

ALFA Fine Grain Dataflow Machine

LORENZO VERDOSCIA

*Istituto per la Ricerca sui Sistemi Informatici Paralleli - CNR
via P. Castellino, 111 80131 Napoli - ITALY*

and

*International Computer Science Institute, 1947 Center Street, Suite 600
Berkeley, CA 94704*

Email : lorenzo@icsi.berkeley.edu

ABSTRACT

This paper presents a new version of ALFA [VER92], a fine grain dataflow machine. This machine uses the static dataflow execution model and is constituted by 128 clusters; each cluster groups 128 identical Functional Units (FU) with homogeneous I/O conditions. Characteristic of this machine is that it is possible to map directly part of dataflow graph programs in hardware because the simple FU design [VER94b]. Since ALFA does not support the conventional processor cycle, at cluster level its behavior is asynchronous and determinate when part of a dataflow graph is mapped and executed on it. After compile time, in ALFA, instructions and data are no longer related, and no control token is generated during the computation, but only data tokens. Consequently, during the execution phase no memory is required to contain the partial results exchanged among FUs of a same cluster. Furthermore, in this paper we explain the reasons and the architectural choices of this static dataflow machine and present also the architecture of the new communication system.

1. Introduction

There are many research and commercial efforts currently underway to build high-performance parallel computers. Approaches that can be classified as general-purpose MIMD systems usually employ interconnections of conventional von Neumann processors. Since the early 1970s the data-flow model of execution has been proposed as an alternative to the conventional von Neumann execution model. In principle the difference lies in what is decisive in the process of computation in the individual models:

- in control flow it is the sequence of instructions,
- in data flow it is the availability of data.

In a conventional control-flow computer, the program is stored in memory as a serial sequence of instructions. The program is executed by fetching successive instructions from memory and executing them in the processor. Thus, the course of computation is given by the sequence of instructions in the program, e. g. by the flow of control. Even if the required operands are available, instructions are not executed until their turn comes in the program. In a data-flow computer, the course of computation is controlled by the flow of data in the program, and it is the presence of operands that activates an instruction. These operands can either be fetched as input data or are the product of preceding instructions in the program. The order of execution of the instructions is determined by the natural structure of the algorithm being executed rather than the strict sequencing of instructions in memory. In this way, the data-flow model of computation exploits in a simple manner the natural parallelism of algorithms, whereas for MIMD machines based on traditional von Neumann processors the automatic detection of adequate parallelism remains a difficult problem, in spite of recent advances in compiler technology. Since the first conceptual designs started by Dennis and Misunas' ideas [DEN75], a number of prototypes have been proposed or have become operational [BIC91, LEE94, HIC93, GUR85]. Recent advances in the design of high speed multi-processor data flow computers have given data flow computation a promising future, by offering possible performance improvement over the conventional control flow multi-processors [CUL90, GRA89, SAK89].

However, studies from past dataflow projects revealed some inefficiencies in dataflow computing [BIC91]. For example, compared to its control flow counterpart, the dataflow model's fine-grained approach to parallelism incurs more overhead in instruction cycle execution. The overhead involved in detecting enabled instructions and

constructing result tokens generally results in poor performance in applications with low degrees of parallelism. Another problem with the dataflow model is its inefficiency to handling data structures (for example, arrays of data). The execution of an instruction involves consuming tokens at the input arcs and generating result tokens at the output arcs. Since tokens represent scalar values, the representation of data structures, as collection of many tokens, poses serious problems. In spite of this shortcomings, we believe that there are at least four reasons to continue the research in this field of data flow:

- 1) the fine grain structure of data flow machines is well suited to the implementation of large, scalable multi-processor systems using VLSI;
- 2) it is possible to realize for data flow graphs a simple control for the thousands of units that constitute these machines;
- 3) memory usage during the computation can be eliminated because there is a natural flow of data through the computing units ;
- 4) the data flow model supports functional languages well; these languages as result of the absence of concepts such as state and variables, enable more elegant and less error prone programming.

These reasons have aroused interest in research communities, at both universities and computer industries, in data flow architecture as an alternative to massively parallel systems where thousands of identical components work together to perform a task.

2. Reasons of a Choice

In principle dataflow architectures can be grouped into static and dynamic ones (recently a new proposal, which presents some advantages of both models, has been presented in [ABR90]) according to their execution model. The static model allows only one token at a time to reside on an arc, while the dynamic one allows essentially unbounded token queues on arcs with no ordering, but each token carries a tag to identify its role in the computation.

As opposed to fine grain computation of a static model, a dynamic one seems better tailored for coarse grain computation [BOH90]. Although several interesting proposals based on dynamic models have been formulated in these last years [CUL90][IAN88][EGA91], it preserves, through the conventional processor cycle, the von Neumann model at some lower level of its implementation, making it very difficult to directly implement dataflow graphs in hardware. Furthermore, because of the coarse grain computation present in the dynamic architectures, the context switching still constitutes a heavy limit in performance [CUL93], even though much has been done in some hybrid architectures [CUL91] to alleviate this problem.

On the contrary, the static model, even though it preserves its dataflow nature, has received several criticisms because of its fine grain computation. An objection is that the overhead of fine grain instruction scheduling prohibits the attainment of acceptable efficiency [GAJ82]. However, in [DEN83][DEN88][VER94b] it has been shown how fine grain instruction scheduling can be done by simple and efficient hardware mechanisms. Another drawback of the model is that since the task switching occurs at the instruction level, it cannot take advantage of the instruction level locality which is present in the programs. This drawback is also arguable. For example, the reason for which 128 FUs have been grouped in a cluster, and two of them are connected to the same cell of the Mail Box System of ALFA [VER92] is just to preserve locality in programs. In this work it has been shown how the communication penalty drastically decreases when sequential threads of code can be incapsulated in a cluster.

Because its fine grain nature, a static dataflow graph is well suited to be implemented directly in VLSI technology allowing a thousand Functional Units to cooperate and efficiently execute threads of a program. This one-to-one mapping between actors of a dataflow graph and FUs of a dataflow engine is possible only if actors of the static dataflow model have homogeneous I/O conditions [VER94]. Homogeneous I/O conditions mean they have one output and two input arcs, and consume and produce only data tokens.

Despite their simplicity, with these actors it is possible to generate determinate graphs, including iterative constructs, where:

- 1) no feedback interpretation is needed to execute a program correctly. This means that no check needs to verify whether the output token of an actor has been consumed, so, only one-way token flow is present;

2) no synchronization mechanism is needed to control the token flow. Thus, the model is completely asynchronous.

The only requirement, to generate these graphs, is that each actor must obey the following rules:

- a) it must have two input arcs and consume two tokens, one for each arc;
- b) it must have one output arc and produce at the most one token;
- c) it must be token-level functional.^a

This result is, in our opinion, of paramount importance because this characterization, besides supplying a very simple control for the data flow, reduces the architecture design complexity. First of all it facilitates enormously the massive employment of the VLSI technology to integrate in a single chip a cluster of hundred identical FUs. Consequently, these FUs can cooperate tightly to execute efficiently threads of a program without using memory to store partial results. Then it allows the compiled program to be split into operation code set and data value set which are logically and physically separated. Consequently, before computation begins no memory location needs to be accessed to execute the program part inside a cluster. In fact, the operation codes enable the necessary hardware of the FUs to execute its elemental operation, mapping thus the graph onto ALFA, while the data values, constituting the tokens, can flow along the FUs in a completely asynchronous mode. Finally it allows a cluster, when a cycle can be fully contained in it, to go in *free oscillation*, once it receives its input values, until the result is produced, if, of course, the cycle converges. In fact, in this case we do not need to resend the operation codes every time a new cycle must be executed, but we allow data values to evolve freely among the FUs involved in the loop computation.

3. Design Considerations

In determining the ALFA architecture we have followed the *language first* approach [KEN84]. Starting from the computational model proposed by Backus [BAC78], a machine has been defined where the data driven execution model is that defined in [VER94]. The computation power of ALFA is made up of thousands of identical FUs. Each FU has specialized hardware and is able to execute any elemental language operation [VER94b]. It has a simple register where the operation code is stored to set and enable part of its hardware, configuring thus the FU to execute the required operation. It is necessary for this configuration to happen in the program execution phase that precedes the crossing token phase in order for the dataflow mechanism to operate correctly. This means that a program is first translated to data flow graph, then mapped onto ALFA to be executed in data flow mode, associating one FU to one actor and carrying out thus the one-to-one correspondence between the execution model and the physical system afore mentioned.

Now we consider the four aspects that drove the ALFA design process: language choice, control philosophy, memory latency, and communication mechanism.

3.1. Language Choice

In the past, a clear distinction has been done between the data driven and demand driven execution models [TRE82]. However, in these years this distinction has become less and less evident. In fact, Davis and Keller [DAV82] observed that data driven execution is like demand driven execution in which all data have already been requested, while Wei and Gaudiot [WEI88] presented a demand driven evaluation system which enables execution in a data driven environment. One of the reasons, in our opinion, that have made these two execution models similar is that they share the same computational model: functional. In fact, the most appropriate programming style for dataflow computers is the functional programming style represented by such languages as Val [ACK79][MCG82], SISAL [MCG85], and Id [NIK86][NIK90]. Furthermore, as Vegdahl observed [VEG84], functions in functional language machines may be evaluated either top-down, where a function is evaluated when requested by another function that requires it as an argument, or bottom-up, where a function is evaluated as soon

^a Token level functionality means that an actor with the same tokens on its incoming arcs will always produce the same token on its outgoing arc, independently from the arrival times of the incoming token and the computation state.

as its arguments are available. The bottom-up approach is known as data driven computation, while the top-down approach is known as demand driven evaluation.

According to these remarks, we propose to call dataflow machines that implement functional computational model *demand-data driven* machines. In fact, in the data driven model the graph construction can be made in *demand* mode, while in the demand driven model, the execution of active instructions, after having reached the atomic operands, can be made in *data* mode.

Consequently it was natural to utilize a pure functional language for the computational model. Moreover since functional languages are referentially transparent [BAC72], programs written in these languages can be considered static objects. This means that an expression in a functional language depends on the meaning of its component subexpressions and not on the history of any computation performed prior to the evaluation of that expression. Notions such as time dependence, side effects and writable memory do not exist.

Among functional languages, the FP style was chosen instead of lambda style (like LISP) because, besides the reasons expressed in [ALD89], with the former it is possible to produce new functions by applying functionals (i.e. combining forms) to functions [WIL82]. In fact combining primitive operators it is possible to change small programs into larger ones using the rule of *metacomposition*. For ALFA the following sub-set of primitive operators constitutes the elementary function set of the language that are implemented directly in hardware:

ADD, SUB, MULT, SEL, EQ, NOEQ, GE, GT, LE, LT, AND, OR, NOT.

3.2. Control Philosophy

Data flow nets are bi-dimensional programs in which there is a dependence between data and operations. They are usually expressed through direct graphs in which nodes or actors represent operations and branches or arcs represent data paths along the graph. As the adopted enabling rule is strict, each node is fired if and only if there is a token on each input arc. Since data flow computation is completely asynchronous *i)* it can happen that along some arc there may be a token queue if the corresponding receiving node does not compute quickly or if the token is not available on the other input arc ; *ii)* the program statements are executed in a non-deterministic manner so no functionality is guaranteed at the global (input/output) level unless proven.

ALFA does not permit any queue but guarantees the global functionality according its execution model. So, if an FU produces a new data, but the previous one is not yet consumed, this new data will substitute the old one. Regarding the functionality, we remember that an FU is the hardware implementation of an actor which obeys the listed rules in the previous section. Because only iterative and conditional structures could compromise the global functionality, in [VER94] it has been proved that dataflow graphs employing actors with homogeneous I/O conditions are well-behaved even though they represent such structures. The model, besides providing a simple synchronization mechanism, suggests how to execute the control in communication among FUs. In ALFA an FU can be fired or not without destroying the graph determinate behavior using only information about data validity. We have augmented the token with the concept of *validity* that denotes whether a data in ALFA is ready to be processed or not. In this mode, we have associated the condition of token presence on a node input arc of the model with the condition of valid data on an FU input channel. So in ALFA a token is constituted by two parts: data value and ready value. When in an FU an operation occurs, if the produced data is useful to the next computation, the validity information, attached to it, will be set. In this mode, according to the model, only the FU that receives a token with the ready value set (valid data) will continue the computation.

An example is represented by the parallel execution of the case construct. When the conditions are analyzed, only one of them may be satisfied. Then, only the FU associated to that condition will produce the output token. Consequently, its result will have attached the label of ready value, and the relative branch can be activated. Because such a value can be coded with only one bit, it constitutes a very simple control that can be implemented directly in hardware with no significant cost. This avoids the creation of algorithms to control the determinate behavior of the graph execution. We point out that no control token is generated during the computation, so, during this phase, in ALFA there will be only data tokens that flow.

3.3. Memory Latency

In a conventional computer the job of a program is to change memory contents. This is done using an instruction set that modifies a data set into another data set, a CPU, and a memory that contains instructions, data, and partial results. Latency is the time that elapses between making a request and receiving the associated response. This communication between memory and CPU, which constitutes the von Neumann bottleneck [BAC78], is one of the reasons for performance degradation in multiprocessor systems [ARV87]. The use of functional languages reduces in part this problem by eliminating variables in the program, which create continual LOAD and STORE operations. Furthermore, the fine grain nature of dataflow is another factor in reducing further the memory latency problem. In fact, in an ideal dataflow machine with an infinite number of completely connected FUs, to execute a program it is sufficient that each FU have one output and two input registers where to store the input and output data and a register where to store the operation code.

Starting from these considerations and because ALFA is constituted by thousands of FUs, we have divided it into k groups of m FUs and carried out a radical choice: we have decided to eliminate conventional memory from each group of FUs. In ALFA each FU has its own memory environment constituted by a set of four registers where to store the operation code and the one output and two input tokens. Because we have reduced a program's instructions to an elemental operation set that configures ALFA's FUs, and this can be made before run time, memory is no longer needed to contain program and partial results. These partial results can thus flow through ALFA's FU, and this migration allows to dataflow paradigm to take place. When a computation is running, each FU executes its operation code on incoming tokens and puts the result on its output. In this way it is possible to eliminate memory references and so the latency problem during an FU group computation.

3.4. Communication Mechanism

Unfortunately, in massively parallel systems, there is another serious bottleneck in addition to von Neumann's: the *communication bottleneck* [BER89]. Independent of the parallel architecture that is considered (MIMD, VLIW, etc.) the communication bottleneck appears evident when we have to manipulate data structures. The access to complex data structures in massively parallel systems introduces inefficiency because of communication rather than "von Neumann". In a large computation system, it is not possible to have data that are all local or locally transmitted unless we have an unlimited machine completely connected. Furthermore, in [REE87] Reed and Grunwald observed that it is very difficult, if not impossible, to find an interconnection network optimal for all applications, but it is possible to choose one that fits well the requirements of certain classes of applications; so, communication problems have required particular attention. In ALFA, to reduce the communication bottleneck effects, a hybrid communication mechanism has been adopted: i) direct communication among FUs in a group, ii) message passing among groups of FUs. This hybrid communication mechanism has outlined the boundary between the part of the machine with and without memory. While the communication among FUs in a group, as observed in the previous point, can happen without using memory, communication among FU groups does need memory.

4. General Architecture

In ALFA the execution of a program passes through three phases. Since these phases require different resources, the adopted policy has been to assign to each phase its own set of resources. The three major components of ALFA are a *host*, a *communication system*, and n FUs which are grouped in k clusters constituted by m interconnected FUs. The cluster can be realized with VLSI technology and it is possible to integrate up to 128 interconnected FUs per chip with parallelism of 40 bit per token. The machine which will be described is shown in Figure 1 and is constituted by 16,384 FUs grouped in 128 clusters with 128 FUs each with parallelism of 32 bit per token.

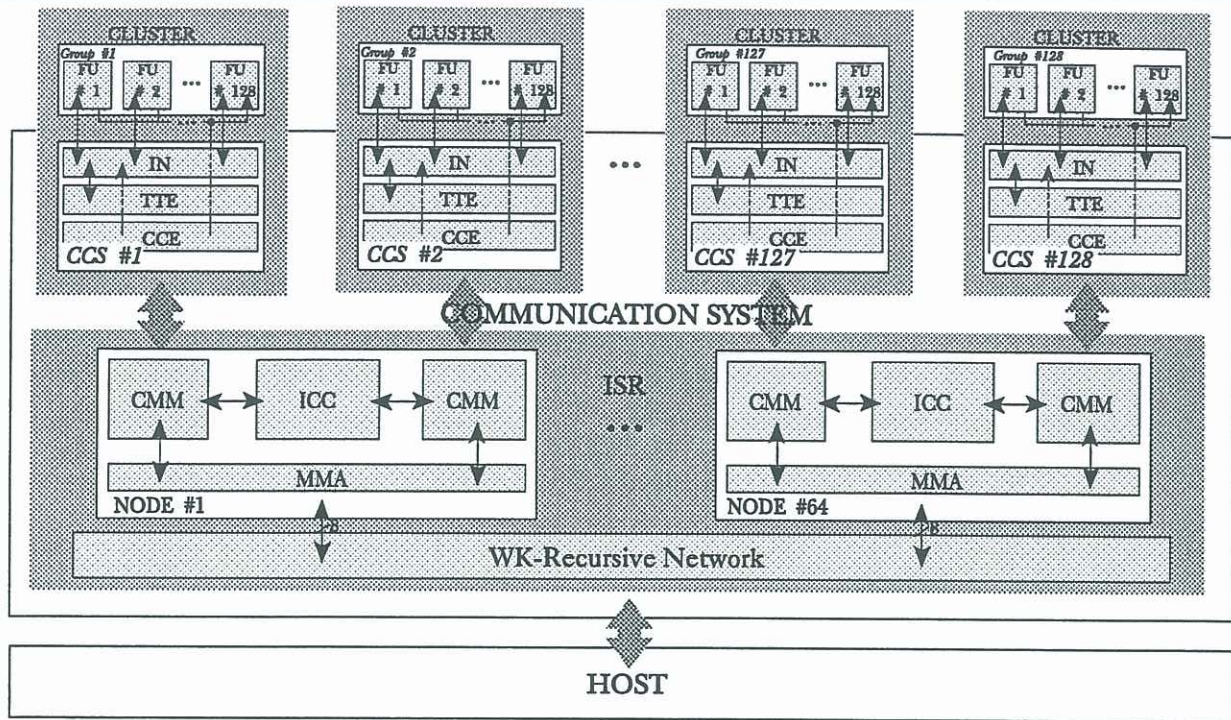


Fig. 1. ALFA Architecture

4.1. Host

The host provides all software support for writing the program and translating it into two tables: the *graph description table* and the *input value table*. The first one contains all the information to configure both the FUs to execute the assigned operation and the communication system to allow the token flow, so that the part of dataflow graph assigned to that cluster can be mapped onto it. Furthermore, this table contains also the information to manage the communication among clusters and between clusters and host. The second one contains the program input data and the information to assign them to the corresponding FUs.

Because the ALFA's FU does not support the conventional processor cycle, after compilation phase, data and instructions do not need more any relation, so that in each instruction there is no reference to any data. The result of this flat division allows activation of the graph description table at a different time from that of the input values table. When the computation starts, the host, on the basis of the graph description table, sends the operation codes to the FUs. It also sends codes and configuration instructions to the cluster interconnection network to map the data flow graph onto FUs. All these operations are executed under the control of a mapper that resides on the host. After mapping the graph onto ALFA, the host sends the input values to activate the computation. When the computation terminates, the host receives the output values from the communication system and through its I/O functions makes the results available.

4.2. Communication System

As the complexity of a computation, in terms of number of actors involved in a graph or amount of data utilized, could require resources bigger than ALFA ones, it is not always possible to map all the dataflow graph onto ALFA completely in one step. Therefore, a trade-off solution has been adopted to satisfy both data and instruction transfer requirements. The resulting communication environment is constituted by three functional levels: *i)* communication with the HOST; *ii)* communication among clusters; *iii)* communication among FUs into a cluster. These three levels of communication are realized utilizing an Intelligent Router System, a Cluster Communication System, and an Interconnection Network System.

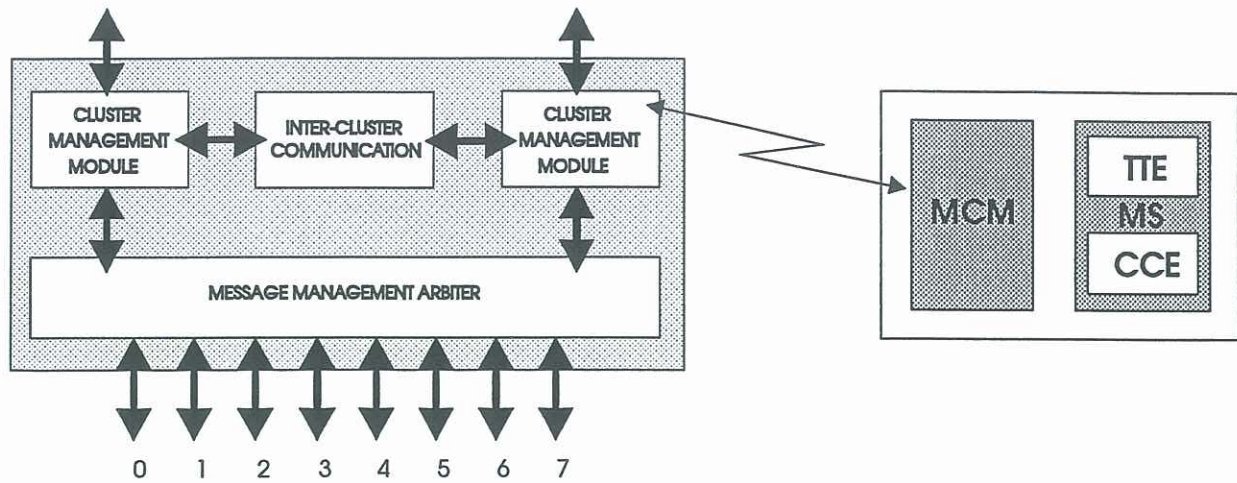


Fig.2. Communication Node Architecture

Intelligent Router System (IRS)

The IRS is an interconnected set of 64 communication nodes. Each node, whose architecture is shown in Figure 2, serves two clusters and is constituted by two Cluster Management Modules (CMM), a Inter-Cluster Communication Module (ICM), and a Message Management Arbiter (MMA). While the ICM and MMA are shared by the two CMMs, each cluster has its own CMM.

The MMA supervises all communication to/from the node. Under its control, incoming messages are evaluated, and a decision is taken to accept or not them. If a message has reached its destination node, the MMA transfers it to the corresponding cluster; otherwise, the MMA, through some of its output links, routes the message forward to its destination. A message coming from a cluster of the node, instead, is directly forwarded to the destination node through an output link.

The CMM is constitute by two parts: Message Control Manager (MCM) and Memory Space (MS). The MCM supervises all the communication operations to/from the cluster. It receives directives from the mapper to manage the part of the graph assigned to its cluster. Consequently, it knows which FUs will receive tokens for that subgraph and when an eventual new configuration can be transmitted to the cluster or accepted from the mapper. Furthermore, under its control messages for other nodes are prepared. If the message is for the other cluster, it is tranfered through the Inter-Cluster Communication.

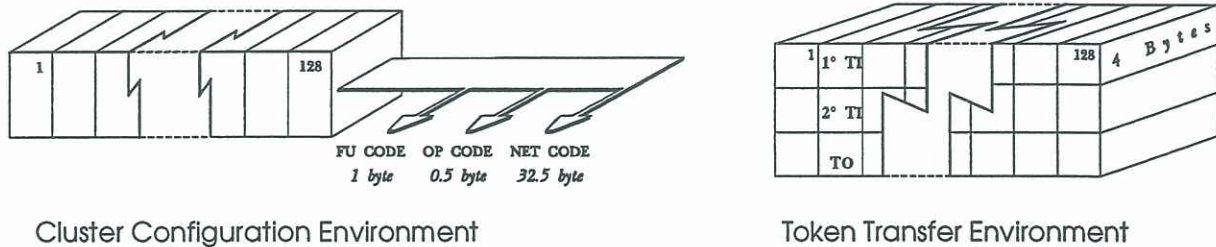


Fig.3. The Memory Space

The MS, shown in Figure 3, contains the part of the graph description table and input value table assigned to the cluster. To avoid any conflict of access on the MS, it is divided into two sections. One constitutes the Cluster Configuration Environment (CCE), the other constitutes the Token Transfer Environment (TTE). In the TTE both input and output tokens have their own space.

When a program must be executed, the host activates the mapper which distributes the graph description table among the CCEs and gives instructions to the MCMs to manage the assigned subgraphs. When a MCM receives its own part of this table, it enables its cluster interface and begins to transfer this information to the corresponding cluster. After finishing this task, it will also send the input tokens if they are available. Once a cluster terminates its computation, it will communicate the resulting tokens to the CMM to be transferred. When the problem size, in terms of elemental operations, is bigger than the machine size, in terms of FUs, the graph description table will be split in subgraphs, and these subgraphs will be mapped onto ALFA. The only difference, in this case, is that partial results of previous subgraph computation could be the token values for the input value table of successive subgraphs.

We point out that, even though we use memory, its amount is just sufficient for storing part of the graph description table and the corresponding tokens. In our case the MS size is only 6 KB. Furthermore, since the CCE can be read from the cluster independently from the TTE writing, these operations can be overlapped to augment the throughput. In fact, having split the memory environment for each FU in parts with different purposes, no conflict can arise when read/write operations occur because each operation happens only in its assigned space. Therefore, no complex context switchings occur, and this reduces enormously cache utilization.

Cluster Communication System (CCS)

The CCS allows communication among the FUs in the cluster. It is constituted by a CCE, a TTE, and an Interconnection Network (IN). While the CCE and the TTE are the image of the homonymous memory residing in the cell, the IN, shown in Figure 4, is a crossbar switch network.

The IN constitutes a mesh with 128×3 rows and three groups of 128 columns. On column links of the first two groups input tokens travel while on the links of the other group the cluster output tokens travel. Dots are permanent connections between the row and column link. Little squares (127 for each group) are the switch elements of the IN. When a cluster receives its assigned part of the graph, it sets each FU involved in the computation to execute the corresponding operation and configure the IN. When tokens arrive, they flow through the FUs and execute, thus, the dataflow graph.

It must be pointed out that another advantage to have two separated and unrelated environments for data and configuration is when same instructions must be executed on different data. In fact, after configuring the cluster, it can be considered as a pipeline stage in which we have spatial parallelism.

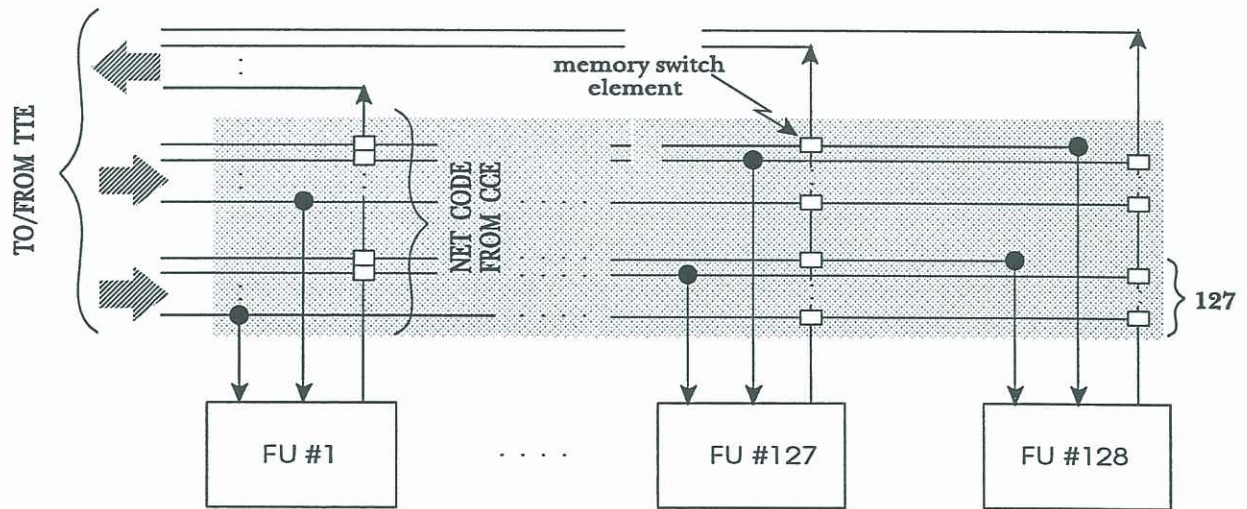


Fig. 4. Interconnection Network inside a Cluster

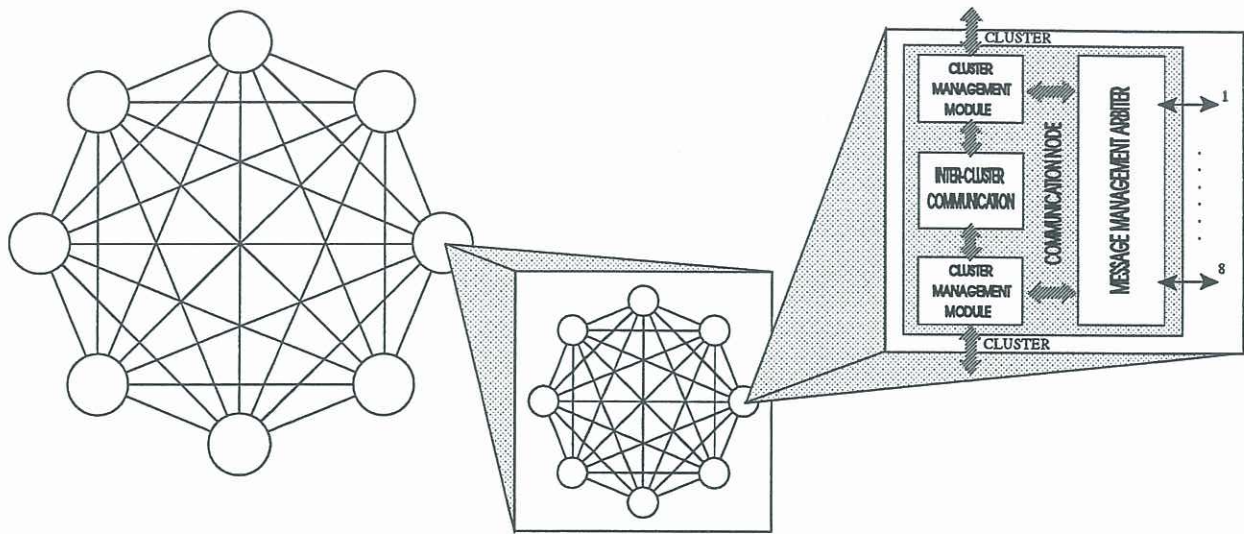


Fig. 5. The ALFA Interconnection Network

Interconnection Network System (INS)

To allow an easy communication policy and a simple scalability of the system a WK-recursive topology [DEL88] with node degree $k=8$ and expansion level $L=2$, shown in Figure 5, has been chosen as interconnection network. The choice takes place from some interesting properties of this topology as, i.e., message broadcasting through the set of shortest paths [FER94] and fault diameter [DUH94] (for fault tolerance), message self-routing (simple control to route messages), invariant bisection width at any expansion level (same connection topology for every virtual node).

5. Related Works

An example of a machine whose architecture constitutes a small grain multiprocessor system and is based on the same computational model is given in [MAG89]. While in the FFP Machine, which has the reduction execution model, the computation takes place in a memory system constituted by a linear array of cells (L cells), ALFA presents a fine grain architecture where the computation takes place in a system constituted by thousands of identical FUs. In ALFA, where the operation assignment is made before run time, the hardware of each FU may configure itself, after receiving its operation code, to execute only the operations that constitute the elementary functional set of the language.

In the static architecture proposed by Dennis in [DEN87] the program instructions are loaded into specific memory locations in the machine before computation begins, and, for each selected instruction, only when the Signal System counter reaches zero, the Signal System sends a fire command to the Execution System so that the instruction is executed. In ALFA, instead, each instruction is before split into operation code and data value, that become logically and physically separated, and then assigned directly to each FU. Furthermore, heterogeneous I/O conditions of the static model actors presented in [DEN80] and [DEN88] constitute not only a performance drawback when dealing with iterative constructs but also an obstacle to the VLSI implementation of FUs in a single chip limiting, thus, the benefits of fine grain computation.

Dynamic architectures, as for example P-Risc [NIK89] and TAM [CUL91], also support fine grain computation. They differ from ALFA because, besides their execution model, they are based on multithreading technique. In the context of multithreading, a thread is a sequence of statically ordered instructions running on a von Neuman machine where once the the first instruction is executed, the remaining instructions execute without

interruption [SCH91]. So, these architectures are hybrid dataflow ones that can be viewed as von Neumann machines extended to support fine-grained interleaving of multiple threads.

6. Conclusions

Until the 1985 the primary goal was to explore the data-flow approach by means of prototypes and simulation models. The data-flow architectures have been implemented with a small number of processing elements, based at the lowest levels on the von Neumann paradigm. More recently the emphasis has been posed to the direct compilation into the silicon of the architectural ideas. So, our approach is based on large, scalable multi-processors using VLSI to implement in fine-grain mode the static data-flow paradigm. The principal features that distinguishes this architecture from the others are the absence of conventional processor and memory in the FUs. Because the homogeneous I/O conditions of the model's actors, ALFA is constituted by thousand identical FUs that allow direct mapping of dataflow graph programs into hardware creating a one-to-one correspondance between actors of the graph and FUs of the machine. FUs do not generate any classical control token during the graph execution, but only data tokens. The machine configuration and the operations that must be executed are defined into the host at compile time and constitute the graph description table. By a simulator implemented on the Ncube parallel machine with 576 nodes at Caltech, it has been observed that the architecture exploits the fine-grain parallelism available in typical numerical algorithms. Currently, it is in development another simulator to evaluate the performance of the Intelligent Router System that constitutes the critical part for the performance of all system.

7. References

- [ABR90] Abramson D. and Egan G., *The RMIT Data Flow Computer: A Hybrid Architecture*, The Computer Journal, Vol. 33, no. 3, 1990, pp.230-240.
- [ACK79] Ackerman W.B. and Dennis J.B. *Val-A value-oriented algorithmic language, preliminary reference manual*. Technical Report MIT/LCS/TR-218, MIT Laboratory for Computer Science, Cambridge, MA, June 1979.
- [ALD89] Alderighi M., Sechi G.R., Vaccaro R., and Verdoscia L. A Computing Unit for FFP Function Evaluation in Support of Correctness Proofs. *Proc. 22nd Annual Workshop on Microprogramming and Microarchitectures*, ACM press, Dublin, August 14-16, 1989, pp.160-162.
- [ARV87] Arvind, and Iannucci R.A. *Two fundamental issues in Multiprocessing*. Memo 226-6, MIT Laboratory for Computer Science, Cambridge, MA, May 1987.
- [BAC72] Backus J.W. *Reduction languages and Variable Free Programming*. Research Report RJ-1010, IBM Yorktown Heights, NY, April 1972.
- [BAC78] Backus J.W. *Can programming be liberated from von Neumann style? A functional style and its algebra of programs*. Communications of the ACM, August 1978, pp. 613-641.
- [BER89] Bertsekas D.P., and Tsitsiklis J.N. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Englewood Cliffs, N.J., 1989.
- [BIC91] Bic L., and Gaudiot J-L. (Eds.). *Advances Topics in Data-Flow Computing*. Prentice-Hall, 1990.
- [BOH90] Böhm A.P.W., and Gurd J.R. *Iterative Instructions in the Manchester Dataflow Computer*. IEEE Trans. Parall. and Distrib. Systems, Vol. 1, Apr. 1990, pp.129-139.
- [CUL90] Culler D. E. and Papadopoulos G. M. *The Explicit Token Store* J. Parallel and Distr. Computing, 10, 4 (Dec. 1990) 289-308
- [CUL91] Culler D.E. et al. Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstracted Machine *Proc. 4th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 1991.
- [CUL93] Culler D.E., Schauser K.E., and von Eicken T. Two Fundamental Limits on Dataflow Multiprocessing *Proc. IFIP Working Group 10.3*, Jan. 1993.
- [DAV82] Davis A.L., and Keller R.M. *Data Flow Program Graph*. IEEE Computer, Vol. 15, Feb. 1982, pp. 26-41.

- [DEL88] Della Vecchia G., and Sanges C. *A Recursively Scalable Network VLSI Implementation*. Future Gener. Comp. System, Oct. 1988, pp. 235-243.
- [DEN80] Dennis J.B. *Data flow supercomputers*, IEEE Computer, November 1980, pp. 48-56.
- [DEN87] Dennis J.B. *Dataflow computation: A case of study*. In Milutinovic V. (Eds.). *Computer Architecture: Concepts and Systems*. Elsevier, 1987.
- [DEN88] Dennis J.B., and Gao G.R. An efficient pipelined dataflow processor architecture. *Proc. of Supercomputing '88*, IEEE Computer Society Press, November 1988, pp. 368-373.
- [DEN83] Dennis J.B., Lim W.L-P., and Ackerman W.B. The MIT data flow engineering model. In *Information Processing*. IFIP 1983, pp. 553-560.
- [DEN75] Dennis J.B., and Misunas D.P. A Preliminary Architecture for a Basic Data Flow Processor. *Proc. 2nd Annual Symposium on Computer Architecture*, New York, 1975.
- [DUH94] D.R. and Chen G.H. *Topological Properties of WK-Recursive Networks*. JPDC n.23, 1994, pp.468-474.
- [EGA91] Egan G.K., Webb N.J., and B hm W., Some Architectural Features of the CSIRAC II Data-Flow Computer. In *Advanced Topics in Data-Flow Computing*, ed. by J.L. Gaudiot and L. Bic. Prentice Hall, 1991.
- [FER94] Fernandes R., Friesen D.K., and Kanevsky A. Efficient Routing and Broadcasting in Recursive Interconnection Networks. *Proc 23rd Conference on Parallel Processing*, 1994.
- [GAJ82] Gajski D., Padua D.A., Kuck D.J., and Kuhn R.H. *A second opinion on data-flow machines and languages*. IEEE Computer, February 1982, pp. 58-69.
- [GRA89] Grafe V. G., Davidson G. S., Hoch J. E. and Holmes V. P. The Epsilon data-flow processor *Proc. 16th Ann. Symp. Computer Architecture*, 1989, 36-45
- [GUR85] Gurd J. R., Watson I. and Kirkham C. C. The Manchester prototype data-flow computer *Commun. ACM*, 28, 1 (Jan. 1985) 34-52
- [HIC93] Hicks J., Chiou D., Ang B.S., and Arvind *Performance Studies of Id on the Monsoon Dataflow System*. JPDC. Special Issue on Dataflow and Multithreaded Architectures. Vol 18, July 1993, pp273-300.
- [IAN88] Iannucci R.A. Towards a Dataflow/von Neuman Hybrid Architecture. *Proc 15th Int'l Symp. Computer Architecture*, IEEE Computer Society Press, 1988, pp. 131-140.
- [KEN84] Kennaway J.R., and Sleep M.R. The Language First Approach. In Chambers F.B., Duce D.A., and Jones G.P. (Eds.). *Distributed Computing*. ACADEMIC PRESS, 1984, pp. 111-123.
- [LEE94] Lee B., and Hurson A.R. *Dataflow Architectures and Multithreading*. IEEE Computer, 27, Aug. 1994, pp. 27-39.
- [MAG89] Mago' G., and Stanat D.F. The FFP Machine. In Milutinovic V.M. (Ed.). *High-Level Language Computer Architecture*. Computer Science Press, 1989, pp. 430-468.
- [MCG82] McGraw J.R. *The Val language: Description and analysis*. ACM Trans. Programming Languages and Systems, January 1982, pp. 44-82.
- [MCG85] McGraw J.R., et al. *SISAL: Streams and iterations in a single assignment language: Language reference manual, version 1.2*. Technical Report TR M-146, University of California, Lawrence Livermore Laboratory, March 1985.
- [NIK90] Nikhil R.S., and Arvind *Id: A language with implicit parallelism*. Technical Report CSG 305, MIT Laboratory for Computer Science, Cambridge, MA, 1990.
- [NIK86] Nikhil R.S., Pingali K., and Arvind *Id nouveau*. Technical Report CSG 265, MIT Laboratory for Computer Science, Cambridge, MA, July 1986.
- [REE87] Reed D.A., and Grunwald D.C. *The Performance of Multicomputer Interconnection Networks*. IEEE Computer, June 1987, pp. 63-73.
- [SAK89] Sakai S., Yamaguchi Y., Hiraki K., Kodama Y. and Yuba T. An architecture of a data-flow single chip processor *Proc. 16th Ann. Symp. Computer Architecture*, 1989, 46-53
- [SCH91] Schauser K.E. et al. Compiler-Controlled Multithreading for Lenient Parallel Languages *Proc. 5th ACM Conf. Functional Programming Languages and Computer Architecture*, ACM, New York, 1991, pp.50-72.
- [TRE82] Treleaven P., Brownbridge D.R., and Hopkins R.P. *Data-Driven and Demand-Driven Computer Architectures*. ACM Computing Surveys, March 1982.

- [VEG84] Vegdahl S.R. *A Survey of proposed Architectures for the Execution of Functional Languages*. IEEE Trans. Comput., December 1984, pp 1050-1071.
- [VER92] Verdoscia L., and Vaccaro R. ALFA: a Static Dataflow Architecture. *Proc. 4th Symp. on the Frontiers of Massively Parallel Computation*. IEEE Computer Society Press, Mc Lean, Virginia, October 19-21, 1992, pp. 318-325.
- [VER94] L. Verdoscia and R. Vaccaro, Conditional and Iterative Structures Using a Homogeneous Static Dataflow Graph Model in *Proc. of 1st Int. Conf. on Massively Parallel Computing Systems '94*, Ischia, Italy, May 1-6, 1994, IEEE Computer Society Press.
- [VER94b] L. Verdoscia and R. Vaccaro. Actor Hardware Design for a Static Dataflow Model *2nd Int'l Workshop on Massive Parallelism: Hardware, Software, and Applications*, October 1994.
- [WEI88] Wei Y., and Gandiot J. *Demand-Driven Interpretation of FP Programs on a Data-Flow Multiprocessor*. IEEE Trans. on Comput., August 1988, pp. 946-966.
- [WIL82] Williams J.H. Notes on the FP style of Functional Programming. In Darlington J., Henderson P., and Turner D.A. (Eds.). *Functional Programming and its applications: an advanced course*. Cambridge University Press, 1982, pp 73-101.