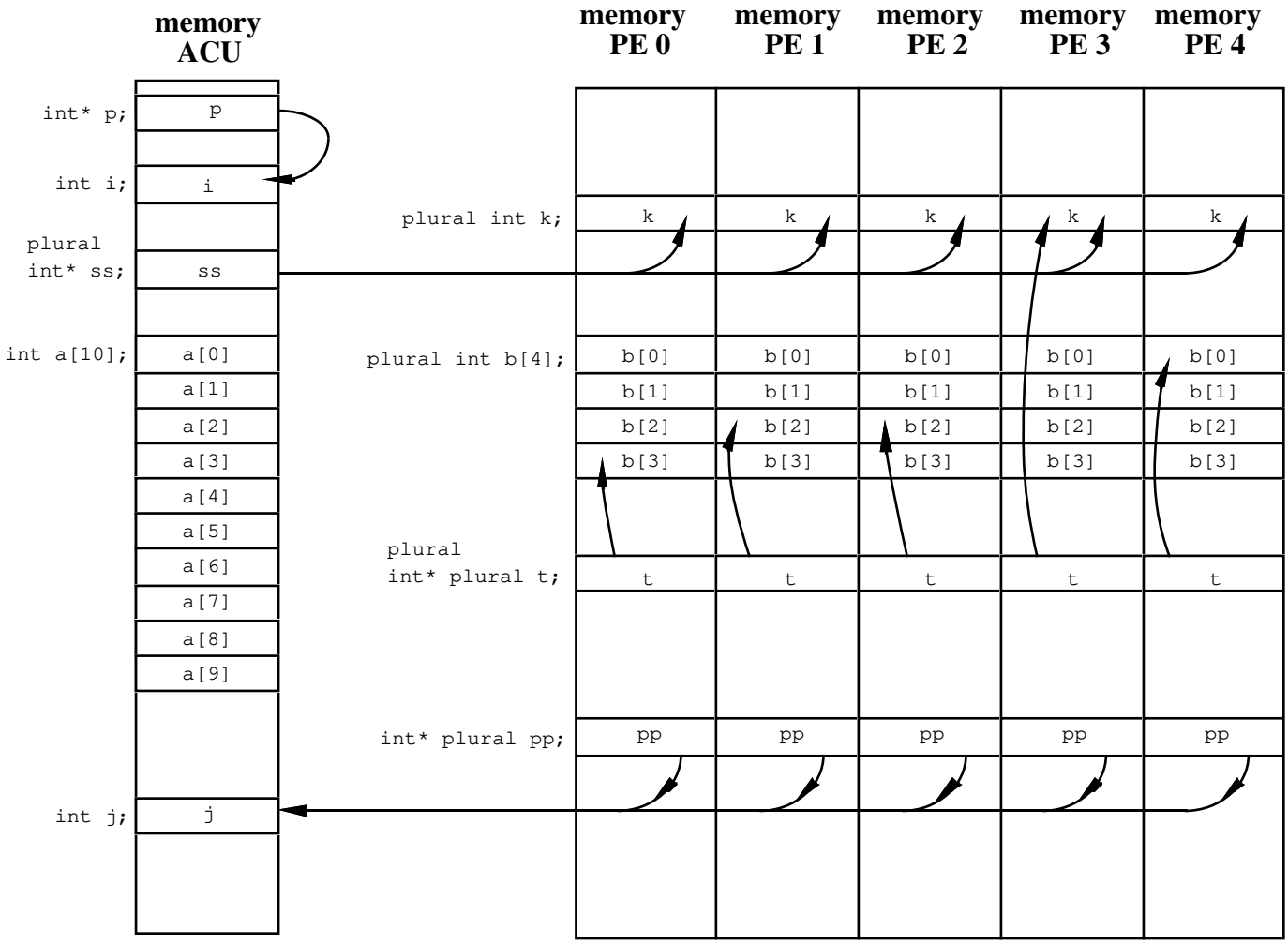


Figure 3.1: Representation of the variables in memory on the ACU and the PE's

3.4. Presentation of the experimental measures results

These results can be presented by a table or a graphic. In any case the measure reported is an execution time in clock cycles, so an integer number. What is measured is always an instruction in C language, thanks to the macro `CYCLES()`. Generally speaking the words ACU*i* (resp. PE*i*) represent the name of a variable stored in the ACU (resp. in the PE's) and of a variable type (`char`, `short`, ...). The *i* sign of these variables (1,2 or 3) helps us to distinguish several variables. Type 1 and 2 variables are randomly initialized uniformly on their permitted interval value, and the ACU2 variables (resp. PE2) will always be non zero to get exceptions, especially during arithmetic calculations. When these variables are floating, the `float` and `double` type, they are randomly initialized on the]-1,0[U]0,1[interval (uniformly), so as not to go past its capacities. Type 3 variables are used as left-value. We initialized each variable with the `uniform(a,b)` function (resp. `p_uniform(a,b)`) defined in the `TOOLS.H`. header file. Notice the given times are averages made on several iterations, often with a random content. However in the macro this random value remains the same for all

iterations. The variables which serve as a parameter (such as distance for example) are declared as global variables, and not `register`. The word ACUr (resp. PEr) is a variable of the ACU (resp. of the PE) declared as register. `CONST` represents an immediate value of a variable type (`char`, `short`, ...). It is not a variable but a macro fixed arbitrarily in the `TOOLS.H` header file. Most tested instructions are combined to assignments. In order the compiler to generate the desired statement code it must be a function call or an instruction (not an expression). For purely practical reasons the variables of the `long long` type are simply denoted `long`, which facilitates the reading of tables or graphs.

3.5. Elementary arithmetic operations and basic operations

It is necessary to remark that all variables used are global and usually non `register` (see paragraph above).

Addition	char	short	int	long[3]	float	double
`ACU3=ACU1+ACU2`	400	337	14	35	800	400
`ACU3=ACU1+CONST`	229	199	11	26	653	390
`PE3=PE1+PE2`	63	119	239	476	612	616
`PE3=PE1+ACU2`	125	122	170	343	549	465
`PE3=PE1+CONST`	39	80	163	327	487	487
Multiplication						
`ACU3=ACU3*ACU2`	438	376	51	955	1159	921
`ACU3=ACU1*CONST`	268	238	46	947	1021	751
`PE3=PE1*PE2`	88	195	454	1162	972	968
`PE3=PE1*ACU2`	161	203	384	1021	905	821
`PE3=PE1*CONST`	72	161	384	1020	842	821
Division						
`ACU3=ACU1/ACU2`	443	381	56	1536	1592	1193
`ACU3=CONST/ACU2`	274	244	54	1536	1454	1183
`PE3=PE1/PE2`	132	271	637	1757	1405	1406
`PE3=PE1/ACU2`	601	630	564	1604	1341	1254
`PE3=ACU1/PE2`	592	605	561	1610	1321	1263
`PE3=CONST/PE2`	468	509	561	1610	1263	1262
Comparison						
`if(ACU1<ACU2)`	339	279	12	211	542	271
`if(ACU1<CONST)`	170	137	8	204	395	266
`if(PE1<PE2)`	72	110	190	346	407	416
`if(PE1<ACU2)`	211	222	120	211	343	260
`if(PE1<CONST)`	71	93	112	195	280	284
Selection						
`if(PE1)`	48	69	110	192	-	-
`if(ACU1)` (=0)	109	93	6	247	-	-
`if(ACU1)` (=1)	107	90	4	247	-	-
`if(iproc==CONST)`	45	45	45	45	-	-
`if(iproc==ACUr)`	74	80	61	-[4]	-	-
`if(iproc==ACU1)`	186	176	68	137	-	-
`PE2=globalor(PE1)`	45	88	170	343	-	-
`ACU2=globalor(PE1)`	92	114	100	192	-	-
`if(globalor(PE1))`	31	52	93	431	-	-
Call for an empty function	171 (without parameter passage)					

3.6. Table analysis

[3] Note that "long" means `long long`.
[4] Not measured.

Transtyping

Generally speaking we notice that at the level of the ACU the operations on variables of the `char` and `short` type (with a size less than to 32 bits, see paragraph on the simple types sizes) take more time than operations on `int` type. This is due to the compiler (see the documentation of the MasPar concerning this subject, in [Ma]) that generates a code systematically transforming these variables in 32 bits (`int`). A transtyping from `char` into `int` is also longer than the transtyping from `short` to `int`. The number of transtyping is directly linked to the variable number which the instruction holds. So then, for the instruction `ACU3=ACU1+ACU2`, the program code makes 3 transtypings and an addition operation, which pretty clearly masks its own execution time. Consequently we can only get a fairly clear idea of the actual operation time on the `int` and `long` types for the integer arithmetic (we will also be able to keep in mind the assignment time of the operation thanks to the table in the paragraph on communication and assignment). In the case of instructions that give a constant (the word `CONST` is present), a transtyping is avoided (it is actually done during the compilation). There is no memory reading of this data since it is a macro whose value was known ever since the compilation, a value that we find directly in the executable machine code. For operations on the PE's, there is no transtyping as opposed to what is said in the MasPar documentation [Ma] (for example the addition time on the variables of the `short` type (16 bits) is two times faster than the same operation on `int`s (32 bits), see `PE3=PE1+PE2` in the instruction table). However, when a constant shows up in an instruction (word `CONST`) the time is generally inferior, because a pointer reading is avoided (one less address calculation) since it's an immediate value.

Floating arithmetic

We notice that there is also a transtyping problem for the `float` type. All the calculations on the ACU are indeed done in double precision thanks to the `double` type. For each `float` variable the machine makes a first transtyping for the calculation in double precision, then a second one to return to the `float` type, compatible with the initial variable. So there are two transtypings per `float` variable for the floating arithmetic operations. The float calculation times are greatly increased (the cost of these two transtyping is roughly of 400 cycles for an instruction with 3 `float` variables) and even be twice as the time for the same `double` operation (for example Addition). For the floating calculation on PE's the transtyping seems to be absent and the calculation times are more or less identical between `float` and `double`.

Comparison of calculation speeds between the ACU and PE's

We must point out that the calculation times of the ACU are superior to the PE's, especially for the `int` type. For the calculation times on 64 bits, ACU barely dominates (except for the addition, which is fast for integers). Of course, calculation on PE becomes advantageous because of the transtyping on the ACU.

PE's selection

We notice that the executing instruction times of the `if(condition_PE)` construction do not depend on the number of active PE's the time of evaluation[5].

Example:

```
plural int c=(iproc<512);
if(iproc%2)
{ if(c) { instructions } }
```

In the preceding code, the instruction `if(c)` is only done on odd numbered PE. The `{ instructions }` block is therefore only executed by the odd PE's, with a number < 512. We must point out that the compiler optimizes the code, when there is no active PE. In the example above, if we had initialized the condition with `c=(iproc<0)` the `{ instructions }` block wouldn't have been executed. The execution time of such a sequence would then be constant (see the table for basic operations). Also, in the example above we could have used a `short c`, rather than an `int` (the selection on the PE's thanks to `iproc` can be coded on 10 bits, for 2^{10}>`iproc` maximum; then a `char` is not enough). Following the calculation table, the `if(c)` is indeed quicker in `short` than in `int` (69 against 110 cycles). This can be major for performances of a program containing many repetitions. However, calculation time of that particular condition (c=...) is lengthened if we use variables on ACU in the calculation, which creates very restricting transtypings for the performances.

The `globalor` function does not depend on the number of active PE's, and very little on the number of PE's whose variable is a true value (we must add 2 cycles if all the variables are zero). This lack of optimization for this global *OR* is regrettable, but the instruction is done parallely and absolute execution time is relatively short.

3.7. Maximal performances obtained on the MasPar

In this paragraph we give time measures on arithmetic operations (floating and integer) done on the PE's grid. To obtain a peak speed we use the PE's registers. All the variables used are then declared as `register plural` local variables.

[5] We must remind you that the evaluation of the conditional instructions in parallel is ruled as a C [K&C], that is that a conditional expression will be considered as true if and only if its evaluation is non zero. During such instructions all PE that would give a zero evaluation at its condition is turned inactive.

Results table (in cycles)

Operation	char	short	int	long	float	double
`PEr1=PEr2`	2	6	14	31	14	31
`PEr3=PEr1+PEr2`	15	29	57	116	404	243
`PEr3=PEr1*PEr2`	50	110	277	800	763	601
`PEr3+=PEr1`	2	8	21	51	118	174
`PEr3*=PEr1`	38	89	241	728	194	531
`PEr3+=PEr1*PEr2`	50	112	284	812	873	743
`PEr3=PEr1/PEr2`	108	210	454	1383	1198	1039
`if(PEr1<PEr2)`	26	31	40	56	247	116

Floating operations on registers

Generally speaking, the results confirm theoretic data given in [Usr] pretty well, except for the types with the `float` data, which, in spite of the measure repetitions, goes into the margin for addition, multiplication and division. We must note that by using the `x`Δ`=y` syntax (in place of `x=x`Δ`y`,were Δ is an operator) we obtain better results, even with the `float` type. According to the table above, we obtain the number of million of operations per second. This number is calculated while considering 1024 PE's and supposing that the average time of an arithmetic operation is the average of the times for an addition and a multiplication.

Variables' sizes	char	short	int	long	float	double
Million of operations/second	640	264	98	33	82	36

These results are the figures rounded to the closest whole number.
To calculate them, we the average number of cycles for an addition and a multiplication: w=(x+y)/2 where x and y represent the times for lines 4 and 5 of the preceding table. Then z=w/12.5MHz, which gives us a time in µs. We convert this last results in instruction per second: 1024/z Mflops. This gives us, in a synthetic manner: z=25600/(x+y) Mflops.

4. Communication procedures

4.1. Generalities

We can classify the different communication types according to the physical elements of the MasPar that come into action.

The various levels of communication are:

a) FE ↔ DPU communication

b) ACU ↔ PE communication

c) PE ↔ PE communication

At first we will look towards the internal DPU communication, that is type b) and c) communication. The b) communications are resumed into the simple assignments, whereas the c) communications can be subdivided into three classes : `proc`, `xnet` and `router`.

4.2. Assignments

In this brief paragraph we include the PE←PE assignments, as well as the indirections. We must note that the assignments of the ACU←PE type are forbidden by syntax (it is impossible for the compiler to transtype a `plural` variable into a non parallel variable). However, communication from PE's to ACU is possible by indirection, they are assignments of the *PE←PE type, or else *PE←ACU, where *PE is a pointer situated on the PE's to an object situated on the ACU (see figure 3.1)

Assignment table

Direct assignment	char	short	int	long	float	double
`ACU1=ACU2`	166	152	7	19	7	19
`ACU1=CONST`	59	60	4	12	4	13
`PE1=ACU2`	113	100	64	143	64	145
`PE1=CONST`	11	27	64	142	64	144
`PE1=PE2`	29	66	139	287	139	287
Indirect assignment[6]						
`ACU2=&ACU1`	2	2	2	2	2	2
`PE2=&ACU1`	64	64	64	64	64	64
`PE2=&PE1`	2	2	2	2	2	2
`*ACU2=ACU1`	257	211	10	19	10	19
`*PE2=ACU1`	456	409	284	321	284	321
`*PE2=PE1`	142	207	340	605	340	605

Analysis of the ACU ↔ PE assignments

These assignments are done thanks to a special bus that has no link with the PE's connection grid. These communications are relatively fast. We will notice the abnormality for the assignments of variables with 8 and 16 bits (see paragraph with basic operations). For the address assignments (X=&Y) we notice that the times are constant no matter what the variable type is. We must then consider the &Y address instead of Y, for &Y has a constant size (see the table on the variable sizes of simple type), independent of the size of Y. The relatively short time of these last assignments is due to the fact that the address of each variable is known starting from the source program compilation. We can then bring these instructions of the direct assignments closer to the constants.

Analysis of the PE ← PE assignments

These assignments are done in parallel and don't need transtyping. We also notice that an immediate value does not need a reading nor an address calculation. There are as many addresses as variables in the instruction, which explains that the assignments with constants are almost twice as quick.

4.3. `proc` communication

This instruction permits us to transmit information with only one processor in a variable of the simple type. With the `proc` construction it is not possible to transmit more than 8 bytes at a time (64 bits maximum with `long long` or `double`). For greater information we will have to use `proc` in a loop or else

[6] For the indirect affectation tests, the variables were declared locally, or else the standard indirections of the C [K&C] (& and *) are forbidden.

another type of communication, global this time, while only selecting one PE. (for example while using `if(iproc==0) ...`).

Syntax: `proc[index].variable`
or else `proc[index_y][index_x].variable`

`index` must be a integer expression representing the number of the selected processor for communication (between 0 and 1023). `variable` is a variable name of a `plural` variable and of simple type (see paragraph on simple types). With the expressions `index_x` and `index_y` it is possible to select a PE by its coordinates with as origin PE #0. `index_y` represents the row number whereas `index_x` the column number. The `proc` construction can be used *left-value* or *right-value* (that is, to figure at the left or right of the equal sign of assignment).

`proc` communication's table

`proc` communication	char	short	int	long	float	double
`ACU1=proc[0].PE1`	130	152	137	232	137	233
`PE2=proc[0].PE1`	78	100	180	353	180	353
`proc[0].PE1=ACU1`	152	140	90	172	90	172
`proc[i].PE2=proc[0].PE1`	122	147	223	397	223	397
`ACU1=proc[0][0].PE1`	150	174	155	252	155	252
`PE2=proc[0][0].PE1`	97	122	180	353	180	353
`proc[0][0].PE1=ACU1`	174	158	90	172	90	172
`proc[i][0].PE2=proc[0].PE1`	139	164	222	397	222	397

Analysis of the `proc` communication's table

First of all, `proc` time is proportional to the variable size: the times on the `int` and `float` variables (resp. `long` and `double`) are identical. The `proc` construction never depends on its index. We varied the PE number in the `proc` construction from 0 to 1023 (in the 8 series of tests on the right and left) without noticing any variations in the results (the variable `i` is not `register`). When it comes to the communications between ACU and PE, we find the problem of the compiler for the transtypings of the 8 and 16 bit variables (see paragraph on basic operations). The `proc` instructions placed in *left-value* are quicker, thus, to write on the grid (send) is quicker than to read (get). Indeed, in order to read, starting from a PE, we must go look for the information on the target PE and bring it back, whereas in order to write, all we need to do is to send the message on the destination PE. When it comes to the `proc[i][j]` syntax, it is slower than the `proc[i]` syntax, only for the `char` and `short` types. When a 32 bits or 64 bits type is used, times are identical for both syntaxes.

Transfering 1024 data from the PE to the ACU can be done either by including in a `i` loop the instruction: `ACU=proc[i].value`, either by making an indirect

assignment of the `*PE=value` type (where PE is a `plural` variable pointing on an ACU variable). The second way of proceeding is quicker. For example, it is done in 132168 cycles (for 1024 `int`) which corresponds to an elementary transfer time of about 132 cycles (to be compare with the first line of the `proc` communication table).

4.4. X-Net communication

It is actually a family of functions allowing all the active PE's to each gain access to a PE found at a uniform distance and in one of the 8 directions, as shown in figure 2.2.

Syntax: `xnet[distance].variable`

`distance` must be a natural (if it is worth 0 there is an internal communication of each active PE with itself). `variable` fulfills the same condition as the `proc` syntax. The direction is given by the the instruction name itself: `xnetN`, `xnetNE`, `xnetE`, `xnetSE`, `xnetS`, `xnetSW`, `xnetW`, `xnetNW` (see figure 2.2).

As for `proc`, `xnet` can be used in *left-value* or *right-value*, and the size of the information is limited to 8 bytes. We will not distinguish more than 4 different sizes in bits of the variable. We need to have knowledge of: 8 bits, 16 bits, 32 bits, 64 bits (in the programs giving the communication tests the floating types are not shown anymore). In the tests the chosen direction does not come into action (this has been verified).

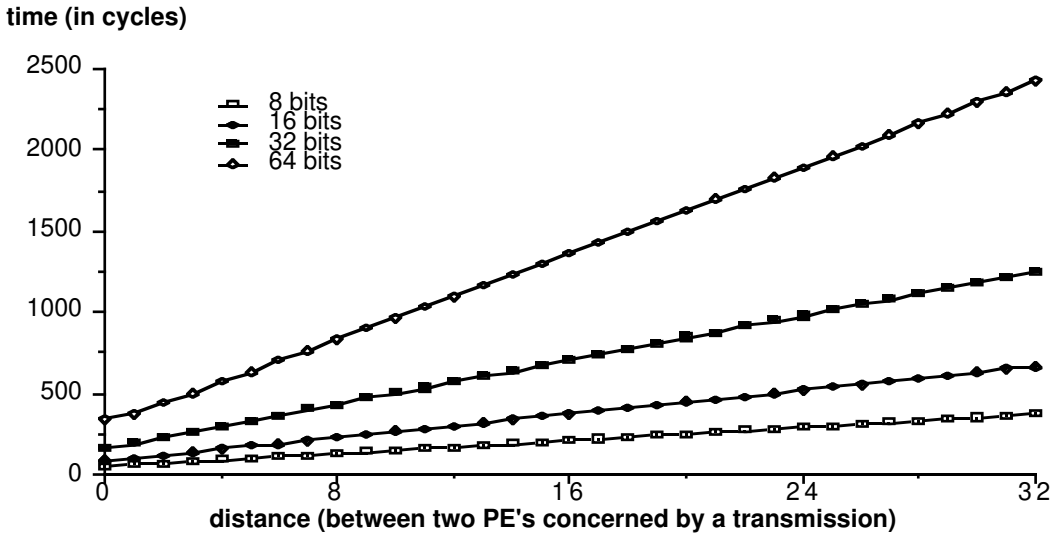


Figure 4.1a: Behavior study of `xnet` as a function of distance

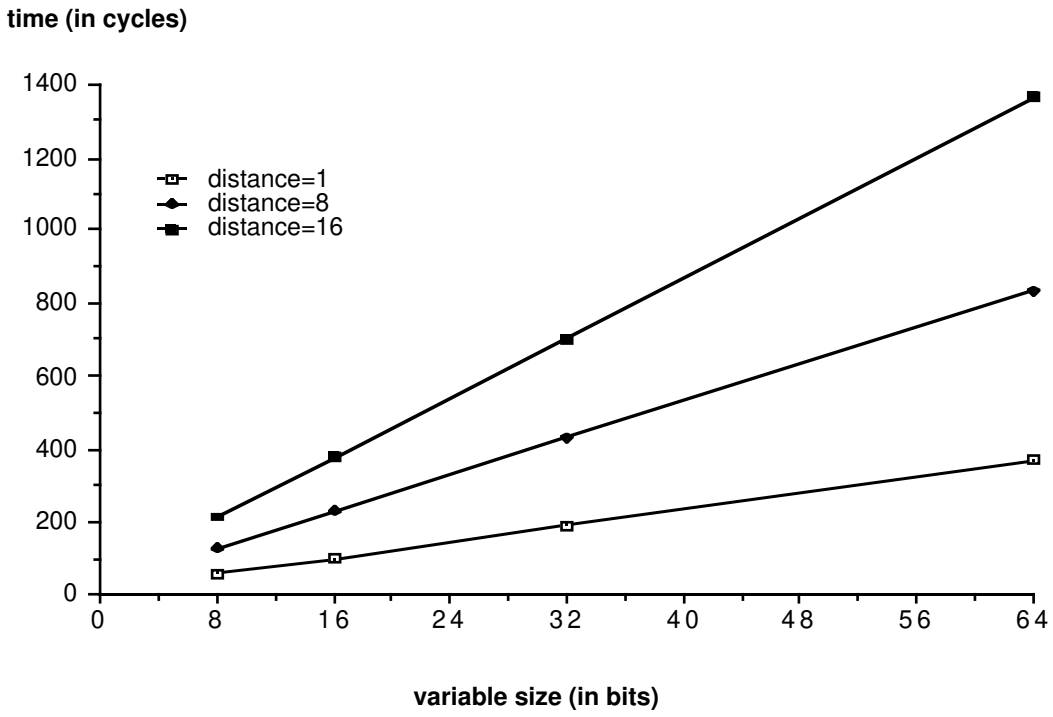


Figure 4.1b: Behavior study of `xnet` as a function of message size

Figures 4.1a and 4.1b analysis

First of all the curves show `xnet` used in *left-value*. They show the dependency of `xnet` on the variable distance and size. Figure 4.1b also shows that `xnet` is linearly proportional to the number of information bits, but that the slope coefficient grows with distance. If we call $\theta(\delta,\tau)$ the time taken by `xnet` for a variable of τ bits transmitted between PE's, at a distance called δ, then:

$$\text{figure 4.1a} \Rightarrow \frac{\partial\theta(\delta,\tau)}{\partial\tau} = a\delta+b$$

$$\text{figure 4.1b} \Rightarrow \frac{\partial\theta(\delta,\tau)}{\partial\delta} = c\tau+d$$

where $\theta(\delta,\tau)=(a\delta+b)\tau+e=(c\tau+d)\delta+e \Rightarrow \theta(\delta,\tau)=a\delta\tau+e$

`xnet` is therefore proportional to the distance*size product.

`xnetp` communication:

This instruction gives us rapid communication between two or several PE's of an information with a maximal size of 8 bytes. The path between the transmitting PE's is pipelined and the intermediary PE located on the connection line (see figure 2.2)

must be inactive. This instruction can be used in *right-value* or *left-value.* As well as for `xnet,` we specify the direction by the corresponding function by adjoining a letter combination taken from {N,E,W,S} in the name of the function.

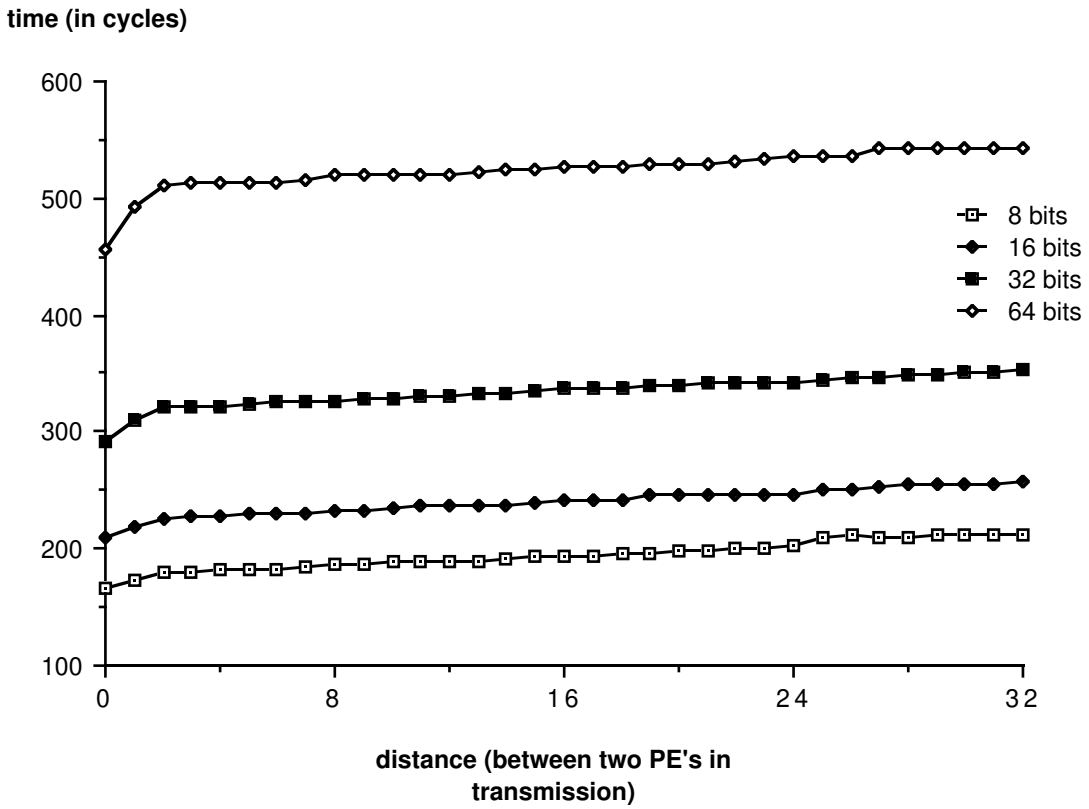


Figure 4.2a: Behavior study of `xnetp` as a function of distance

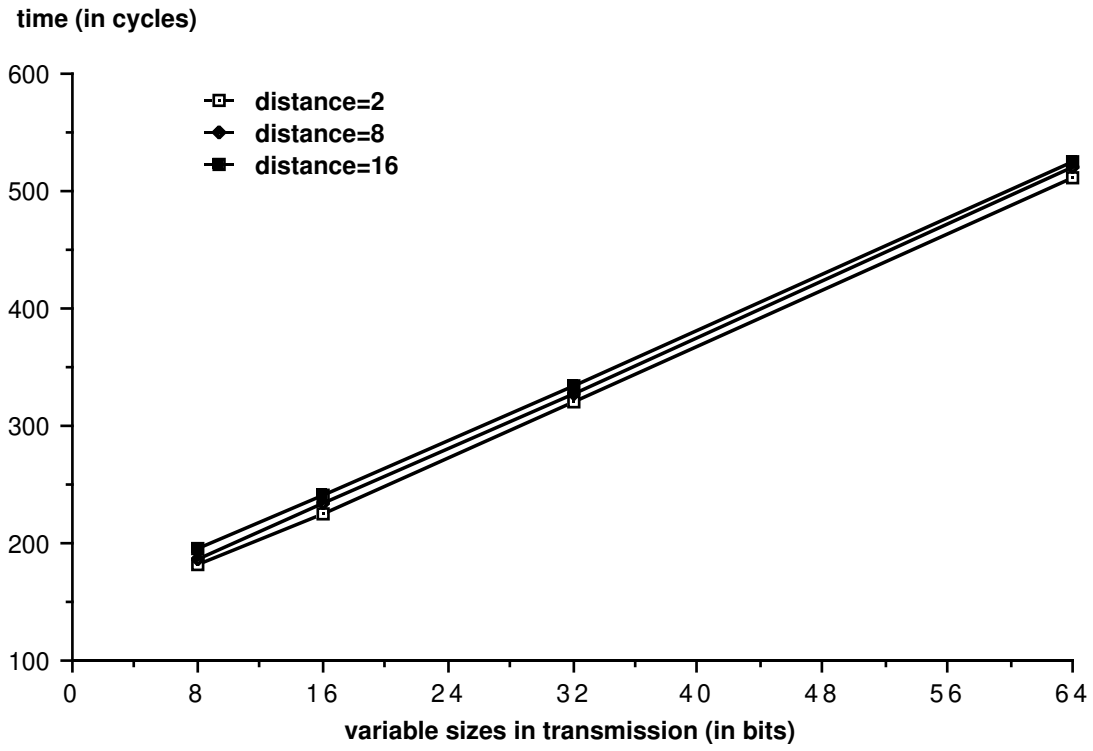


Figure 4.2b: Behavior study of `xnetp` as a function of the size of the message

Figures 4.2a et 4.2b analysis

For of all we must point out that during the tests, the `xnetp` instructions are used in *left-value*. On the other hand, because of the blockage of intermediary PE's, only one PE is selected: It is good to keep into account the selection done in the measure (here the selection is done using an instruction of the `if(c)` type, see the basic operations table). The very light slope of the curve in figure 4.2a indicates the very low dependency of `xnetp` with the distance, however we do have a linear model in ß+Lτ with τ=1.11 cycle/distance (ß being the function of the size of the variables). So the time is especially function of the message size (figure 4.2b), which also gives a linear model in ß+Lτ with τ=5.93 cycle/bit (ß being the distance function between the transmitting distances).

We can explain the curve break for 4.2a. When the distance is zero, the `xnetp` is executed, but there is a simplification of the X-Net communication operating principle (see figure 4.6, page 30) for there is no inter-PE's exchange. Then the 2^d transfer of figure 4.6 does not take place, but the duplication (first transfer) is indeed done, or else the 4 curves would all start from the same point. If the distance is 1, `xnetp` is optimized, for there is no need to block the non existing intermediary PE. This explains that for figure 4.2b we did not represent the curve where **distance=0** and **distance=1**.

`xnetc` communication

It allows us to transmit in a pipelined fashion from one PE towards another by recopying the intermediary PE's. These are inactive. We must point out that `xnetc` does not work correctly if it is a *right-value*, as it is indicated in the [Ma] documentation (without causing any error, the PE's concerned by the communication do not receive the message at all). As well as for `xnet` we specify the direction by the corresponding function, adjoining a combination of letters amongst {N,E,W,S} in the name of the function. For tests, only one PE is selected, the one that sends or receives the message, by a conditional instruction. We must then subtract from the observed times the time of such a condition (see the basic instructions table).

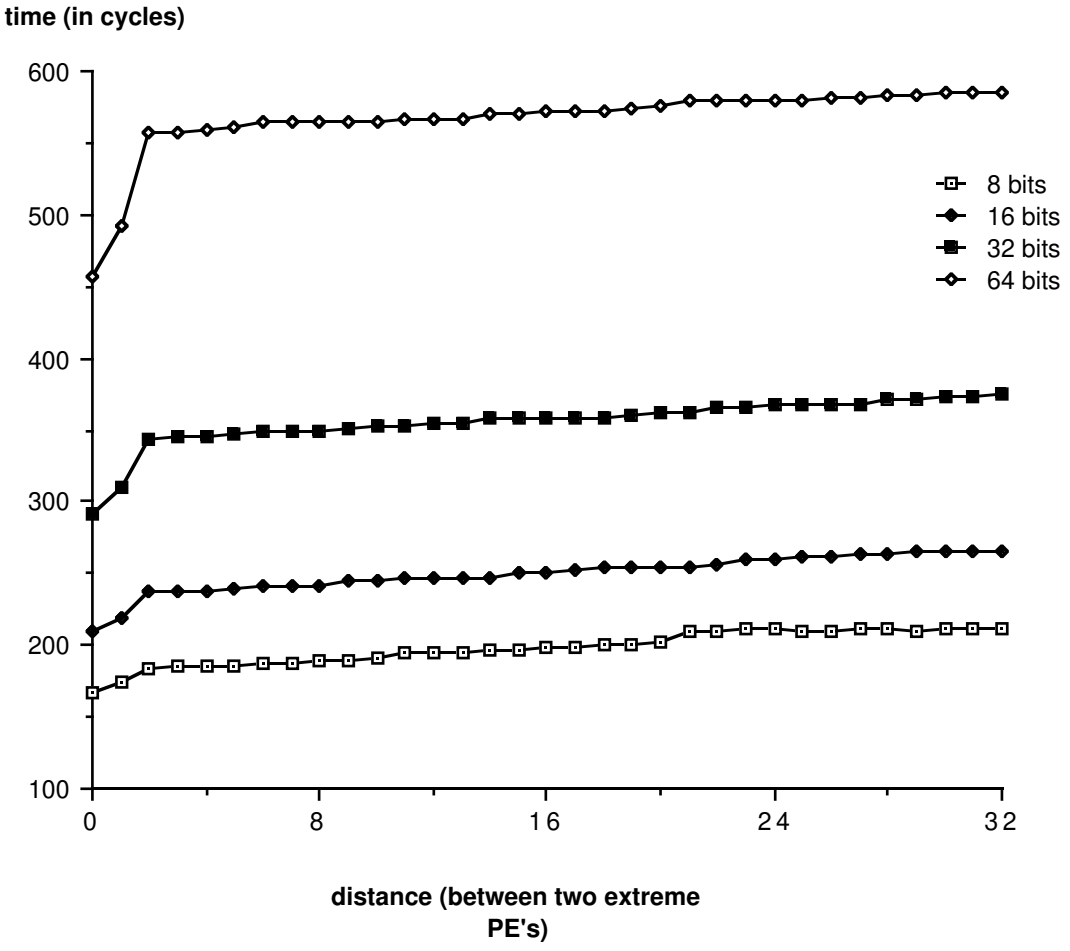


Figure 4.3: Behavior study of `xnetc` as a function of distance.

Figure 4.3 analysis

We notice an appearance globally similar to the 4.2a curve. We have a linear model in ß+Lτ with the same τ as for `xnetp`, that is τ=1.11 cycle/distance. As for

xnetp, in the test, we must of course take into account the fact that xnetc is used in *left-value* and also consider the selection time. For distances inferior or equal to 1, we can also make the same remarks as for xnetp and we notice that in this case the functions are optimized (because for xnetc and xnetp it is the same times for such distances). However, xnetp is quicker, this being due to a lower initialization time than for superior distances.

xsend and xfetch communications

This type of communication allows us to move an information message of variable size (and bigger than 8 bytes). xsend and xfetch are close to the xnet construction (non pipelined communication), where xsend corresponds to xnet used in *right-value*, and where xfetch corresponds to a *left-value*.

Syntax: *type*_x*function*(dy,dx,scr,dest,n)

dy and dx are relative displacements (signed integers) of a destination compared to the source. Beware, for we must note that dy is oriented from bottom to top whereas dx goes from left to right. scr is a pointer on the data zone that we must move, and dest points on the target memory zone. n is the number of bytes of scr that must be transferred. *function* is either send or fetch. *type* can be ss, ps, sp or pp, depending on whether the variables scr and dest are singular or plural pointers (we will call p_scr and p_dest pointers stored on PE's, whereas s_scr and s_dest are pointers located on the ACU[7]). So we have 4 types of the following instructions:

```
ss_xfunction(dy,dx,s_scr,s_dest,n)
sp_xfunction(dy,dx,s_scr,p_dest,n)
ps_xfunction(dy,dx,p_scr,s_dest,n)
pp_xfunction(dy,dx,p_scr,p_dest,n)
```

[7] There exists two kinds of pointers on the elements situated in the PE memory (plural void*, see figure 3.1). On one hand there are pointers situated in the ACU memory (for example plural void* s_scr), they are used when the address of the pointed element is the same on all the PE's.

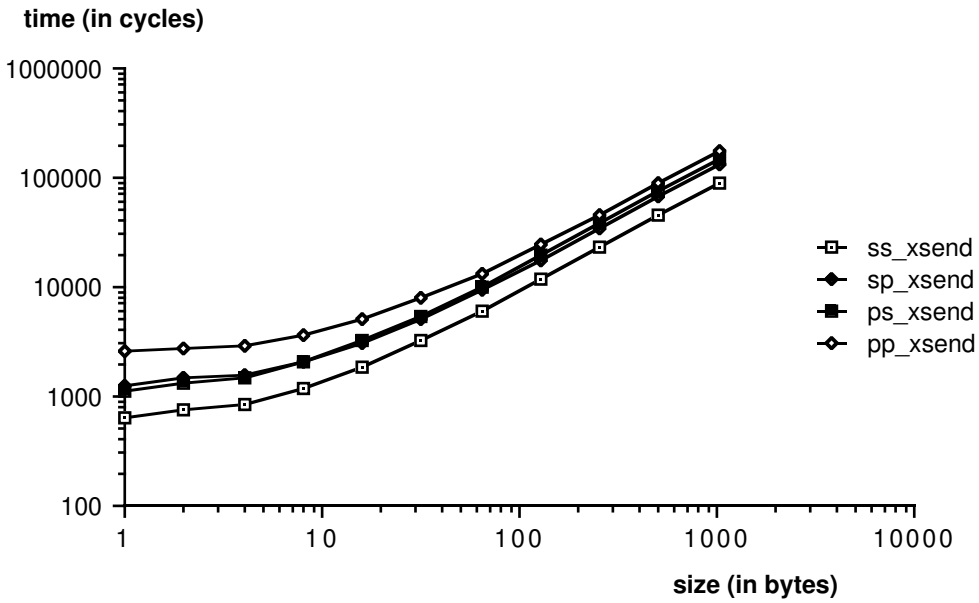


Figure 4.4a: Study of `xsend` communication, as a function of the message size transferred (from 1 byte to 1KBytes).

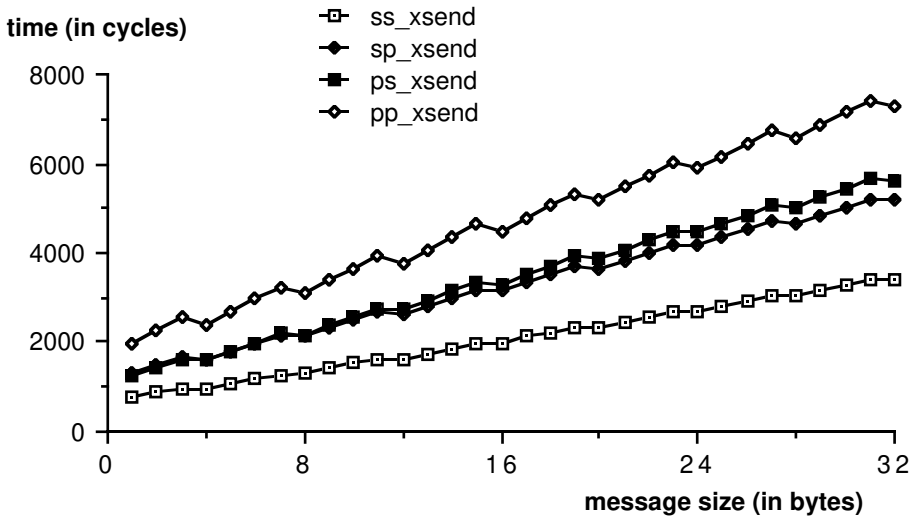


Figure 4.4b: Study of `xsend` communication, as a function of the message size transferred (from 1 to 32 bytes).

Figures 4.4a and 4.4b analysis

The curves were obtained using distance parameters `dx=0` and `dy=1`. Note that in Figure 4.4a, from a certain size onward, `xsend` time is linear, as a function of size. This curve presents axes on a logarithmic scale to display variable sizes from 1 byte to 1K byte. If we look at the small values of the message size, we notice several breaks in the curve lines, and that is for the multiple values of 4 of the

abscissa. We can then think that the message is transferred by groups of 32 bits (4 bytes) during the `xsend` communications. The same characteristics were obtained with `xfetch`.

We are going to study the `xsend` instruction time as a function of the distance. The problem is to determine which distance is to be considered (it has to correspond to the length of the path truly covered by data). Yet the `xsend` function contains two parameters, `dx` and `dy`. Two paths seem possible (see figure 4.5c). The associated distances are `d1=|dx|+|dy|` and `d2=max(|dx|,|dy|)`. We will study the time for `xsend` as a function of distance, while considering both paths.

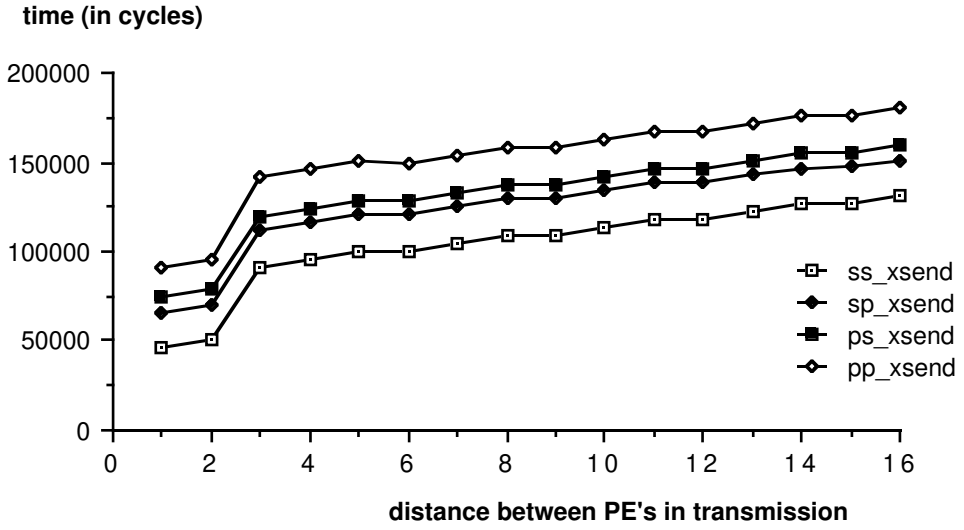


Figure 4.5a: Study of `xsend` communication as a function of distance `d1=|dx|+|dy|`, where `dx=d1/3` and `dy=d1-dx`.

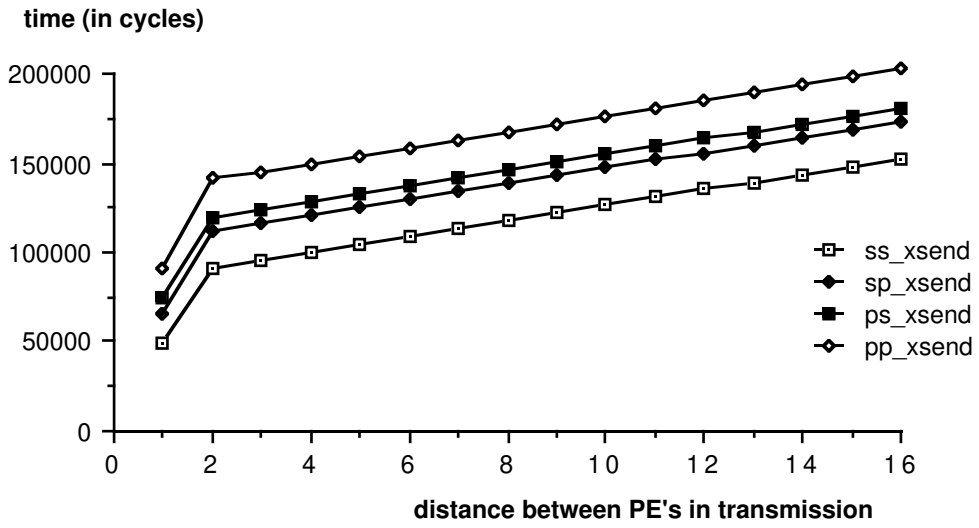


Figure 4.5b: Study of `xsend` communication as a function of distance `d2=max(|dx|,|dy|)`, where `dx=d2` and `dy=d2-1`.

Figure 4.5a and 4.5b analysis

The figures have been obtained from a communication of 512 bytes. We must notice that the distance parameter in the curves depend, in various ways, on `dx` and `dy` following the model chosen for the message course on the grid (see figure 4.5c). In both tests we have chosen `dx` and `dy` in such a way that the transmitted messages cannot be transferred without a direction change, meaning `dx` or `dy` non zero and $|dx| \neq |dy|$. For curve 4.5b, we could not do this condition for `d2=1`, which explains the quicker transfers for this distance. We will notice that for the first model (figure 4.5a) distances `d1=1` and `d1=2` correspond to a message circulation without a direction change (so the first two measures are lower). All that is left to do is to define which is the curve that corresponds to the communication mode. It appears on figure 4.5a that the `xsend` time for certain distances is the same. For example, `d1=5` and `d1=6`, we measure roughly 100000 cycles for `ss_xsend`. For `d1=5`, we have `dx=1` and `dy=4`, and for `d1=6`, we have dx=2 and `dy=4`. Let's now consider `d2` with the same values of `dx` and `dy`. We then obtain `d2=4` in both cases. That means that time is function to distance `d2` and not `d1`. This hypothesis is confirmed by the appearance of the curve 4.5b which is linear and very regular (for `d2>1`). We will notice that the path defining `d2` presents an optimization compared to `d1`. Indeed we still have $d2 \leq d1$ no matter what the transmitting PE's are. So we have for xsend a linear model for d2>1 in function of distance L between the PE's involved the communication: $ß+L\tau$ with τ=5753.07 cycles/distance for a transfer of 512 bytes (ß depending on the type of function, `ss`, `sp`, `ps` or `pp_xsend`).

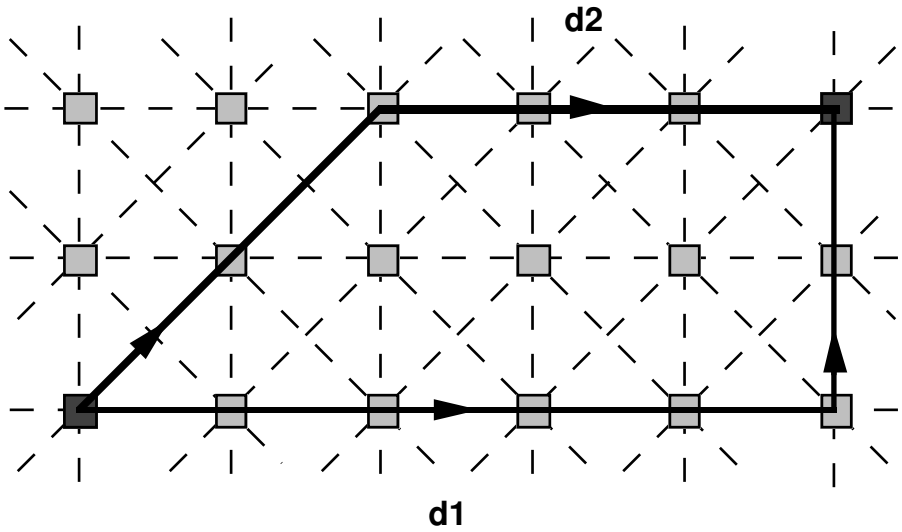


Figure 4.5c: Two distance models (`d1` and `d2`) for `xsend` and `xfetch` communications.

`xsend` and `xfetch` communication Results

Just as the `xnet` communications, the `xsend` and `xfetch` instructions depend on the distances, which are `dy` and `dx`. They also depend on the number of transferred bits (see the size of the simple types of the variables table). We will also notice the importance of left-values (that is `xsend`). We can also say that the times depend on the parameter *type* (`ss`, `ps`, `sp`, or `pp`; see syntax of these communications). These times decrease when the pointers (`scr` and `dest`) are in the ACU memory. When all the variable addresses are the same, the pointer is then situated in the ACU and the sequencer can then order a transfer of bytes directly to all the PE's. But if the variable addresses are not the same for all PE's, the sequencer can only send, to the PE's, instructions for adresses decoding and then send transfer instructions. So then the PE's have to perform more instructions.

Results on X-Net type communications

An X-Net communication used in *left-value* is always quicker than the same instruction *right-value*. Performances depend neither on direction nor on the number of active PE's (this is because of the uniform character of the communication), but linearly depend on the size of the distributed information and the distance. The sensitivity of the two parameters is more or less identical in the non pipelined case (proportional to distance*size), whereas in the pipelined case (`xnetp` and `xnetc`) time is especially sensitive to the size. Conflicts may occur when there are simultaneous writings. That happens when for example all the PE's send data to their right neighbor, whereas each predecessor on the left simultaneously crunches the data. The problem is resolved (see figure 4.6) by a duplication of data on each PE before transmitting them through the grid to the destination PE. This is not serious for the `xnet` but it becomes so with `xsend` and `xfetch` where the message length can be critical for the PE source's memory

zone. In any case we need a memory space on the PE source, at least twice as big as the size of the message transmitted.

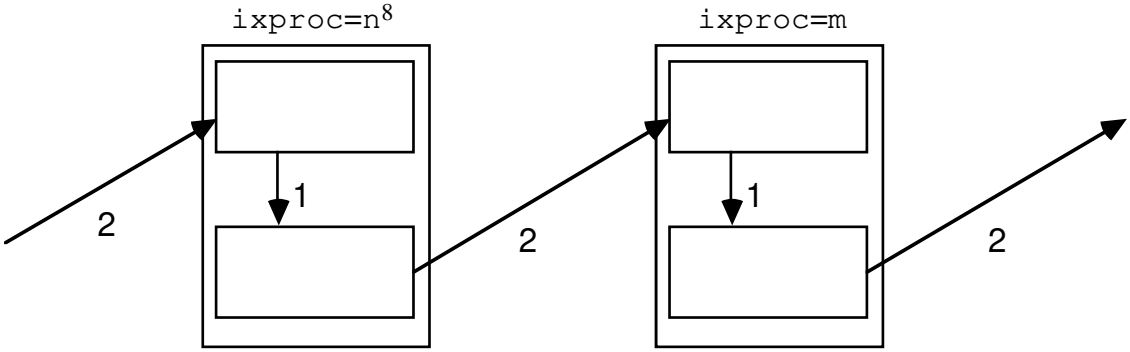


Figure 4.6: Operating principle of X-Net communication between two PE's with a distance of |m-n|. 1- Message duplication; 2- Inter-PE transfer

In the principle given above there cannot only have two steps because for the second step the memory of a particular PE is actually reading (sending of the message duplicated to its right neighbor) as well as writing (reception of the message that came from the left). Such a memory with accesses to simultaneous reading/writing is actually difficult to do. The principle is without a doubt more complex, with for example a PE out of two which is reading, then, in another time, it goes to the writing mode whereas the other half did the opposite exchange. Only the whole principle is correct.

X-Net communication optimization

There are two main types of simplification of the source code for the optimization in times of the X-Net communications: the distance reduction (critical element for that type of communication) and the use of *left-values*. Also in the case of a communication with the PE's neighboring by a distance of 1, keeping the results in mind, it is better to do: `xnet` rather than `xnetp` or worse yet `xnetc`. However the message sizes cannot go over 64 bits.

a) Distance optimization

In all `xnet` statements, as soon as distance `d`>16 we can use the same instruction with an identical direction but going the opposite way, and a new distance parameter worth (32−d)[9].

Example a_1: `{xnetN[20].x}` ⇔ `{xnetS[12].x}`
Example a_2: `{ss_xsend(20,dx,s,p,n)}` ⇔ `{ss_xsend(-12,dx,s,p,n)}`

[8] `ixproc` is the PE position following the horizontal direction (line).

[9] The ϕ diameter of a k-tore to p PE being $\phi=\sqrt[k]{p}$, that is here $\sqrt[2]{1024}=32$.

Of course, all other variants of `xnet` statements are allowed, such as `xnetpNW`, `sp_xfetch`,..., except for `xnetc`, whose result clearly depends on the path followed by the message.

b) Optimization of *right-values* in *left-values*

This simplification can be carried out only when an X-Net type function appears to the right of an assignment symbol. It involves exchanging roles of the receiving and sending PE's in a given message. Some problems arise from PE selection. Let us first suppose that all PE's are active, due to the prefix `all` added before a sequence of instructions:

Example b1: `{y=xnetN[d].x}` ⇔ `{xnetS[d].y=x}`
Example b_2: `{sp_xfetch(dy,dx,scr,dest,n)}`
⇔ `{ps_xsend(dy,dx,dest,scr,n)}`

Of course, in example b_1, `y` can be substitute by `x`, and the construction `xnetp` is allowed (the problem does not arise for `xnetc`, which it cannot be a *right-value*), regardless of direction. In example b_2 the pointers are exchanged (along with the prefix of the function). This condition also remains valid when the pointers `scr` and `dest` are of the same type (`ss` or `pp`).

Now let us suppose that only some of the PE's are active at the time of communication. Let R represent the total of active PE's at the time of an X-Net communication, executed in *right-value*: $R=\{r_1,...,r_n\}$ with $n \in [0,$`nproc`$[$ (where `nproc` is the total number of PE's, and is also a global C variable containing total number of processors). And let L represent all of the PE numbers receiving information from the PE's of R: $L=\{l_1,...,l_m\}$ with $m \in [0,$`nproc`$[$. This therefore gives us[10]:

Example b1': `{if(R) y=xnetN [d].x}` ⇔ `{if(L) xnetS[d].y=x}`

The transposition of example b_2 is similarly determined by the same rule of conditional permutation. We will note that through this procedure it is possible to write on a non-active PE, which can be problematic, for example, in any conditional instruction such as `if`, `for`, `while`, `switch`....

In the tables of appendices 8.1 and 8.2, it is possible to estimate the time saved by using X-Net instructions, as a function of distance and of syntax used, in *right-value* or in *left-value*.

[10] One can easily describe by C the entire sets of R and L by:

```
 i) plural int R,L;
ii) R= ((iproc==r1) || (...) || (iproc==rn));
iii) L= ((iproc==l1) || (...) || (iproc==lm));
```

4.5. `router` communication

This instruction allows a global communication in parallel with several processors. The parallelization of this instruction is only partial; there is a sequential part, and is very dependent on the notion of clustering (see Chapter 2). With the number of *conflicts*, we will represent the number of active PE's in a single cluster, wishing to transmit either with the PE's of the same cluster or with those of a different cluster. We immediately note that multiple reading or writing on a same PE by *n* different PE's will produce *n* conflicts (see operating principle, figures 4.9 and 4.10). There are three possible routing functions: `router`, `rsend` and `rfetch`.

`router` communication

This instruction allows a generalized communication, with message contained in a simple variable, the size thus limited to 64 bits.

Syntax: `router[index_PE].variable`

where `index_PE` is a `plural int` and the value of the instruction `proc[i].index_PE` indicates the number of PE's sought by PE `#i` for communication. `variable` satisfies the same conditions than function `proc` (see paragraph on `proc` communications). This construction can be used in *left-value* or in *right-value*.

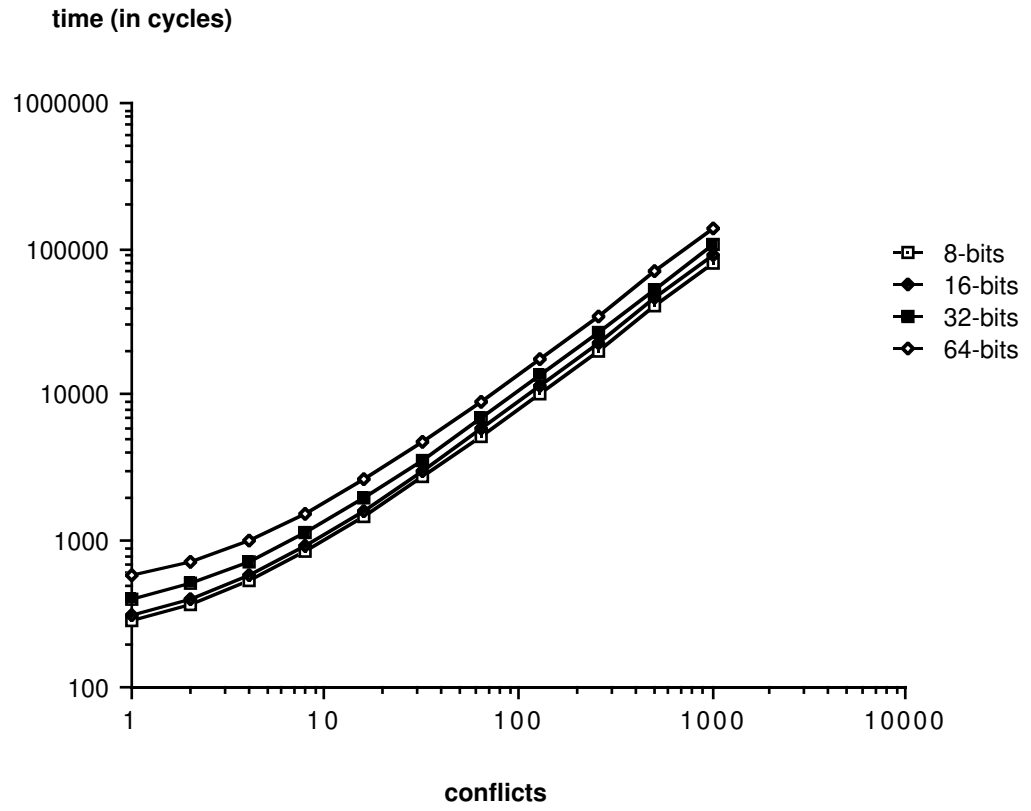


Figure 4.7a: `Router` communication as a function of the number of conflicts per cluster

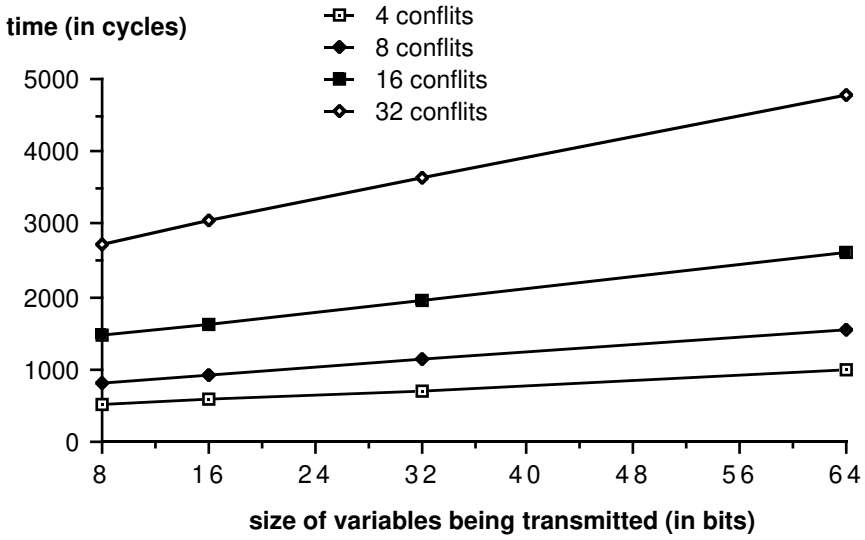


Figure 4.7b: `Router` communications as a function of message size.

Analysis of figures 4.7a and 4.7b

In the tests we varied the number of conflicts yielded by the function `conflit()` (see `TOOLS.H`, appendix 8.5). This was done by way of the instruction `if(c)`

`PE2=router[f].PE1` where `c=(iproc<i)` and `f=(plural int)0`. At each iteration, the number `i` doubles (i.e. the number of conflicts doubles, since with `f=0` all PE's tend to transmit with the same PE, each time producing the maximum number of conflicts per cluster and per active PE[11]) which explains why the scale of Figure 4.7 is logarithmic in both abscissa and ordinate (the number of conflicts is, successively, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024). We can assume that from 16 conflicts onward, `router`'s executing time is proportional to the number of conflicts. Figure 4.7b presents the same characteristics as Figure 4.1b for `xnet`.

We note that `router` is faster in *right-value,* whereas X-Net communication is faster in a *left-value*.

`rsend` and `rfetch` communications

Following the example of `xsend` and `xfetch` functions, there are two `router` type functions which allow us to send or to receive a message of variable size.

Syntax: *`type_xfunction`*`(index_PE,scr,dest,n)`

where `index_PE` has the same meaning as in `router`, and where `type`, `function`, `scr`, `dest`, and `n` have the same meaning as in `xsend` and `xfetch` (see paragraph on X-Net communications).

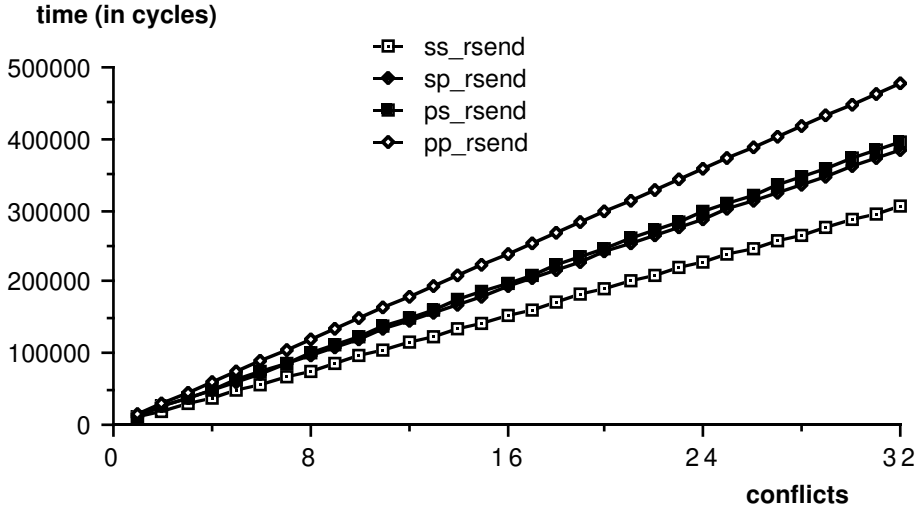


Figure 4.8a: Study of `rsend` as a function of the number of conflicts

[11] i.e. the number i.

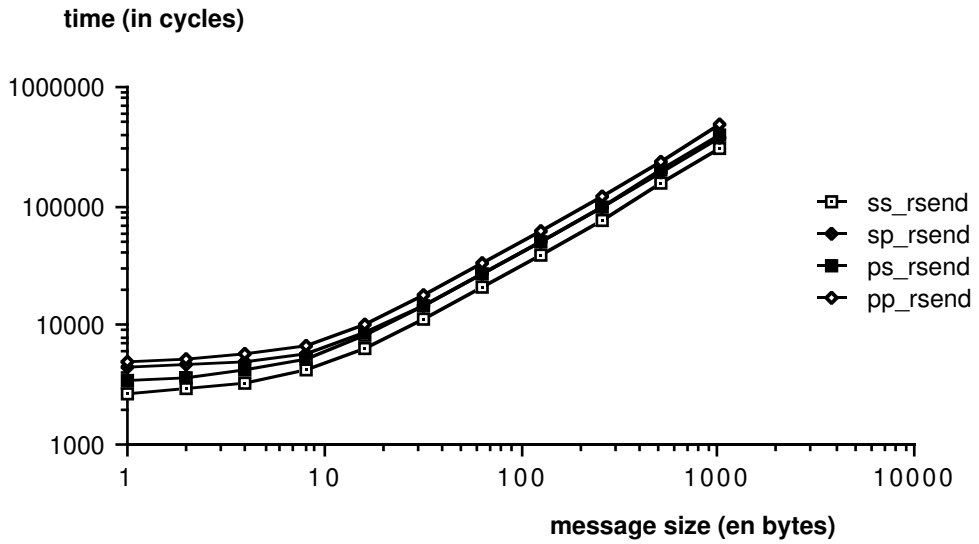


Figure 4.8b: Study of `rsend` as a function of the size of the message transferred

Analysis of figures 4.8a and 4.8b:

It appears that the executing time of `rsend` statement is proportional to the number of conflicts, just like `router` (Figure 4.8a, corresponding to the transfer of 512 bytes). Regarding the size of the message transmitted, `rsend` increases linearly from 16 bytes onward (Figure 4.8b, corresponding to 16 conflicts). For smaller sizes, `rsend` seems to be optimized (the logarithmic curve has a less steep slope for smaller size values).

Results on `router` type communications

Communication is independent of the distance between PE's. Unlike in X-Net group communications, here, there is no duplication (on PE's) of the messages sent (see operating principle, figure 4.9), which can prove useful for `rsend` and `rfetch`. This type of communication depends (linearly) on the number of conflicts and on the size of the message transferred.

Unlike X-Net communications, `router` communications are faster in *right-value* than in *left-value*, owing to a much different operating principle. Precise times are shown in appendix 8.3.

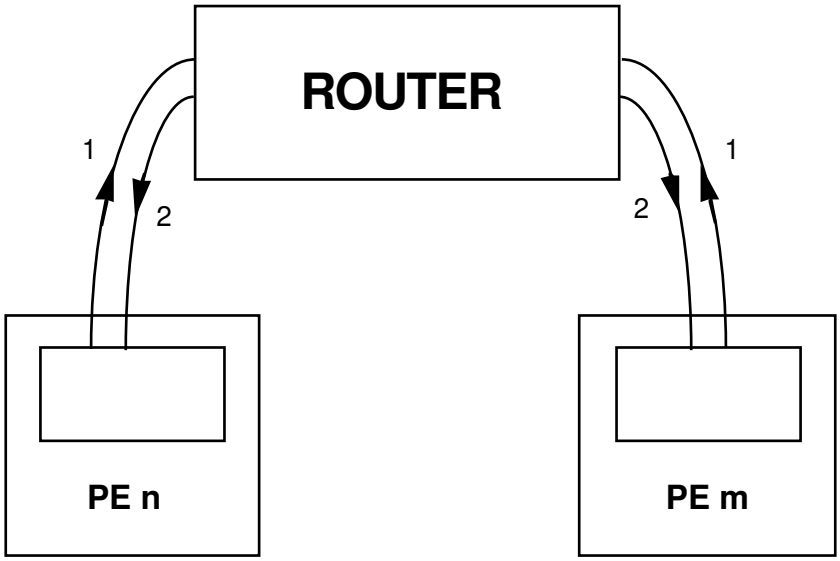


Figure 4.9: Operating principle of `router` communications between PE #`n` and PE #`m`

This principle is also quite simplistic. There are in fact three hierarchical levels of router connections. The first router level enters with 1024 connections and exits with 64 ones (64=1024/16): the copy of the message of PE's seeking communication is carried out, but only one PE per cluster is retained. The second router, linked to the first, enters and exits with 64 connections. In addition, this router has an internal *crossbar* connection between entries and exits (one and only one entry is connected to exactly one exit. A PE cannot, for example, read or write over two other PE's, located on different clusters at the same time). The third router, linked to the second, thus has 64 entries versus 1024 exits connected directly to the PE's.The third router level performs writes in PE memory, whereas the first performs reads.

Through this process it is thus possible in a single operation to exchange (*swapping*) two memory zones situated on two PE's of different clusters.

Example of *swapping* program:

```
plural int f=0,x=iproc;
if((iproc==0)||(iproc==4)) /* 2 PE on 2 clusters */
{ proc[0].f=4;
  proc[4].f=0;
  p_printf("x=%d ",x);
  x=router[f].x;
  p_printf("x=%d ",x);
}
```

This program displays `x=0 x=4`, then `x=4 x=0`.

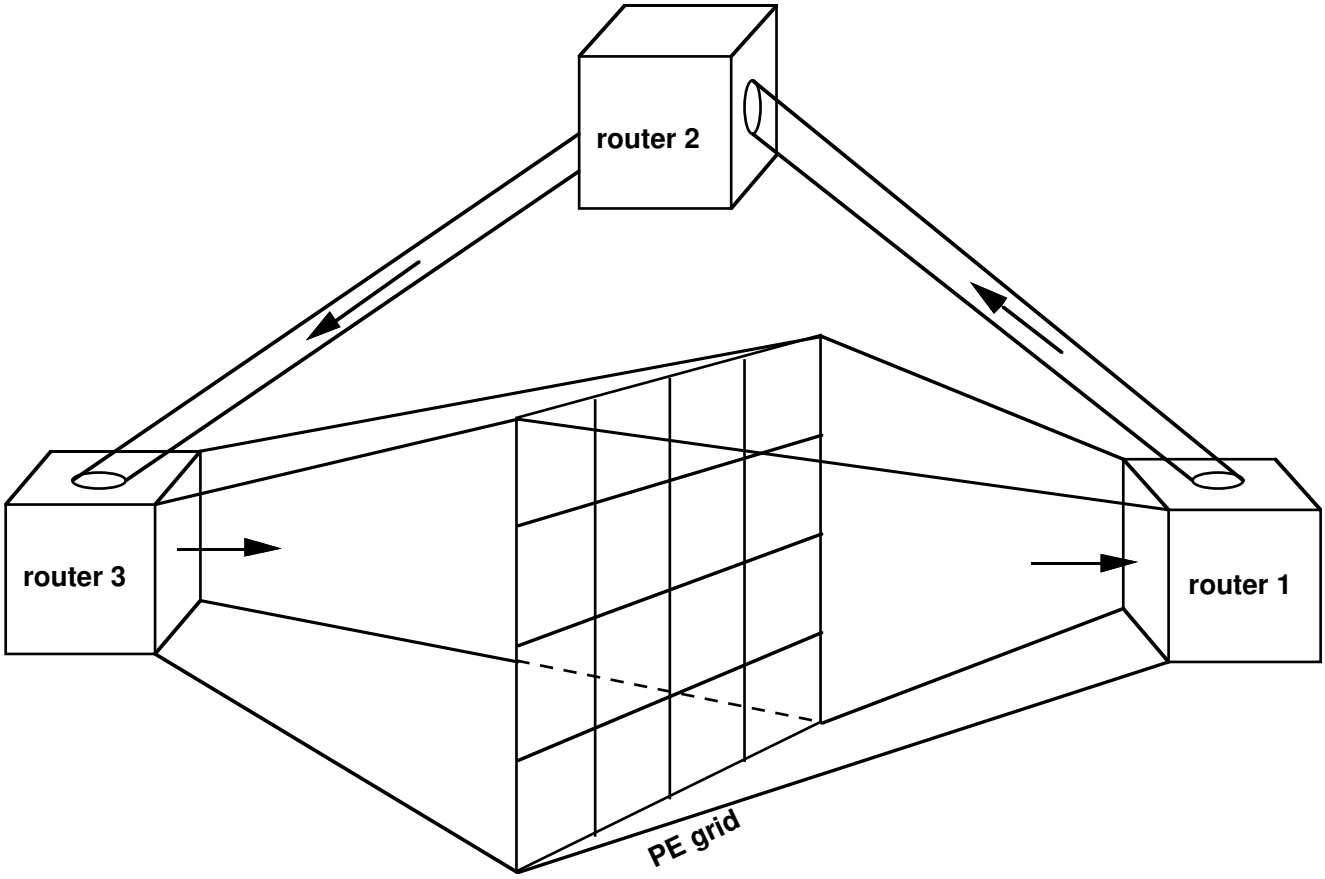


Figure 4.10: Routing principle on three levels

4.6. Summary of communication procedures

In this summary, all assignments have been removed (see paragraph 4.2 on assignments). We have used the same notation as in the paragraphs previous.

```
int i,j,dx,dy;
unsigned int d,n;
plural int f;
plural type_simple[12] variable;
plural char * (plural)[13] scr;
plural char * (plural) dest;
```

`proc[i].variable`
`proc[i][j].variable`
Transmits to the ACU the value of `variable` of PE number `i`, or the `PE` located on row i and column j of the PE grid (communication limited to 8 bytes).

`xnetN`[14]`[d].variable`
`xnetpN[d].variable`
`xnetcN[d].variable`
Transmits to the PE's the content of `variable` located in direction `N` and at a distance `d` (communication limited to 8 bytes).

`type`[12]`_xsend(dy,dx,scr,dest,n)`
`type``_xfetch(dy,dx,scr,dest,n)`
Transfer of `n` bytes from `scr` to `dest` for all active PE's in the direction (the same for all PE's) determined by relative displacement (`dx`,`dy`).

`router[f].variable`
Transmits to the PE's the content of `variable` located on indicated target PE's (each active PE indicates its target by the integer `f`).

`type``_rsend(scr,dest,n,f)`
`type``_rfetch(scr,dest,n,f)`
Transfer of `n` bytes from `scr` to `dest` for all active PE's toward target PE's (determined on each PE by the integer `f`).

---

[12] Must be a simple type, among `{char, short, int, long long, float, double}`.
[13] See precise syntax in paragraph on `xsend` and `xfetch` transmissions.
[14] Direction indicated by a capital letter, among `{N, NW, E, SE, S, SW, W, NW}`.

5. Typical communication

We wish to carry out more complex communications among PE's within the grid. For each communication, several solutions are available. Regarding the time required for these solutions, for which we give the program, we must take the executing time for each procedure into account (see time for basic operations).

5.1. One to all

The aim is to transmit a message contained by one of the PE's to the whole grid. The type of information emitted is simple and is 64 bits in size. We will assume that the information to be sent is on PE number 0. The variable `x` on the PE's (`plural` and declared global) is to have the same value as `proc[0]`, at the end of execution of a solution.

Emission through `router`

We use here the last method of communication. The obvious advantage of using `router` is that the distance does not interfere. But being able to associate only two PE's at a time brings the number of conflicts to a maximum (each PE waits for access with the same PE, number 0).

Program 1: performance=139693 cycles=11175.44 µs

```
OneToAll_router()
{ x=router[(plural int)0].x; }
```

Emission through `xnet` *doubling*:

Here, the number of simple PE to PE communications is minimized. We proceed in two stages, as in program 2, first in a row, and then parallelly in columns (see Figure 5.1).

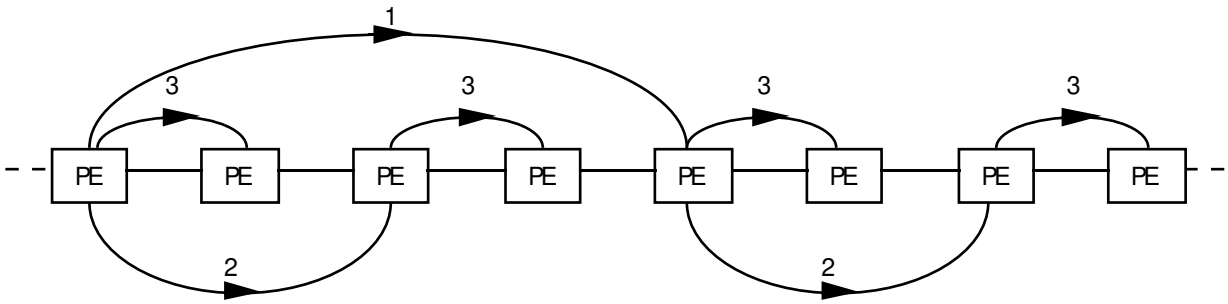


Figure 5.1: Emission of a message through `xnet` *doubling* on an 8-PE ring (torus of dimension 1). The numbers indicate the succession of transfers.

We encounter a complexity in the number of elementary `xnet` communications in O(log$n$) if n is the total number of PE's in the grid. With the same degree of complexity, we could have programmed an emission simultaneously in the two dimensions, but the resulting program would be less efficient, since it produces a greater number of test instructions.

Program 2: performance=12697 cycles=1015.76 µs

```
OneToAll_xnet_doubling()
{
   int puis,i;

   puis = nxproc;
   for (i=1;i<=lnxproc;i++)
      if (ixproc%puis==0) xnetE[puis/=2].x=x;

   puis = nxproc;
   for (i=1;i<=lnxproc;i++)
     if (iyproc%puis==0) xnetS[puis/=2].x=x;

}
```

The variable `lnxproc` has the value $\ln_2$(`nxproc`), which, in this case, is the integer 5.

Emission of information through an `xnetc`

We emit the information in the first row of the grid, then, parallelly, in all the columns.

Program 3: performance=1193 cycles=95.44 µs.

```
OneToAll_xnetc()
{ if (!iproc) xnetcE[nxproc-1].x=x;
  if (!iyproc) xnetcS[nyproc-1].x=x;
}
```

The values of `nxproc` and `nyproc` variables are, respectively, the number of PE's in a row and in a column (in our case, `[nxproc-1]` and `[nyproc-1]` could have been replaced by `[31]`, for a slight time savings).

Emission of information through ACU

For this first solution, we use the special bus twice, between ACU and the grid. At the end of an initial time, the content of PE number 0 can be read, then at the end of a second time, all PE's can simultaneously be assigned this value:

Program 4: performance=542 cycles=43.36 µs

```
OneToAll_ACU()
{ x=proc[0].x; }
```

Note that the two uses of bus ACU↔PE can be represented by a single assignment statement. We also could have written:

```
{ long long t;
  t=proc[0].x;
  x=t; }
```

which is slower to execute, due to the writing and the reading of local variable `t`. In the case of single assignment, the data is probably moved through an ACU register, which is faster than using a local variable and avoids using calculations of address.

Summary table: *one to all*

Method used	time in µs
`OneToAll_ACU()`	43.36
`OneToAll_xnetc()`	95.44
`OneToAll_xnet_doubling()`	1015.76
`OneToAll_router()`	11175.44

5.2. Transposition

The aim is to transpose a `nxproc*nyproc` matrix denoted `x`, for which each element is a simple type of information, of 64 bits (in this case, the matrix is 32x32).

Transposition through routing

A `router` construction is used, with index function `f=iyproc+nyproc*ixproc`, (since we still supposedly have true equality: `iproc=ixproc+nxproc*iyproc`).

Program 5: performance=2673 cycles=213.84 µs

```
Trans_router()
{ x=router[f].x; }
```

Program 6: performance=2783 cycles=222.64 µs

```
Trans_router_bis()
{ if(c) x=router[f].x; }
```

`router` is used in *right-value*, as this is the fastest means of execution. In program 6, `c=(ixproc!=iyproc)`, which is calculated in the program designed for this

function, allows us to deactivate any diagonal PE's which are not susceptible to transposition. But this does not lower the number of conflicts (in the event of a transposition, this number is 16). Here we must add the time for one more selection (see table of basic operations).

Transposition through `xnetp`

We will consider all straight lines parallel to the main diagonal of the matrix to be transposed (see Figure 5.2). The program consists in exchanging, successively, each of these lines (each exchange is transmitted in parallel through an `xnetp`). Since the result must appear in the same entry variable `x`, we duplex `x` in `y`, to avoid loss of data during *swapping*, which is impossible to achieve directly through X-Net communication.

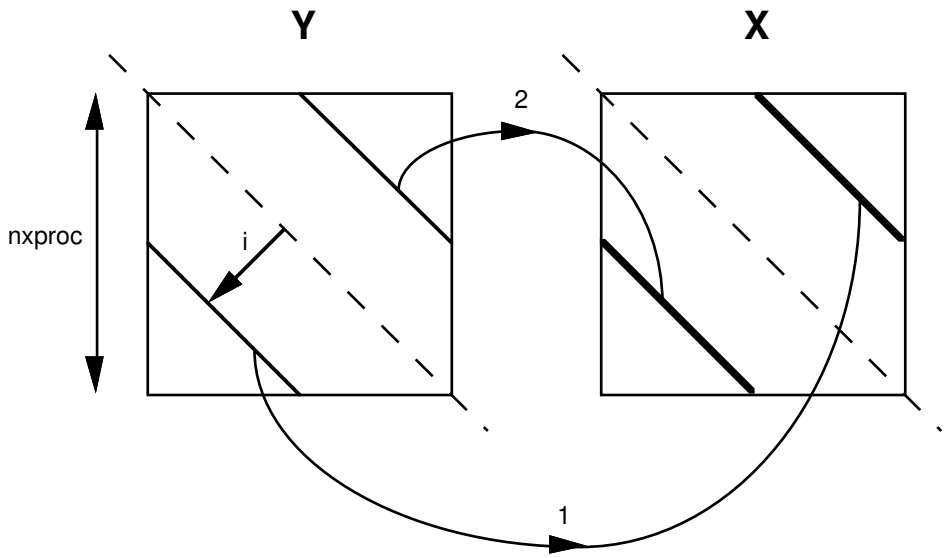


Figure 5.2: Method of transposition through `xnetp,` at stage number `i`

Program 7: performance=36381 cycles=2910.48 µs

```
Trans_xnetp()
{ register int i;
  y=x;
  for(i=1;i<nxproc;i++)
   { if (ixproc-i==iyproc) x=xnetpSW[i].y;
     if (iyproc-i==ixproc) x=xnetpNE[i].y;
     }
}
```

The following solution is a good example of optimization of X-Net communication. We proceed by optimization in *left-value*, particularly of the distance parameter, considering that the diagonal does not need to be transposed and that a simple

`xnet` is faster than an `xnetp` for a distance of 1. In total, we spare almost 1/4 of the execution time (exactly 24.3%).

Program 8: performance=27541 cycles=2203.24 µs

```
Trans_xnetp_bis()
{ register int i,j;
  y=x; /* lower part of y is used */
  if (ixproc-1==iyproc) xnetSW[1].x=x;
  if (iyproc-1==ixproc) xnetNE[1].x=y;

  for(j=2;j<=16;j++)
   { if (ixproc-j==iyproc) xnetpSW[j].x=x;
     if (iyproc-j==ixproc) xnetpNE[j].x=y;
     }

  for(j=15;j>=2;j--)
   { i=32-j; /* If i is replaced, it's longer ! */
     if (ixproc-i==iyproc) xnetpNE[j].x=x;
     if (iyproc-i==ixproc) xnetpSW[j].x=y;
     }

  if (iproc==31) xnetNE[1].x=x;
  if (iproc==992) xnetSW[1].x=y;

}
```

Note that X-Net communications do not depend on the number of active PE's. This program's strategy to deactivate PE's not in use is thus a bad one. The process prolongs the global execution time for a duration equal to the selection time (`if`).

Summary Table: *transposition*

Method used	time in µs
`Trans_router()`	213.84
`Trans_router_bis()`	222.64
`Trans_xnetp()`	2910.48
`Trans_xnetp_bis()`	2203.24

5.3. Personalized one to all

The algorithm involves transmitting a message on all PE's, starting from one PE in particular, as in the classic *one to all* . But in this case the information transmitted is personalized for each PE. The messages to be sent are collected in a table of `nproc` entries (called `scr`) and the type of a personal message is `char`. The final result is a matrix `x` (declared `plural char`). As previously, we assume that PE number 0 functions as transmitting processor, and the variables handled, `x` and `scr`, are defined as global (on the PE's).

Emitting with `proc` communication

The first solution involves assigning each division of the table of PE number 0 directly to the PE destinations, using an iteration of the sequence: `proc[i].x=proc[0].scr[i].`

Program 9: performance=201345 cycles=16107.60 µs

```
OneToAllP_proc()
{ register int i;
  for(i=0;i<nproc;i++) proc[i].x=proc[0].scr[i];
}
```

The following solution uses another type of assignment in the same iteration: `x=proc[0].scr[i]`. According to the table of `proc` communications, we see that `PE2=proc[0].PE1` is faster than `proc[i].PE2=proc[0].PE1` (for the type `char` 78 vs. 122 cycles). But for each assignment, only PE number i should be active, which adds the following condition: `(if(iproc==i)` (61 cycles, see Table of Basic Operations).

Program 10: performance=226513 cycles=18121.04 µs

```
OneToAllP_proc_bis()
{ register int i;
  for(i=0;i<nproc;i++) if(iproc==i) x=proc[0].scr[i];
}
```

A new total of 78+61 vs. 122 cycles is thus obtained for program 9: more precisely, a loss of 17 cycles. We obtain as a theoretical time 201345+1024*17=218735 cycles. The variable `i` is declared as register on the ACU, and `register` variables on the ACU occasionally correspond to a zone of ACU memory. We can thus take 78+68 cycles as the initial time for the conditional instruction (considering `i` as a local variable in ACU memory). This yields a theoretical time which is closer to the experimental time: 201345+1024*24=225921 cycles (with less than 0.3% error).

Emission through rotation

The table of PE number 0 is seen as `nxproc` tables of `nyproc` bytes (a byte=an elementary message=a `char`). For each sub-table, the last non-transferred element is sent into PE number 1; next, each sub-table is shifted one column to the right (see Figure 5.3). In a second time period, the same thing is carried out, moving all columns downward in parrallel, considering each sub-table as `nyproc` divisions of `char`.

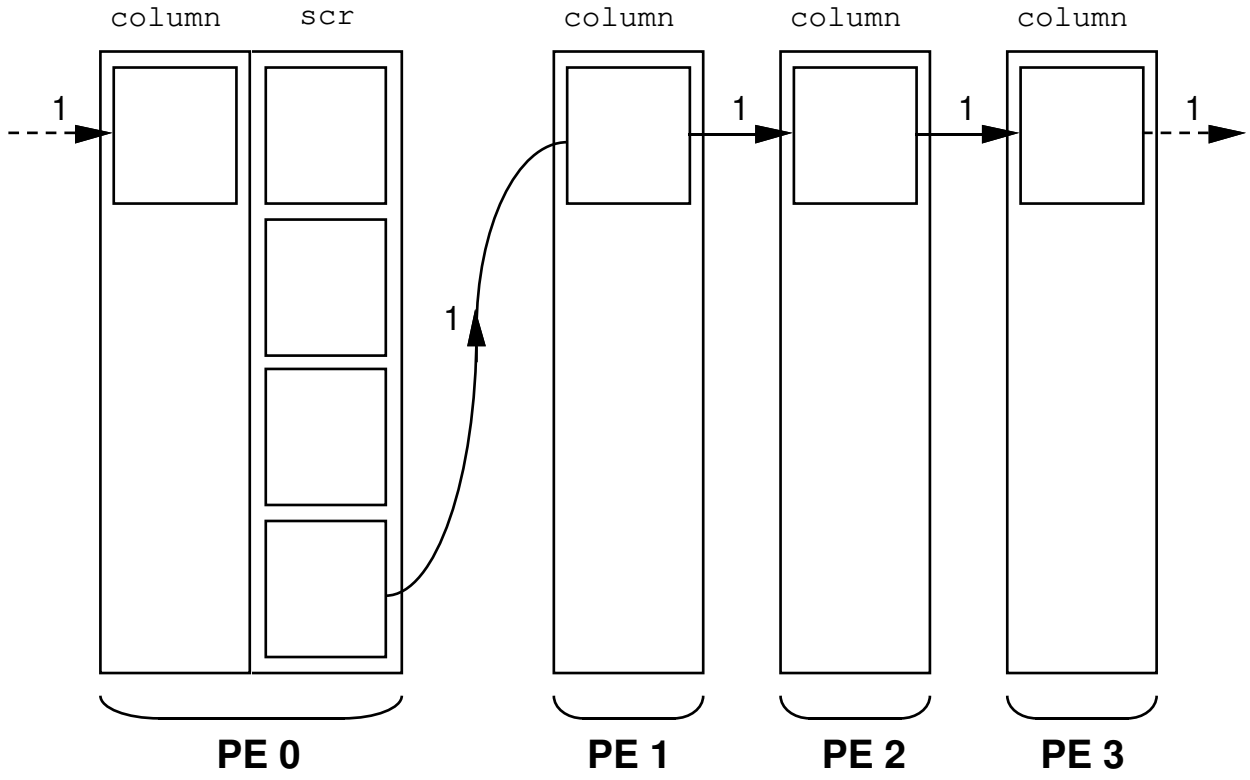


Figure 5.3: Emission through rotation, with the help of a `ps_xsend(0,1,pscr,column,sizeof(column))` on a 4-PE ring, where `pscr` is a pointer of value `scr` on PE #0 and `column` on other PE's[15].

Note that through this method, the correct value of the main table PE 3 is attained in only 3 steps.

Program 11: performance=222316 cycles=17785.28 µs

```
OneToAllP_ps_xsend()
{ register int i;

pscr=column;
```

[15] Contrary to the diagram, the memory zones indicated by `column` and `scr` are allocated on all PE's (and not just on PE number 0, or just on the first row). Moreover, instructions allowing memory allocation on the PE's cannot be personalized.

```
if (iproc==0) pscr= &(scr[nproc]);
for(i=1;i<nxproc;i++)
 { if (iproc==0) pscr-=sizeof(column);
/* pointer #0 points toward the last non-transferred
elememt of scr[] */
  ps_xsend(0,1,pscr,column,sizeof(column));
 }
if   (iproc==0)   for(i=0;i<sizeof(column);i++)
column[i]=scr[i];

pscr= &x;
if (iyproc==0) pscr= &(column[sizeof(column)]);
for(i=1;i<nyproc;i++)
{ if (iyproc==0) pscr-=sizeof(x);
/* pointer#0 points toward the last non-transferred
elememt of column[] */
  ps_xsend(-1,0,pscr,&x,sizeof(x));
  }
if (iyproc==0) x=column[0];

}
```

Caution: the result produced is the transposition of matrix `x`, with respect to programs 9 and 10, i.e. the elements must be entered differently to yield the same result.

An alternative of this program would be to use an `ss_xsend`, rather than a `ps_xsend` (slower). But this would mean recopying the element to be shifted beforehand. A time savings is not apparent, especially since the number of shifts is no longer `nxproc-1`, but `nxproc`.

Summary table: personalized *one to all*

Method used	time in µs
`OneToAllP_proc()`	16107.60
`OneToAllP_proc_bis()`	18121.04
`OntToAllP_ps_xsend()`	17785.28

6. Data processing through communications

In this chapter we will speak briefly about complex arithmetical operations, about *sort*, and about *scan*, *reduce* and *sendwith*. type functions. We will not clarify the syntax of these functions; for further details, see [Ma].

6.1. Scan, reduce and sendwith functions

All functions have been tested with signed parameters.

For `scan` functions, the segment (SEG) parameter is initialized to zero, which corresponds to a calculation upon all PE's.

Scan	char	short	int	long	float	double
PE3=scanAnd?[16](PE2,SEG)	2089	2987	4115	7996	-	-
PE3=scanOr?(PE2,SEG)	2089	2781	4115	6914	-	-
PE3=scanMax?(PE2,SEG)	2163	4057	4262	7135	6409	7944
PE3=scanMin?(PE2,SEG)	2155	2862	5328	8315	5293	7724
PE3=scanAdd?(PE2,SEG)	2089	2774	4125	6951	5546	8166
PE3=scanMul?(PE2,SEG)	2476	3581	6310	13736	6310	11776

It can be said that `scan` functions depend neither on the number of active PE's nor on the segment lengths. In other words, the time of `scan` functions is constant. This means that `scan` functions are not optimized for these two parameters in all cases.

Reduce	char	short	int	long	float	double
PE3=reduceAnd?(PE2)	1095	1347	1610	2509	-	-
PE3=reduceOr?(PE2)	211	263	262	502	-	-
PE3=reduceMax?(PE2)	431	586	825	1158	1329	1183
PE3=reduceMin?(PE2)	487	708	1067	1610	1743	1844
PE3=reduceAdd?(PE2)	1095	1315	1610	2509	3120	3781
PE3=reduceMul?(PE2)	2508	2101	3810	9345	3879	7356

For `sendwith` functions, the destination (D) parameter, comparable to the index function of a `router`, is random.

[16] The "?" indicates that the name of the instruction varies according to the type of parameters. The name of the instruction must contain an element from the list {8,16,32,64,f,d} (see [Ma]).

SendWith	char	short	int	long	float	double
PE3=sendwithAnd?(PE2,D)	40431	40653	42260	45855	-	-
PE3=sendwithOr?(PE2,D)	40598	41316	42200	45736	-	-
PE3=sendwithMax?(PE2,D)	39880	40653	41980	45515	44411	45736
PE3=sendwithMin?(PE2,D)	38664	39438	40984	44852	43416	45073
PE3=sendwithAdd?(PE2,D)	39769	40580	41906	45442	44410	45810
PE3=sendwithMul?(PE2,D)	39548	40377	42090	46896	44116	46311

6.2. Sorting and ranking functions

Sort	char	short	int	long	float	double
PE3=psort?(PE1)	13544	18885	27946	47062	31592	50156
PE3=rank?(PE1)	23194	27835	36307	54024	39659	57007

6.3. Cosine, square root, arctangent and absolute value functions

In these complex functions, it appears that the times depend, sometimes significantly, on the content of the variable. Therefore calculating cos(0) or cos(π) is faster than calculating cos(0.27). Note that the times are not averages of several random variable values, but are averages of several repetitions of the instruction on the same random value (see Paragraph 3.4). The times shown represent those times obtained most often with the test program.

Cosine	float	double
ACU3=cos(ACU2)	7934	15791
ACU3=cos(CONST)	10075	15791
PE3=cos(PE2)	5741	13210
Square Root		
ACU3=sqrt(ACU2)	1789	2320
ACU3=sqrt(CONST)	1647	2310
PE3=sqrt(PE2)	2610	3614
ArcTg(y/x)		
ACU3=atan2(ACU2,ACU1)	6288	5515
PE3=atan2(PE2,PE1)	10376	19511
Absolute Value		
ACU3=fabs(ACU2)	329	61
ACU3=fabs(CONST)	193	60
PE3=fabs(PE2)	600	597

7. Conclusion

In this report, I have sought to create a model of calculation and communication operations on the MasPar.

The arithmetical operations are slow, as each node is a 4-bit processor, but are globally efficient, with 1024 processors.

For communications, the grid structure makes distance highly essential. But for uniform and local communications (from processors to neighbor processors) the MasPar is very efficient. Personalized communications (from processors to processors at varying distances) are less efficient.

The magnitude of calculation times and of communications is the same. This implies that the high efficiency is due to optimal use of these two types of operations.

The adaptation and experimentation have been applied to an area other than that of the present study. These results were put into use and confirmed by practical work done on the same machine, by C. POIZAT, for the sort, and by S. SALANEUVE, for the polynomial root search.

8. Appendix

8.1. Tables of `xnet`, `xnetp` and `xnetc` communications

All tests on the transfer of data apply to variables of 8, 16, 32 and 64 bits. As the speed of X-Net instructions does not depend on direction, we have arbitrarily chosen North (N). Note that, although distances greater than 32 are useless for the X-Net's (except for `xnetc`), these instructions are not optimized as a function of distance (it would, for example, be possible to keep only 4 bits of weak weight in the distance parameter). For pipelined `xnetc` and `xnetp` functions, the time is independent of the original processor's number (and thus of the final processor's number). We will make use of PE 0. The selection is made by way of an `if(c)` type instruction, where `c=(iproc==0)` is a constant calculated prior to the test instruction (therefore, for the actual time, take away 110 cycles).

Reading XnetN[d]	char	short	int	long
PE3=XnetN[0].PE1	65	92	174	337
PE3=XnetN[1].PE1	73	100	190	370
PE3=XnetN[2].PE1	86	122	226	436
PE3=XnetN[4].PE1	110	164	295	569
PE3=XnetN[8].PE1	160	247	431	833
PE3=XnetN[16].PE1	258	407	705	1361
PE3=XnetN[32].PE1	452	726	1279	2420

Writing XnetN[d]	char	short	int	long
XnetN[0].PE1=PE2	51	92	175	339
XnetN[1].PE1=PE2	56	100	193	371
XnetN[2].PE1=PE2	68	122	229	438
XnetN[4].PE1=PE2	88	157	298	571
XnetN[8].PE1=PE2	129	232	433	835
XnetN[16].PE1=PE2	211	376	706	1363
XnetN[32].PE1=PE2	371	664	1251	2420

Writing XnetpN[d]	char	short	int	long
if(c)XnetpN[0].PE1=PE2	166	210	291	456
if(c)XnetpN[1].PE1=PE2	174	219	310	493
if(c)XnetpN[2].PE1=PE2	179	226	321	512
if(c)XnetpN[4].PE1=PE2	183	228	321	514
if(c)XnetpN[8].PE1=PE2	186	233	327	520
if(c)XnetpN[16].PE1=PE2	194	242	337	526
if(c)XnetpN[32].PE1=PE2	211	257	352	542

Reading XnetcN[d]	char	short	int	long
if(c)PE3=XnetcN[0].PE1	183	224	307	472
if(c)PE3=XnetcN[1].PE1	195	239	331	512
if(c)PE3=XnetcN[2].PE1	210	258	365	579
if(c)PE3=XnetcN[4].PE1	211	262	368	579
if(c)PE3=XnetcN[8].PE1	216	271	376	589
if(c)PE3=XnetcN[16].PE1	233	284	395	608
if(c)PE3=XnetcN[32].PE1	266	321	431	640

Writing XnetcN[d]	char	short	int	long
if(c)XnetcN[0].PE1=PE2	166	210	291	456
if(c)XnetcN[1].PE1=PE2	174	219	310	493
if(c)XnetcN[2].PE1=PE2	183	237	343	558
if(c)XnetcN[4].PE1=PE2	186	238	346	559
if(c)XnetcN[8].PE1=PE2	190	242	350	564
if(c)XnetcN[16].PE1=PE2	198	251	358	572
if(c)XnetcN[32].PE1=PE2	211	266	376	586

8.2. Tables of `xsend` and `xfetch` communications

The variables used for the instruction tests are the same as in the paragraph on syntax in `xsend` and `xfetch`. To simulate the distance parameter on the two variables `dx` and `dy`, we do it as a function of `d`: `dx=d/3` and `dy=d-dx`. We therefore still have $d=|dx|+|dy|$ and $|dx|\neq|dy|$ (at least for $d>1$). We deliberately limit the distance to 16, since greater distances do not generally exist in a program (see Optimization of X-Net Instructions on Distance).

ss_xsend(dy,dx,s_scr,s_dest,n)

Instruction \ d	1	2	4	8	16
ss_xsend(...,1)	655	679	986	1030	1117
ss_xsend(...,2)	775	800	1221	1312	1429
ss_xsend(...,4)	844	878	1345	1451	1626
ss_xsend(...,8)	1200	1270	2091	2303	2644
ss_xsend(...,16)	1907	2053	3582	3998	4678
ss_xsend(...,32)	3322	3611	6549	7395	8757
ss_xsend(...,64)	6152	6732	12501	14172	16900
ss_xsend(...,128)	11814	12976	24408	27740	33187
ss_xsend(...,256)	23122	25452	48174	54863	65776
ss_xsend(...,512)	45739	50412	95795	109122	130956
ss_xsend(...,1024)	91036	100327	190962	217645	261310

sp_xsend(dy,dx,s_scr,p_dest,n)

Instruction \ d	1	2	4	8	16
sp_xsend(...,1)	1279	1318	1658	1812	2073
sp_xsend(...,2)	1463	1535	1977	2155	2465
sp_xsend(...,4)	1580	1656	2126	2346	2714
sp_xsend(...,8)	2089	2197	3027	3341	3883
sp_xsend(...,16)	3107	3300	4823	5347	6208
sp_xsend(...,32)	5145	5458	8392	9354	10891
sp_xsend(...,64)	9217	9807	15592	17360	20275
sp_xsend(...,128)	17372	18486	29956	33385	38997
sp_xsend(...,256)	33664	35844	58665	65436	76453
sp_xsend(...,512)	66258	70561	116122	129527	151423
sp_xsend(...,1024)	131446	139992	231009	257706	301323

ps_xsend(dy,dx,p_scr,s_dest,n)

Instruction \d	1	2	4	8	16
ps_xsend(...,1)	1150	1166	1460	1504	1591
ps_xsend(...,2)	1351	1370	1763	1864	1981
ps_xsend(...,4)	1520	1550	2041	2158	2365
ps_xsend(...,8)	2090	2166	3009	3229	3606
ps_xsend(...,16)	3238	3379	4926	5365	6067
ps_xsend(...,32)	5530	5843	8784	9637	11038
ps_xsend(...,64)	10127	10705	16492	18167	20910
ps_xsend(...,128)	19304	20430	31927	35237	40690
ps_xsend(...,256)	37691	39879	62797	69337	80265
ps_xsend(...,512)	74466	78831	124523	137583	159404
ps_xsend(...,1024)	148015	156734	247995	274084	317676

pp_xsend(dy,dx,p_scr,p_dest,n)

Instruction \d	1	2	4	8	16
pp_xsend(...,1)	2542	2585	2917	3081	3326
pp_xsend(...,2)	2817	2861	3324	3523	3819
pp_xsend(...,4)	2956	3027	3580	3802	4188
pp_xsend(...,8)	3651	3748	4701	5028	5576
pp_xsend(...,16)[17]	5070	5220	6929	7466	8380
pp_xsend(...,32)[18]	7835	8164	11406	12364	13908
pp_xsend(...,64)	13397	14010	20321	22091	25016
pp_xsend(...,128)	24519	25723	38225	41610	47281
pp_xsend(...,256)	46731	49149	74025	80653	91701
pp_xsend(...,512)	91190	96002	145629	158667	180601
pp_xsend(...,1024)	180101	189705	288835	314693	358450

[17] For n<=16, an equivalent instruction `pp_xfetch` is faster.
[18] From n>=32 onward, `pp_xsend` becomes faster than `pp_xfetch`.

ss_xfetch(dy,dx,s_scr,s_dest,n)

Instruction \ d	1	2	4	8	16
ss_xfetch(...,1)	727	742	1119	1190	1282
ss_xfetch(...,2)	875	906	1428	1537	1702
ss_xfetch(...,4)	930	969	1537	1684	1891
ss_xfetch(...,8)	1291	1361	2310	2533	2923
ss_xfetch(...,16)	2010	2155	3819	4264	4985
ss_xfetch(...,32)	3453	3745	6842	7703	9125
ss_xfetch(...,64)	6336	6896	12882	14577	17393
ss_xfetch(...,128)	12091	13214	24966	28322	33915
ss_xfetch(...,256)	23619	25854	49177	55816	67025
ss_xfetch(...,512)	46666	51152	97564	110806	133196
ss_xfetch(...,1024)	92766	101724	194357	220771	265523

sp_xfetch(dy,dx,s_scr,p_dest,n)

Instruction \ d	1	2	4	8	16
sp_xfetch(...,1)	1205	1221	1586	1661	1757
sp_xfetch(...,2)	1432	1463	1977	2090	2265
sp_xfetch(...,4)	1568	1609	2197	2322	2531
sp_xfetch(...,8)	2172	2248	3192	3436	3821
sp_xfetch(...,16)	3377	3524	5185	5660	6398
sp_xfetch(...,32)	5789	6103	9197	10099	11524
sp_xfetch(...,64)	10597	11255	17226	18993	21792
sp_xfetch(...,128)	20255	21488	33226	36741	42313
sp_xfetch(...,256)	39548	41979	65257	72256	83380
sp_xfetch(...,512)	78112	83011	129328	143308	165514
sp_xfetch(...,1024)	155264	165074	257455	285410	329721

ps_xfetch(dy,dx,p_scr,s_dest,n)

Instruction \ d	1	2	4	8	16
ps_xfetch(...,1)	1315	1371	1760	1950	2206
ps_xfetch(...,2)	1561	1625	2192	2397	2729
ps_xfetch(...,4)	1674	1754	2337	2585	2970
ps_xfetch(...,8)	2252	2356	3304	3658	4188
ps_xfetch(...,16)	3413	3580	5261	5808	6706
ps_xfetch(...,32)	5734	6029	9160	10118	11702
ps_xfetch(...,64)	10344	10928	16950	18718	21722
ps_xfetch(...,128)	19591	20725	32521	35937	41757
ps_xfetch(...,256)	38089	40305	63692	70376	81802
ps_xfetch(...,512)	75074	79474	126032	139223	161910
ps_xfetch(...,1024)	149054	157810	250697	276975	322134

pp_xfetch(dy,dx,p_scr,p_dest,n)

Instruction \ d	1	2	4	8	16
pp_xfetch(...,1)	1976	2010	2364	2531	2799
pp_xfetch(...,2)	2294	2357	2858	3058	3392
pp_xfetch(...,4)	2447	2527	3083	3335	3742
pp_xfetch(...,8)	3240	3348	4301	4667	5221
pp_xfetch(...,16)[19]	4822	5009	6751	7318	8222
pp_xfetch(...,32)[20]	7988	8324	11629	12641	14242
pp_xfetch(...,64)	14326	14954	21420	23248	26243
pp_xfetch(...,128)	26995	28214	40874	44482	50265
pp_xfetch(...,256)	52338	54734	79882	86950	98279
pp_xfetch(...,512)	103019	107775	157893	171870	194322
pp_xfetch(...,1024)	204386	213855	313920	341727	386429

8.3. Tables of `router` communications

The number of conflicts is calculated with the function `conflit()`. PE selection is carried out in the following manner: `c(d)=(iproc<d)` represents a condition allowing the selection of exactly `d` PE's (`d` variations from 1 to 1024). We use the index function `f=0` to produce the maximum number of conflicts (see paragraph on `router`). In the

[19] For n<=16, an equivalent instruction `pp_xsend` is not as fast.
[20] From n>=32 onward, `pp_xfetch` becomes slower than `pp_xsend`.

times shown above, the time of one selection (110 cycles) can be taken away, to obtain the actual instruction time of the communication.

`router` in *right-value*

PE2=router[f].PE1	char	short	int	long	conflicts
if(c(1)) PE2=router[f].PE1	291	315	404	585	1
if(c(2)) PE2=router[f].PE1	369	402	505	721	2
if(c(4)) PE2=router[f].PE1	530	579	713	992	4
if(c(8)) PE2=router[f].PE1	842	923	1131	1536	8
if(c(16)) PE2=router[f].PE1	1474	1625	1960	2623	16
if(c(32)) PE2=router[f].PE1	2737	3040	3624	4797	32
if(c(64)) PE2=router[f].PE1	5262	5868	6951	9151	64
if(c(128)) PE2=router[f].PE1	10314	11526	13603	17854	128
if(c(256)) PE2=router[f].PE1	20417	22841	26915	35261	256
if(c(512)) PE2=router[f].PE1	40622	45472	53545	70078	512
if(c(1024)) PE2=router[f].PE1	81034	90732	106787	139710	1024

`router` in *left-value*

router[f].PE1=PE2	char	short	int	long	conflicts
if(c(1)) router[f].PE1=PE2	303	321	409	592	1
if(c(2)) router[f].PE1=PE2	388	420	520	735	2
if(c(4)) router[f].PE1=PE2	564	609	741	1021	4
if(c(8)) router[f].PE1=PE2	918	985	1183	1610	8
if(c(16)) router[f].PE1=PE2	1625	1758	2067	2789	16
if(c(32)) router[f].PE1=PE2	3039	3273	3835	5146	32
if(c(64)) router[f].PE1=PE2	5868	6304	7371	9861	64
if(c(128)) router[f].PE1=PE2	11526	12366	14443	19290	128
if(c(256)) router[f].PE1=PE2	22841	24489	28587	38149	256
if(c(512)) router[f].PE1=PE2	45472	48736	56875	75866	512
if(c(1024)) router[f].PE1=PE2	90733	97230	113451	151301	1024

8.4. Tables of `rsend` and `rfetch` communications

The number of conflicts is calculated int he same way as for `router` communications (see Paragraph 8.4). It is thus fitting to take away the time of one selection instruction (110 cycles) for each value of the table. We limit the number of conflicts to 16, which is the average number of conflicts corresponding, for example, to matrix transposition.

Size \ Conflict	1	2	4	8	16
ss_rsend(f,scr,dest,1)	490	651	948	1537	2752
ss_rsend(f,scr,dest,2)	504	666	997	1648	2973
ss_rsend(f,scr,dest,4)	530	724	1095	1868	3379
ss_rsend(f,scr,dest,8)	579	827	1316	2298	4262
ss_rsend(f,scr,dest,16)	726	1117	1902	3452	6583
ss_rsend(f,scr,dest,32)	1021	1702	3080	5799	11260
ss_rsend(f,scr,dest,64)	1610	2863	5402	10449	20543
ss_rsend(f,scr,dest,128)	2788	5220	10083	19805	39254
ss_rsend(f,scr,dest,256)	5146	9935	19511	38665	76971
ss_rsend(f,scr,dest,512)	9861	19363	38370	76383	152406
ss_rsend(f,scr,dest,1024)	19290	38223	76088	151816	303276

Size \ Conflict	1	2	4	8	16
sp_rsend(f,scr,dest,1)	741	984	1482	2470	4434
sp_rsend(f,scr,dest,2)	765	1039	1592	2695	4906
sp_rsend(f,scr,dest,4)	779	1070	1647	2790	5074
sp_rsend(f,scr,dest,8)[21]	836	1175	1856	3196	5920
sp_rsend(f,scr,dest,16)[22]	1020	1538	2598	4690	8901
sp_rsend(f,scr,dest,32)	1389	2289	4078	7669	14829
sp_rsend(f,scr,dest,64)	2126	3771	7060	13602	26687
sp_rsend(f,scr,dest,128)	3600	6731	12985	25478	50487
sp_rsend(f,scr,dest,256)	6556	12642	24797	49106	97727
sp_rsend(f,scr,dest,512)	12478	24450	48495	96454	192442
sp_rsend(f,scr,dest,1024)	24298	48166	95780	191093	381456

[21] Slower than `ps_rfetch`.
[22] Faster than `ps_rfetch`.

Size \ Conflict	1	2	4	8	16
ps_rsend(f,scr,dest,1)	680	873	1252	2010	3525
ps_rsend(f,scr,dest,2)	692	899	1313	2127	3754
ps_rsend(f,scr,dest,4)	721	953	1416	2343	4187
ps_rsend(f,scr,dest,8)	781	1073	1650	2817	5130
ps_rsend(f,scr,dest,16)	973	1457	2420	4335	8167
ps_rsend(f,scr,dest,32)	1356	2221	3967	7409	14338
ps_rsend(f,scr,dest,64)	2125	3746	7017	13545	26601
ps_rsend(f,scr,dest,128)	3655	6830	13151	25810	51151
ps_rsend(f,scr,dest,256)	6720	12954	25405	50315	100146
ps_rsend(f,scr,dest,512)	12847	25201	49935	99356	198226
ps_rsend(f,scr,dest,1024)	25109	49717	98953	197414	394328

Size \Conflict	1	2	4	8	16
pp_rsend(f,scr,dest,1)	991	1261	1813	2916	5128
pp_rsend(f,scr,dest,2)	1005	1291	1867	3010	5317
pp_rsend(f,scr,dest,4)	1032	1347	1978	3240	5767
pp_rsend(f,scr,dest,8)	1094	1463	2200	3702	6692
pp_rsend(f,scr,dest,16)	1316	1923	3139	5542	10375
pp_rsend(f,scr,dest,32)	1784	2856	4985	9244	17778
pp_rsend(f,scr,dest,64)	2713	4705	8682	16638	32551
pp_rsend(f,scr,dest,128)	4558	8394	16092	31460	62199
pp_rsend(f,scr,dest,256)	8267	15813	30926	61096	121466
pp_rsend(f,scr,dest,512)	15680	30633	60543	120361	239996
pp_rsend(f,scr,dest,1024)	30488	60267	119825	238918	477128

Size \ Conflict	1	2	4	8	16
ss_rfetch(f,scr,dest,1)	476	615	896	1462	2581
ss_rfetch(f,scr,dest,2)	490	647	946	1555	2774
ss_rfetch(f,scr,dest,4)	516	697	1057	1760	3203
ss_rfetch(f,scr,dest,8)	569	800	1260	2199	4042
ss_rfetch(f,scr,dest,16)	715	1095	1852	3367	6398
ss_rfetch(f,scr,dest,32)	1011	1684	3011	5699	11045
ss_rfetch(f,scr,dest,64)	1599	2862	5368	10376	20394
ss_rfetch(f,scr,dest,128)	2773	5218	10080	19804	39252
ss_rfetch(f,scr,dest,256)	5136	9933	19510	38663	76970
ss_rfetch(f,scr,dest,512)	9847	19362	38368	76380	152404
ss_rfetch(f,scr,dest,1024)	19279	38222	76085	151816	303273

Size \ Conflict	1	2	4	8	16
sp_rfetch(f,scr,dest,1)	659	839	1197	1911	3307
sp_rfetch(f,scr,dest,2)	680	873	1259	2028	3563
sp_rfetch(f,scr,dest,4)	708	930	1371	2253	3968
sp_rfetch(f,scr,dest,8)	769	1040	1586	2674	4850
sp_rfetch(f,scr,dest,16)	956	1413	2332	4164	7824
sp_rfetch(f,scr,dest,32)	1328	2163	3827	7150	13800
sp_rfetch(f,scr,dest,64)	2076	3658	6813	13139	25758
sp_rfetch(f,scr,dest,128)	3573	6646	12805	25107	49712
sp_rfetch(f,scr,dest,256)	6564	12630	24769	49048	97576
sp_rfetch(f,scr,dest,512)	12550	24603	48705	96896	193252
sp_rfetch(f,scr,dest,1024)	24520	48536	96567	192627	384751

Size \ Conflict	1	2	4	8	16
ps_rfetch(f,scr,dest,1)	728	947	1390	2273	4040
ps_rfetch(f,scr,dest,2)	748	983	1449	2386	4263
ps_rfetch(f,scr,dest,4)	775	1039	1568	2638	4742
ps_rfetch(f,scr,dest,8)[23]	832	1151	1794	3083	5647
ps_rfetch(f,scr,dest,16)[24]	1024	1537	2568	4630	8755
ps_rfetch(f,scr,dest,32)	1409	2309	4101	7697	14868
ps_rfetch(f,scr,dest,64)	2192	3857	7194	13881	27267
ps_rfetch(f,scr,dest,128)	3742	6950	13360	26179	51814
ps_rfetch(f,scr,dest,256)	6832	13138	25736	50931	101318
ps_rfetch(f,scr,dest,512)	13024	25514	50488	100435	200326
ps_rfetch(f,scr,dest,1024)	25398	50266	99992	199443	398342

[23] Faster than `sp_rsend`.
[24] Slower than `sp_rsend`.

Size \ Conflict	1	2	4	8	16
pp_rfetch(f,scr,dest,1)	945	1215	1757	2862	5072
pp_rfetch(f,scr,dest,2)	954	1241	1813	2957	5257
pp_rfetch(f,scr,dest,4)	992	1315	1974	3260	5844
pp_rfetch(f,scr,dest,8)	1043	1423	2162	3636	6620
pp_rfetch(f,scr,dest,16)	1276	1878	3083	5514	10376
pp_rfetch(f,scr,dest,32)	1753	2808	4960	9229	17793
pp_rfetch(f,scr,dest,64)	2673	4668	8664	16638	32585
pp_rfetch(f,scr,dest,128)	4536	8390	16120	31590	62531
pp_rfetch(f,scr,dest,256)	8254	15846	31003	61353	122054
pp_rfetch(f,scr,dest,512)	15686	30713	60769	120881	241105
pp_rfetch(f,scr,dest,1024)	30595	60506	120346	239997	479340

8.5. `TOOLS.H` Header file

In this paragraph we present all the definitions and declarations of C [K&C] that were used in all the tests.

```
#include <mpl.h>
#include <stdio.h>
#include <sys/file.h>
#include <ppeio.h>
#include <limits.h>
#include <math.h>

#define min(x,y) ((x)<(y)?(x):(y))
#define max(x,y) ((x)>(y)?(x):(y))
#define abs(x) ((x)<0?(-x):(x))
#define FORI 228
#define STARTUP 10

void initprof()
{ asm("cmov32 $0x10, c1");          /* TEUOPS_TACUOPS_TCOMOPS */
  asm("csmov32 c1, cs3");           /* PROFCFG */
}

unsigned int profcnt()
{ asm("csmov32 cs4, c1"); }         /* PROFC1 */

unsigned int top(u,i)
unsigned int u;
int i;
/*
```

```
** Gives the number of units corresponding to the number of
** clock cycles, expressed by CLOCK.  It should be
** noted that the maximum duration is 343"60=5'43"60.
** The number of loops must also be specified.
** u is the total number of cycles observed and i is the
** number of the repetition of the measure. To obtain the
** time in micro seconds, divide by the frequency in
** MHz, i.e. for PE's 12.5 MHz.
*/
{ return((u-FORI)/i-STARTUP);
}

/*
** Gives the number of times of execution of a program
** sequence, in clock cycles.
** The program P, which must not contain any ";"
** at the end, is executed n times. The total time is put
** in a variable which must be global and be an
** unsigned int , due to the macro and the assignment
** u=profcnt(). It seems that if the CLOCK macro is repeated
** too quickly the 16 bits of strong weight in u
** are initialized only randomly,
** with a probability of about 1%. The error of the
** CLOCK function is slightly proportional to n and to P, but it has
** a potential of approximately 20. Of course, this number is
** negligible when n and P become large. It must be
** noted that the number u of cycles is always even.
** For macros, local variables must
** end with "_"
*/
#define CLOCK(P,n,u)                    \
{ register unsigned int i_;             \
  initprof();                           \
  for(i_=0;i_<(n);i_++) {P;}            \
  u=profcnt();                          \
  }

/*
** Macro similar to CLOCK. Here, to lessen the chances of
** error, the measure is separated into two equal repetitions,
** thus obtaining two measures. We can evaluate the variation
** of the measure: if it exceeds 5%, we take the smallest,
** or the largest, if it is arbitrarily less than
```

```
** 2 (4 being the shortest duration of type ACU=CONST);
** otherwise, we determine the average. This is only efficient if
** the number of the repetition is N>1, because if N=1 nothing further
** can be done. The macro uses the top() function and thus gives back
** the number of cycles for the instruction {P}.
** It can be seen that N is considered even.
** As is CLOCK, the variable unsigned int T must be global.
*/
#define CYCLES(P,N,T)                                                    \
{ register unsigned int i_,p_=(N)/2;unsigned int u1_,u2_,m_;             \
  initprof();for(i_=0;i_<p_;i_++) {P;} u1_=profcnt();                    \
  initprof();for(i_=0;i_<p_;i_++) {P;} u2_=profcnt();                    \
  m_=((u1_<1)||(u2_<1)) ? max(u1_,u2_) : min(u1_,u2_);                   \
  T=((fabs(u1_-u2_)/(double)m_)>0.05)?top(m_,p_):top((u1_+u2_)/2,p_);  \
}

double uniform(a,b)
double a,b;
/*
** Returns a random uniform value in [a,b].
** Caution:  a must be <b.
*/
{ double x;
  x=(double)random()*(double)M_LN2+(double)M_E;
  x=fabs(x);
  x=x-floor(x);
  x=x*(double)(b-a)+a;
  return(x);
}

plural double p_uniform(a,b)
plural double a,b;
/*
** Similar effect to uniform(a,b), but here the
** parameters (a,b) can be different for
** each processor.
*/
{ plural double x;
  x=(plural double)p_random()*(plural double)M_LN2+(plural double)M_E;
  x=p_fabs(x);
  x=x-p_floor(x);
  x=x*(plural double)(b-a)+a;
  return(x);
```

```
}

void initrand()
/***********************************************
Initializes the pseudo-random number generating
function. The functions srand(), srand48() and
p_srandom() do not work. In addition, times always
sends back an invalid number, not corresponding to
"man times". The remaining random values always
being the same, by default, we randomly obtain this
table; it's complicated, but it works!
***********************************************/
{
int handle;
register int s=0,n;
char c;

system("date >! /tmp/tampon.date");
handle=open("/tmp/tampon.date",O_RDONLY);
while (read(handle,&c,sizeof(char))>0) s+=(int)c;
close(handle);
system("\\rm /tmp/tampon.date");

for(n=0;n<s;n++) {random();p_random();}
}

void string(c,n)
/***************************************
Reproduces the character c, n times.
***************************************/
char c;
int n;
{ int j;for(j=0;j<n;j++)printf("%c",c);}

#define type char
#define stype "char"
/*
** Defines the type of an element in a table.
** Caution: TX must be at least equal to the number
** of active processors ordered to receive the table of
** processor#0. A problem can also arise regarding
 ** memory size, since the xsend routine seems to
** recopy data in a memory buffer zone
```

```
** on the PE's before transferring it to its destination
*/
#define TX 1024                    /* maximum size in elements */
plural type scr[TX];      /* works for xfetch,xsend,rfexth,rsend */
plural type dest[TX];
plural type * plural pscr=scr;
plural type * plural pdest=dest;

void tx_start()
/*************************************************************
Randomly determines the tables scr and dest, above.
These variables are considered zones of 8*TX bytes.
*************************************************************/
{
register int i;
for(i=0;i<TX;i++)  {  scr[i]=(plural  char)p_random();  dest[i]=(plural
char)p_random(); }
}

#define ck 123
#define sk 12345
#define ik 12345678
#define lk 1234567890
#define fk 12345.12345
#define dk 123456789.123456789

char cacu1,cacu2,cacu3;
short sacu1,sacu2,sacu3;
int iacu1,iacu2,iacu3;
long long lacu1,lacu2,lacu3;
float facu1,facu2,facu3;
double dacu1,dacu2,dacu3;

plural char cpe1,cpe2,cpe3;
plural short spe1,spe2,spe3;
plural int ipe1,ipe2,ipe3;
plural long long lpe1,lpe2,lpe3;
plural float fpe1,fpe2,fpe3;
plural double dpe1,dpe2,dpe3;

void start()
/*************************************************************
 Randomly determines the global variables above.
```

```
 The variables ??2 are non-zero and the
 variables f?2 et d?2 are in ]-1,0[ U ]0,1[.
**********************************************************/
{
cacu1=(char)random();
while ((cacu2=(char)random())==0);
sacu1=(short)random();
while ((sacu2=(short)random())==0);
iacu1=(int)mrand48();
while ((iacu2=(int)mrand48())==0);
lacu1=(long long)mrand48();
while ((lacu2=(long long)mrand48())==0);
facu1=(float)drand48();
while ((facu2=uniforme(-1.0,1.0))==0);
dacu1=(double)drand48();
while ((dacu2=uniforme(-1.0,1.0))==0);

cpe1=(plural char)p_random();
while ((cpe2=(plural char)p_random())==0);
spe1=(plural short)p_random();
while ((spe2=(plural short)p_random())==0);
ipe1=(plural int)p_random();
while ((ipe2=(plural int)p_random())==0);
lpe1=(plural long long)p_random();
while ((lpe2=(plural long long)p_random())==0);
fpe1=(plural float)p_random();
while ((fpe2=p_uniforme((plural double)(-1.0),(plural double)1.0))==0);
dpe1=(plural double)p_random();
while ((dpe2=p_uniforme((plural double)(-1.0),(plural double)1.0))==0);
}

void printl(z)
plural long long z;
/*****************************************************************
Displays a rectangle of nxproc*nyproc divisions, each containing the
value of z on a processor. The directions x and y are respected.
The upper left angle is processor 0. The variable to be displayed must
be long long. A line is skipped at the end of the display.
*****************************************************************/
{ int x,y;
printf("\n");
for(y=0;y<nyproc;y++)
  { for(x=0;x<nxproc;x++) printf("%ld ",proc[y][x].z);
```

```
    printf("\n");
  }
printf("\n");
}

void printd(z,n)
plural long long z;
int n;
/*****************************************************************
Displays a rectangle of nxproc*nyproc divisions, each containing the
value of z on a processor. The directions x and y are respected.
The upper left angle is processor 0. The variable to be displayed
must be entire (8 to 64 bits). A line is skipped at the beginning and
at the end of the display. The number n specifies the number of digits
to be displayed.
*****************************************************************/
{ int x,y;
printf("\n");
for(y=0;y<nyproc;y++)
  { for(x=0;x<nxproc;x++) printf("%*ld ",n,proc[y][x].z);
   printf("\n");
  }
printf("\n");
}

int conflit(indexe)
plural int indexe;
/*
** This function sends back the number of conflicts for instructions
** of router, rsend and rfetch communications.
** The parallel index variable must be the number of a processor.
** All PE's are not necessarily active.
*/
{ plural int x;
  __routerCount=0;
  x=router[indexe].x;
  return(__routerCount);
}
```

9. References

[Usr] *MasPar Parallel Application Language (MPL), User Guide (version 2.0)*

[Ma] *MasPar Parallel Application Language (MPL), Reference Manual (version 2.0)*

[K&R] *Kernighan & Ritchie*