University of Southern California
Los Angeles, California
(213) 743-0249

ABSTRACT: Partial differential equations can be found in a host of engineering and scientific problems. The emergence of new parallel architectures has spurred research in the definition of parallel PDE solvers. Concurrently, highly programmable systems such as data-flow architectures have been proposed for the exploitation of large scale parallelism. The implementation of some Partial Differential Equation solvers (such as the Jacobi method) on a tagged token data-flow graph is demonstrated here. Asynchronous methods (chaotic relaxation) are studied and new scheduling approaches (the Token No-Labeling scheme) are introduced in order to support the implementation of the asynchronous methods in a data-driven environment. New high-level data-flow language program constructs are introduced in order to handle chaotic operations. Finally, the performance of the program graphs is demonstrated by a deterministic simulation of a message passing data-flow multiprocessor. An analysis of the overhead in the data-flow graphs is undertaken to demonstrate the limits of parallel operations in data-flow PDE program graphs.

1 Introduction

Partial Differential Equations are an important problem in scientific and engineering areas. Much research has been devoted to the study of the numerical solutions of these equations. This paper will show the implementation of several PDE solvers on a special class of multiprocessor systems, namely data-flow architectures.

The data-flow principles of execution (Arvind and Iannucci, 1983) offer the programmability needed to synchronize at runtime the many parallel processes on a large scale multiprocessor. Instead of relying on the conventional central program counter, the availability of data instead 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. It is the purpose of this paper to evaluate the respective effect of the data-flow synchronization methods on the final performance of the machine for a certain class of numerical applications.

In this paper, we consider the Jacobi and Chaotic relaxation methods for solving PDEs. Although we will contrast the implementation of both methods in a data-driven graph, we do not intend to reflect on the respective intrinsic merits of the two approaches to solving PDEs. Instead, in both cases, we study the rules of construction of a U-interpreter data-flow graph (Arvind and Gostelow, 1982). In order to enforce chaotic behavior of the data-flow graph, we also introduce specific solutions such as the *token no-labeling approach* in which actors are activated without regard for the tag of the incoming token. Special high-level data-flow language program constructs are introduced in order to properly describe the asynchronous behavior of data in a chaotic algorithm. We have conducted a deterministic simulation of a large tagged token data-flow multiprocessor system. Performance evaluation of the simulated data-flow architecture centers around the speed-up obtained and analyzes the influence of the "data-flow overhead" (*i.e.*, the actors needed to ensure proper data-driven synchronization). We show how much of this overhead cannot be easily parallelized and thus presents an unsurmountable impediment to delivering high speed-ups even in the presence of large degrees of parallelism.

Overall, the goal of this paper is to demonstrate how such different implementations of PDE solvers as conventional Jacobi and chaotic relaxation 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 multiprocessor system. In section 2, the elementary data-flow principles of execution and several methods for the solution of Partial Differential Equations are shown. The Jacobi method and the chaotic relaxation method along their associated data-flow implementation are discussed in section 3. While section 4 presents the results of a deterministic simulation of the MIT Tagged Token Data-flow Architecture, performance observations are analyzed in section 5 and concluding remarks are made in section 6.

2 PDE's on Data-Driven Machines

In this section, we first introduce the data-flow principles of execution and review the essentials of the PDE solvers which are intented to be implemented on these machines.

[1]This material is based upon work supported in part by the US Department of Energy under Grant No. DE-FG03-87ER25043

- Actors are purely functional and execution produces no side-effects.

Data-flow programs are represented by directed, acyclic graphs which consist of actors connected together with arcs. Arcs represent the data dependencies between actors and carry tokens which are the data values being passed between actors.

2.1.1 Interpretation Models and Structure Representation

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 been put forward regarding the interpretation of data-flow graphs (Dennis, 1974) (Arvind and Gostelow, 1982).

Large structures cannot be easily modified under data-flow principles of execution since the underlying principle of *single assignment* prevents the *updating* of any data structure. Copying operations are expensive albeit logically acceptable; therefore, several schemes(Arvind and Thomas 1980)(Gaudiot 1985, 1986)*et al.* have been designed to alleviate these problems.

2.1.2 The MIT Tagged Token Data-flow Architecture

This machine implements a version of the U-Interpreter, while array handling mechanisms support I-structure concepts (Arvind, Kathail and Pingali, 1980). An abstracted structure of a PE is shown in Fig. 1. In this distributed architecture model, each PE is independent from its neighbors and there is no global controller. A hypercube communication network allows the transmission of data-flow tokens between PEs. Store-and-Forward capabilities are provided so that a pair of PEs which are not directly linked may still communicate.

2.2 PDE Solvers

We now describe in general terms the computational problems posed by Partial Differential Equations. A partial differential equation is an equation which contains one or more partial derivatives. An example of a PDE is Laplace's equation.

2.2.1 Jacobi Method

In the evaluation of basic iterative methods (Varga, 1962), we assume that A is a nonsingular nxn (sparse or dense) matrix. Let b be a given column vector, then the system of linear equations can be expressed as $Ax = b$. The Jacobi iterative method is derived as:

... We can then terminate the iterations and accept $x^{(k)}$ as the answer when the following condition is satisfied : $|x^{(k)} - x^{(k-1)}| < \epsilon$

2.2.2 Chaotic Relaxation

In the asynchronous approach (Baudet, 1978), communication between processes is achieved by reading the dynamically updated variables, while each process continues the execution for the updating of the common variables. A subset of asynchronous methods, called *chaotic relaxation schemes*, was introduced by Chazan and Miranker (1969) to solve linear systems. Generally, the chaotic relaxation method is more satisfactory in practical applications although the definition of asynchronous methods is mathematically more rigorous.

2.3 High-Level Language Constructs

The numerical PDE solvers which we have described must also be expressed by high-level language constructs. We have chosen for this purpose SISAL (Streams and Iterations in a Single Assignment Language), a high-level applicative language developed by McGraw, Skedzielewski, *et al.* (1985). In addition, we will demonstrate our new synchronization constructs necessary to describe chaotic behavior. As we have seen before, we can distinguish between two forms of PDE solvers: *synchronous* or *asynchronous* methods. Each of these two approaches will be expressed by different high-level language program constructs which we will now study in turn.

2.3.1 Synchronous Constructs

Assume that we have to implement a relaxation problem with a stopping criterion evaluated by the function *Notconverge*(). The function *Notconverge*() checks the array Y at every iteration to determine whether it converges. Therefore, the whole relaxation procedure behaves in a synchronous manner. The corresponding program construct is shown below:

```
(1)  Function relax ( x:OneDim
(2)                   N:Integer
(3)                   returns OneDim)
(4)  for initial
(5)  Y :=x;
(6)  while ( Notconverge(Y) ) repeat
(7)  T := for I in 1, IMAX
(8)  returns array of F ( old Y[I-1] , old Y[I], old Y[I+1] )
(9)  end for
(10) Y := array[ 0 : X[0] ]
(11)          || T
(12)          || array[ IMAX+1 : X[IMAX+1] ]
(13) returns value of Y
(14) endfor
(16) Endfunction
```

—gorithm. In the above example (line 8), we now allow the function F to execute over different versions of the Y's. This means that whenever a value of index I completes the evaluation of the function F, it proceeds with the execution of the next computation without waiting for other index values which are executing the function. Hence, we can rewrite the above program for asynchronous computations by changing line(6) to:

while (Notconverge(Y(I))) async-repeat

Once an index I has completed the evaluation of F in line(8), the value of the grid point at index I is checked by the function *Notconverge*(). If the function returns a true value (i.e., all the points still do not converge), then the index is allowed to produce a new iteration.

3 Program Graph Constructions

The numerical methods of PDE solvers described in the previous sections can be easily adapted to the data-driven principles of execution.

3.1 Jacobi Method

As described earlier, the Jacobi method is an iterative solution of a system of linear equations.

3.1.1 The Token Relabeling Implementation of Jacobi

In order to insert the data inputs into the Jacobi data-flow graph, special actors must be used for tag modification. In particular, these actors modify the context and the iteration number portion of the tag. In the actual implementation, the matching of the iteration number portion of the tag of the data tokens is handled through the use of two primitive actors δ_R and δ_W. The function of the δ_R actor is to extract the iteration number from the input token while that of the δ_W actor is to set the iteration field of the tag of one of the input tokens with the data field of another input token. A detailed presentation of these actors was made by Gaudiot (1985) and by Gaudiot and Wei (1986).

3.1.2 Data-flow Implementation

In the Jacobi method, the major computation consists of repeatingly computing the value of $x^{(k+1)}$ from (1) and evaluating the termination criterion for n steps, where n is the size of the matrix A. The data-flow graph of this algorithm will accept a matrix A (nxn, dense), a column vector b, an initial column vector x and two constants n and ϵ. The output would be the solution vector x with required accuracy ϵ. In Fig. 2, the block diagram of the Jacobi computation are presented. Note that each block of the body of the graph is iteratively performed:

- *Distributor*: The set of input data tokens from the two dimensional array A is routed to the distributor which separates diagonal elements from non-diagonal elements.
- *Major Jacobi Computations*: These data tokens are then forwarded along with the other input data tokens to the terminals of the major Jacobi computations block (body of the graph). This block processes paired tokens and generates the new tokens (new x vector) by using equation (1).
- *Termination Computations*: This stage computes the error vector from the old and the new vectors of x.
- *Loop Control*: The loop control block then checks whether a solution has been reached by checking the output of the terminator block against the prescribed accuracy ϵ and makes a decision of whether to loop the tokens back or to route them to the output module.

3.2 Chaotic Relaxation Method

As was seen before, chaotic relaxation is another approach which is particularly successful in parallel environments. In order to efficiently implement it on a data-flow multiprocessor, we introduce here the *token no-labeling approach* in order to observe the appropriate asynchronous behavior in a data-flow environment.

3.2.1 The Token No-labeling Approach

In order to solve PDE problems by chaotic relaxation, we propose the *token no-labeling* scheme based on the MIT Tagged Token Data-flow Architecture. In the MIT architecture, the U-interpreter uses $< u, c, s, i >$ to describe every activity (a single execution of an operator), where u is the context field, c is the code block name, s is the instruction number, and i is the iteration number. For synchronous methods, the U-interpreter can be used for the data-flow graph (token relabeling). However, it is not directly suitable for chaotic relaxation. We therefore introduce the following scheme: we *no-label* the iteration field, that is, we ignore the i field in the $< u, c, s, i >$ label. When the interpreter compares the tokens in the matching store, only u, c, and s need to be compared. If two input tokens have the same $< u, c, s >$, then they can be matched and the corresponding activity scheduled for execution. This is shown in Fig. 3a. When tokens A and B enter the F actor, as long as A and B have the same u, c, and s then they can be executed to produce the output.

In order to guarantee the proper behavior of the no-labeling actor, we also introduce the notion of *locks* at the inputs of the actors. 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

... matching method in the Matching Store:

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

2. When another token arrives, actor F will be fired and will produce an output token. Note that the iteration tag is undetermined because the output will be used by other *no-labeling* actors.

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.

3.2.2 Data-flow Implementation

The implementation of chaotic relaxation in a data-flow graph form is quite similar to the Jacobi method mentioned in the previous section. When executing the chaotic relaxation algorithm, every x_i in equation (1) is kept evaluating by taking the latest updated values of other points. This process will terminate as soon as, for each x_i, the condition: $| x_i^{(k)} - x_i^{(k-1)} | < \epsilon$ and $| x_i^{(k-1)} - x_i^{(k-2)} | < \epsilon$ where k is the number of iteration steps, is satisfied. The main difference between the two methods is in the loop control block of Fig. 2. The loop control block must wait for the completion of all the processes, before it can proceed with the next evaluation. In the chaotic method, every process can proceed uninterrupted without waiting for other processes. This unsynchronized loop control represents the chaotic behavior which simplifies the design of the graph. Note that translating the graph into such a low-level graph construct can be easily effected by the compiler upon encountering an *async-repeat* high-level language instruction as was described in section 2.3.2.

4 Simulation Results

The principles developed in the previous sections were implemented in several data-flow graphs. Their execution was verified by a deterministic simulation of a data-flow machine.

4.1 Simulation Assumptions

The architecture model of the Arvind/MIT Tagged Token Data-flow Architecture 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 on the two PDE solvers, we made appropriate assumptions on the various hardware and software delays. We also assume that all functional units as well as each network node all require a single unit of delay to perform their function.

4.2 Simulation Results

Both the Jacobi method and the chaotic relaxation approach have been programmed in a tagged token data-flow graph and their execution simulated. When no confusion is possible, they will be respectively referred to as " non-chaotic" and "chaotic" from now on. In both sets of experiments, the following data were noted:

- *Execution time*: Program sizes range from an 8x8 matrix (*i.e.*, an 8-point grid) to a large 32x32 problem (*i.e.*, a 32-point grid). Figs. 4a through 4c show the effect of increasing the machine size for 8, 16, and 32 point grids and display the execution time of the given problem size for various machine configurations (from 1 to 64 Processing Elements).

- *Speed-up*: The speed-up can be obtained by comparing the execution time of N PEs with the execution time of one PE. While the results obtained in the previous section indicate that any computation will be taxed by the overhead in processing, they do not directly explain the serious speed-up limitations observed in the simulation (Figs. 5a through 5c). Indeed, it appears that several factors are responsible for this effect: A large overhead portion cannot be parallelized (Amdahl's law), the increasing communication costs and increased *forced* sequentiality (also referred to as *resource dependency*), the I/O problem, as well as the allocation problem. We will analyze the relative effects of these factors in the next section.

- *Overhead analysis*: Table 1 displays a dynamic count of the actors executed during the processing of an 8x8 problem in both the chaotic and the non-chaotic relaxation modes for various machine configurations. For our purposes, we define a non-overhead actor as an actor directly involved in the algorithm. Table 2 shows, for the chaotic relaxation algorithm, the count of sequential overhead actors and sequential non-overhead actors in an 8-PE machine for an 8x8 problem size and in a 16-PE machine for a 16x16 problem size. In an iterative part of the algorithm, sequential overhead actors are located on a *critical path* of the loop while non-sequential overhead actors can be executed in parallel. The *total number of actors* has simply been obtained by counting, during the simulation, the number of actor executions. This number gives a measure of the complexity of the problem. The number of *overhead actors* has then been evaluated in a similar fashion: it was obtained by counting only the execution of certain *marked actors*. Marking actors involves two steps:

  1. Mark "overhead" actors as those not involved directly in the algorithm itself but instead participate only in operations related to the U-interpretation model such as production of iteration indices with the appropriate tags, etc. The rest are marked as "nonoverhead" actors.

2. Mark actors according to their functional relations with other actors in the graph structure as "sequential overhead" or "sequential nonoverhead".

We use a simple program graph whose function is to square the diagonal values of a matrix to illustrate the *actor marking* operation: in Fig. 6, assume we use n^2 PEs for a n×n matrix. There are five actors (A to E) in each PE to perform this function. The E-actor (SQUARE) is the only actor relevant to the problem, while others are just inserted for the purpose of insuring correct synchronization and safe execution. Therefore, we *mark* the E-actor as "*nonoverhead*" while the actors A, B, C, and D are marked as "*overhead*". By a closer inspection, a "sequential" path can be found (either A-C-D-E or B-C-D-E). These two strings represents a forced sequential execution. Hence, if we choose A, C, and D as "sequential overhead" actors, the B-actor remains simply an overhead actor, because it could be executed in parallel with other actors. Note that the marking of actors as sequential only refers to their execution relative to other actors in the neighboring program graph construct. Indeed, it will be seen in the next section that these sequential actors will cause a "drying-up" of the various pipelines in the machines.

- *Utilization*: Table 3 shows the utilization ratio of each functional unit (Matching Store Unit, Instruction Fetch Unit, ALU, and Token Formating Unit) in a Processing Element for chaotic relaxation with different machine sizes.

5 Performance Analysis

Although the rate of convergence in chaotic relaxation is hard to obtain theoretically, the simulation results shown in Figs. 4a-4c indicate that the chaotic relaxation is faster than Jacobi's method. The chaotic relaxation can perform 2 to 5 times better than the conventional Jacobi approach. This is due to the fact that in Jacobi's method, each processor must ***synchronize*** with other processors at each iteration which creates a bottleneck.

The result of more crucial relevance to this paper, however, is the fact that both of these two methods have a limited speedup (Figs. 5a through 5c). The speedup is limited to 3 by using 8 PEs in an 8×8 problem size, and the speedup is 5 by using 16 PEs in a 16×16 problem size, while it is limited to 9 by using 32 PEs in a 32×32 problem size. As mentioned, there are several factors to explain this. The main reasons for this phenomenon are two-fold:

- High amount of sequential overhead.
- The existing type of parallelism associated with the algorithm itself is not suitable to this implementation.

From Table 1, it can be seen that the overhead associated with the computation is large: only 20 % of the actors involved in the computation are actually directly related to the algorithm, while most of the remaining overhead actors (80 %) have to be sequentially executed. In the following discussion, we extend Amdahl's law to analyze this phenomena.

Our data-flow multiprocessor is in fact a collection of pipelined processors (since each data-flow processor (Fig. 1) is composed of 4 sequential blocks, the Matching Store Unit, the Instruction Fetch Unit, the ALU, and the Token Formating Unit). For a parallel algorithm on such a *multi-pipelined-processor* system, we should consider two types of parallelism: *pipeline*-type parallelism and MIMD-type parallelism (sometimes referred to as concurrency). The *pipeline*-type parallelism is a temporal parallelism which is suitable for execution on a multi-stage pipeline processor. At the same time, MIMD-type parallelism is a spatial parallelism which can be efficientiy executed in several simple Processing Elements. Therefore, a process with MIMD-type parallelism can be divided into several parts for several processors with a speedup P (the number of processors). This is not, however, the case for a process with *pipeline* parallelism. Assuming that we have a process containing *pipeline*-type parallelism which can be executed in a single processor with throughput T, it cannot be claimed that by using P processors for the same process, every processor will still have a throughput T and gain a speedup P for these processors. Matrix multiplication is a case in point:

Matrix multiplication can be written as $\mathbf{A} \times \mathbf{B} = \mathbf{C}$, where $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are n×n matrices. For every element $c_{i,j}$ in $\mathbf{C}$, $c_{i,j} = \sum_{k=1}^{n} (a_{i,k} \times b_{k,j})$. The operations of multiplication are of MIMD-type parallelism and the operations of summation are of *pipeline*-type parallelism. Hence, a total of n^3 MIMD-type parallelism (the multiplications) and n^2 of *pipeline*-type parallelism (the summations) can be found. If we use a pipelined processor to execute the n^3 multiplications, n^3 PEs can achieve a speedup of n^3. However, for the n^2 summations, n^2 PEs will not obtain a speedup of n^2. The reason is that if only one PE is used, many summations can share the stages of the pipe and saturate it. However, when many PEs become available, it correspondingly becomes possible to allocate only one operation to one PE. Therefore, the pipe cannot be fully utilized due to the data dependency inside the summation operation. This automatically degrades the speed-up.

According to Amdahl's law, the total execution time (ET) of a program can be expressed as:

$$ET(1) = (Sequential\ part) + (Parallel\ part) \quad with\ 1\ PE \qquad (2)$$

$$ET(P) = (Sequential\ part) + (Parallel\ part)/P \quad with\ P\ PEs \qquad (3)$$

The total execution time can also be rewritten as:

$$ET(1) = (Sequential\ part) + (Pipeline\ part) + (MIMD\ part) \qquad (4)$$

$$ET(P) = (Sequential\ part) + (Pipeline\ part/P) \times T(P) + (MIMD\ part/P) \qquad (5)$$

P : number of PEs.
$T(P)$: interoutput time = 1/throughput, for each PE of P PEs.

The speed up can be written as:

$$S_p(P) = \frac{ET(1)}{ET(P)} \tag{6}$$

In equation (5), the value of T(P) would be 1 in the ideal case which means that an output is generated at each pipeline cycle. Instead, T(P) will increase when the number of processors (P) increases. This is due to the fact that pipeline stages become unsaturated and that resources become idle. The amount of actors in each part of equation (5) (*Sequential part*, *Pipeline part*, and MIMD *part*) can be obtained from the simulation results summarized in Tables 1 and 2. For example, there is a total of 4065 dynamic actors (Table 1) for an 8×8 matrix in chaotic relaxation using 8 PEs. We separately counted 64 actors in the *Sequential part* due to the sequential input block. The actors in the *Pipeline part* can be calculated from Table 2: add the total sequential nonoverhead actors in all PEs (734) and the total sequential overhead actors in all PEs (2436), for a total of 3170. Finally, there are 831 actors in the MIMD *part*. This last figure has been obtained by subtracting the number of actors in the *Sequential part* and in the *Pipeline part* from the total number of actors. If we assume that T(1) is equal to 1, from equation (6), we obtain $ET(P) = ET(1) \,/\, S_p(P)$, in which $S_p(P)$ is the speedup observed during the simulation and $ET(1)$ is obtained from equation (4). Hence, we can calculate $ET(P)$ and obtain T(8) = 2.7 Similarly, we can also obtain T(2) = 1.1 , T(4) = 1.6 Therefore, when the number of processors increases, the increasing values of T(P) show that it becomes harder to fully utilize every functional units in a PE. This is a good intuitive explanation for the poor speed-up behavior with increasing numbers of Processing Elements: adding more Processing Elements simply lowers the load on each PE with no possible gain due to the high sequentiality of the program graph.

By inspecting Table 3, we can further verify that the utilization actually decreases in each stage of functional units as more PEs are used. The problem has also been noticed by Gajski *et al.* (1982) for *pipeline*-type parallelism. For example, in the matrix multiplication case, the summation operation can be executed by using the *tree height reduction* method. In a data-driven machine, when an algorithm contains *pipeline*-type parallelism (such as our PDE solver), those overhead actors which are needed to insure the correctness of computations, will be forced to execute sequentially since they are usually loop index producers, etc. which are highly sequential operations. Therefore, it is not possible to use the traditional methods such as *tree height reduction* to handle the problem.

In the *macro-actor* scheme (Gaudiot and Ercegovac, 1985), several operations are grouped into a single actor. *Macro-actors* can also be used in this context if we lump all the sequential overhead actors. This would reduce the execution time by reducing the length of the sequential path. However, it should be noted that the inverse throughput $T(P)$ of a PE would slightly increase since each actor requires more processing time but the pipelining will be improved. Indeed, the sequential overhead actors in each PE can be *grouped* in various sizes of *macro-actors*. From equation (5), it can be seen that the amount of *Pipeline part* actors are actually reduced by the above scheme. For example, if a *macro-actor* contains two sequential overhead operations, then the number of actors in *Pipeline part* of equation (5) will be reduced to 253, which is the sum of the sequential nonoverhead actors and half of the sequential overhead actors in a PE for an 8×8 problem size. If we assume the throughput of each PE remained unchanged, the speedup, according to equations (4), (5), and (6), will tend to increase while the actors in *Pipeline part* are reduced. The speedup, calculated from equation (6), using 8 PEs for an 8×8 matrix problem, can be predicted in Fig. 7 with respect to different amounts of sequential overhead *macro-actors* in each PE. It should be noted that the reduction in the amount of sequential overhead actors is limited. This is due to the fact that the *sequential part* is not independent of the rest of the computation but in fact presents certain data dependencies with the rest of the computation. For instance, it could be estimated that the number of sequential actors could not be less than 100 in Fig. 7.

6 Conclusions

In this paper, we have demonstrated how two different approaches to solving Partial Differential Equations could be implemented on a data-driven multiprocessor architecture. The two PDE solvers were chosen for their known inherent parallelism of execution: the conventional Jacobi method and the chaotic relaxation approach. While the "*conventional*" principles of the U-interpreter were used in the graph construction of the Jacobi method, chaotic behavior could not be easily sequenced in this model of interpretation. We therefore proposed a new scheme for the implementation of chaotic relaxation: the "*token no-labeling*" scheme proceeds with the execution as soon as *any change* has been detected on the input arcs, instead of allowing execution upon arrival of a matched token set. This low level data-flow graph scheme can be easily inserted in the program graph by the compiler, provided that our proposed special, asynchronous, high-level language constructs can be included in the target language. An extensive simulation of the execution of these two PDE solvers on a simulated data-flow machine was carried out. It demonstrated that a combination of excessive overhead ratio, large sequentiality of overhead program graph, high communication costs, and poor locality of allocation could seriously limit final speed-up, regardless of the machine configuration.

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 numerical evaluation of highly concurrent numerical algorithms such as PDE solvers. Indeed, we have demonstrated the graph construction of data-driven Jacobi.

2. While previous data-flow research has concentrated on "conventional" mechanisms of execution, asynchronous execution (chaotic relaxation) can also be enforced. Due to their low inherent communication costs, this type of algorithms will be more efficient in large-scale, distributed multiprocessors. Our token no-labeling approach has easily and efficiently solved the problem.

3. Programmability of these two 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 no-labeling program graphs.

4. Even these highly parallel applications can entail a large amount of overhead processing (up to 80 %). In addition, lack of pipelining within this overhead is a major hindrance to speeding up the computations. The creation of larger computing entities must be undertaken in order to ensure better resource utilization.

Future research issues will indeed include using *macro-actor* techniques and designing an optimal instruction set to reduce the overhead.

REFERENCES

[1] Arvind, and Gostelow, K.P., "The U-interpreter," in *IEEE Computer*, Vol. 15, No. 2, February 1982.

[2] Arvind and Iannucci, R.A., *Two fundamental issues in multiprocessors: the data-flow solution*, MIT Lab, for Comp. Sc. Technical Report MIT/LCS/TM-241, Sept. 1983.

[3] Arvind, Kathail, V., and Pingali, K., *A data-flow architecture with tagged tokens*, Lab. for Comp. Sc. (TM-174), MIT, Cambridge, Massachusetts, Sept. 1980.

[4] Arvind, and Thomas, R.E., *I-structures: An efficient data type for functional languages*, Rep. LCS/TM-178, Lab. for Computer Science, MIT, June 1980.

[5] Baudet, G.M., "Asynchronous iterative methods for multiprocessors," in *Journal of ACM*, April 1978.

[6] Chazan, D., Miranker, W., "Chaotic relaxation," Linear Algebra and Application, 1969, pp. 199-222

[7] Dennis, J.B., "First version of a data flow procedure language," in *Programming Symp.: Proc. Colloque sur la Programmation* (Paris, France, Apr. 1974), Lecture notes in Computer Science, vol. 19, Springer-Verlag, 1974, pp. 362-376.

[8] Gajski, D.D., Padua, D.A., Kuck, D.J., and Kuhn, R.H., "A second opinion on data-flow machines and languages," in *IEEE Computer*, February 1982, pp. 58-69.

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

[10] Gaudiot, J.L., and Wei, Y.H., "Token relabeling in a tagged data-flow architecture," in *Proc. 1986 International Conference on Parallel Processing*, August 1986.

[11] Gaudiot, J.L., "Structure handling in data-flow systems," in *IEEE Transactions on Computers*, June 1986.

[12] Gostelow, K.P., and Thomas, R.E.,"Performance of a simulated data-flow computer," in *IEEE Transactions on Computers*, Vol. C-29, No. 10, October 1980.

[13] McGraw, J.R., and Skedzielewski, S.K., *SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual, Version 1.2*, Lawrence Livermore National Lab. TR M-146, March 1985.

[14] Varga, R.S., *Matrix iterative analysis*, Prentice Hall,1962.

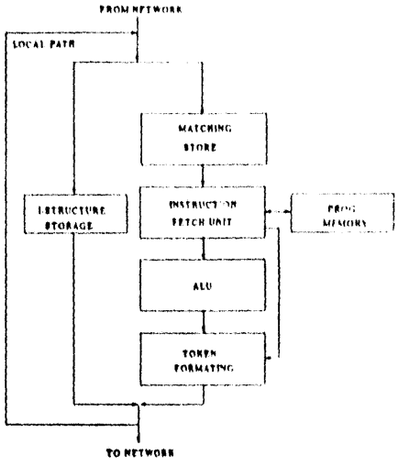


Fig. 1 Simplified model of a PE in the MIT Tagged Token Data-flow Architecture

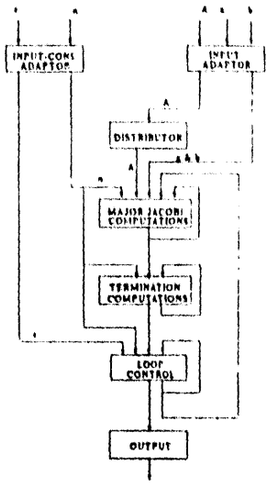


Fig. 2 Block diagram of Jacobi

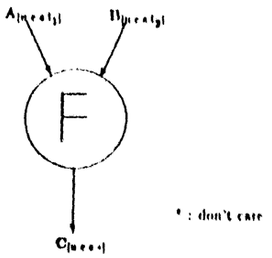


Fig. 3a Token No-labeling Scheme

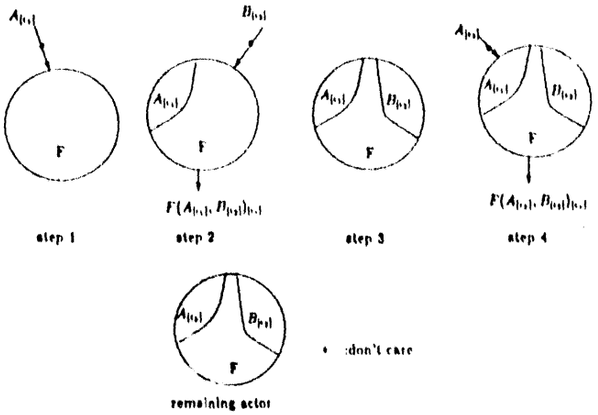


Fig. 3b Token no-labeling matching

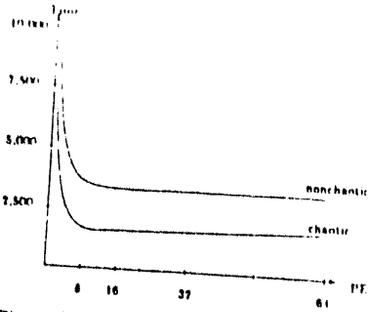


Fig. 4a Execution time with problem size 8x8

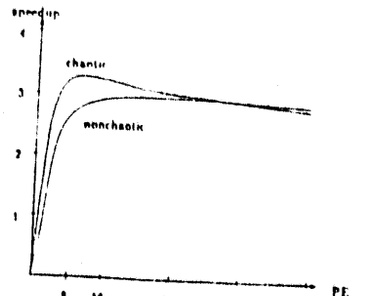


Fig. 5a Speedup with problem size 8x8

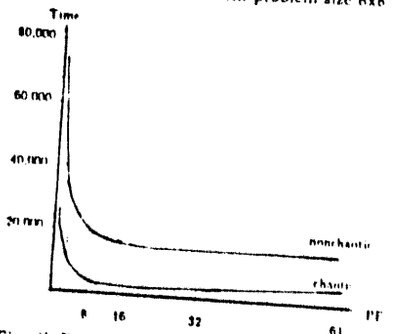


Fig. 4b Execution time with problem size 16x16

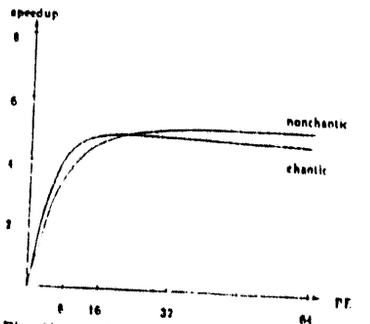


Fig. 5b Speedup with problem size 16x16

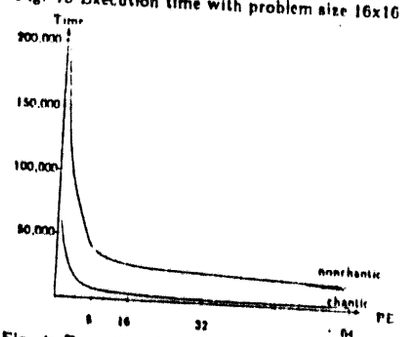


Fig. 4c Execution time with problem size 32x32

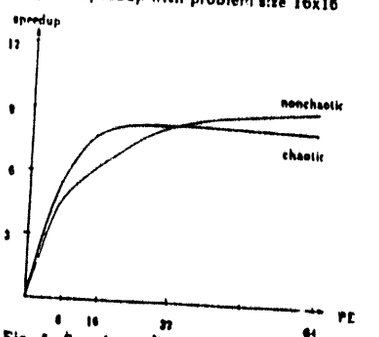
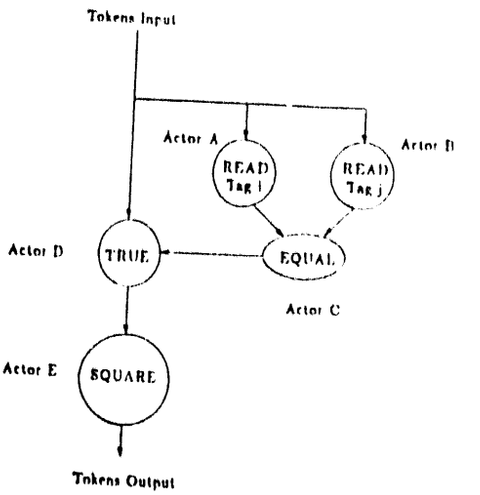


Fig. 5c Speedup with problem size 32x32



Actor A : overhead, sequential overhead
Actor B : overhead
Actor C : overhead, sequential overhead
Actor D : overhead, sequential overhead
Actor E : nonoverhead, sequential nonoverhead

Fig. 6 An example of Marking Actors

	number of PE	1	2	4	8	16	32	64
non-chaotic	number of static actors	61						
	number of dynamic actors	5276						
	number of overhead actors	4550						
	percentage	86.3						
chaotic	number of static actors	80						
	number of dynamic actors	3512	4067	3907	4065	3828	3384	3512
	number of overhead actors	2846	3273	3181	3273	3090	2724	2846
	percentage	81	80.4	80.6	80.5	80.7	81.2	81

$$\text{percentage} = \frac{\text{number of overhead actors}}{\text{number of dynamic actors}} \times 100 \text{ percent}$$

Table 1. Actor traces

	PE id	0	1	2	3	4	5	6	7
8x8	sequential nonoverhead	69	95	95	95	95	95	95	95
	sequential overhead	231	315	315	315	315	315	315	315
16x16	PE id	0	1	2	3	4	5	6	7
	sequential nonoverhead	87	175	178	174	174	179	174	175
	sequential overhead	344	692	707	687	687	717	687	692
	PE id(continued)	8	9	10	11	12	13	14	15
	sequential nonoverhead	179	179	175	174	174	174	174	174
	sequential overhead	712	713	692	687	687	687	687	687

Table 2. Sequential actor traces

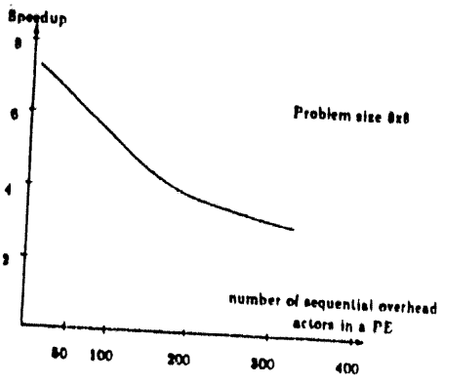


Fig. 7 The effect of reducing the number of sequential overhead actors by macro-actors on the speed-up

	1 PE	4 PE's	8 PE's
Matching Store Unit	78	50	25
Instruction Fetch Unit	64	41	21
ALU	66	42	22
Token Formating Unit	97	63	31

Table 3 Utilisation ratio of each functional unit with increasing machine size (8x8 matrix).

DISCLAIMER

END

DATE
FILMED
12/9/93