Communication and Performance Trade-offs in a Systolic Machine

Denis Archambaud **Ivan Saraiva Silva** **Pascal Faudemay**

MASI Laboratory - Université Paris 6
4, place Jussieu, 75252 Paris Cedex 05, France
email : {archambaud, dasilva, faudemay}@masi.ibp.fr

Abstract *- This paper presents the trade-offs concerning Input-Output communications at the interface of a SIMD systolic machine, and inside the machine. These problems have been raised by the accelerating board we are currently designing. We discuss about the issues involved by VLSI design: cascadability, scalability, fault tolerance, and feasibility. We present the main features of our architecture, and the performances measured with an actual application which proves that despite of the host computer slowness, the I/O bottleneck is not a penalty for the calculation acceleration.*

Introduction

When implementing an application into a given computer, one of the most important parameter that must be considered is the execution time. A given system can be considered too slow either because the application consists in a real-time processing and the computation cannot stand the data throughput, or because the application requires such an amount of calculation that it would need years, decades or even centuries to run. Automatic image recognition can be classified in the first category, human genetic code alignment in the second.

Evolution in processor design (RISC machines), or technology (clock frequency multiplication allowing up to 500 MHz internal clock) does not speed up machines sufficiently. A well known solution is parallelism, based on the following rule : "That which can be done by one computer, should be done twice faster by two computers".

A parallel architecture is usually made of a large number of processors, the whole being connected to an host (i.e. it behaves as a co-processor). Actually, most of the massively parallel machines built so far have been expensive, complicated and had to face some communication problems. I/O bottleneck appears at the interface because each processor requires typically data, instruction and provides another datum at each cycle.

To avoid the bandwidth to be linearly proportional to the number of processors, some parallel architectures feature simplified interprocessor communications. Systolic design belong to that category. In a systolic architecture, a mesh links each processor to its neighbours, making possible parallel data transfer. Moreover the instruction is the same for all the processors (SIMD organization). Processors are mounted on a pipe-line in which data are shifted in parallel from a processor to another. The idea of data steadily flowing from a processing element (PE) to another explains the biologically-inspired name "systolic".

Whatever the parallel structure may be, its architecture implies some specific I/O bandwidth problem. For instance it appears that a one-dimensional systolic net needs a much lower data throughput than bi-dimensional nets. The way the data are processed inside the systolic net also determines the bandwidth requirements.

The first chapter presents the different systolic architectures with their advantages and disadvantages concerning I/O bandwidth and VLSI feasibility. We then detail in chapter 2 our own architecture and justify our choices. Chapter 3 discuss of the performances of our system and more precisely the efficiency of the interface communication.

1 - Systolic features: problems and trade-offs

1.1 - Definition

In systolic arrays, each processor reads its data input stream from another processor through a mesh (i.e. a local linking bus between two topologicaly neighbouring processors). Only the processors on the edge need data stream from outside (i.e. an host computer). This considerably reduces the I/O requirements of the systolic array and offers a parallel system for which the parallelism factor doesn't affect the I/O bandwidth.

Systolic design basically goes with SIMD organization (single instruction stream, multiple data stream) and mesh topologies. SIMD means only one instruction is broadcast to the processors at each cycle. Therefore, instruction fetching complexity is not multiplied by the parallelism factor. It has the complexity and the I/O requirements of a single processor.

Due to its particular connection features, systolic architecture can only execute (and accelerate) particular algorithms. Problems with a lot of computation on each datum fit well this design as data will pass through all the processors successively. Data dependency is also very important. Communication between neighbouring processors favour the local dependency.

1.2 - Feasibility

Nowadays possibilities in VLSI implementation allows to instanciate multiple processors in a single circuit. Thus, one can build a massively parallel systolic array on a board far less expensive than parallel main-frames as CM5 or so.

To adapt with on-going technologies, the architecture has to be scalable. The PE must be designed in such a way that the number of PEs in a circuit does not matter concerning feasibility. As connections between the PEs are usually negligible regarding space, the number of PEs in a systolic circuit will grow linearly with the integration scale.

Scalability is not sufficient. To be able to build interesting systolic arrays, it is necessary to design cascadable circuits, i.e. circuits that can be interconnected to make one bigger PE matrix. Cascadability means the circuit interface can interconnect other circuits in the same way that each PE is connected to other PEs. This globally implies that the interconnection net can be exhaustively extended beyond the limits of the circuit.

1.3 - Two dimensional nets

In ideal world, the more communication between processors, the more powerful will be the systolic net. Performing 2-D matrix calculation would require 2-D systolic array. However, VLSI implementation and I/O communication bandwidth imply some limitations.

Considering some multi-PEs circuits, the scalability propriety could not be applied to 2-D matrix of PEs because the interface (or more generally the interconnection bandwidth) would be too large and would grow with the circuit complexity. Therefore, designers had to build either mono-circuit systems, or mono-PE circuits (the number of circuits on the board being then the number of PEs). In both cases, high parallelism factor cannot be achieved. The only solution would be to implement a lot of deliberately small PEs such as 1-bit processors [4] but this lead to a particular topic that is outside the framework of this paper.

Some designers decided to implement pseudo-2-D arrays of PEs in a single circuit. For instance, distance calculation between two texts (e.g. ascii strings) can be solved by a 2-D net. Discarding PEs on the North-East and South-West corners does not change the result if we assume limits in the distance (distance score between completely different texts is meaningless). Thus, the array shape can be limited to a trapezoid, however this is still insufficient to achieve high parallelism [9]. Besides, reducing the structure lead to a specific architecture that can only implement a reduced set of algorithms.

When 2-D systolic arrays are applied to matrix computation, each value is coded in a processor, the link between the processors correspond to the data dependency. Matrix multiplication is a basic problem, it is also possible to implement more complex algorithms as the distance calculation between two texts or biological sequences (cf figure 1), or the algebraic path problem [17].

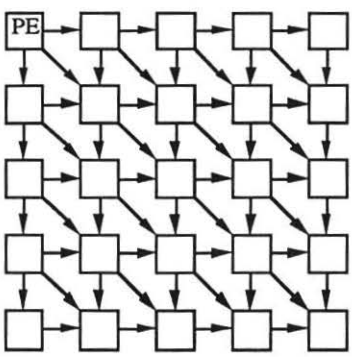

Figure 1 : architecture of a 5x5 2-D systolic net: three meshes are designed (vertical, horizontal and diagonal). Such a structure could be applied to the parallelization of sequence comparison described by Edmiston and Wagner [5]. A host would have to provide 2.N.(N-1)+1 data word at each cycle and probably read-out as much (N is the number of PEs on each side of the net).

1.4 - One dimensional nets

One-dimensional systolic array solves scalability and circuit interface problems. It offers a reduced I/O bandwidth: the interface size of a circuit is equivalent to the one of a PE. Moreover, 2-D systolic algorithms can often be efficiently mapped into a 1-D architecture.

A 1-D design presents scalability and cascadability due to a more simple communication scheme. In addition it supports fault-tolerance. As we deal with scalable design, layout usually reach the technology limitations in size and consumption, therefore the risk of defaults in the silicon circuits is greater than ever. Being able to use some circuits with one or more faulty PE in it gives the possibility to reach an higher parallelism degree for a given price, it has been often applied to massively parallel architecture such as CAM [11] and neural networks [22]. Fault tolerance can be applied to 2-D architectures [15], however it is far more simple to implement in a 1-D organisation.

In a 1-D structure a faulty PE can be discarded from the active PEs by a mere initialisation test. Detection of bad PEs is simply done by running the same test program to all the PEs in parallel (SIMD test). A PE with a wrong result is disabled by resetting a flag that condition its

possibility to interact with the other PEs. While in 2-D topologies adequate routing must be calculated, in a 1-D topology by-passing is sufficient.

This flag orders the mesh to by-pass the faulty PE so that it is completely ignored from the computation: this does not affect the sequencement of the instructions. Most of the program can be made to be independent from the number of active PEs in the array.

Typical applications that can be implemented into a PE row are cartesian products and associated operations such as aggregation, filtering, data retrieval and pattern matching [1]. Some bi-dimensional problems have also been adapted to 1-D systolic net because of their particular data dependency. Distance calculation between two texts as defined by Wagner and Fischer [20] imply local dependency (each matrix element can be calculated according to the values of 3 neighbouring elements). This problem has a lot of applications, from the dictionary approximate-search function [12] to the DNA protein sequences alignment [4]. A way to adapt this 2-D matrix computation in a 1-D systolic machine is explained in [2]. Implementing 2-D matrix-matrix multiplication in a 1-D net has also been studied in [16].

According to the data-dependence of the algorithm, the mesh can be duplicated so that multiple data transfer can be performed at each cycle. For instance Lopresti's machine [10] and Edmiston's one [5] features two meshes to compute some protein similarity scores, because the algorithm requires the calculation of the elements of a 2-D matrix, each value being deduced from 3 neighbouring values (cf Needleman and Wunsch [13], and Smith and Waterman [18] algorithms). Another example is [16] that uses up to 8 meshes with different time-behaviour to perform matrix multiplication.

However, it can become disadvantageous to exaggeratedly duplicate the number of meshes and consequently to increase the size of the PEs. Some of the data-dependency can be solved by programming the PEs and using a unique mesh successively. This is the solution we adopted in our architecture.

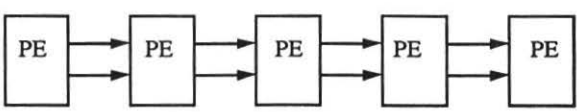

Figure 2: a 5 PEs systolic row. This row is able to perform in one-dimensional fashion the computation that fit in figure 1 two-dimensional array. The row replaces a diagonal in the 2-D array, it supports Smith and Waterman algorithm as explained in [5]. The global interface does not depend on the number of PEs.

1.5 - Instruction flow and format

The way the instructions are sent to the PEs is also characteristic of a systolic system.

The most regular model which is the direct application of the SIMD definition is a unique instruction sent to all the PEs that execute it at the same time. From the program, a PE is "anonymous", it has no address, no particular position, it only processes a data and sends it to its neighbour.

The main disadvantage of the instruction broadcast is that it is a slow way to provide instructions to the PEs because the instruction bus is topologically very long (hence very capacitive) and has to drive a lot of PEs. Bufferization is necessary and requires time. This problem can be solved by sending the instructions to the PEs via a mesh. The global sequencer only feeds the first PE, and then the instruction is successively shifted from a PE to the following one until the end of the row. This still is a SIMD model of execution even though it has a very different behavior considering the connection between data-flow and instruction-flow

[7]. In Rapid-2 we chose to broadcast micro-instructions as it leads to a simple and more powerful programming model.

The instruction format definition also defines a systolic machine. Most of the architectures minimise the instruction bus size, and each PE has to decode the instruction to command its resources. All in all, this is not so advantageous as the decoder is duplicated in each PE, requiring space that could be used to instanciate extra PEs and so increasing the parallelism. It could be of some interest to directly map the PE resources command in the instruction format.

2 - Systolic and associative capabilities

We are currently developing a board made of VLSI circuits and that would be used as an accelerating co-processor for application requiring a massive processing over a somewhat reduced amount of data.

Our architecture (Rapid-2) mixes several ideas and can be defined by the following points :

2.1 - A systolic organization

The PEs are organised into a 1-D systolic row. A mesh links each PE to its neighbours, allowing systolic data shifting in both directions. Three morphologies of the mesh can be defined: a loop (mono-dimensional projection of a torus), a 1 entering line (the end of the line being ignored or readable by the host), or a 0 entering line.

Each PE is basically made of a memory, some auxiliary registers and an ALU (Arithmetic and Logical Unit). The size of the word is 32 bits (plus 6 extra bits used for internal marking facilities). The 32-bit ALU can be devided into 4 parallel 8-bit ALUs, so that the parallelism factor in 32-bit mode is multiplied by 4 when dealing with bytes.

The ALU offers common basic operations (+, -, 1-bit shift, Boolean operations, comparisons) in both 8-bit and 32-bit mode. Multiplication and division are also possible in multiple cycles.

2.2 - A global data-bus

A global data bus makes Rapid-2 pseudo-systolic according to Kung classification [8]. This bus offers important possibilities such as providing a datum to all the PEs in parallel directly form the host. Another possibility is broadcasting [19]: one PE can send its result to some other PEs. This is not in accordance with the SIMD functioning as one PE behaves differently from the others, this will be explained in point 3.

2.3 - A paginated associative organization

The architecture is both systolic and associative. Associativity means that a set of PEs can be accessed based on some assertion (logical condition). This set of PEs can be marked for further processing. One of the marked PEs can also be chosen by an arbitration device to allow the host to read-out a PE content through the data bus. Associativity is an address-less data access method which is different from the systolic method. This combination gives a great flexibility to the architecture. Such combination was also proposed by Herman and Sodini [6].

Associative memories can be full-associative, paginated or set-associative. In paginated and set-associative memories, each PE is connected to a particular register file. A same number register is accessed at any time in all the PEs. In set-associative architectures, the current register number is a hashing value of the evaluated data. Hashing is done by hardware.

Rapid-2 is a paginated and set-associative memory. Each PE is designed to be part of a memory that supports sets and data storage with wired hashing. More than half of the space in the PE is dedicated to memory. The global control yields an address to all the PEs' memory so that a

particular word in each PE can be accessed and processed simultaneously. The basic idea is the CAM (Content Addressable Memory) since a word in a PE can be selected and then processed and/or read-out by the host computer - this is the address-less data retrieval method [20]. However CAM only do comparisons, while an associative architecture as Rapid-2 also executes other calculations. Set-associativity is due to three devices, the token unit, a global data bus, and the PE memory addressing:

The token unit :
Actually a standard systolic machine is somewhat limited by the fact that each datum passing through the systolic row has inevitably to be processed in each PE successively. To give more flexibility in programming a systolic treatment, it is interesting to implement some marking flags (that we call tokens), with the possibility to inhibit the execution of an instruction in the PEs according to the values of the flags. This property is a characteristic of associative architectures.

There are 8 tokens in each Rapid-2 PE :

1 for storing the result of a comparison in 32-bit mode
4 for storing the result of a comparison in 8-bit mode (1 bit per byte)
1 to provide the PE from executing the instructions
1 to indicate which PE is allowed to emit onto the data-bus
1 to indicate whether a PE is valid or not (fault tolerance mechanism)

Memory addressing :
The PE memory can be addressed by a global address unit. It is a single address for multiple data (SAMD). A given instruction addresses the same word in each PE. The memory addressing combined with the token manipulation give a way to read any word in any PE. This is a way to access in non-parallel fashion beyond the possibilities of the systolic functioning. Moreover, the address can be calculated by the PE, using its internal ALU. This is a way to perform indirect addressing and to manage structured data.

2.4 - A VLSI design

We aim at a 256 PEs system. The board would be made of about 12 VLSI CMOS circuits of which one is the controller and the others are execution circuits.

This controller provides the data and the instruction to all the PEs, it is connected to the host (PC or compatible machine) via an EISA bus.

The execution circuits only contain PEs. The fact that the PE layout is symbolic and that the number of PEs in the circuit does not affect its behavior or its interface make the execution circuit scalable. Technologies to come will offer higher integration scale and thus higher performances for future implementation of our architecture. The execution circuit is also cascadable, its interface is composed of the data bus, the instruction bus and the mesh (about 160 pads). Assembling the execution circuit is thus as easy as assembling the PEs.

2.5 - Board programming facilities

Even though the systolic row only need an instruction and a datum per cycle, this is still a constraint that would reduce its performances if the information had to come from the host. The EISA bus is slow compared with the mesh bandwidth, therefore the row would have to wait a part of the time for data from the host, and thus would never reach its maximum calculation power.

We precisely studied the I/O bottleneck between the host and the board (cf chapter 3), and the kind of program that uses such a row. Actually, programs are often based on recursive treatment over data (aggregate, filtering, similarity calculation), therefore there relatively short

and contain very repetitive loops. This propriety lead us to build a storage device into the controller.

However, after further study, it was decided to implement the architecture as a micro-programmed one. Instruction store and micro-instruction store were merged into a single micro-instruction RAM. Therefore the device is programmable at a very low level (registers, multiplexer commands), which offers a high utilisation flexibility. A counterpart is the need for software tools to help developers implement and debug programs.

It is a 128 Kbit ram which is loaded before the beginning of the systolic program, it generates the control flow towards the PEs while the host bus only deal with the data. Synchronisation between data and program execution is ensured by a semaphore system (handshake flags at the interface with the host).

As in classical micro-programmed architecture, the micro-instruction contains a branch code that allows conditional branches and loops with a counter. Finally the control is completely internal and the host only sees a system that reads its data and yields the results. Some words of the ram can even be used to store constant necessary to the program, so that the host has not to send them during computation.

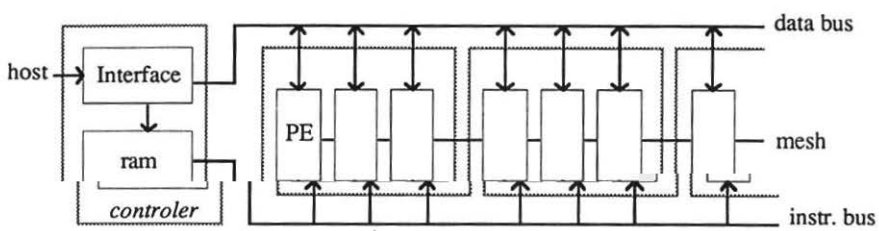

Figure 3 : The Rapid-2 architecture with its communication buses.

3 - Host-computer interface : I/O performances

The communication between the host machine and the board is managed by a data transfer program in the machine and a micro-program in the board. This simultaneous processes require some control mechanisms. This mechanism has to prevent data transfer which speed would be too fast for the board or host possibilities. These problems are part of the I/O performance considerations since a correct speed adaptation between the board and the host ensures an optimised data transfer. We will first present the hardware architecture of the interface implemented in the board, then we will describe how the data transfer program is written in the host. The performances will be also presented.

3.1 - Controller protocol

The communication protocol between the board and the host is based upon some registers and associated semaphores. A semaphore is a 1-bit flag that indicates whether the corresponding register is valid or not. These semaphores can be read by the host in a status register which always valid. The semaphore management rules are as follows:

- If the semaphore is set, the register is valid, it can be read but not over-written.
- If the semaphore is not set, the register is empty, writing-in is allowed, but reading is not.

The semaphores synchronise data transfer between the host and the board by conditioning the execution of transfer programs in the host and the execution of the micro-program in the board. Basically, the semaphores are automatically set by a host transfer, they are reset by the micro-program when it doesn't need the data anymore. The interface registers are:

- **Instruction**: The host writes in this register the instructions which must be executed in the board. It contains a branch address in the micro-program. This register is under

ol of a semaphore.
: The host writes in this register the data that have to be sent to the PEs. This s under the control of a semaphore.
ut: The host reads this register to retrieve some data stored in the PEs. This s under the control of a semaphore.
In: The host reads this register to know the current state of the board. This is always valid, therefore it can be unconditionally read. No semaphore are ed to *Status_In*.
Out: The host writes in this register to change the board state. Since the board n be changed at any time, this register can be unconditionally written. ently there is no semaphore associated to *Status_Out*.

as written a new datum in *Data_In* or *Instruction*, it first has to check the re to get sure it will not overwrite a valid datum. In the same way, when the n in *Data_Out*, it first has to check the *Data_Out* semaphore to get sure of the necking the semaphores is performed by reading register *Status_Out*. These extra data transfer that may be considered as time penalty. However this nous transfers. When the host data transfer program and the micro-program ronous fashion, the semaphore checking procedure can be by-passed.

oint of view, the micro-program execution is conditioned by the semaphores. exists conditional branches that can be used to micro-program loops that wait the host.

ication functions

s board interface registers by calling some functions of a "driver" program. respect the protocol rules as explained before. They read the *Status_Out* e the semaphores and behave depending on their values.

nctions that perform I/O access to the interface with or without semaphore riptive list of these functions is given below:

his function reads registers *Data_Out* or *Status_Out* . When reading *Data_Out*, vant semaphore is checked. If it is set, then the function reads *Data_Out* and into host ram. A TRUE code is then returned. If it is not set, it means the s not yet available, so it is not read and a FALSE code is returned. When *Status_Out* , for which there is no associated semaphore, the word is tionally read and the returned code is always TRUE.
ait: This function performs a task equivalent to the previous one. Moreover, ts to read *Data_Out* until its semaphore is OK (whereas function *Read* gives up first semaphore check). Thus, the returned code is always TRUE since the loops until the operation is successful.
This function allows to write in register *Data_In, Instruction*, or *Status_In*.. aphore is checked when writing in registers *Instruction* and *Data_In*. If the re is not set (i.e. the register is free), the word indicated by the user is written RUE code is returned. If the semaphore is set (i.e. the register still contains a atum), the word is not written and a FALSE code is returned. No semaphore necessary when writing in *Status_In* : the word is unconditionally sent and a de is returned.

- **Write_wait**: This function is equivalent to the *Write* function but loops until the semaphore has turned to a correct value (i.e. the register is free). Therefore the function is always successfully executed, and always returns a TRUE code.
- **Write_over**: This function writes a word in *Instruction* or *Data_In* without checking the relevant semaphore. It is used to force some values when the micro-program is not able to manage the semaphores (when loading the micro-ram for instance). In any case and always return a TRUE code.
- **Read_block_async**: This function reads the successive values in register *Data_Out* or *Status_Out* according to the stream throughput. The number of words to be read before ending the function and the address where to store the words are parameters of the function. When reading *Data_Out*, the relevant semaphore is checked before reading each word.
- **Read_block_sync**: This function allows to read a data stream as *Read_block_async* without checking the semaphores. When using this function, the board has to be faster than the driver function, so that it can provide each word before the driver reads it.
- **Write_block_async**: This function writes the content of an array, word after word, towards *Instruction*, *Data_In*, or *Status_In*. Parameters indicate the number of words to be sent and the array pointer. In relevant cases, the semaphore is checked. When the semaphore has not the correct value, a loop continuously reads it until it turns right.
- **Write_block_sync**: This function writes the content of an array to the interface registers *Instruction*, *Data_In*, or *Status_In*. Semaphore are not checked, hence the board must use the data at least as fast as they are sent by the host.

The data transfer program has to use the fastest driver function in order to achieve the maximum I/O throughput. This means semaphore checking must be ordered only when necessary. A transfer function that checks the semaphore lasts 26 cycles (we assume a 33MHz clock): 8 cycles are spent to read *Status_Out*, 10 cycles are spent to evaluate the needed semaphore and perform the corresponding branch, and the 8 lasts cycles are spent to perform the transfer. For block transfers, 16 extra cycles are necessary to manage the arrays. In addition, each function needs approximately 40 cycles for initialisation and to return to the data transfer program.

We can notice that calling a block transfer function is more convenient that calling successive word transfer functions. Transferring N words (N>1) with *Write_wait*() will costs N.(26+40), while transferring a N words block only costs N.(26+16)+40.

The following section we will apply these consideration to a real-size application (genetic sequence comparisons). We will verify that Rapid-2 communication devices do not limit the parallel processing capabilities of the board.

3.3 - Performances for a real size problem

Genetic sequences can be seen as character strings. Geneticists compare the sequences to find some possible similarities between them and thus classify them. Sequence comparison is basically obtained by calculating a 2-D array [12, 16], nevertheless some implementations fit in a systolic 1-D net [9]. Calculating the similarities between two sequences **A** and **B** is performed in Rapid-2 within three phases: phase P1 during which the micro-program is loaded in the board micro-ram, phase P2 during which sequence **A** is loaded in the PEs, and phase P3 which is the similarity calculation obtained by successive systolic shiftings. The characters of **B** are provided to the PEs row in a systolic fashion, i.e. one character for a shift.

In our implementation, the systolic shift lasts several cycles (it is called a step). One of these cycles is used to transfer the **B** character to the PEs row. As soon as the character has been sent to the PEs, it is of no more use in the interface, the semaphore is immediately reset so that the host will be able to sent a new datum to the interface. More precisely, the new character is needed at the beginning of the step, if it is not yet received the micro-program loops until the host send the datum (this is controlled by the semaphore). The character is then sent to the PEs, and the semaphore is reset so that the host has the rest of the step time to send the following

datum. While waiting for the next data word, the loop let the PEs inactive (since they are waiting for this word). If waiting cycles are spent in each step, this means the board parallelism is used under his possibilities - therefore such a situation must be avoided.

We have calculated the duration of each phase for a 256 PEs configuration (256 32-bit PEs equivalent to 1024 8-bit PEs). This allows to compare in a single pass a 1024 character sequence with an arbitrary length sequence. Results below are given for two 1000 characters sequences.

The phase P1 needs a data transfer to load the micro-program ram. This transfer is performed in the host by a *Write_block_sync* function. The execution time is 0.15 ms. During this phase, the PEs are inactive, this can be considered as an unavoidable initialisation phase.

Phase P2 uses a *Write_block_sync* function to load sequence **A**. We do not need to check the semaphore since the number of cycles necessary to load (with no computation) the sequence is less than the cycles required by the data transfer. The board being always ready to read a new data word, it is not necessary to check the semaphore before sending the words. Execution time for this phase is 0.24 ms, 13.13% of this time being used to calculation in the PEs. This low utilisation rate is due to the fact that the board has to wait for data most of the time.

Phase P3 uses a *Write_block_async* function to yield the **B** characters at a rhythm of 1 per step. We use an asynchronous transfer (i.e. we systematically check the semaphore before writing) because a step lasts longer than a one-word transfer. If the semaphores were not checked, the host could over-write some data words in the interface that have not been yet used by the PEs row. Execution time for this phase is 2.2 ms. 99.51% of this time is spent in calculation in the PEs. This indicates a satisfactory use of the hardware during this phase. Since phase P3 lasts much longer than the others phases (it takes 85% of the overall execution time), it is important that this is the most optimised one.

Actually, geneticists usually compare one sequence with, say, 1000 other sequences from a database. Comparing a sequence **A** with successively 1000 sequences **B** only requires one phase P1, one phase P2 and 1000 phase P3. The execution time is then 0.15+0.24+2200 = 2200.39 ms in which phase P3 (using 99.51% of the board power) represent 99.98% of the overall time. In such a situation, phases P1 and P2 can be ignored as regards to the performances.

Thus, using a slow EISA bus does not slow down our architecture. First tests with real genetic sequences show that Rapid-2 can speed up today's software implementation on work stations of a factor 50 to 200 depending on the algorithm complexity (similarity accuracy needed, mutation matrices...).

Conclusion

We presented the different systolic structures, their advantages and disadvantages. We do think that a 1-D associative systolic array is the architectures that fits the best with a VLSI implementation to build an accelerating board. Moreover, we believe that adding to a classic systolic structure some set-associative features (global data bus, tokens...) increases the possibilities and flexibility of the architecture. Rapid-2, the accelerating board we are currently developing, is based upon those features. Implementing some existing applications that require too much execution time in software showed that the interface between the board and the host does not raise some redhibitory I/O bottleneck.

Acknowledgements

The authors wish to thank Alain Greiner (director of the MASI laboratory), François Dromard, Laurent Winckel, and Jean Penné for their fruitful contribution. The project is partially funded by GdR ANM (Novel Machine Architecture), GdR "Informatique et Génome", and GIP GREG (Groupement de Recherches et d'Etudes sur les Génomes).

References

1. D. Archambaud, P. Faudemay, "Rapid-2, A Paginated Set-Associative Circuit That Performs String Processing", Proceedings of the 1st south american workshop on string processing, pp 1-13, September 1993

2. D. Archambaud, P. Faudemay, A. Greiner, "Rapid-2, An Object Oriented Associative Memory Applicable to Genome Data Processing", Proceedings of the 27th annual Hawaii international conference on system sciences, vol 5, pp 100-110, January 1994

3. R. Barman et al., "Silt: The Bit-Parallel Approach", pp 332-336, IEEE 1990

4. A. F. W. Coulson et al., "Protein and nucleic acid sequence database searching : a suitable case for parallel processing", Computer Journal, vol 30, n°5, pp 420-424, June 1987

5. E. Edmiston and R. A. Wagner, "Parallelization of the Dynamic Programming Algorithm for Comparison of Sequences", Proceeding of the International Conference on Parallel Processing, pp 78-80, 1987

6. F. Herman and C. Sodini, "A Dynamic Associative Processor for Machine Applications", IEEE Micro, June 1992

7. R. Hughey and D.P. Lopresti, "Architecture of a Programmable Systolic Array", Proceedings of the international conference on systolic arrays, may 1988, pp 41-49

8. H. T. Kung, "Why Systolic Architectures", Computer, vol 15, pp 37-46, January 1982

9. D. Lavenier, "An Integrated 2-D Systolic Array for String Comparison", Proceedings of the 1st south american workshop on string processing, pp 117-121, September 1993

10. D. P. Lopresti, "P-NAC: A Systolic Array for Comparing Nucleic Acid Sequences", Computer, vol 20, pp98-99, July 1987

11. A. J. McAuley & C.J. Cotton, "A Self-Testing Reconfigurable CAM", IEEE journal of Solid-State Circuits, vol. 26, N° 3, March 1991

12. M. Motomura et al., "A 1.2-Million Transistor, 33-MHz, 20-b Dictionary Search Processor (DISP) ULSI with a 160-Kb CAM", IEEE journal of Solid-State Circuits, vol. 25, N° 5, October 1990

13. S.B. Needleman & C.D. Wunsch, "A General Method Applicable to the Search for Similarities in the Amino-Acid Sequence of Two Proteins", Journal of Molecular Biology, vol 48, pp 443-453, 1970

14. T. Ogura et al., "A 20-Kbit Associative Memory LSI for Artificial Intelligence Machines", IEEE journal of Solid-State Circuits, vol. 24, N° 4, August 1989

15. S. P. Popli and M. A. Bayoumi, "A Reconfigurable VLSI Array for Reliability and Yield Enhancement", Proceedings of the international conference on systolic arrays, may 1988, pp 631-642

16. V. K. Prasanna Kumar and Yu-Chen Tsai, "Architecture of a Programmable Systolic Array", Proceedings of the international conference on systolic arrays, may 1988, pp 51-60

17. T. Risset, "Applying Semi-Systolic Techniques to SIMD Programming", Proceedigs of the IFIP 1994, Appl. in Parallel and Distributed Computing, pp 103-112, C. Girault editor

18. T. F. Smith & M.S. Waterman, "Identification of Common Molecular Subsequences", Journal of Molecular Biology, n° 147, pp 195-197, 1980

19. Q. F. Stout, "Mesh-Connected Computers with Broadcasting", IEEE trans. on computers, vol C-32, n°9, September 1983

20. R. A. Wagner & M. J. Fischer, "The String-to-String Correction Problem", Journal of the Association for Computing Machinery, vol. 21, N° 1, pp 168-173, January 1974

21. P. Wayner, "Smart Memories", Byte, pp 147-152, March 1991

22. M. Yasunaga et al., "A Self-Learning Digital Neural Network Using Wafer-Scale LSI", IEEE journal of Solid-State Circuits, vol. 28, N° 2, February 1993