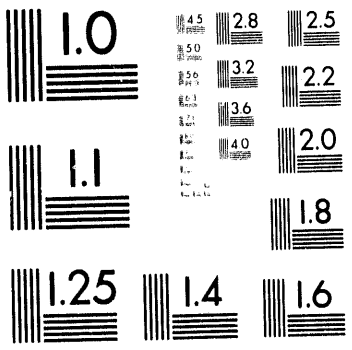
1.0
4.5
2.8
2.5
5.0
5.6
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

1 of 1

~~[illegible]~~
CONF-8912178--1

Proceedings of IFIP WG 10.3
Working Conference on
Decentralized Systems
Dec. 11-13, 1989
Lyon, France

PERFORMANCE OF ASYNCHRONOUS ALGORITHMS IN MULTI-LEVEL DATA-DRIVEN SYSTEMS *

Jean-Luc Gaudiot and Chih-Ming Lin

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California

ABSTRACT: Asynchronous algorithms are efficient methods in solving scientific and engineering problems. Much research has been devoted to the study of asynchronous algorithms in different areas. This paper will show asynchronous algorithms applied to logic circuit simulation, communication networks, partial differential equations (PDE) and artificial neural networks, and as well as implementations of these asynchronous algorithms on a special class of multiprocessor systems, namely Multi-level Tagged-token Data-flow (MTD) architectures.

1 Introduction

The data-flow principles of execution [1] offer the programmability needed to synchronize at runtime the many parallel processes simultaneously active in a large scale multiprocessor. Instead of relying on the conventional central program counter, the availability of data renders an instruction executable. However, in spite of the simplicity of these principles, much overhead may be introduced in order to respect the functionality of execution. A Multi-level Tagged-token Data-flow (MTD) architecture has been proposed [9] in order to reduce the execution time by exploiting the available parallelism while keeping communication overhead at a minimum.

The asynchronous behavior of data in a chaotic algorithm is very complex. Generally, in the asynchronous approach, communications between processes are achieved by reading the dynamically updated variables, while each process continues the execution for the updating of the common variables. It is thus very difficult to implement asynchronous algorithms in a data-driven system. Hence, we must introduce some special high-level language program constructs. A specific firing rule in the Matching Stores of processors will also be introduced in order to execute efficiently asynchronous computations in a data-flow graph. In this work, we will conduct a deterministic simulation of a MTD system and show how the actors can be easily formed and thus deliver a high speed-up in the presence of large degrees of parallelism. Performance of the simulated architecture will be evaluated and the influence of partitioning in program graphs will

*This material is based upon work supported in part by the U.S. Department of Energy, Department of Energy Research, under Grant No. DE-FG03-87ER25043 be analyzed.

The goal of this paper is to demonstrate how such different implementations of the different asynchronous algorithms can be executed on a data-driven machine. This includes the expression in a high-level language, the construction of the data-flow graphs, as well as the performance of the simulated MTD system. In section 2, the elementary data-flow principles of execution is shown. The high-level language constructs and new execution mechanism are introduced in section 3. Several asynchronous algorithms along with their data-flow implementation are discussed in section 4. While section 5 presents the results of a deterministic simulation of the MTD system, performance observations are analyzed in section 6, and concluding remarks are made in section 7.

2 Data-Driven Principles

In this section, we introduce the data-flow principles of execution which are used to implement asynchronous algorithms.

2.1 Data-flow Principles

Programmability is a major issue in the design of large scale multiprocessor systems. Programmers cannot be expected to be able to schedule and synchronize the hundreds or thousands of tasks that are required to utilize fully the resources of such machines. The data-flow model of computation has been introduced to alleviate this problem. Data-flow principles offer the runtime synchronization of operations based on their data dependencies.

2.1.1 Interpretation Models

Once a data-flow graph has been constructed and allocated, the issue arises of how to actually execute the graph. Although the graph defines the operations to be performed and how they are related to one another, it contains no information about arc capacity, precise firing rules, actor execution order, token consumption order, simultaneous actor execution order, etc. Several approaches have thus been put forward regarding the interpretation of data-flow graphs. These include the Acknowledgment Scheme, the Queueing Interpreter, and the Unraveling Interpreter. Under the Acknowledgment Scheme [5], an actor can fire not only when its input arguments are ready but also when its output arcs are empty. Alternatively, in the U-interpreter [2] (a dialect of which we will exclusively use in this paper), tokens are tagged with information pertaining to their context of creation. Only two tokens with identical tags (also called color) can activate an actor.

2.1.2 Multi-level Data-flow Systems

In spite of the simplicity of the execution principles, the fine grain data-flow model encounters much overhead in order to respect functionality of execution. The overhead in token handling, poor performance for low levels of parallelism, and inefficient storage and array handling are important problems in a fine grain computation model. The multi-level data-flow architecture [9], based on the concept of *macro-actors* (several operations are grouped into a single actor) as described by Gaudiot and Ercegovac [7], is shown to bring a solution. Indeed, with actors of various sizes, the amount of non-compute operations and the cost of communication can be significantly reduced. However, one should also note that the increasing size of an actor (with larger granularity) may reduce the parallelism available in a program.

3 High-Level Language Constructs and Execution Mechanism for Asynchronous algorithms

In order to efficiently implement asynchronous algorithms on a data-flow multiprocessor, we describe new constructs in high-level language and a new *firing* rule.

3.1 High-Level Language Constructs

We have chosen "SISAL" (Streams and Iterations in a Single Assignment Language) [11], a high-level applicative language developed at the Lawrence Livermore National Laboratory in cooperation of other institutions (Colorado State University, University of East Anglia, University of Manchester, Digital Equipment Corporation) for our high-level language. New synchronization constructs will have to be added to SISAL in order to describe chaotic behavior.

3.1.1 Asynchronous Constructs

Indeed, asynchronous computations cannot be easily implemented by traditional constructs. Therefore, we designed the **async-repeat** and the **async-for** operations. The new construct of *async-repeat* allows its inside procedures to be concurrently evaluated without any constraints of synchronization between each other. While the construct of *for* in SISAL allows every index value to be synchronously executed in parallel, the *async-for* construct releases the synchronization between each index value and allows independent execution of every index value in parallel.

Chaotic relaxation, for example, can be expressed in SISAL by using the new constructs. In Fig. 1 , it indicates that under the *async-repeat* construct, both the relaxation procedure and the termination check can be executed concurrently without any dependency between each other. Furthermore, inside the *async-for* construct, each index value can concurrently proceed with the execution of the next computation without waiting for other index values which are executing the same function.

3.2 A New Matching Store with Locks

In order to guarantee the proper asynchronous behavior of asynchronous algorithms, we introduce the notion of **locks** at the inputs of the actors. In other words, we create *locks* inside the matching store for the firing of an actor. Note that the implementation of *locks* in actors corresponds to the *Async-repeat* and *Async-for* constructs in a high level language. The *locks* will be attached to the input actors of a subgraph which represents the processes that can be executed asynchronously under the *Async-repeat* and *Async-for* constructs.

Under the new firing rule, when an actor is fired, the input tokens remain in the input lock until the next input token is received. In this fashion, the incoming token will replace the stored value and will activate the actor once more. Fig. 2 shows the step-by-step operation of the new firing rule of an actor:

1. Initially, when either token A or token B comes into the actor F, it is *locked* inside the actor.

2. When another token arrives on the opposite arc, actor F is fired and produces an output token.

3. After firing actor F, both input tokens remain *locked* inside the actor. In other words, the rules of execution are *non-swallowing.*

4. When another token is later received by the actor, the actor is fired with the *locked* token on the other port and the new value on the first port. The incoming token will remain locked in the actor. Note that it overwrites the previous token value.

4 Asynchronous Algorithms In MTD System

The **async-repeat** and **async-for** constructs implemented with the *Matching Store with Locks* scheme can actually be used to express various asynchronous algorithm on our MTD system. In the followings, we describe the data-flow graphs for several asynchronous algorithms from different application areas.

4.1 Asynchronous Digital Systems Simulations

A logic digital system can be either synchronous or asynchronous. If it is synchronous, every logic gate is synchronized by the system clock and execute the logic functions step by step. If it is asynchronous, there may be no system clock or several different clocks for different logic gates and the arrival time of a signal to a gate is unpredictable. To design an asynchronous digital system, one must ensure that the system is *hazard-free.* In other words, the functions must be exactly as expected by the user. A detailed discussion of the merits of asynchronous circuits can be found in [6].

General speaking, it is difficult to simulate an asynchronous logic system on a conventional single processor because the program counter will implicitly synchronize the processes and force the whole execution in a deterministic manner. In a data-flow computer, the *Matching Store with Locks* scheme allows an actor to be executed when a token arrives. For example, a hazard-free logic circuit is shown in Fig. 3. The simulation of this circuit can therefore be translated into the data-flow graph shown in Fig. 4.

4.2 Asynchronous Shortest Path Algorithms

Finding the shortest path from one node to another, which can be solved by the well-known Dijkstra algorithm in a conventional computer, is an important issue in communication network systems.

In [3], a distributed asynchronous version of the Bellman-Ford algorithm is described to find the shortest distance from one node to another. According to the description of the algorithm, each node will keep calculating the possible shortest distance until the whole system reaches a stable state.

In a data-driven approach, the data-flow principles and the *Matching Store with Locks* scheme can offer the algorithm suitable environment. In the data-flow graph, every actor represents a node of the actual graph and executes the computation whenever a new data token arrives. An example of the graph is shown in Fig. 5 with the distances between each node and we try to find the shortest path from node 3 to 12. The corresponding data-flow graph is shown in Fig. 6

4.3 Chaotic Relaxation for PDE

Solving PDE is one of the most important research issues in scientific computing. A PDE can be solved numerically either by direct methods (*e.g.* Gaussian elimination) or by iterative methods. Iterative methods can further be classified as synchronous or asynchronous methods. Synchronous methods, like the Jacobi method and the Gauss-Seidel method have been discussed and extensively used. Asynchronous styles of iterations (*e.g.* chaotic relaxation) have been developed more recently.

In the *chaotic relaxation* scheme [4], each grid point on the discretized problem independently replaces the old value with the newly updated values in its neighboring nodes. It is in contrast with the Jacobi method when every point completes the relaxation in a *lock step* manner. A detailed analysis of this kind of asynchronous relaxation can be found in [8]. The data-flow execution for a two-dimensional problem will be simulated and analyzed in the next section.

4.4 Neural Networks Simulations

Artificial neural networks have been recently studied to achieve high performance in AI-related research. Generally, the model of neural networks is specified by the net topology, and node characteristics.

One basic model of neural networks contains many nonlinear computational elements connected by biological neural nets. Inside the elements, computational models can simulate the neurons [10]. In Fig. 7, a generic artificial neuron is drawn with multiple input arcs and one output arc. The input signals of an artificial neuron can be either binary codes or continuous values. In the actual execution, a new signal will be sent out from the output arc, whenever the combination of the input signals satisfy some conditions of the properties of the cell. When we apply the data-flow principles of execution to simulate neural networks, every neuron can be treated as a macro-actor connected to others according to the biological neural nets. Due to the fact that an input signal will asynchronously arrive at a neural cell, the actor that represents the cell needs to be fired at any time when an input token arrives. In a data-flow machine, the *Matching Store with Locks* scheme can exactly represent the behavior. An example of a simulated neuron can be seen in Fig. 8.

5 Simulation Results

The principles developed in the previous sections have been implemented in several data-flow graphs. Their execution has been verified by a deterministic simulation of a multi-level data-flow machine.

5.1 Simulation Assumptions

The architecture model of a MTD system within the von Neumann execution mode was adopted for the simulator. It consists of a multiprocessor system with a maximum of 64 processors interconnected by a packet switching 6-dimension hypercube network. In order to gather reasonable performance statistics, we made appropriate assumptions on the various hardware and software delays: we assumed that all functional units (Matching Store Unit, Instruction Fetch Unit, and Token Formating Unit) as well as each network node all required a single unit of delay to perform their function. We also assumed that each single elementary actor would take a single unit of delay in the ALU, while a macro-actor would take the amount of time needed to execute all the micro-instructions inside the macro-actor, and the execution time of the broadcast actor would be proportional to the number of tokens to be produced.

5.2 Simulation Results

Both the chaotic relaxation (asynchronous) and Jacobi method (synchronous) have been programmed in a tagged token data-flow graph and their execution simulated. The termination criterion of both methods is chosen to be $||x^{(k)} - x^{(k-1)}|| < 10^{-3}$, where the norm is the maximum norm. In both sets of experiments, with a 16×16 matrix problem size, the following results have been observed:

- *Execution time* : The execution times of chaotic relaxation in various system configurations with pure micro-actors and with combined micro and macro actors are reported in Table 1. The execution times of the Jacobi method in various system configurations with pure micro-actors and with combined micro and macro actors are reported in Table 2.

- *Speed-up* : The speed-up can be obtained by comparing the execution time of a data-flow graph on N PEs with the execution time of the same graph on a single PE. The reports of the speed-ups in various system sizes for both chaotic relaxation and the Jacobi method are attached in Tables 1. and 2. while Fig. 9 shows the trend of the speed-ups with increasing PEs for the two different relaxation methods.

- *System tune-up* : In order to observe the effect of network communication time and matching time in the matching store, we change the time delay in the above facilities to verify the original assumptions. The execution times of both methods in macro execution mode are compared in Table 3, while that in micro execution mode is reported in Table 4.

- *Robustness*: One of the most important parameters to evaluate a multiprocessor system is to observe the system performance with various problem sizes. Hence, the robustness of a system is defined as :

$$Robustness\ (N, M) = \frac{Exe.\ time\ of\ M\ on\ one\ PE}{Exe.\ time\ of\ M\ on\ N\ PEs}$$

where M is the problem size that is optimal to the system size N

Essentially, robustness is an indication of how well the architecture/execution model will scale up when machine sizes and problem sizes are increased. In fact, the robustness property of a system can actually indicate its potential performance. We thus express the robustness property of data-driven architectures by showing the speed-ups in a large number of PEs. We exploit the trend of speed-ups in many different problem sizes with various system configurations. We start with the matrix problem size from 8×8 up to 64×64 and the machine size from 1 PE to 64 PEs. The report is shown in Table 5. and the curves are shown in Fig. 10.

6 Observations

In this section, we observe the simulation results and analyze the relative merits of the Jacobi method and of chaotic relaxation as well as of the macro and the micro execution modes.

6.1 Observations

The simulation results, shown in the previous section, indicate many interesting features with respect to the two algorithms as well as the architecture:

- Fig. 9 shows that the combined macro and micro actor graph always needs less execution time than the pure micro actor graph in both chaotic relaxation and Jacobi method. This is due to the reduction of overhead in the macro-actor execution mode.

- The execution time shown in Fig. 9 indicates that the speed-up in chaotic relaxation is always better than the speed-up of the Jacobi method in both macro and micro execution modes. In a single processor environment, we find that chaotic relaxation needs more time to execute than the Jacobi method. In a multiprocessor system, on the contrary, it can be seen that chaotic relaxation reduces the execution time much more than the Jacobi method.

- In chaotic relaxation, a *superlinear* speed-up can be sometimes observed due to the undeterministic property of the algorithm itself. Indeed, the random sequence of relaxations may lead to a faster convergence in multiprocessor systems.

- The inter-PE communication delay and the slow matching mechanism in the matching store may defeat the performance of a data-driven system. Tables 3, 4 show that chaotic relaxation is less sensitive to this kind of time delay than the Jacobi method. A paradoxal phenomenon can also be observed in that chaotic relaxation may need less time in a slower system (delay 10 times higher).

- In Fig. 10, we know that there are almost linear increasing speedup curves for the two methods in each operation mode. This is a very promising feature for data-driven multiprocessor systems. Indeed, the robustness property of data-flow architectures can guarantee the performance in multiprocessor systems for various problem sizes. For example, from Table 5, the speed-up of chaotic relaxation for 64×64 problem size can reach up to 52 in a 64 PEs system with the macro execution mode.

7 Conclusions

In this paper, we have demonstrated how asynchronous algorithms could be implemented on a data-driven multiprocessor architecture. In summary, it can be said that this paper has demonstrated the following points:

1. Data-flow principles of execution can be used to provide high-programmability efficiently in the evaluation of highly concurrent asynchronous methods. Indeed, we have demonstrated the graph constructions of data-driven asynchronous algorithms.

2. While previous data-flow research has concentrated on "conventional" mechanisms of execution, asynchronous execution can be enforced. Due to their low inherent communication costs, this type of algorithms will be more efficient in large-scale, distributed multiprocessors. Our Matching store with locks approach has easily and efficiently solved the problem.

3. Programmability of these types of algorithms can be verified not only at the low-level discussed in the previous points (graph construction) but also in the high-level language. To this end, we have introduced new asynchronous program constructs which can be used to create program graphs.

4. The robustness property of data-driven architectures can promise high performance in multiprocessor systems for numerous ranges of problems. In fact, this feature has been verified by our simulation results.

Future research issues will indeed include studying *syntax-directed* partitioning instead of *graph-directed* partitioning to form *macro-actors* and designing efficient mechanism to execute the graph that contains variable resolutions of actors.

References

[1] *IEEE Computer, Special issue on data-flow systems*, February 1982.

[2] Arvind and K.P Gostelow. The U-interpreter. *IEEE Computer*, February 1982.

[3] D. Bertsekas and R. Gallager. *Data Networks.* Prentice-Hall International Editions, 1987.

[4] D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and Application*, 2:199–222, 1969.

[5] J.B. Dennis. *First version of a data flow procedure language*, pages 362–376. Springer-Verlag, New York, April 1974.

[6] A. D. Friedman and P. R. Menon. *Theory and Design of Switching Circuits.* Computer Science Press, 1975.

[7] J.L. Gaudiot and M.D. Ercegovac. Performance evaluation of a simulated data-flow computer with low resolution actors. In *Journal of Parallel and Distributed Computing*, November 1985.

[8] J.L. Gaudiot, C.M. Lin, and M. Hosseiniyar. Solving Partial Differential Equations in a Data-Driven Multiprocessor Environment. In *Proceedings of the 15th International Symposium on Computer Architecture, Honolulu,Hawaii*, May 1988.

[9] J.L. Gaudiot and W. Najjar. Macro-actor execution on multilevel data-driven architectures. In *Proc. of IFIP WG 10.3 Working Conference on Parallel Processing, Pisa, Italy*, April 1988.

[10] R. Lippmann. An introduction to computing with neural nets. *IEEE ASSP MAGAZINE*, April 1987.

[11] J.R. McGraw and S.K. Skedzielewski. *SISAL: Streams and Iterations in a Single Assignment Language, Language Reference Manual, Version 1.2.* Technical Report Technical Report M-146, Lawrence Livermore National Laboratory, March 1985.

```
while Err < N  ASYNC-REPEAT
    X := ASYNC-FOR i in 0, N
         temp1 := for j in 1, N
                    ......

                    RELAXATION PROCEDURE
                    ......
         end for;

 Err := for i in 1,-N
                     ......

                    TERMINATION CHECK
                     ......
  end for
```

Figure 1: A Sample Program of Asynchronous Constructs

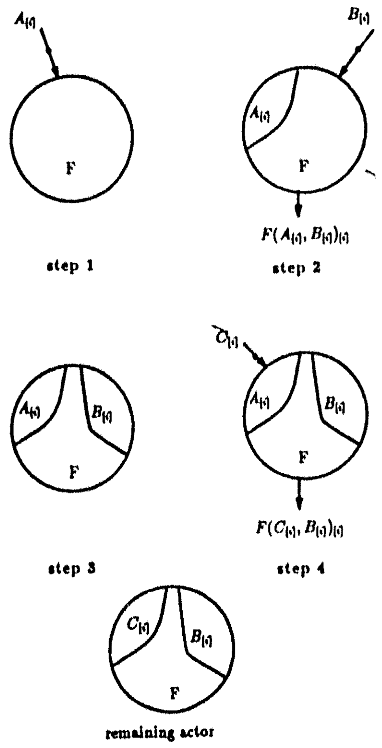

Figure 2: A New Firing Rule with locks in Matching Store

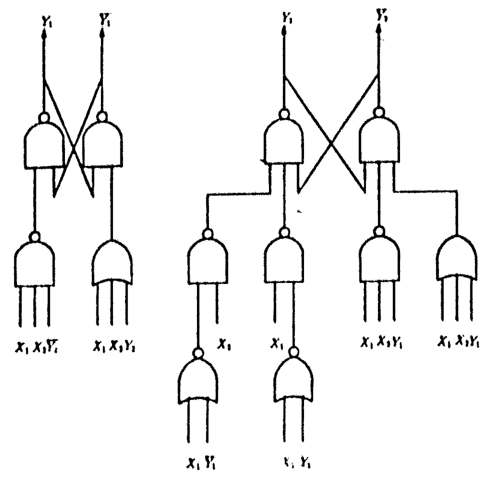

Figure 3: An Example of a Hazard-free Circuit

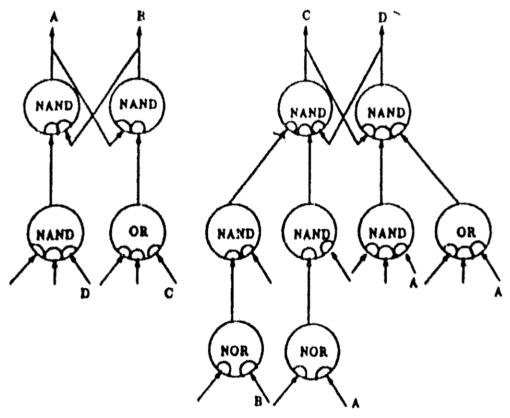

Figure 4: The Data-flow Graph for an Asynchronous Circuit

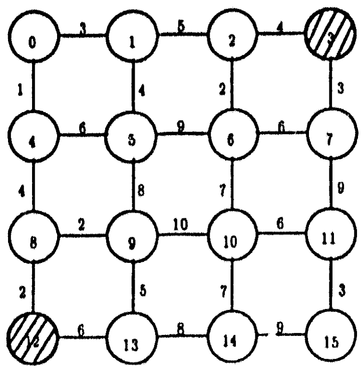

Figure 5: A 4x4 Mesh Connected Computer Network

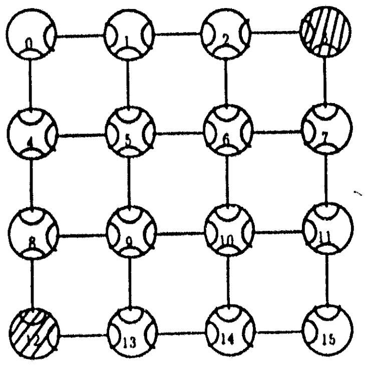

Figure 6: Data-flow Graph Representation for

a Mesh Connected Computer Network

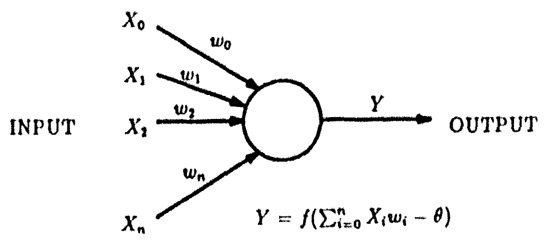

Figure 7: A Generic Abstract of a Neuron

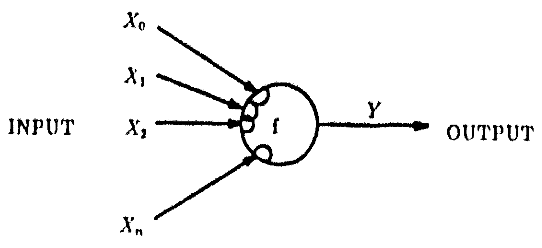

Figure 8: The Data-flow Graph Representation of a Neuron

Problem Size = 16 × 16				
System Size	Chaotic(Macro)		Chaotic(Micro)	
number of PEs	exe. time	speedup	exe. time	speedup
1 PE	108291	1	108990	1
2 PEs	56690	1.91	54430	2.002
4 PEs	26999	4.01	27174	4.01
8 PEs	13548	7.99	14840	7.34
16 PEs	8050	13.45	10538	10.34
32 PEs	6867	15.76	9708	11.22

TABLE 1

Problem Size = 16 × 16				
System Size	Jacobi(Macro)		Jacobi(Micro)	
number of PEs	exe. time	speedup	exe. time	speedup
1 PE	79924	1	92203	1
2 PEs	42112	1.89	49399	1.86
4 PEs	23109	3.45	27901	3.30
8 PEs	13640	5.86	18219	5.06
16 PEs	9759	8.18	14470	6.37
32 PEs	9244	8.64	13971	6.59

TABLE 2

Network: 1
Match: 1
Fetch: 1
ALU: 1
Router: 1

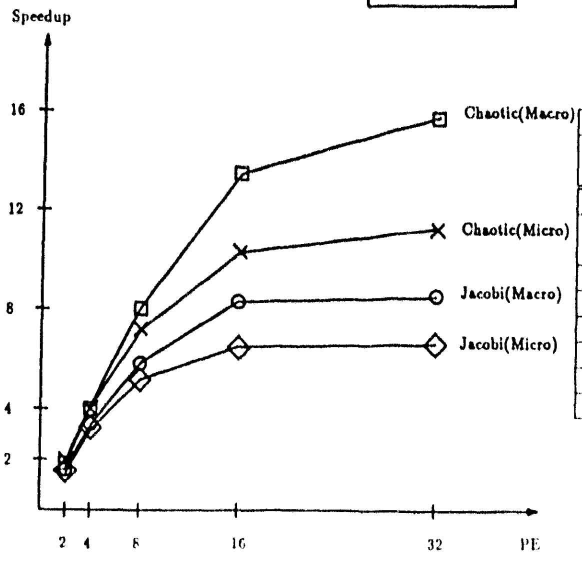

Figure 9: Speedup with Problem Size : 16 × 16

Problem Size = 16 × 16						
⋆ Network=1, Match=1, Fetch=1, ALU=1, Router=1						
♡ Network=0.1, Match=0.1, Fetch=1, ALU=1, Router=1						
System Size	Chaotic(Macro)			Jacobi(Macro)		
number	⋆	♡		⋆	♡	
of PEs	exe. time	exe. time	%	exe. time	exe. time	%
1 PE	108291	108290	99	79924	79477	99
2 PEs	56690	56689	99	42112	41101	97
4 PEs	26999	28388	105	23109	21904	94
8 PEs	13548	13476	99	13640	12312	90
16 PEs	8050	7107	88	9759	7873	80
32 PEs	6867	5681	82	9244	7256	78

$$Percentage = \frac{\text{Exe. time on short delay systems}}{\text{Exe. time on long delay systems}} \times 100\%$$

TABLE 3

Problem Size = 16 × 16

★ Network=1, Match=1, Fetch=1, ALU=1, Router=1

♡ Network=0.1, Match=0.1, Fetch=1, ALU=1, Router=1

System Size	Chaotic(Micro)			Jacobi(Micro)		
number	★	♡		★	♡	
of PEs	exe. time	exe. time	%	exe. time	exe. time	%
1 PE	108990	108989	99	92203	90832	98
2 PEs	54430	54429	99	49399	47464	96
4 PEs	27174	27233	100	27901	25772	92
8 PEs	14840	14613	98	18219	15054	82
16 PEs	10538	8839	83	14470	10811	74
32 PEs	9708	8382	86	13971	11669	83

$$Percentage = \frac{Exe.\ time\ on\ short\ delay\ systems}{Exe.\ time\ on\ long\ delay\ systems} \times 100\%$$

TABLE 4

Speedups of Various Problem Sizes					
Number of PEs	Problem Size	Chaotic (Macro)	Chaotic (Micro)	Jacobi (Macro)	Jacobi (Micro)
8 PE	8 × 8	7.15	5.54	4.28	3.45
16 PEs	16 × 16	13.45	10.34	8.18	6.37
32 PEs	32 × 32	26.26	20.30	16.05	12.21
64 PEs	64 × 64	52.15	39.77	31.49	23.70

TABLE 5

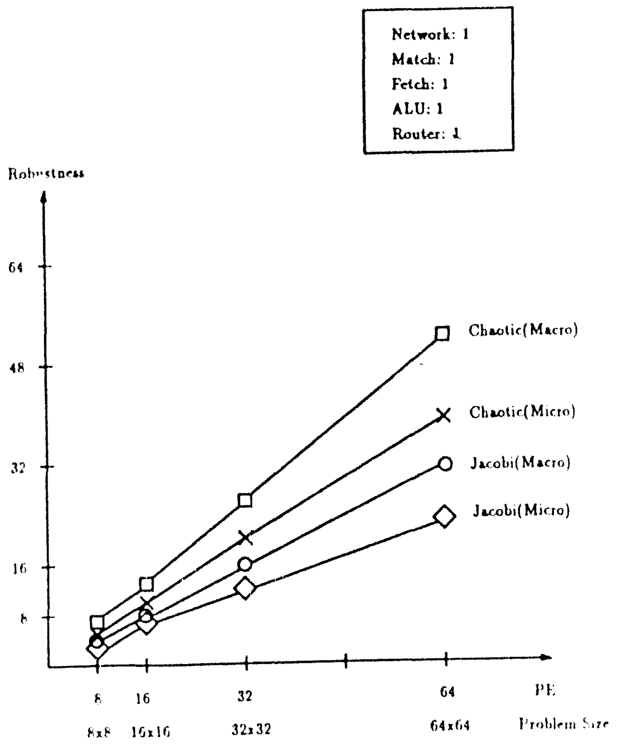

Figure 10. Robustness in Data flow Architectures