# Dataflow dot product on networks of heterogeneous digit-serial arithmetic units

Jean Duprat, Mario Fiallos-Aguilar


*Laboratoire de l'Informatique du Parallélisme*

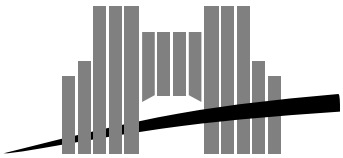
Ecole Normale Supérieure de Lyon
Institut IMAG
Unité de recherche associée au CNRS n°1398

# *Dataflow dot product on networks of heterogeneous digit-serial arithmetic units*

Jean Duprat
Mario Fiallos Aguilar

March 1993

Research Report N$^{O}$ 93-10



**Ecole Normale Supérieure de Lyon**
46, Allée d'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).

# Dataflow dot product on networks of heterogeneous digit-serial arithmetic units

Jean Duprat
Mario Fiallos Aguilar

March 1993

**Abstract**

In this paper we deal with a new high precision computation of the dot product. The key idea is to use hundreds of digit-serial arithmetic units that allow a *massive* digit-level pipelining. Parallel discrete-event simulations performed on a memory-distributed massively parallel computer show that with a limited number of arithmetic units, the computation of dot product when performed using a *"classical"* algorithmic technique (i.e. serial cumulative multiplications) is almost as fast as the case where an *"optimal"* divide-and-conquer algorithmic technique is used. Interconnection networks for both algorithmic techniques are considered.

**Keywords:** Dataflow, dot product, massive pipelining, digit on-line computation.

**Résumé**

Ce document décrit un produit scalaire à haute precision. L'idée principale est d'utiliser plusieurs centaines d'unités arithmétiques permettant le "pipeline" au niveau du chiffre. Des simulations parallèles d'évènements discrets faites sur des machines parallèles à mémoire distribuée montrent que lorsque le produit scalaire est calculé avec un nombre fixe d'unités, un ordononnancement de multiplications cumulatifs est presque aussi rapide qu'un ordononnancement "divide-and-conquer". Les réseaux d'interconnection pour les deux techniques sont aussi presentés.

**Mots-clés:** flot de données, produit scalaire, "pipeline" massive, calcul en-ligne.

# Dataflow dot product on networks of heterogeneous digit-serial arithmetic units*

Jean Duprat and Mario Fiallos Aguilar†
Laboratoire de l'Informatique du Parallélisme (LIP)
Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
mfiallos@lip.ens-lyon.fr duprat@lip.ens-lyon.fr

## 1 Introduction

Matrix and vector operations based on dot product computation occur frequently in engineering and scientific applications. A lot of work has been performed in order to obtain better algorithms and efficient implementations on parallel computers [16], [22], [21] [17], [5], [23].
Unfortunately, in computations of arithmetic algorithms that deal with the approximation of real numbers by floating-point representations, inaccurate calculations and representations lead to completely wrong results.
These errors are produced by *cancellation* and *truncation* of the floating-point numbers [27], [28]. A computer that allows the size of operands and results to be large enough to compute according to the needs of accuracy potentially resolves these problems.
However, as high accuracy is achieved using very-long precision arithmetic, the representation of numbers needs a lot of bits, typically several thousands. It is more practical to carry all these bits serially than in parallel.
In *digit on-line* mode of computation [11], [10], the operands and the results flow through the arithmetic operators or units (aus) serially, digit by digit, starting with the most significant, allowing a digit-level pipelining.
This paper deals with the *digit on-line* mode computation of dot product. Floating-point *digit on-line* adders and multipliers are used to compute with a maximum accuracy of 1024 digits.
Two different algorithmic techniques for computing the dot product are studied. The first one is the "divide-and-conquer" technique, (a technique frequently used in parallel machines). The second consists basically in computing the dot product using cumulative multiplications, a technique frequently used in SISD computers.
A comparison of the two algorithmic techniques is performed using analytical methods and parallel discrete-event simulation on MasPar MP-1.

## 2 On-line and dataflow modes of computation

As stated above in *digit on-line* computation, the operands and the results flow between arithmetic units serially, most significant digit first (*MSD*). Similarly, *LSD* means least significant digit.
A consequence of this flow is the need of a redundant number system[1]. In such systems, addition

*This work is part of a project called CARESSE which is partially supported by the "PRC Architectures Nouvelles de Machines" of the French Ministère de la Recherche et de la Technologie and the Centre National de la Recherche Scientifique.
†Supported by CNPq and Universidade Federal do Ceará, Brazil.

is carry free and can be performed in parallel, or in any serial mode (*MSD* or *LSD*). The most usual arithmetic operations can be calculated in *MSD* mode too. *Digit on-line* arithmetic is the combination of *MSD* and redundant number system.
An interesting implementation of a radix-2 carry-free redundant system is the Borrow Save notation, *BS* for short. In *BS*, the $i^{th}$ digit $x_i$ of a number $x$ is represented by two bits $x_i^+$ and $x_i^-$ with $x_i = x_i^+ - x_i^-$. Then 0 has two representations, (0 0) and (1 1). The digit 1 is represented by (1 0) and the digit $\overline{1}$ is represented by (0 1).
A *BS* floating-point number $x$ with $n$ digits of mantissa and $p$ digits of exponent is represented by $x = m_x 2^{e_x}$, where $m_x = \sum_{i=1}^{n} m_{x_i} 2^{-i}$ and $e_x = \sum_{i=0}^{p-1} e_{x_i} 2^i$. In our system the exponents and the mantissas circulate in *digit on-line* mode, exponent first.
The *digit on-line* systems are characterized by their *delay*, that is the number $\delta$ such that $p$ digits of the result are deduced from $p + \delta$ digits of the input operands. When successive *digit on-line* operations are performed in digit pipelined mode, the resulting delay will be the sum of the individual delays of operations and communications, and the computation of large numerical jobs can be executed in an efficient manner. We will assume that any communication has a delay of 1. See figure 1.

As we can see from figure 1, the computations in *digit on-line* mode can be described as a *data*

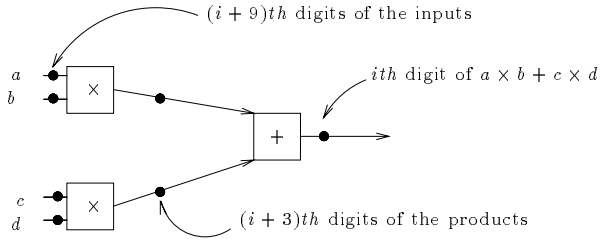



Figure 1: Digit-level pipelining in digit on-line arithmetic

*dependence graph* or *dataflow graph*, *DFG*. These graphs consist of nodes, which indicate operations executed on arithmetic units, and edges from one node to another node, which indicate the flow of data between them. A nodal operation can be executed only when the required information, a digit from all the input edges is received. Typically a nodal operation requires one or two operands and produces one result. Once that the node has been activated and the computations related to the input digits inside the arithmetic unit performed (i. e. the node has fired), the output digit is passed to the destination nodes. This process is repeated until all nodes have been activated and the final result obtained. Of course, more than one node can be *fired* simultaneously.

## 2.1 Pseudo-normalization

In classical binary floating-point representation, a number is said normalized if its mantissa belongs to $[1/2, 1[$ or $]\overline{1}, \overline{1}/2]$. Normalization of numbers leads to more accurate representations and consequently results. In *BS* representation, to check if a number is normalized may be necessary to examine all its digits. For this reason, we replace the concept of normalized numbers by that of pseudo-normalized numbers. A number is said pseudo-normalized if its mantissa belongs to $[1/4, 1[$ or $]\overline{1}, \overline{1}/4]$. It is very easy to ensure that a number is pseudo-normalized: it suffices to forbid a mantissa beginning by 01, $0\overline{1}$, $\overline{1}1$ or $1\overline{1}$. This pseudo-normalization is performed serially.
In the next sections we will describe briefly the two *digit on-line* floating-point arithmetic units (aus) used in the computation of dot product.

# 3 Fully digit on-line floating-point adder

As in classical floating-point adders, the fully *digit on-line* adder (*addflo*, for short)[7], [8] performs three basic operations: exponent calculation, mantissa alignment and mantissa calculation. Figure 2 shows the different blocks of the adder:

- A serial maximizer computes the maximum of the two exponents.
- A serial aligner performs the mantissa alignment with a shift register using the difference between the exponents.
- A serial adder calculates the sum of the aligned mantissas.
- A synchronizer is used to guarantee that only an unavoidable carry appearing in the sum of the mantissas will provoke the truncation of the mantissa and the incrementation of the exponent of the result.

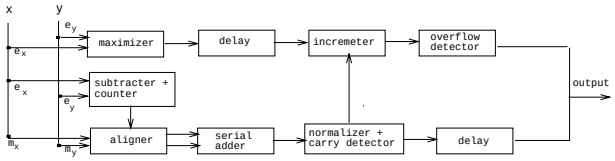



Figure 2: The on-line floating-point adder

The synchronizer (normalizer + carry detector in figure 2) must test the carry digit and the more significant digit of the mantissas sum. When these digits are equal to 1 0, $\overline{1}$ 0, 1 1 or $\overline{1}$ $\overline{1}$ respectively, the incrementation of the exponent and the truncation of the mantissa (the last digit is lost) are performed. When these digits are equal to 0 1, 0 $\overline{1}$, 1 $\overline{1}$ or $\overline{1}$ 1, the exponent is not modified and by substituting 1 $\overline{1}$ by 0 1 or $\overline{1}$ 1 by 0 $\overline{1}$, when necessary, the carry digit of the sum is 0.
The operation performed by the synchronizer is different from the systematic incrementation of the exponent during all addition proposed by Tu [32]. In this last case, the truncation of the mantissa of the result leads to a needless loss of information. In the solution we have proposed [8] the incrementation is performed only when necessary. After the tests of the first two bits of the mantissas sum, the exponent is incremented iff the modified carry digit is not 0. Since the decision of incrementing or not can be made when the last digit of the exponent result is outputting the incrementer (see figure 3), the on-line delay of *addflo*, $\delta_{add}$ becomes 3.
The digit pair 1 1 is used to transmit a 0 and the pair 0 0 for a non significant transmission, so that the synchronization is insured automatically.

# 4 Fully digit on-line floating-point multiplier

The floating point multiplier consists of three different parts[7], [8]:

- A serial adder for the exponents.
- A serial multiplier for the mantissas.
- A synchronizer, which ensures that if two input numbers are pseudo-normalized the output will be pseudo-normalized too.

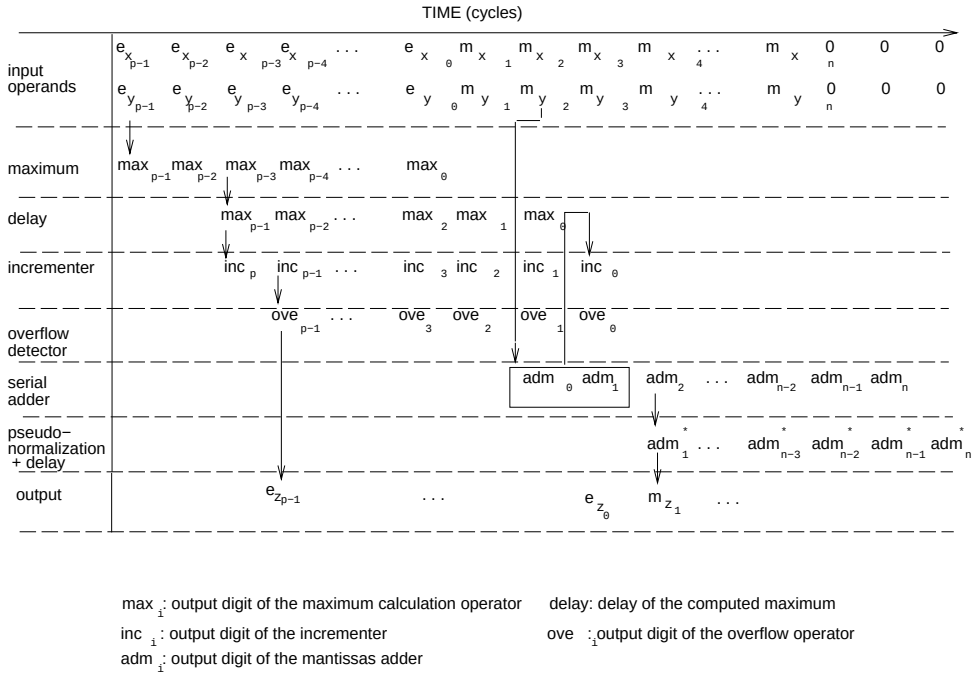



Figure 3: Synchronization in the on-line floating-point adder

The serial multiplier is described in [3], [9], [14], [2]. The serial adder and the pseudo-normalizer, are similar to the floating-point adder ones and will not present here[8], [7].
Since the two input mantissas belong to $[1/4, 1[$ (in absolute value), the product belongs to $[1/16, 1[$ thus, if we want maintain the result pseudo-normalized it is necessary to shift up the mantissa up to 2 positions to the right. This kind of normalization requires the dynamic subtraction of 0, 1, or 2 to the exponent of the result. To generate the final exponent, the last two digits of the exponent are controlled by the three digits of the mantissa product. The normalizer contains a decrementer followed by an overflow detector which is similar to the one of the adder. The *digit on-line* delay of the multiplier, $\delta_{mul}$ is 6. The internal synchronization is similar to the *adfflo* case.

# 5 Computing dot product in digit on-line mode

With the adder and the multiplier presented in the last two sections we envisage computing the dot product of two vectors in a massive digit-pipelined mode.
The dot product $C$ of two vectors, $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_n\}$, is given by:

$$C = \sum_{i=1}^{n} a_i b_i \tag{1}$$

The first *"fast"* mode of computation arises immediately: compute the products in parallel first, and after use adders to perform operations of reduction following a divide-and-conquer technique. See figure 4. The resulting *dataflow* graph is a complete binary tree, $CBT$ for short, with $\lceil log_2 n \rceil + 1$ levels.
We can also try to compute dot product by cumulative multiplications (see figure 4): we compute the first product. This result is added then, to the next using an adder and so on until the final result is reached. The $DFG$ resulting is a linear array of operators ($LA$) [1].

[1] The graph can be defined also as a strictly binary tree with $n$ levels.

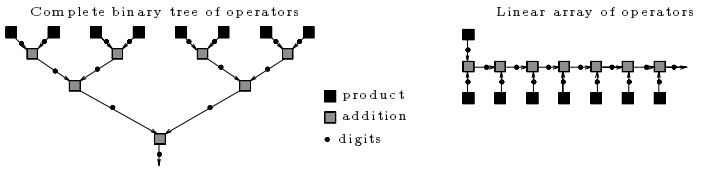



Figure 4: Two resulting graphs for dot product computation

Three cases arise when the dot product is performed in *digit on-line* mode:

- The number of arithmetic units is greater than the number of operations to perform and a minimum delay is obtained.
- The number of arithmetic units is less than the number of operations to perform but, reusing the idle operators, it is possible to compute with minimum delay.
- The number of arithmetic units is less than the number of operations to perform and though the idle operators are reused as soon as possible, it is not possible to compute with minimum delay.

Note that in the last two cases a scheduling policy must be used.

## 5.1 Computing dot products with a number of aus larger or equal than n

With $\delta_{mul} = 6$, $\delta_{add} = 3$ and assuming that any communication has a delay of 1, it is easy to find that the minimum delay for computing a dot product using the divide-and-conquer technique (see fig. 4) is:

$$\delta_{CBT} = \delta_{mul} + (\delta_{add} + 1)\log_2 n = 6 + 4\log_2 n \tag{2}$$

Similarly, the minimum *on-line* delay for the computation of the dot product using cumulative multiplications is:

$$\delta_{CM} = \delta_{mul} + (\delta_{add} + 1)n = 6 + 4n \tag{3}$$

As we stated the two cases above are not *realistics*.

## 5.2 Reusing the aus to compute with minimum delay

The problem of reusing operators is unavoidable in a *real machine* where the number of them cannot be grown indefinitely. However, it is possible to reuse the aus to obtain a minimum delay.
On a not *digit on-line* computer, (i. e. it receives all digits of the operands in parallel), reusing is simple. For the complete binary tree this can be achieved with $n \div 2$ adders and $n$ multipliers.
In a similar way it is possible to reduce the number of operators to compute in *digit on-line* mode. But, here the situation is more complex because as the numbers are transmitted serially, digit by digit, the *predecessor* operators may be computing the last or some middle digits of the result number whose other digits are being consumed by the *successor* nodes, and then the *predecessors* cannot be reused until them have produced the last digit of the result. A similar situation occurs in the $LA$ of operators.
We will present first the case for the complete binary tree graph and after the case for the linear array of operators.

### 5.2.1 Reusing aus in the tree

It is not possible to reach the minimum $\delta_{CBT}$ without using $n$ multipliers. The problem is to know how many adders are necessary to compute with minimum delay having $n$ multipliers. We note $u_i$ as the level on the $CBT$ of operators (the multipliers are at level 1), and $L$, the number of digits used to code the numbers. It is possible to reuse only the first level of adders. The beginning and ending time of adders ($tbeg_a$, $tend_a$) and multipliers ($tbeg_m$, $tend_m$) are:

$$tbeg_m = 0 \tag{4}$$
$$tend_m = \delta_{mul} + L - 1 = 5 + L \tag{5}$$
$$tbeg_a = (u_i - 2)\delta_{add} + \delta_{mul} + u_i - 1 = 4u_i - 1 \tag{6}$$
$$tend_a = tbeg_a + \delta_{add} + L - 1 = 4u_i + 1 + L \tag{7}$$

The adders that have not begun their first digit computation when the ones at the second level of the $CBT$ have finished, are those with:

$$u_j \geq \lceil (11 + L) \div 4) \rceil \tag{8}$$

For example, if $L = 64$ and $n = 524288$, the number of adders *saved* by reusing is only 1!. For $CBT$, searching for the minimum *digit on-line* delay is not realistic.

### 5.2.2 Reusing aus in the linear array

If we enumerate the adders of the $LA$ from the left to right as $a_0, \ldots, a_n$ and the multipliers as $m_0^{'}$ and $m_0, \ldots, m_n$ (see figure 4), we can compute the beginning and ending time of the operators as following:

$$tbeg_{ai} = \delta_{mul} + 1 + (\delta_{add} + 1)i = 7 + 4i \tag{9}$$
$$tend_{ai} = tbeg_{ai} + \delta_{add} + L - 1 = 9 + 4i + L \tag{10}$$
$$tbeg_{mi} = tbeg_{ai} - \delta_{mul} - 1 = 4i \tag{11}$$
$$tend_{mi} = tbeg_{ai} + L - 2 = 5 + 4i + L \tag{12}$$
$$\tag{13}$$

For the first adder, $tend_{a0} = 9 + L$. It can be reused at time $tbeg_{re} = 10 + L$. Then, we look for the adders whose beginning time are greater or equal than $tbeg_{re}$:

$$tbeg_{rea} \geq 10 + L \tag{14}$$
$$7 + 4i \geq 10 + L \Rightarrow i \geq \lceil (3 + L) \div 4 \rceil$$

We note that $i$ is the subindex that identifies each adder. For the other aus the situation is similar. These results indicate that computation of dot product with minimum delay can be performed by using a number of operators which depends only on the length of the format of the number $L$, and not on the number of products, $n$.
Table 1 summarizes the number of operators required to compute with minimum delay as function of the number of digits used to represent the numbers (for adders, $\lceil (3 + L) \div 4 \rceil + 1$ and for multipliers, $\lceil (6 + L) \div 4 \rceil + 1$).

| Number of digits | Adders needed | Multiplier needed |
|---|---|---|
| 64 | 18 | 19 |
| 128 | 34 | 35 |
| 256 | 66 | 67 |
| 512 | 130 | 131 |
| 1024 | 258 | 259 |
| 2048 | 514 | 515 |
| 4096 | 1026 | 1027 |
| 8192 | 2050 | 2051 |
| 16384 | 4098 | 4099 |

Table 1: Number of operators to compute with CM technique with minimum delay

#### 5.2.3 A first comparison between the techniques and networks

The *CM* technique permits the computation of dot product with a number of aus that is independent of the number of products to be performed. This contrasts with the divide-and-conquer technique where the computation depends on both, the length of operands, $L$, and the dimension of the dot product $n$.
It is true that the *digit on-line* delay of the $LA$ is greater than the delay using the divide-and-conquer technique but it is interesting to investigate what will happen in a *real-world* situation, where the number of operators will be limited. In the next sections we will show that the advantages of computing with the divide-and-conquer method may vanish when the number of operators is limited and the number of digits to represent the numbers increases.
Moreover, it is easy to see that as the operators must be reused, the computation of dot products by using cumulative multiplications can be performed on a ring of operators instead of on a $LA$. That is, the output of the last adder of the computation is feedback to one of the inputs of the first adder. See figure 5. From now we suppose that the computation of dot products using cumulative multiplications will be performed on a ring of aus. Note also that the "divide-and-conquer" technique suggests immediately to use a set of $CBT$ interconnected aus.

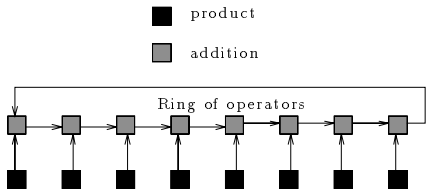



Figure 5: Computing dot-product using a ring of operators

### 5.3 Computing dot product with a greater delay than the minimum

We suppose now that we have less aus than necessary to compute with minimum delay.
To compute the dot-product we adopt a two phases scheduling algorithm. The first phase is the assignment of a priority number for each task. Priorities are in decreasing order. The second phase schedules the tasks according to their priority number and the number of available aus. As there are two types of aus, adders and multipliers, the scheduling can be performed independently and in parallel for each type. We will present this scheduling algorithm with more details in section 7.3.
Let us present first the host computer where the scheduling algorithm and the dot product were performed.

# 6 Parallel simulation and the host machine

We use discrete-event parallel simulation [26], [24], [20], [6], [18], [4], [12], [19],[31], [13], [30], [13] in our work. In the discrete approach to system simulation, state changes are represented by a series of discrete changes or events at specific instants of time. In our case the events are the input and output of digits of the aus.
MasPar MP-1, the host computer of the simulation is a SIMD massively parallel computer [29], [25]. In MP-1 all processors change state in a simple, predictable fashion. The parallelism in MasPar is achieved from the execution of single operation simultaneously across a large set of data. In MP-1, it is easy to determine the program state, because all processes are either active or inactive and the full synchronization guarantees that the value each processor retrieves is correct.
The processors (PEs) are interconnected by an *xnet* toroidal neighborhood mesh and a global multistage crossbar router network. The programming language used is MPL, a superset of C that includes commands for the data-parallel programming mode.
The key idea to simulate several aus on MP-1 is to map to several PEs, several aus processes. It is possible to map several aus of the same or different types to each PE, but all the processors would simultaneously simulate the same type of operator, since MP-1 is a SIMD computer.

# 7 Description of the performed simulation

We have performed both the scheduling algorithm and the computations of dot-product using the parallel facilities of MP-1.
The simulation can be viewed as a finite succession of two different steps: *computation* and *communication*. In fact, due to the data-parallel programming model of MasPar problems of synchronization between the different arithmetic units are easily solved. The computations are performed in one type of operator at a time. Static or dynamic scheduling may be applied to our problem. We use dynamic scheduling. As the *digit on-line* delay of the adders and multipliers is fixed static scheduling seems more natural at first. But, in *digit on-line* mode of computation there are some arithmetic operations as the division that cannot be computed with a constant *digit on-line* delay and consequently the static scheduling cannot be used. However, in both cases of scheduling, the results will be identical.
Some other features of the simulation are:

- The event list is partitioned or distributed on the PEs. In fact, each PE has a variable called (*priority*) that contains its priority relatively to the other tasks of the same type.
- It exist a *global counter* for counting the number of cycles used to perform the computations and *local counters* to describe the state of the operator. The local counters are used to control the computational progress on the node they belong to. The global and local counters always progress forward.
- The time is advanced according to the production of the next event. That is, after one step of computation and communication, the time is incremented in one unit.
- Using the data-parallel paradigm it is guaranteed that the simulated computation time of each node that produces output digit, is less than the virtual or simulated receive time of the node that consumes the output digit.

## 7.1 Simulation of the fully-digit on-line floating-point operators

Each node of the simulated *DFG* performs its discrete-event simulation by repeatedly processing the inputs, performing some computation and outputting its results. In our simulation a *BS digit* is represented by two bits. The floating-point *BS* format chosen may have from 54 to 1014 digits for

the mantissa and from 10 to 16 digits for the exponent. Control of each arithmetic unit process is assumed by a status variable. The process works like a *global* automaton which controls *local* ones ( maximum, overflow detector and pseudo-normalizer, etc) and circuits (serial adder and incrementer, etc)[7].

## 7.2 Mappings

It is necessary to map the tree[15] and the linear array of operators on the mesh of MP-1. The mappings of the tree and of the linear array of processors *DFGs* for a 128-multiplications dot product are shown in figures 6, 7 respectively.

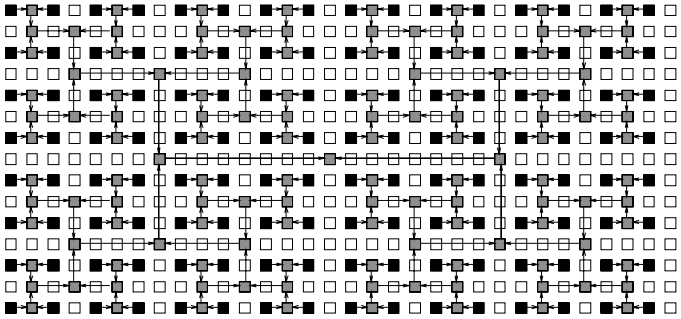


Figure 6: Mapping of a 128-products tree on a mesh

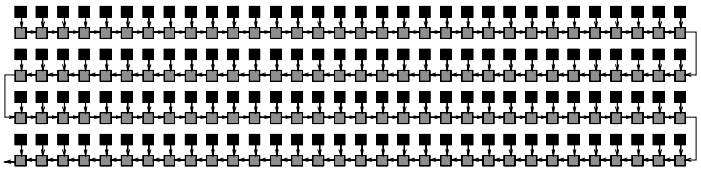


Figure 7: Mapping of a 128-products linear array on a mesh

### 7.2.1 The simulation of the interconnection network

From figures 6 and 7 we see that the communication distances are short. In order to take advantages of this, the networks are simulated using the static mappings.
With this mode of simulation we will activate a number of operators less than or equal to the number of available operators. The result is that the communication distances will be kept short and the computation will be performed fast.
Note also that at this level of abstraction the simulation of the linear array of operators and the ring are equivalent.

## 7.3 The scheduling algorithm and its simulation

From now, the terms task and node will be used as synonymous. A flag will be used to indicate when an operator has been scheduled. Counter $C$ will store the number of iterations of the algorithm. An information table will contain the beginning and ending time of computation for each operator in

the computation ($t_{beg}$, $t_{end}$). If the two predecessors of a node have produced valid digits [2], then we will say that the node is ready.
We present the scheduling algorithm applied to the $CBT$. The case for the $LA$ of operators is similar. The algorithm can be stated as follows:

1. Assigns priorities to the nodes that represent the additions from one side to the other of the $CBT$ beginning at the first level of adders and ending at the level of the root. The node with 0 priority has the highest priority and the node $n-1$ the lowest one. In a similar way, assign priorities to the the nodes that represent the multiplications.
2. Set counter $C$ to zero.
3. As long as there are nodes to be scheduled, do the following:
   (a) For each type of task determine the number of ready nodes. Scheduled the maximum number of ready tasks according to the number of available operators of the type.
   (b) Set in the beginning time, $t_{beg}$, of the arithmetic units selected in the last item, to the value of C. Compute for each node its $t_{end}$ too.
   (c) Wait computations of the cycle to be performed.
   (d) Return to the group of available operators, whose which interval of computation have expired.
   (e) Increment $C$.
4. End.

As one of the inputs of an operator may be delayed in relation to the other, a synchronization must be provided. Latches are used to delay the input that is ready first.
The scheduling algorithm was performed using MPL and the parallel facilities of MP-1. The scheduling were performed in a type of operator at a time, but using data-parallel statements.

# 8 Performance of the techniques and networks

In order to compare the networks we adopt the following measures of performance:

1. Number of cycles means the number of necessary cycles to perform the computations. In fact the number of cycles is equal to the *digit on-line* delay +length of the operands.
2. The speed-up of computing with $n$ operators of each type is defined as the ratio of the number of necessary cycles to compute with 1 operator of each type and the number of necessary cycles to compute with $n$ operators of each type.
3. Efficiency is the ratio of the speed-up and the number of operators used.

Finally, traces show how the utilization of the different operators along the time are.

[2] first outputs differents from 00

## 8.1 Number of cycles to perform computations

The figures 8 and 9, show that when the number of multiplications of the dot product is fixed, the cumulative multiplications technique when performed on the ring has performances comparable to those of the binary tree. When the number of available operators begins to increase, the performance of the binary tree, as can be expected, begins to be better.

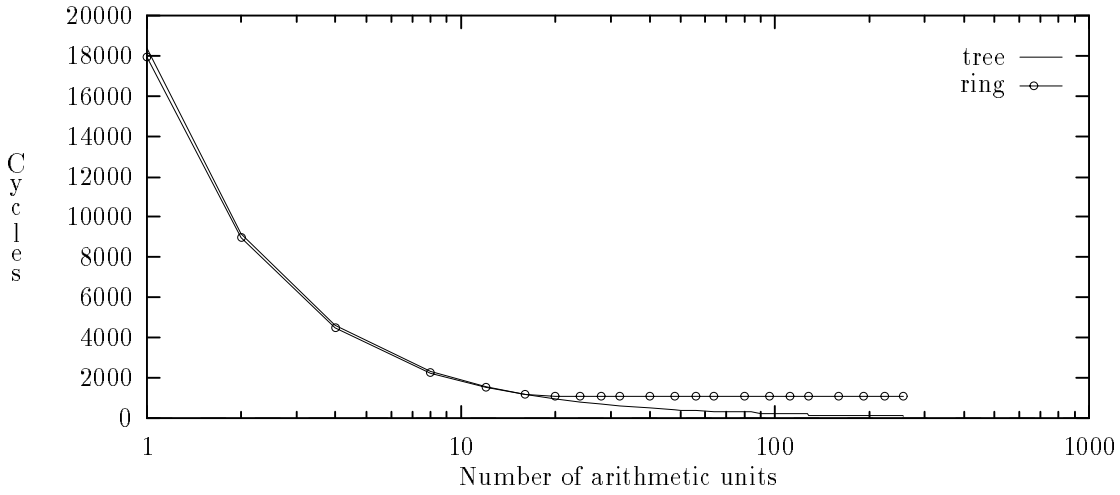



Figure 8: Number of cycles needed to perform a 256-elements dot product with L = 64

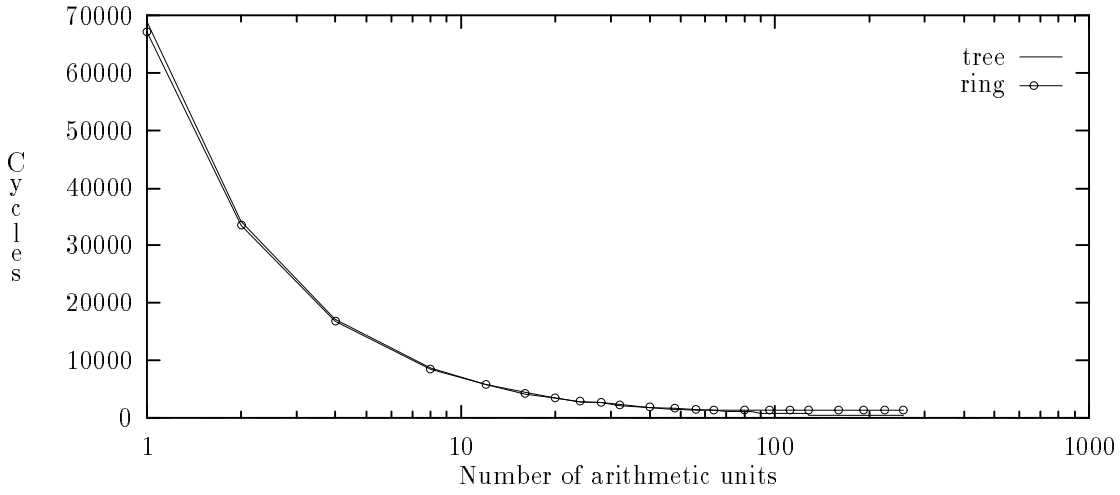



Figure 9: Number of cycles needed to perform a 256-elements dot product with L = 256

## 8.2 Speed-up

The figures 10 and 11 show the speed-up obtained for the tree and the ring respectively. In the tree the speed-up is better when the number of aus is a power of 2. For the ring the speed-up reaches a maximum value relatively fast.

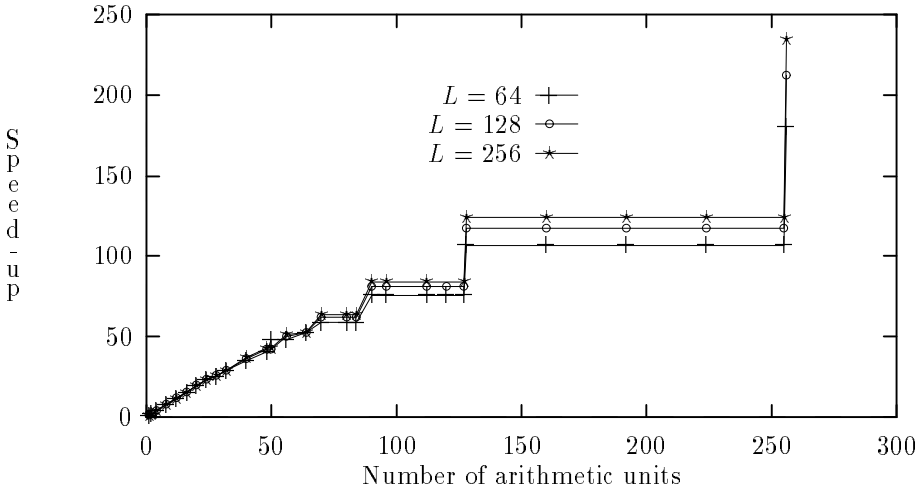



Figure 10: Speed-up on the tree for a 256-element dot product

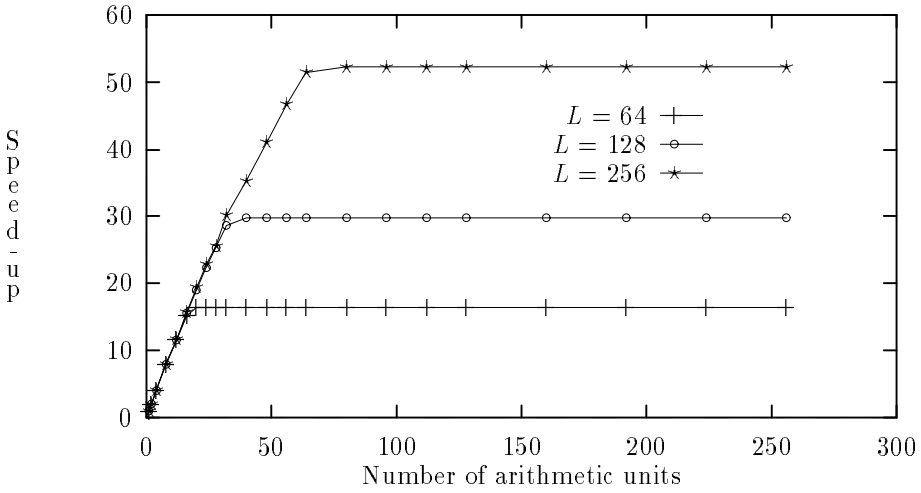



Figure 11: Speed-up on the ring for a 256-element dot product

## 8.3 Efficiency

In the tree (fig. 12) the efficiency is better when the number of aus is a power of 2. The efficiency for the ring is self explanatory (see fig 13).

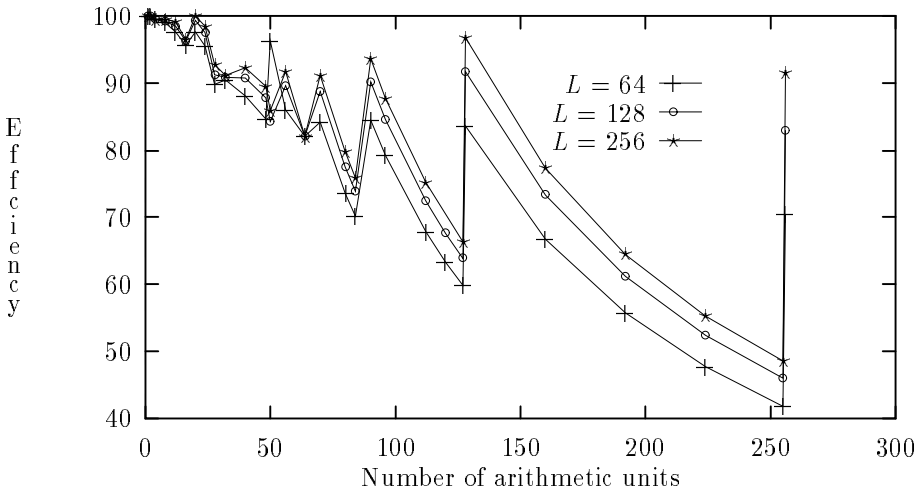



Figure 12: Efficiency on the tree for a 256-element dot product

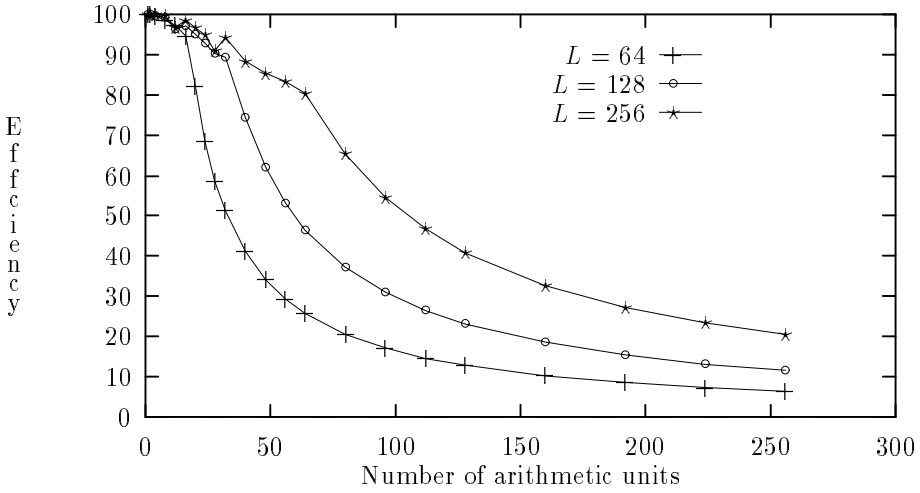



Figure 13: Efficiency on the ring for a 256-element dot product

## 8.4 Traces of adders and multipliers

From figure 15, we see that the peak value of the adders used for the tree is reached a number of times equal to the ratio of $n$ (the dimension of the dot product), per number of available aus. For the ring (figs. 16 and 17) the maximum number of multipliers and adders is reached fast and maintained practically constant until the end of computation.

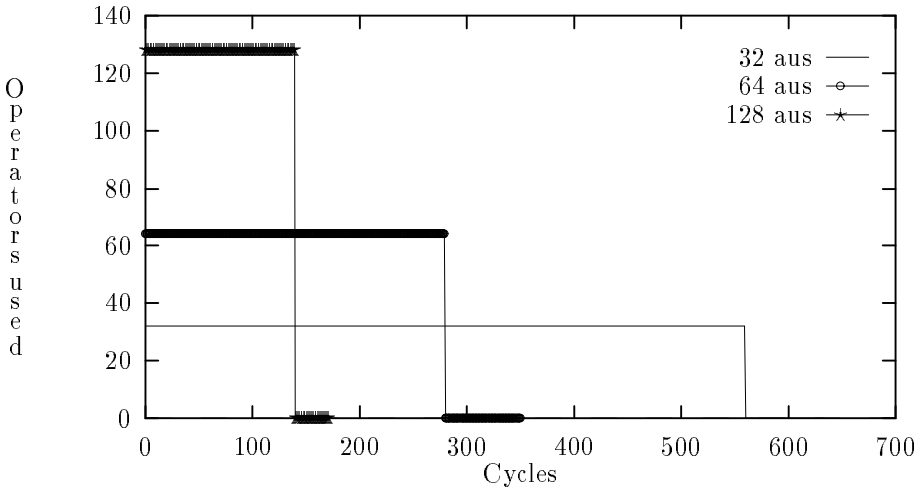



Figure 14: Traces of utilization of multipliers on the tree for a 256-elements dot product with L = 64

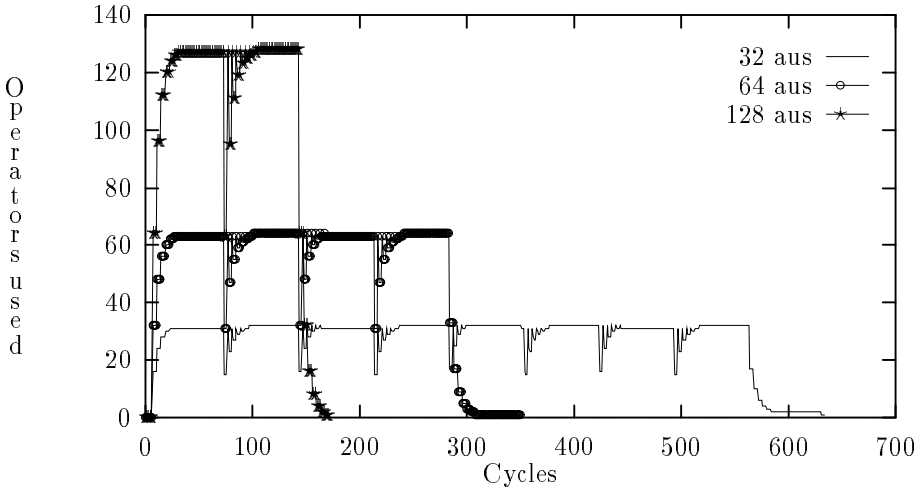



Figure 15: Trace of utilization of adders on the tree for a 256-elements dot product with L= 64

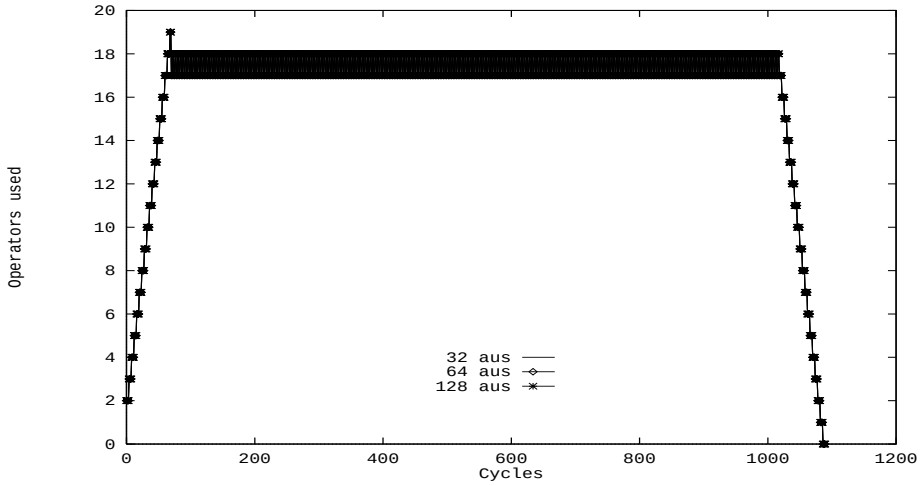



Figure 16: Traces ofutilization of multipliers on the ring for a 256-elements dot product with L = 64

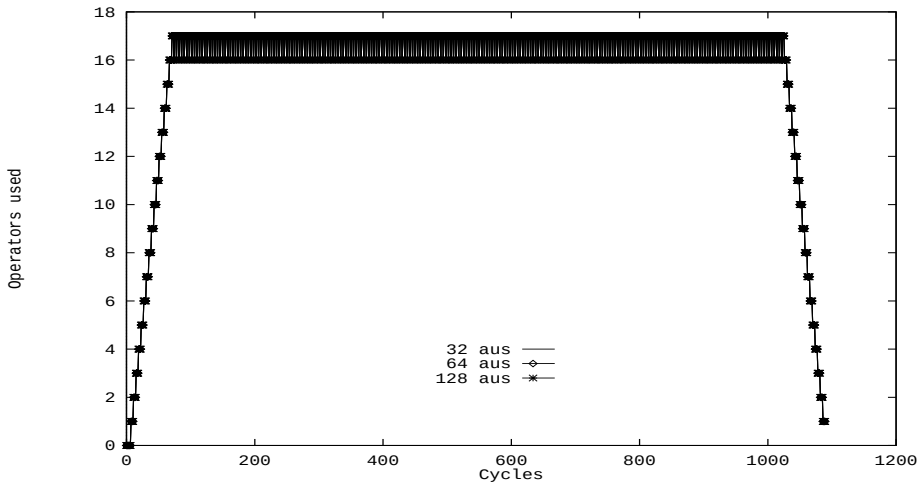



Figure 17: Traces of utilization of adders on the ring for a 256-elements dot product with L = 64

# 9 Concluding remarks and future work

Here we have described a heterogeneous computer made up of *digit on-line* adders and multipliers working on the dot product problem. We have described the simulation of the machine on a massively parallel computer, the MasPar MP-1.

The main conclusion is that, due to the natural pipeline at digit level in *digit on-line* mode, linear arrays have performances very near of binary trees when the dimension of the problem is large compared to the number of arithmetic operators. This phenomena is augmented by the fact that working at the digit level, the dimension of the problem is the product of the number of inputs and the length of the number. Another interesting fact is the ability of a ring of *digit on-line*

arithmetic units that with a reasonable number of them, may perform high precision calculus with large numbers.
Other numerical computations are under study. This includes polynomial evaluation and the Gauss elimination algorithm to solve linear equations.
We are working in a project to simulate and to build a *digit on-line* machine called CARESSE, the french abbreviation of Serial Redundant Scientific Computer, that will made of heterogeneous *digit on-line* arithmetic units. A VLSI prototype of the multiplier has been projected and tested.

# References


[1] A. Avizienis. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, 10:pp 389–400, 1961.

[2] J.C. Bajard. *Evaluation de fonctions dans des Systémes Redondantes d'ecriture des Nombres.* PhD thesis, Ecole Normale Supérieure de Lyon, February 1993.

[3] J.C. Bajard and J.M. Muller. On-line power series. In *International Conference on Signal Processing Applications and Technology, Boston, USA 1992*, 1992.

[4] J. Bezivin and H.Imbert. Adapting a simulation language to a distributed environment. In *3rd International conference on distributed computing system*, pages 596–603. IEEE, 1983.

[5] P. Bjorstad, F. Manne, T. Sorevik, and M. Vajtersic. Efficient matrix multiplication on simd computers. *SIAM Journal of Matrix Anal. Appl.*, 13(1):386–401, January 1992.

[6] E. Debenedictis, S. Ghosh, and M. YU. A novel algorithm for discrete-event simulation. *IEEE Computer*, pages 21–33, 1991.

[7] J. Duprat and M. Fiallos. On the simulation of pipelining of fully digit on-line floating-point adder networks on massively parallel computers. In *Second Joint Conference on Vector and Parallel Processing*, Lecture Notes in Computer Science, pages 707–712. Springer-Verlag, September 1992.

[8] J. Duprat, M. Fiallos, J. M. Muller, and H. J. Yeh. Delays of on-line floating-point operators in borrow save notation. In *Algorithms and Parallel VLSI Architectures II*, pages 273–278. Noth Holland, 1991.

[9] J. Duprat, J. M. Muller, S. Kla, and J.C. Bajard. Some operators for radix 2 on-line computations. *Journal of Parallel and Distributed Computing.* To Appear.

[10] M.D. Ercegovac. On-line arithmetic: an overview. In SPIE, editor, *SPIE, Real Time Signal Processing VII*, pages pp 86–93, 1984.

[11] M.D. Ercegovac and K.S Trivedi. On-line algorithms for division and multiplication. *IEEE Trans. Comp.*, C-26(7):pp 681–687, 1977.

[12] C.T. Evens, M. Gargeya, and T. Leonard. Structure of a distributed simulation system. In *3rd International conference on distributed computing systems*, pages 584–589. IEEE, 1983.

[13] R. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):28–53, October 1990.

[14] A. Guyot, Y. Herreros, and J. M. Muller. Janus, an on-line multiplier/divider for manipulating large numbers. In *9th Symposium on Computer Arithmetic*, pages 106–111. IEEE Computer Society Press, 1989.

[15] E. Horowitz and A. Zorat. The binary tree as an interconnection network: Applications to multiprocessor systems and vlsi. *IEEE Transactions on Computers*, c-30(4):247–253, April 1981.

[16] K. Hwang and Y. Cheng. Partitioned matrix algorithms for vlsi arithmetic systems. *IEEE Transactions on computers*, c-31(12):1215–1224, December 1982.

[17] H. V. Jagadish and T. Kailath. A family of new efficient arrays for matrix multiplication. *IEEE Transactions on computers*, 38(1):149–155, January 1989.

[18] D.R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[19] J. P. Katoen. Simulation of doom, a loosely coupled multiprocessor system. Master's thesis, Computer Science Department of the University of Twente, November 1987.

[20] J. kent Peacecock, j. Wong, and E. Manning. Distributed simulation using a network of processors. *Computer Networks*, 3:44–56, 1979.

[21] C. King, W. Chou, and L. Ni. Pipelined data-parallel algorithms: Part 2 - design. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):487–499, October 1990.

[22] C. King, W. Chou, and L. Ni. Pipelined data-parallel algorithms: Part1 - concept and modeling. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):471–485, October 1990.

[23] Andrewas Knoefl. Fast hardware units for the computation of accurate dot products. In P. Kornerup and D. Matula, editors, *10th Symposium on Computer Arithmetic*. IEEE, IEEE Computer Society Press, June 1991.

[24] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):559–565, July 1978.

[25] MasPar Computer Corporation. *MasPar Parallel Application Language(MPL) - User Guide*, 1991.

[26] J. Misra. Distributed discrete-event simulation. *Computer Surveys*, 18(1):39–65, March 1986.

[27] Jean-Michel Muller and Philippe Francois. Faut-il faire confiance aux ordinateurs? Rapport de recherche, Laboratorire de L'Informatique du Parallélisme, ENSL, 1990.

[28] J.M. Muller. *Arithmetique des Ordinateurs*. Masson, 1989.

[29] J. Nickolls. The design of the maspar mp-1: A cost effective massively parallel computer. In IEEE, editor, *IEEE Compcon Spring 1990*, pages pp 25–28, 1990.

[30] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. Lebecks, J. C. Lewis, and D. A. Wood. The winsconsin wind tunnel: Virtual prototyping of parallel computers. In *1993 ACM SIGMETRICS Conference*, May 1993.

[31] T. R. Stiemerling. *Design and Simulation of an MIMD Shared memory multiprocessor with interleaved instruction streams*. PhD thesis, Department of Computer Science, University of Edinburgh, November 1991.

[32] P. K. Tu. *On-line Arithmetic Algorithms for Efficient Implementation*. PhD thesis, Computer Science Department, UCLA, 1990.