Numerical Partial Differential Equations Solvers on Variable-grain Data-flow Multiprocess Systems

Chih-Ming Lin

CEng Technical Report 91-13

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Electrical Engineering)

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA. 90089-0781

Acknowledgements

I would like to express my most profound gratitude to Professor Jean-Luc Gaudiot, my dissertation chairman, for his valuable guidance and for his constant encouragement. I have benefited greatly from his extensive technical knowledge and experience during the course of this research. I am also grateful to the other members of the dissertation committee: Professor V. K. Prasanna Kumar and Professor W. Proskurowski for their useful suggestions and constructive criticism.

Thanks also to my friends and colleagues in the Department of Electrical Engineering-Systems at USC for various assistance and discussions, among whom I would like to mention Dr. Yi-Hsiu Wei, Dr. Liang-Teh Lee, Dr. Christoph Scheurich, Dr. Paraskevos Evripidou and Mr. Andrew Sohn. Thanks also go to Mary Zittercob and William Bates for their most capable administrative help.

I would also like to acknowledge the financial support received from the U.S. Department of Energy under grant No. DE-FG03-87ER25043. Finally, I would like to dedicate this work to my parents

Wan Chieu-yueh Lin and *Howu-Tsai Lin*

and my wife,

Jyu-Mei Chen.

Contents

List Of Figures

Abstract

This dissertation addresses the research results from designing parallel multigrid algorithms for solving Partial Differential Equations (PDE), generating macro-actor graphs in the process of compiling data-flow languages, and implementing numerical asynchronous algorithms on data-driven multiprocessor systems. In order to develop not only parallel but also efficient algorithms, theoretical analysis as well as experiments of numerical *Parallel V-cycle Multigrid* (PVM) algorithms are developed. Besides parallel algorithms design, new techniques in compiler and architecture designs are studied to achieve high performance. In the software design, micro-actor graphs are lumped into macro-actor graphs based on *Actors*, *Tags*, and *Tokens* partitioning schemes. In the hardware design, techniques for implementing asynchronous algorithms, such as *Matching Store with Locks*, are developed to execute the algorithms accurately. Besides the conventional speedup and efficiency measures of multiprocessor systems, new performance measures, the *Growth Factor* and the *Scalability Factor*, are developed to evaluate systems more precisely. In this research, the study of parallel multigrid algorithms is targeted to general multiprocessor systems, while the study of the macro-actor graphs, asynchronous algorithms implementation, and performance measures is centered around the Variable-grain Tagged-token Data-flow (VTD) systems.

Chapter 1

Introduction

Solving Partial Differential Equations (PDE) is an important problem in scientific and engineering areas [12, 34]. Developing numerical solvers of PDE in a parallel environment is a challenge which has only recently been exploited. New issues have been raised in the areas of algorithm design, compiler development, and architecture implementation. It is therefore the aim of this research to design parallel PDE solvers, to develop macro data-flow graphs, and to study the implementation issues in data-driven multiprocessor systems.

When the solution of a PDE is solved numerically, iterative methods are better suited for execution in a multiprocessor environment due to their inherent parallelism. Two approaches can then be considered: We may either parallelize an existing algorithm for the target machine, or develop a new parallel algorithm. The challenge of the first approach is to detect as much parallelism and data dependency in the existing algorithm. This can be automatically done by analyzing the algorithm and then map it onto the target machine. On the other hand, the goal of the second approach is to develop a not only efficient but also

parallel algorithm. In other words, the new algorithm not only can solve a PDE faster but also can be easily executed in parallel. However, besides the study of implementation techniques on multiprocessor systems, this approach requires numerical analysis of the algorithms in order to achieve faster convergence rate.

The data-flow principles of execution have been proposed as an alternative to the von Neumann model. Instead of relying on the conventional central program counter, the availability of data instead renders an instruction executable. The foremost advantage of this model is to offer the programmability needed to synchronize at runtime the many parallel processes on a large scale multiprocessor [2, 6]. In fact, it has been shown that single-assignment or side-effect free languages do not introduce any artificial data-dependencies and therefore can be compiled into a data-flow execution graph that models exactly the intended algorithm. However, in spite of the simplicity of these principles, much overhead may be introduced in order to respect the functionality of execution [29].

New techniques in compiler and architecture designs must be studied in order to reduce the execution time by exploiting the available parallelism while keeping communication overhead at a minimum. Variable-grain architectures have been advocated in the past decades to exploit parallelism at varying levels of granularity [18, 19]. At the software design level, the compiler must detect the parallelism at different levels and decide the size of processes so that can be executed efficiently. At the hardware design level, the architecture must execute each

instruction and handle data structures promptly. In addition, accurate performance measures of multiprocessor systems must be exploited in order to design and evaluate the systems precisely.

The main goal of this research is to develop parallel PDE solvers, to study many variable-grain data-driven execution issues which includes software and hardware, and to combine the algorithms and the architectures to achieve high performance in numerical computation. While the first issue is targeted for general multiprocessor systems, the study of the latter two issues will be centered around a special class of multiprocessor systems, namely Variable-grain Tagged-token Data-flow (VTD) systems.

In chapter 2, a survey of previous research in the areas of numerical algorithms, software development, and hardware design issues will be conducted. In chapter 3, we will introduce a new highly parallel and efficient multigrid algorithm that can be implemented on parallel machines. This includes the description of the new algorithm, the architecture, as well as the theoretical analysis and the experimental results of the new algorithm on the desired machine. In chapter 4, we will demonstrate several partitioning schemes to generate macro data-flow graphs. These schemes will then be applied to computationally intensive numerical applications and compare the performance of different grain sizes of actors and tokens in data-flow graphs. In chapter 5, we will take a coarse-grain approach in the VTD system to implement asynchronous methods for solving linear systems as opposed to the fine-grain data-flow systems. The objectives of this chapter are therefore to evaluate the efficiency of the VTD system for numerical applications

in both fine-grain and coarse-grain execution models and to compare the performance of asynchronous algorithms with that of synchronous algorithms. Finally, the conclusion remarks will be made in chapter 6. The programs of a multigrid algorithm to solve Elliptic PDEs in SISAL will also be attached in the appendix.

Chapter 2

Background and Research Issues

This chapter presents a survey of previously conducted research in the areas of numerical algorithms, software development, and hardware design. In the area of numerical PDE algorithms, focus is on the iterative solvers. In software development, research issues and previous results of data-flow compilers will be introduced. In the architecture design domain, the investigation of data-driven multiprocessor systems will be discussed.

2.1 Parallel Numerical Applications

We describe in general terms the computational problems posed by Partial Differential Equations and introduce several approaches for numerical solutions.

2.1.1 Numerical Computations

A partial differential equation is an equation which contains one or more partial derivatives. Such equations occur most frequently in modeling physical phenomena, in applications of mathematics, and in almost all fields of engineering. An elliptic PDE can be represented in general form with proper boundary conditions as:

$$F(X, Y, U, U_x, U_y, U_{xx}, U_{yy}, U_{xy}, \ldots) = 0 \ in \ \Omega \subset R^2 \tag{2.1}$$

An example of a PDE is Laplace's equation:

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = 0 \ in \ \Omega \subset R^2 \tag{2.2}$$

PDEs could be solved (if a solution exists) using the finite difference approximation method. An $n^{th}-$order, one-dimensional forward finite difference is recursively defined by the following formula:

$$\begin{cases} \triangle^{(n)} f(x_i) = \triangle^{(n-1)} f(x_{i+1}) - \triangle^{(n-1)} f(x_i) \\ \triangle^{(0)} f(x_i) = f(x_i) \end{cases} \tag{2.3}$$

The basic concept in the Finite Difference method is to subdivide the domain of solution of the given PDE by a net with a finite number of mesh points. The derivative at each point is then replaced by a finite difference approximation. Alternatively, one can visualize the discretization procedure as the replacement of

the solution of the PDE with an interpolating polynomial and the differentiation of this polynomial. Three steps are required for the solution of a given PDE.

1. Choose an approximate finite difference approximation for the derivatives appearing in the governing equation and rewrite it as a difference equation.

2. Decide on an appropriate mesh size to be used and setup the corresponding system of equations. Then insert all known boundary so as to obtain a system of equations for the unknown values, in the form $Ax = b$.

3. Obtain a solution for the system of equations derived in step 2.

Methods of solution in step 3 may be classified as direct, involving a fixed number of arithmetic operations, or as indirect or iterative, involving the repetition of certain steps until the required accuracy is achieved. Iterative methods have often been preferred to direct methods for solving large and sparse sets of equations because they use only the non-zero coefficients and so require the least amount of storage.

2.1.2 Iterative Numerical PDE Solvers

We now describe in general terms of the computational problems posed by Partial Differential Equations and introduce several approaches for numerical iterative solutions.

2.1.2.1 Basic Iterative Methods

In the evaluation of basic iterative methods [34], we assume that A is a nonsingular nxn (sparse or dense) matrix, and that its diagonal entries a_{ii} are all nonzero numbers. Let b be a given column vector, then the system of linear equations can be expressed as $Ax = b$:

$$\sum_{j=1}^{n} a_{ij}x_j = b_i \quad for\ 1 \leq i \leq n \tag{2.4}$$

There are several basic iterative methods to solve the above equations, the simplest of which include Jacobi, Gauss-Seidel, and Successive Over Relaxation. Besides the above methods, a special asynchronous relaxation scheme called *chaotic relaxation*, which is particularly suitable in multiprocessor systems, also belongs to the basic methods. In any iterative scheme, one should note that the evaluation of the convergence, and the termination criterion have to be taken into account.

1. **Jacobi Method :** Since the diagonal entries a_{ii} of A are nonzero, the following iterative method is derived from (1):

$$x_i^{(k+1)} = \frac{-\sum_{j\neq i, j=1}^{n} a_{ij}x_j^{(k)} + b_i}{a_{ii}} \quad for\ i = 1, \ldots, n\ and\ k \geq 0 \tag{2.5}$$

where the $x_i^{(0)}$'s are initial estimates of the components of the unique solution of x. The above equation is the Jacobi iterative method.

2. **Chaotic Relaxation :** Partial differential equations can be solved by either synchronous or asynchronous methods. The Jacobi method belongs to the synchronous methods. In the asynchronous approach [10], 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 [13] to solve linear systems. In a chaotic relaxation scheme, practical constraints on the asynchronous behavior are imposed. It varies slightly from this definition in that, while an asynchronous algorithm imposes no restriction on how "old" a value may be (*i.e.*, how many iterations ago it was produced), chaotic relaxation requires that the updated value of a point be received within a fixed amount of time. In this sense, the chaotic relaxation method is more satisfactory in practical applications although the definition of asynchronous methods is mathematically more rigorous.

2.1.2.2 Advanced Iterative Methods

Besides basic iterative methods, many advanced iterative methods have been developed. Some of the algorithms conduct more complex procedures but still apply the above mentioned basic iterative methods.

Multigrid : The basic idea of the multigrid methods is to use different grid sizes to reduce different frequency domains of errors. One can deduce, based on Fourier analysis, that the fine grid relaxation can reduce the *high frequency*

error rapidly, while the coarse grid relaxation can efficiently reduce the *low frequency error* [26]. Many variants derived from the basic multigrid method can be used to solve PDE problems. One of the most interesting challenges is to find an efficient parallel multigrid algorithm and implement it in a parallel processing environment. Gannon and Rosendale have introduced the Concurrent Iteration(CI) algorithm in [16]. The idea of their work is to concurrently send the current residual errors down to the coarse grid levels and receive the current correction terms from the coarse grid levels.

2.2 Data-flow Computations

Advanced computing has been centered around the concept of parallel processing. For years, the design of computers has been based on the *von Neumann* execution model. In this execution model, a parallel programming language and an "*intelligent*" compiler are required to specify and detect implicit parallelism and convert the potential parallelism into a parallel format. Usually, this needs a complex analysis in the compiling process and can only achieve limited parallelism.

As an alternative to the conventional von Neumann model, data-flow principles have been proposed for efficient execution in multiprocessor systems [1]. Instead of relying on the conventional central program counter, the availability of data instead renders an instruction executable. The foremost advantage of this model is to offer the programmability needed to synchronize at runtime the many parallel processes on a large scale multiprocessor [6]. However, in spite of

the simplicity of these principles, much overhead is introduced in order to respect the functionality of execution [2]. Therefore, new techniques in compiler and architecture must be studied in order to reduce the execution time.

2.2.1 Basic Principles

In the course of parallel processing, *synchronization* and *parallelism* are the primary factors that influence final performance. Over the past few years, attempts have been made to design complex *Supercompilers* to optimize programs that are written in conventional languages. As an alternative to the control-flow model, data-flow principles offer run-time synchronization of operations based on their data dependencies and inherently parallel based upon the availability of arguments to cure the aforementioned two problems.

Data-flow principles can be characterized by two statements:

1. Operations execute only when all required operands are available.

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

In this way, a very large number of different tasks can be efficiently scheduled throughout the entire machine and the delay in network and memory access can be overlaid [6]. Data-flow computation starts from data-flow programs written in data-flow languages to describe algorithms. The compiler then converts the programs into directed graphs which consist of actors connected together with arcs. Arcs represent data dependencies between actors and carry tokens which

are the data values passed between actors. At the last step, the graphs are partitioned and then allocated to the various processors.

Once data-flow graphs have been constructed and allocated, the issue arises of how to actually execute the graphs. Although the graphs define the operations to be performed and how they are related to one another, they contain no information about arc capacity, precise firing rules, actor execution order, token consumption order, simultaneous actor execution orders, etc. Two approaches have thus been proposed regarding the interpretation of data-flow graphs:

1. Static Model: Under this model, proper matching of data-tokens must be observed by ordering the token production. The condition to fire an actor is only when its input arguments are ready but also when its output arcs are empty. The Acknowledgment Scheme [14], is an example of a static data-flow machine.

2. Dynamic Model: In this model, tokens are tagged with information pertaining to their context of creation. An actor is allowed to be executed only when an input token pair with identical tags (also called color) arrive the actor. The U-interpreter and I-structure (also called dynamic tagged-token data-flow system) [5] is an example of a dynamic data-flow machine.

Recently, a combination of static and dynamic execution model has also been studied by Abramson and Egan [2, 3].

2.2.2 Data-flow Languages

Programmability is the main goal in the design of data-flow languages. When data-flow computation is considered, a programming language must avoid such programming difficulty that is not easy to be overcome in the von Neumann model.

From the notion that data *flows* from one function entity to another, the essence of data-flow language design must have the following properties to meet the requirement of functional programming:

1. *Single assignment*: A variable may appear on the left side of an assignment only once within the area of the program in which it is active.

2. *Side effect free*: The sequence of execution constraints should be the same as data dependencies in programs.

3. *History insensitivity*: Functions in programs have no state variables that retain data from one invocation to the next.

This flow concept gives data-flow languages the advantage of allowing program definitions to be represented exclusively by graphs, while it would be extremely inefficient to do this if a program is written in a conventional language.

Many parallel programming languages have been developed in the past [1]. "SISAL" (Streams and Iterations in a Single Assignment Language) [27], developed at the Lawrence Livermore National Laboratory in cooperation with other institutions (Colorado State University, University of East Anglia, University of Manchester, Digital Equipment Corporation), is one of the high-level applicative

languages. In SISAL, six basic scalar types are defined: boolean, integer, real, double real, null and character. Data structures in SISAL may be records, unions, arrays or streams. Each basic data type has its associated set of operations, while record, union, array and stream types are treated as mathematical sets of values just as the basic scalar types. In particular, under the *forall* construct, these types can be used to support identification of concurrency for execution on highly parallel processors.

2.2.3 Structure Handling

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 have been designed to alleviate these problems:

1. *Heaps:* This scheme has been originally described by Dennis [14]. It represents an array as a tree of pointers. When a single array element is modified, only a sequence of pointer modification is involved. The functionality of operations is maintained at all times. For an n-element array, the cost of modifying a heap amounts to $log\ n$ which compares favorably with the cost n of entirely re-copying the original array. There are several disadvantages associated with the heaps. These include a sequentialization of some array operations, a centralization of array accesses, etc. [1].

2. *I-structures:* The I-structures were introduced by Arvind and Thomas [8] to permit pipelining between producer and consumer of a structure. In other

words, an array element should be readable before the entire array has been produced. This scheme can be implemented by associating "presence bits" with every cell of storage. When a request is made to a "full" cell, the data can be forwarded to the requestor. When the cell is found to be empty, a tag can be left in the cell indicating the forwarding address of the requestor. When the data is computed, it is directly forwarded to its intended destination.

3. *Token relabeling:* This scheme was described by Gaudiot [17] for U-interpretation principles. It has been shown that the notion of array can be entirely ignored at the lowest level of execution. Instead, the *tag* associated with each token is used as identification of the index of the array element. In other words, when an array A is created, its $A(i)$ element is tagged with i. In addition, special tag manipulating actors can generate such tokens as $A(F(i))_{[i]}$ for scatter or gather operations.

2.2.4 Data-flow Overhead

In spite of simplicity in data-flow principles, data-flow computing imposes overhead in processors, intercommunication networks, and data-flow computation model itself. Three kinds of overhead can be characterized as execution, communication, and computation overhead:

1. *Execution overhead*: It refers to the overhead which is generated inside a processing unit (PE). In order to execute an actor, a PE must perform a *match* function to gather a token with its partner token(s) and a *routing*

function to deliver results to destination actors. Special hardware must be implemented to support these functions.

2. *Communication overhead*: It refers to the time delay in the interconnection network of a system. In data-flow systems, data-tokens are transmitted from one PE to another PE through network. No matter what topology of the network is, the inherent communicational delay may degrade system performance.

3. *Computation overhead*: It refers to the actors in data-flow graphs which are not directly involved in the algorithm itself but instead participate only in operations related to data-flow execution model. For example, actors in a graph to produce iteration indices with appropriate tags are computation overhead.

Besides communication overhead is a general issue in parallel architecture design, execution and computation overhead is the overhead that is introduced in data-flow computing in order to respect functionality and guarantee correct execution. A classification of the overhead problem in data-flow computing can also been specified as *within an actor, between actors*, and *in the graph* according to data-flow graphs [29].

The overhead problem in data-flow computing is directly affected by the granularity issue in data-flow graphs. In a fine grain graph, each actor represents a single instruction. When it is implemented in a data-flow machine, each actor must be executed in a PE and data tokens flow around the network. Therefore,

execution and communication overhead is inevitably created. Besides, computation overhead is enforced to associate with the graph to insure the correct representation of data-flow graphs. When the grain size of a graph is increased, in other words, when an actor contains several operations instead of one single operation, the overhead can be drastically reduced. First, execution overhead can be reduced due to the fact that the total number of actors is reduced in a large granularity. Second, network traffic can be reduced due to the fact that many instructions will be executed in the same PE. Third, many actors (computation overhead) can be easily eliminated due to the fact that large granularity can simplify the manipulation of data within the same actor.

2.2.5 Data-flow Compilers

When high-level data-flow programs are translated into data-flow graphs, a compiler must analyze and implement the graphs into the target machine. The main processes of a compiler in data-flow computation are to partition, allocate, and schedule many operations for execution. Two important features of data-flow computing regarding the compiling process should be noted.

1. There is no partitioning issue in pure fine grain data-flow graphs. In fine grain computation, each actor is a self-contained operation and therefore the grain size is fixed. In this situation, a compiler only needs to allocate the many fine grain actors into the available processors of a data-flow machine.

2. The scheduling in dynamic data-flow computing is implicit. The time to *fire* an actor which depends on the availability of its arguments is not deterministic. In other words, the actor is *self-scheduled* and do not be prescheduled at compile time or run time.

When the grain size becomes variable, the partitioning issue is more important and much complex. The trade-off between grain size and overhead is considered as the most difficult issue in the partitioning process. A thorough analysis of this problem has been carried out in [30]. In Sarkar's thesis, macro data-flow scheduling (compile-time partitioning and run-time scheduling) and compile-time scheduling (compile-time partitioning and scheduling) problems are expressed as optimization problems, which have also been proved to be NP-complete problems. Besides the theoretical study in partitioning, in practical, one should note that it may not be possible to partition a real program into an arbitrary size mainly due to the limitation of program structures.

The allocation issue in data-flow computing is similar to conventional parallel processing. Both static and dynamic allocation can be studied. The static allocation, in general, is more desirable for data-flow computing due to the fact that it imposes less overhead. In dynamic allocation, unless efficient distribution techniques are developed, the run-time decision-making overhead will dominate the computation time since each computation process only needs relatively short execution time.

2.2.6 Data-flow Architectures

Data-driven architectures can be classified into three categories : static data-flow machines, dynamic data-flow machines, and hybrid machines.

2.2.6.1 Pure Data-flow Systems

Many data-flow machines have been developed during the last decade. A survey of the static and dynamic data-flow machines can be found in [31].

One important feature of static machines is that they cannot did not unfold loops or procedure calls. In the early 70s, Dennis and Misunas proposed a static model, together with a data-flow language VAL [2, 14], is an example of the static machines.

Dynamic machines unfold the loops and attach tags to each token in order to exploit more parallelism at execution time. Monsoon machine [7] in U.S.A. and Sigma-1 [32] in Japan are examples of dynamic data-flow machines.

2.2.6.2 Hybrid Systems

The idea of combining data-flow and control-flow, so as called hybrid systems, has been adopted by various research groups in order to improve performance. A recent development of hybrid systems can be found in [2].

1. **Dataflow/von Neumann Hybrid Architecture** [2] : The main feature of this machine is to obtain the advantages of data-flow principles which are essential for tolerating latencies and synchronization costs. It is incorporated into a *parallel machine language* which fits on top of a von Neumann

base. *Scheduling quanta* is the basic scheduling unit which is bounded at compile time.

2. **Variable-grain Data-flow Architecture** [29] : This project has been studied at USC. The architecture of the machine is organized in a hierarchical fashion. It respects the U-interpreter model at the higher level, but composes powerful hybrid processing elements at the lower level. In each processor, there are Matching Store, Instruction Fetch, ALU, and Token Formation Units. Each PE has its own local memory and is connected by a hypercube network.

Chapter 3

Parallel Multigrid Algorithms

The multigrid approach represents a wide family of methods to efficiently solve problems in many different areas. The basic idea of a multigrid method is to use different grid sizes to reduce different frequency domains of errors.

Many variants have been derived from the basic multigrid method used to solve PDE problems. One of the most interesting challenges is to find an efficient parallel multigird algorithm and implement it in a parallel processing environment. The main difficulty of implementation of a standard general V-cycle multigrid (GVM) algorithm on parallel computers is the sequentiality of the execution of the V-cycle [11]. Therefore, specialized parallel algorithms for multigrid methods must be designed to cure the problem.

Gannon and Rosendale first introduced the Concurrent Iteration(CI) algorithm in [16]. The idea of their work is to concurrently send the current residual errors down to the coarse grid levels and receive the current correction terms from the coarse grid levels. The structure of the algorithm can be perfectly mapped on large grain systolic arrays and be executed by data-driven or pipeline models

[28]. In Greenbaum's approach [23], the residual error is first distributed to all of grid levels, and a special operation is then used to collect those correction terms and bring them back to the finest grid level. Recently, a method of using multiple coarse grid corrections has been studied by Frederickson and McBryan [15]. A recent survey of parallel multigrid algorithms can be found in [12].

In this chapter, we introduce a highly parallel and efficient multigrid algorithm which can be implemented on a data-driven machine. This includes the description of the algorithm, the data-driven architecture, as well as the theoretical analysis and the results of experiments of the algorithm on the desired machine.

3.1 Multigrid Methods

In this section, we first review the essentials of the multigrid algorithms that are intended to be implemented.

3.1.1 Multigrid Methods

Specialized parallel algorithms for multigrid methods are needed to cure the sequentiality of the original algorithms. In the following, we first discuss the basic multigrid methods and some proposed parallel multigrid algorithms.

3.1.1.1 General V-cycle Multigrid Algorithms

Multigrid methods were originally applied to simple boundary value problems which arise in many physical applications. For simplicity and for historical reasons, these problems provide a natural introduction to multigrid methods. As an example, consider the following two-point boundary value problem which could describe the steady-state temperature distribution in a long uniform rod. It is given by the second-order ordinary differential equation

$$-U''(x) + \sigma U(x) = f(x), \quad 0 < x < 1, \quad \sigma \geq 0 \tag{3.1}$$

subject to the boundary conditions $U(0) = U(1) = 0$. Although we can handle this problem analytically, it is instructive to develop numerical methods for its solution. Many such approaches are possible. The simplest one is the finite difference method. The domain of the problem is partitioned into N subintervals by introducing the grid point $x_i = ih$, where $h = 1/N$ is the constant width of the subintervals. One way to improve a relaxation scheme on the grid points is to use a good initial guess. A well-known technique for obtaining an improved initial guess is to perform some preliminary iterations on a coarse grid and then use the resulting approximation as an initial guess on the original fine grid. Relaxation on a coarser grid is less expensive since there are fewer unknowns to be updated. Also, the coarser grid will have a marginally improved convergence rate since the convergence factor depends on the step size h. This suggests that the coarse grids

are worth considering. A general V-cycle multigrid(GVM) algorithm consists of three main steps:

1. Relaxation: it refers to some basic iterative methods, like weighted Jacobi, which can reduce the error on each grid level.

2. Restriction: this operation forms an error correction term from the fine grid level to the coarse grid level, when the fine grid level has completed the relaxations.

3. Interpolation: it projects the correction term of error from the coarse grid level back to the fine grid level, when the coarse grid level has corrected the error.

There are many strategies to perform each of these three main operations. In general, the GVM algorithm on multigrid levels can be expressed as the following procedure (see Fig. 3.1):

1. Relax n_1 times on the fine grid level with the initial guess.

2. Compute the residual error on the fine grid level.

3. Restrict the error term into the coarse grid level.

4. Relax n_2 times on the coarse grid level with the initial guess.

5. Interpolate the correction term back to the fine grid level.

6. Relax n_1 times on the fine grid level.

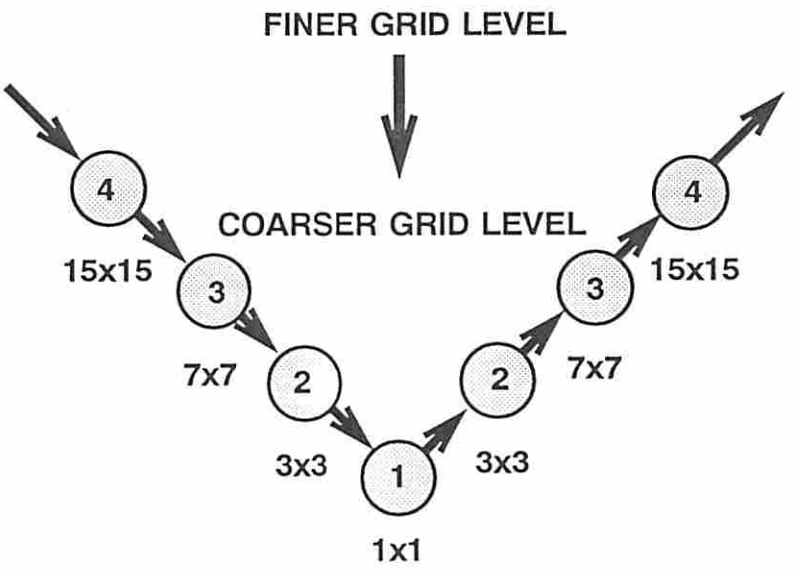

Figure 3.1: Multigrid Principles.

7. Check the stopping criterion and decide either to do another V-cycle or to stop the execution.

From the above, it transpires that the major computation demands in the GVM algorithm come from the relaxations on each grid level and the calculation of the residual error. This is also true for the following parallel multigrid algorithms.

3.1.2 Parallel V-cycle Multigrid Algorithms

As mentioned in the previous section, there are several approaches to achieve parallelism for multigrid methods.

In Greenbaum's approach [23], the residual error is first produced from the finest grid level and restricted to all the coarse grid levels. The whole system is then solved by concurrent relaxation on each grid level. After several concurrent

relaxations, the correction term is collected from the coarse grid levels to the finest grid level. A special collection method is also developed to combine each solution on each grid level with the approximation of the finest grids.

The experimental results from the above algorithm show a high utilization of the computer system due to the concurrent relaxations among all the grid levels. However, the number of V-cycles needed in this parallel algorithm is greater than for the general V-cycle multigrid algorithm. This indicates that the rate of convergence for this algorithm is not better than that of the GVM algorithm.

3.2 The PVM algorithms and Architectures

The purpose of developing this new parallel V-cycle multigrid(PVM) algorithm is to exploit more parallelism and increase the convergence rate. The architecture of the data-driven system is also designed to support the PVM algorithms.

3.2.1 The PVM Algorithms

The basic idea of the PVM algorithms is to perform additional relaxations on the fine grid level, while the coarse grid level is relaxing. In other words, all of the finer grid levels have additional relaxations while the V-cycle execution is on the coarser grid level. Indeed, the PVM algorithm has used those idle processors to achieve highly parallel execution and faster convergence rate.

3.2.1.1 The Procedure of the PVM Algorithms

In the following, the procedure of the PVM for a two-level V-cycle is described:

1. Relax ν_1 times on the fine grid level with the initial guess.
2. Compute the residual error from the fine grid level.
3. Restrict the error term into the coarse grid level.
4. Relax additional ν_2 times on the fine grid level.
5. Relax ν_1 times on the coarse grid level.
6. Interpolate the correction term back to the fine grid level.
7. Relax ν_1 times on the fine grid level again.
8. Check the stopping criterion and decide either to do another V-cycle or to stop the execution.

Note that steps 4. and 5. in the above procedure can always be executed concurrently in an actual implementation. The same concept of parallel execution can also be extended to each level in a multi-level V-cycle. There are several remarks worth mentioning:

- It is a general method; any kind of relaxation scheme, restriction, and interpolation can be applied to the algorithm.
- It is a highly parallel algorithm; most of the grid levels are running at the same time.

- It will yield high system efficiency; the additional relaxations are on the finer grid levels and the finer grid levels utilize the idle processors to execute the relaxations.
- It is a simple method; no new operation is needed to support the algorithm.
- It is a fast method; the convergence rate has improved significantly.

3.2.2 Convergence analysis for PVMs

The convergence theory for the GVM algorithms has been studied extensively. Here, we analyze the convergence rates of the PVMs and make a comparison with the GVM algorithms.

3.2.2.1 Two-grid analysis

First, we assume that U^h is the exact solution to a discrete PDE, where h is the discretization parameter, V^h is the computed approximation to the exact solution, I_h^{2h} is the restriction operator from the fine grid level h to a coarse grid level 2h, and I_{2h}^h is the interpolation operator from the coarse grid level 2h to a fine grid level h. We also use $e^h = U^h - V^h$ and Q as the relaxation matrix, whose spectral radius is less than 1.

The V-cycle of the GVM algorithm, for the case of 2 levels, can then be described as:

1. fine grid relaxation with ν_1 times : $V^h \leftarrow Q^{\nu_1} V^h$
2. restriction : $f^{2h} = I_h^{2h}(f^h - A^h V^h)$

3. coarse correction : $V^{2h} = (A^{2h})^{-1} f^{2h}$

4. interpolation : $V^h \leftarrow V^h + I_{2h}^h V^{2h}$

5. fine grid relaxation with ν_1 times : $V^h \leftarrow Q^{\nu_1} V^h$

Combining the above five steps, we can write the following equation:

$$V^h \leftarrow Q^{2\nu_1} V^h + Q^{\nu_1} I_{2h}^h (A^{2h})^{-1} I_h^{2h} (f^h - A^h Q^{\nu_1} V^h) \tag{3.2}$$

We also can substitute U^h for V^h in above equation:

$$U^h \leftarrow Q^{2\nu_1} U^h + Q^{\nu_1} I_{2h}^h (A^{2h})^{-1} I_h^{2h} (f^h - A^h Q^{\nu_1} U^h) \tag{3.3}$$

By subtracting the last two equations, we can have the error expressed as:

$$e^h \leftarrow (Q^{2\nu_1} - Q^{\nu_1} I_{2h}^h (A^{2h})^{-1} I_h^{2h} A^h Q^{\nu_1}) e^h \tag{3.4}$$

Similarly, the error in the PVM algorithm can be expressed as:

$$e^h \leftarrow (Q^{2\nu_1+\nu_2} - Q^{\nu_1} I_{2h}^h (A^{2h})^{-1} I_h^{2h} A^h Q^{\nu_1}) e^h \tag{3.5}$$

where the ν_2 is the number of extra relaxations on the fine grids.

3.2.2.2 Spectral radius of iteration operator

The iteration operation: $(Q^{2\nu_1+\nu_2} - Q^{\nu_1} I_{2h}^h (A^{2h})^{-1} I_h^{2h} A^h Q^{\nu_1})$ in the last equation decides the convergence rate of the PVM algorithms. In fact, to achieve the

fastest convergence rate one needs to optimize the choice of the values of Q, ν_1, ν_2, I_{2h}^h, and I_h^{2h}. In the following, we will compute the spectral radius of this operator with different values of ν_2, in the simplest case of the two grid levels.

$$1/16 \quad X \quad \begin{vmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{vmatrix}$$

Figure 3.2: A Full Weighting Restriction

$$1/4 \quad X \quad \begin{vmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{vmatrix}$$

Figure 3.3: A Bilinear Interpolation

We first choose the damped Jacobi method as the relaxation scheme with a fixed relaxation parameter w=2/3 :

$$U_{i,j}^{(k+1)} = (1-w) \times U_{i,j}^{(K)} + w \times [(U_{i+1,j}^{(k)} + U_{i-1,j}^{(k)} + U_{i,j+1}^{(k)} + U_{i,j-1}^{(k)} + h^2 f_{i,j})/4], \quad (3.6)$$

the full weighting restriction (Fig. 3.2), bilinear interpolation (Fig. 3.3), and ν_1=1. Then, we adapt the two-grid analysis developed in [33] to calculate the spectral radius for different values of ν_2.

ν_2	0	1	2	3	4
Spectral Radius	0.6639	0.0434	0.0236	0.0173	0.0188
ν_2	5	6	7	8	
Spectral Radius	0.0211	0.0232	0.0254	0.0277	

TABLE 3.1 : Spectral Radius with Different Values of ν_2.

The results of the calculations are shown in Fig. 3.4 and Table 3.1. The case when no extra relaxation is added (ν_2=0), represents the standard GVM algorithm. The results indicate that with the addition of extra ν_2-iterations the spectral radius rapidly decreases, and then slowly starts increasing. A minimum is attained for the value of ν_2=3. Extrapolating this two-grid analysis to the complete V-cycle, we can expect a similar situation where the optimal rate of convergence is achieved when only a small number ν_2 of extra iterations is used. This has been confirmed by the numerical experiments, see section 4.

3.2.3 Architectures for Multigrid Methods

There are many architectures proposed for the multigrid methods. Among those, the form of a pyramid seems to be the most natural structure for the V-cycle multigrid algorithms [11]. The regularity and simplicity of the structure and the local and direct communication paths make the pyramidal architecture suitable for parallel processing.

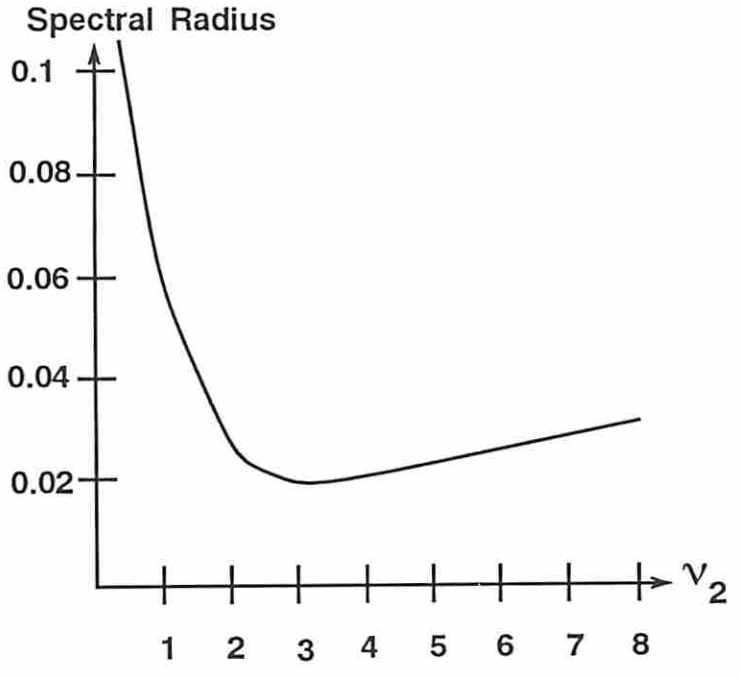

Figure 3.4: Two-Grid Analysis for PVM methods .

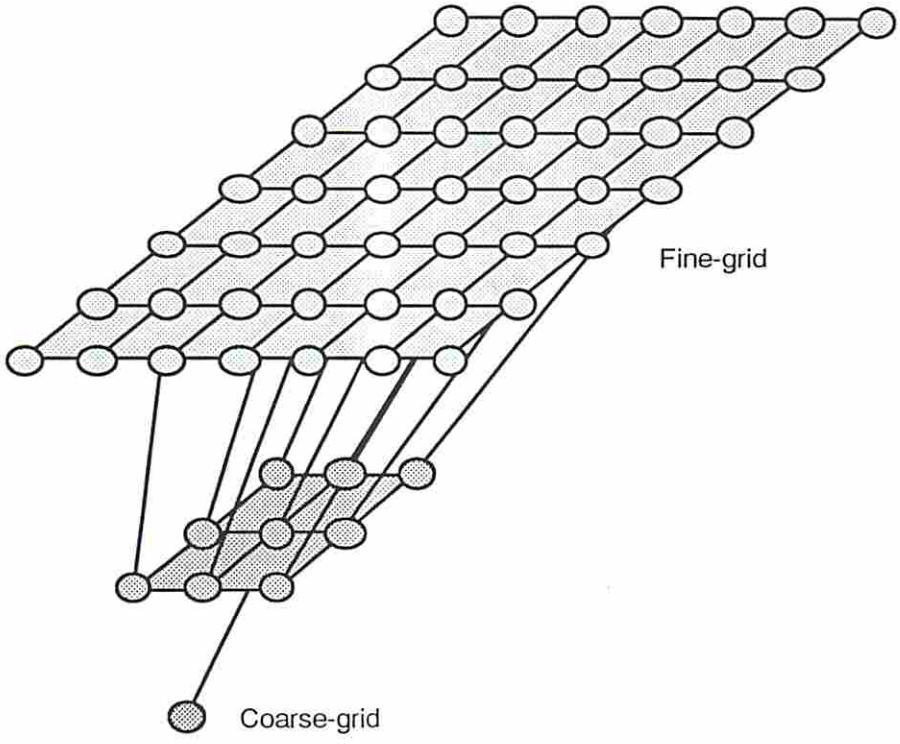

Figure 3.5: A Pyramidal Data-Driven Achitecture Structure.

3.2.3.1 The Pyramidal Architecture for the PVMs

The multigrid architecture, shown in Fig. 3.5, is proposed to form a pyramid-like structure, which is executed under the data-driven model. Each processor in the system has a powerful relaxation mechanism and is connected by a mesh interconnection network within each layer. There are also some direct communicational paths between two adjacent layers and the communications are automatically handled by the data-driven execution principles. Note that this system can be easily utilized by the PVM algorithms: One grid level corresponds to one layer and each grid point corresponds to one processor in each layer.

3.2.3.2 Execution on the Data-Driven System

When a data-driven system executes a multigrid algorithm, one special feature of the system is the asynchronous execution due to the data-driven principles. This feature makes the system not only fit for all multigrid algorithms but also very suitable for the class of asynchronous multigrid algorithms [25]. For an asynchronous algorithm, one can either map one grid point to a processor, if the number of processors in system is sufficient, or map several grid points to a processor, if the number of processors in the system is limited. Indeed, release of the constraints of synchronization in those asynchronous algorithms will perfectly match the data-driven machines, since they have no explicit synchronization mechanism.

3.3 Experimental Results and Comparison

The PVM algorithm developed in the previous sections is used to solve several problems with different step sizes. The convergence rate of PVM and the resources utilization are compared with the standard V-cycle multigrid method.

3.3.1 Architecture Assumption

The pyramidal data-driven architecture, designed in the previous section, has been adopted for the multigrid algorithms. In the finest level, the number of processors is exactly the same as the number of grid points in the discretization model. In the second level, the number of processors will be only one fourth that

on the upper (*i.e.*, finer) level, and so on for each next coarser level. The processors are connected as a mesh interconnection network in each level, and there are some direct connections between different levels for each pair of corresponding processors. When the desired grid level is operating, each processor in this level receives data from its four neighbor processors, then performs the relaxations and the necessary restrictions and interpolations for the coarser and finer grid levels.

3.3.2 Experiments

As the model problem we chose the Poisson's equation on the unit square (0,1)×(0,1), with Dirichlet boundary conditions, and different load functions $f(x,y)$:

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = f(x,y), \qquad 0 < x < 1,\ 0 < y < 1. \tag{3.7}$$

3.3.2.1 Problem definition

In the experiments we first choose the exact solution u to be:

1. $u(x,y) = sin(x)sin(y)$, and
2. $u(x,y)$ =random generated (as a check on the effect of smoothness of the solution).

Then, we generate the proper right hand side of the equation, $f(x,y)$, and the boundary condition. The uniform step size h in both coordinate directions for the finest grid level is chosen as:

1. 1/16 (the corresponding matrix size is 256×256), or

$u(x,y) = sin(x)sin(y)$ $(h = 1/64)$									
ν_2	0	1	2	3	4	5	6	7	8
Number of V-cycles	14	10	7	8	10	13	15	18	20

TABLE 3.2 : Number of V-cycles with Extra Relaxations (ν_2) on Finer Grids.

2. 1/32 (the corresponding matrix size is 1024×1024), or

3. 1/64 (the corresponding matrix size is 4096×4096).

Both of the parallel V-cycle multigrid(PVM) and the general V-cycle multigrid(GVM) methods are tested using the initial guess u=0 and stopping criterion requiring that the residual for two consecutive iterations satisfies:

$$\frac{\|u^{k+1} - u^k\|}{\|u^{k+1}\|} \leq 10^{-6} \tag{3.8}$$

In the following discussion, we will use the term *residual error norm* to refer to the quantity of the above convergence test.

3.3.2.2 Experimental results

We first test a six-level V-cycle (h=1/64) PVM algorithm with different number of ν_2 extra relaxations on the finer grid levels. The results are reported in Table 3.2 and Fig. 3.6. The results show that only 7 V-cycles are needed when the PVM algorithm is employed with ν_2=2 while the standard GVM algorithm requires twice that many V-cycles.

The detailed history of convergence, *i.e.* how the residual errors are reduced in every V-cycle for both the PVM(ν_2=2) and the GVM algorithms, is shown

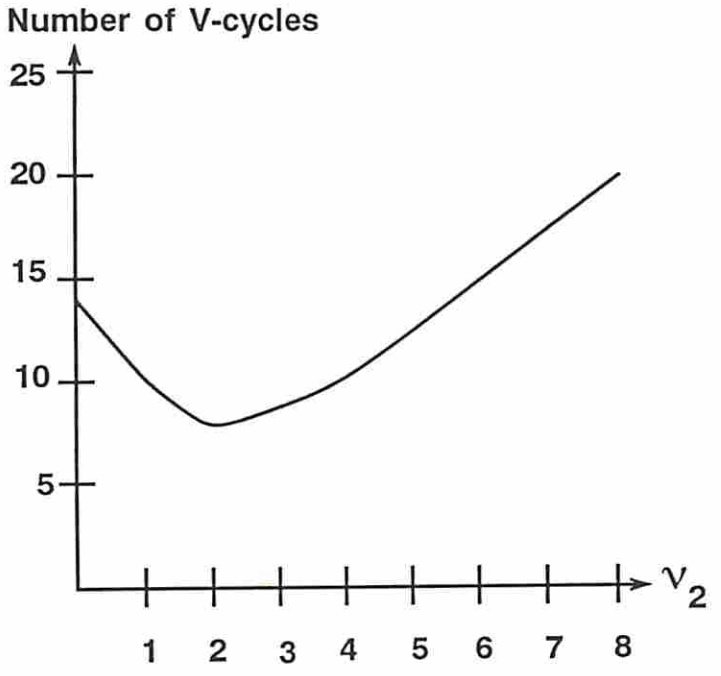

Figure 3.6: Number of V-cycles vs. Extra Relaxations in the PVM.

$u(x,y) = sin(x)sin(y)$ $(h = 1/64)$		
V-cycles	PVM (ν_2=2)	GVM
1	2.25×10^{-2}	5.73×10^{-2}
2	3.25×10^{-3}	1.69×10^{-2}
3	5.94×10^{-4}	6.48×10^{-3}
4	1.06×10^{-4}	2.69×10^{-3}
5	2.01×10^{-5}	1.15×10^{-3}
6	3.62×10^{-6}	4.99×10^{-4}
7	7.20×10^{-7}	2.18×10^{-4}
8		9.56×10^{-5}
9		4.19×10^{-5}
10		1.84×10^{-5}
11		8.10×10^{-6}
12		3.56×10^{-6}
13		1.56×10^{-6}
14		6.90×10^{-7}

TABLE 3.3 : Residual Error in PVM (2 extra relaxations) and GVM.

in Table 3.3 and the curves are drawn in Fig. 3.7. It should be noted that in a multi-level V-cycle, not at every grid level arbitrary, many extra relaxations can be performed. For example, at the second coarsest grid level, one can have only one extra relaxation. This is due to the limitation posed by the execution time of the relaxation on the coarsest grid level.

In Table 3.4, we show the residual error norms, the absolute error norm, and the number of V-cycle iterations with different step sizes for $u(x,y) = sin(x)sin(y)$. The results for $u(x,y) =$ random generated are shown in Table 3.5.

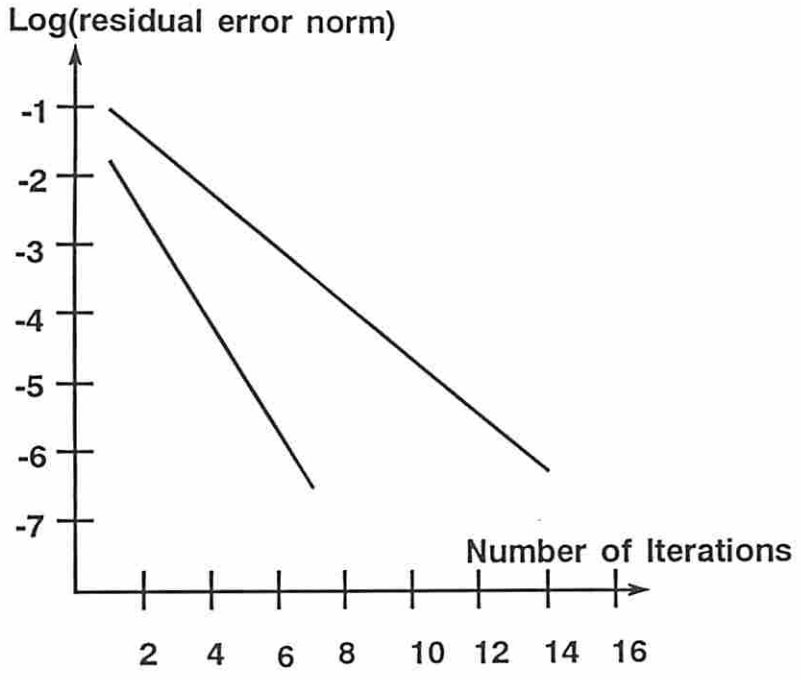

Figure 3.7: Number of Iterations vs. Error Norm (h=1/64).

$u(x,y) = sin(x)sin(y)$		PVM	GVM
h=1/16	Residual Error Norm	2.31×10^{-7}	9.2×10^{-7}
	Absolute Error Norm	9.49×10^{-7}	1.67×10^{-6}
	Number of V-cycles	8	14
h=1/32	Residual Error Norm	9.22×10^{-7}	8.59×10^{-7}
	Absolute Error Norm	3.89×10^{-6}	2.1×10^{-6}
	Number of V-cycles	7	14
h=1/64	Residual Error Norm	7.2×10^{-7}	6.9×10^{-7}
	Absolute Error Norm	3.65×10^{-6}	2.38×10^{-6}
	Number of V-cycles	7	14

TABLE 3.4 : Number of V-cycles in Various Step-sizes (h).

$u(x,y)$ =Random Generated		PVM	GVM
h=1/16	Residual Error Norm	2.95×10^{-7}	9.17×10^{-7}
	Absolute Error Norm	5.11×10^{-7}	7.96×10^{-7}
	Number of V-cycles	8	14
h=1/32	Residual Error Norm	7.81×10^{-7}	7.26×10^{-7}
	Absolute Error Norm	5.56×10^{-7}	9.26×10^{-7}
	Number of V-cycles	8	14
h=1/64	Residual Error Norm	3.32×10^{-7}	6.55×10^{-7}
	Absolute Error Norm	2.18×10^{-7}	1.02×10^{-7}
	Number of V-cycles	9	14

TABLE 3.5 : Number of V-cycles in Various Step-sizes (h).

3.3.3 Comparison

Based on the experimental results, we compare the PVM algorithm with the GVM algorithm in both convergence rate and system utilization.

3.3.3.1 Convergence rate

The convergence rates of these two algorithms are represented by the number of V-cycle iterations in each case. From Tables 3.4 and 3.5, we conclude that the PVM algorithm requires half the number of V-cycles needed by the GVM to reach the same order of accuracy, which means that the PVM algorithm is twice faster than the GVM algorithm.

3.3.3.2 System utilization

To compare the resources utilization of these two algorithms in two dimensional problems, we define W to be the cost of performing one relaxation in the coarsest

grid level that has only one processor, and neglect the cost of intergrid restrictions and interpolations. Then, the GVM algorithm costs total of

$$(2N-1)\ \ W \tag{3.9}$$

work units to perform one single V-cycle, where $N = \sum_{i=1}^{n}(2^i - 1)^2$ and n is the number of grid levels. On the other hand, the PVM costs

$$((2+\nu_2)N - \nu_2 - 1)\ \ W \tag{3.10}$$

work units to perform one V-cycle (assuming that every grid level has ν_2 extra relaxations except the coarsest level).

Thus, the system utilization for PVM compared to GVM is approximately:

$$\frac{2+\nu_2}{2} \tag{3.11}$$

In the numerical experiments reported here, the optimal ν_2 (in the sense of the rate of convergence) is equal to 2. Hence, in our case, where two algorithms are executed on parallel computers, the PVM algorithm has twice as efficient system utilization as the standard GVM algorithm.

3.4 Discussion

In this chapter, we have presented a highly parallel multigrid algorithm for parallel computing. We have also demonstrated how the algorithm can improve the convergence rate and system utilization in a pyramidal structure.

Evaluating the performance in terms of scientific computing, the following remarks can be made about our approach:

1. From the algorithm point of view, the PVM is a simple, general, efficient, and parallel algorithm. It can cooperate with a wide class of multigrid operators, including the relaxation, restriction, and interpolation.

2. From the architecture point of view, the data-driven execution model proposes massive parallelism at the runtime, and no explicit synchronization mechanisms for the processes. These features make it very attractive for a multiprocessor system. Some efficient execution mechanisms for the basic relaxations can be developed inside the processors to speed up the execution and achieve high performance.

In this chapter, we have demonstrated several variants of the PVM algorithms and the integration with an efficient data-driven system. The two-grid analysis of the spectral radius of the iteration operator have also been analyzed to prove the efficiency of our PVM algorithm.

Chapter 4

Macro Data-flow Graph Generators

Variable-grain data-flow computing has been advocated in the past decades to exploit the parallelism at various levels of granularity [18, 29] to reduce overhead and execution time. The major research issue of this approach has been to generate various resolutions of actors in data-flow graphs for data-flow systems. While theoretical analysis has proved that to decide an optimum grain size is an NP-complete problem [30] and also it is impractical to partition a real program into an arbitrary size, the objective of this chapter is to develop several partitioning schemes to generate macro data-flow graphs. These schemes will then be applied to computationally intensive numerical applications and compare the performance of different grain sizes of actors and tokens in data-flow graphs. To this end, we have concentrated on multigrid algorithms for solving PDEs and compared the performance of different partitioning schemes on the graphs of this algorithm. Our research has therefore been aimed in the following directions: specification of the partitioning schemes, translating multigrid algorithms from

high-level data-flow languages into macro data-flow graphs, designing of an appropriate simulation environment, and evaluating the performance.

In this chapter, we will also conduct a deterministic simulator and show how macro data-flow graphs can be executed and thus deliver high efficiency. We start from implementing multigrid algorithms in applicative as well imperative high level languages to compare the codes. Then, the codes are translated into data-flow graphs described by assembly languages. After the graphs are partitioned into macro graphs, the execution is simulated. The simulation will gather the number of static actors in graphs and the number of actual executional instructions. The execution time of various problem sizes with unlimited numbers of processing elements and with a single processing element will be simulated and compared. From the simulation results, the ideal speedup of each partitioned graphs will also be predicted.

4.1 Macro Data-flow Graphs

In fine grain data-flow computation, high overhead needed to respect the functionality in execution will result in poor performance at low levels of parallelism. Indeed, to execute fine grain graphs (where each actor represents a single instruction), execution, communication, computation overhead must be enforced to associate with actual computation actors to insure the correct execution. Therefore, overhead problem will be inevitably created and will degrade performance.

When the grain size of a graph is increased, in other words, an actor now represents several operations instead of one single operation in graphs, the overhead

problem can be easily alleviated. The concept of *macro-actors* (several operations are grouped into a single actor) described by Gaudiot and Ercegovac [19], has been shown to bring a solution to the problems in a fine grain computation model. 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 actors (with larger granularity) may reduce the available parallelism in programs and increase memory and communication latency. Hence, forming macro-actors from fine grain micro actors is a trade-off between latency and parallelism.

4.1.1 Theoretical Work

To decide an optimum size of macro actors from fine grain data-flow graphs has been theoretically proved as an NP-complete problem [30]. In [19], a simple mean-value analytical model of *variable resolution* data-flow has illustrated an optimal resolution based on the criterion of minimizing execution time. The results also described the average performance of a program for a given macro-actor size. However, they did not address the issues in actual partition of a real program which may not be likely partitioned into arbitrary sizes.

4.1.2 Dynamic Partitioning

In general, partitioning data-flow graphs can be characterized at either compile time (static) or run time (dynamic). In *Dynamic Structured Dataflow* (DSDF)

system [22], a partitioning scheme has been proposed to partition fine-grain data-flow graphs at run-time. In DSDF, instead of executing actors corresponding to individual instructions, the system supports Execution Processing (EP) units which are essentially sequential, and von Neumann-like execution threads. When a program is executed, a system scheduler activates such EPs which will progress along paths in the data-flow graphs as far as data availability permits, while the extent of execution is not known at compile time. The design of EP units in the system not only realizes run-time partitioning but also allows the grain of parallelism to remain fine. However, two architectural supports must be developed in order to realize DSDF execution rules. First, a specialized facility must be dedicated to *Context Initiation* (CI) which has been identified as the key overhead cost in DSDF. Second, a significant *look-ahead* mechanism must be employed by EPs for fetching operands in execution.

4.1.3 Static Partitioning

In static partitioning, two automatic partitioning schemes have been studied in [30]. First, in the *macro-dataflow* approach, which is based on compile-time partitioning and run-time scheduling, an objective function, F(II), which gives the cost of partitioning *directed acyclic graphs* (dags) graphs, has been developed. An approximation algorithm to determine the optimal partition based on F(II) is also demonstrated. Second, in *compile-time scheduling* approach which is based on partitioning and scheduling at compile-time, it has shown the problem of determining the optimal scheduling algorithm is NP-complete in the strong sense.

An efficient scheduling algorithm which is close to an optimal solution is presented to demonstrate how an automatic partitioning and scheduling can be realized in multiprocessor systems.

4.1.4 User's Directive Partitioning

Another static partitioning is formed as user's directive partitioning. At this level of partition, the partitioning process takes the advantage of programmer's knowledge of the structures in programs and produces a better partition. Among many interactive processes between users and compilers, one of the most efficient techniques in user's directive partitioning is *Vectorization*. In SIGMA-1 data-flow machine [32], the system allows users to define several kinds of vector operations such as: *for (i=0; i* $\leq$ *limit; i=i+1) a[i] = b[i] + c[i];* or *x[index[i]] = y[index[i]] * x[i] + w[i].* However, a set of *Structure Processing Elements* (SEs) is needed in the system for handling complex data structure in vector operations. In [4], a *delayed updating* scheme is proposed. In this scheme, a buffering mechanism is developed to save several vectors in structured memory units and allows the scheduler to decide the executional sequence of vectors.

4.2 Macro Data-flow Graphs Generation

When data-flow programs are compiled into graphs, a trade-off between overhead and parallelism becomes an important issue. Indeed, if there are fewer actors and tokens (several data elements may be contained in a token (*i.e., fat* token)) in

the system, the degree of parallelism may be reduced [2]. In the course of this research, three-phase partitioning schemes are described in order to generate macro (variable-granularity) data-flow graphs. The first-phase of partitioning scheme is called *Actor-based* partitioning which is based on the concept of lumping sequential micro operations into macro actors. The second-phase of partitioning is *Tag-based* partitioning which gathers many tag-manipulational operations into macro actors. The third-phase of partitioning, called *Token-based* partitioning, exploits *Vectorization* concepts from conventional computers into data-flow computing. In this three-phase partitioning schemes, not only sequential actors but also parallel actors will be lumped together as macro actors. Therefore, the degree of parallelism in partitioned graphs will be reduced due to the increased grain size. In this research, we also choose SISAL to express algorithms before we translate them into macro data-flow graphs.

4.2.1 Basic Data-flow Instruction Sets

In fine grain data-flow computation, actors only perform a simple operation and are micro actors. The basic micro actors (fine grain instruction sets) can be characterized into arithmetic, logic, data structure, tag, and control operations. In the following, we introduce the basic micro actors:

(1) Arithmetic Instructions:

```
ADD a, b, c    ;a=b+c          SUB a, b, c ;a=b-c
MUL a, b, c    ;a=b * c        DIV a, b, c ;a=b / c
INC a          ;a=a+1          DEC a       ;a=a-1
SQU a          ;a=a*a          MOD a, b, c ;a= b mod c
FLR a, b       ;a= floor of b  CLG a, b    ;a= ceiling of b
```

```
CMP a, b, c   ;a= -1 if b > c   FLR a, b    ;a= floor of b
CMP a, b, c   ;a=  0 if b = c   TRC a, b    ;a= truncate of b
CMP a, b, c   ;a=  1 if b < c   NOP         ;no operation
```

(2) Logic Instructions:

```
AND a, b, c   ;a= b AND c       OR a, b, c  ;a= b OR c
XOR a, b, c   ;a= b XOR c       NOT a, b    ;a= NOT b
```

(3) I-structure Handling Instructions:

```
CRE(i)  ; create an array with dimension i
APP(i)  ; append an element to an array with dimension i
SEL(i)  ; select an element to an array with dimension i
DEL(i)  ; delete an array with dimension i
```

(4) Stream Generation and Tag Manipulation Instructions:

```
RTG     ; read tag              WTG     ; write tag
DDD     ; D actor               IDD     ; inverse D actor
LLL     ; L actor               ILL     ; inverse L actor
GST     ; generate stream tokens with iteration tags
```

(5) Control Instructions:

```
TRU     ; true gate             FAL     ; false gate
SWI     ; switch gate           MRG     ; merge gate
TPR     ; token print
```

To represent actors and tokens, we have adopted the U-interpreter ($[u, c, s, i]$) 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. We also use the iteration number i to represent the indices in arrays. For example, an element, A[1][2][3], in array A can be represented as $Value_{[u,c,s,(123)]}$.

4.2.2 Actor-based Partitioning

Once fine-grain data-flow graphs are conducted, we start from the first-phase partitioning. In actor-based partitioning, two kinds of actors are lumped into macro actors: The sequential data-dependent actors and actors that perform a unique function.

4.2.2.1 Sequential Actors Lumping

In this process of partitioning, a straightforward strategy for creating macro-actors is first selected. We inspect the data-flow graphs of an algorithm and then lump the sequential portions of the graphs into macro-actors. The primary benefit of this approach is that one can easily reduce the inter-PE and intra-PE communication overhead without losing parallelism at execution time. In this phase of partitioning, fat tokens, which contain several data in a token, can also be avoided and hence no extra latency will be introduced. In the following, we will demonstrate how this scheme can be applied to data-flow graphs:

```
for i in 1, h cross j in 1, h
    return array of A[i][j][k]
       A[i][j][k]+A[i][j-b][k]
end for ;
```

For example, to perform the above SISAL statements whose data-flow graphs are shown in Fig.4.1, the value of A[i][j-b][k] must be obtained and then the index(tag) of the value must be changed in order to add with the value of A[i][j][k]. From the graph, we find that a sequential path which consists of four sequential actors exists in this function. Hence, we form a macro-actor to perform the

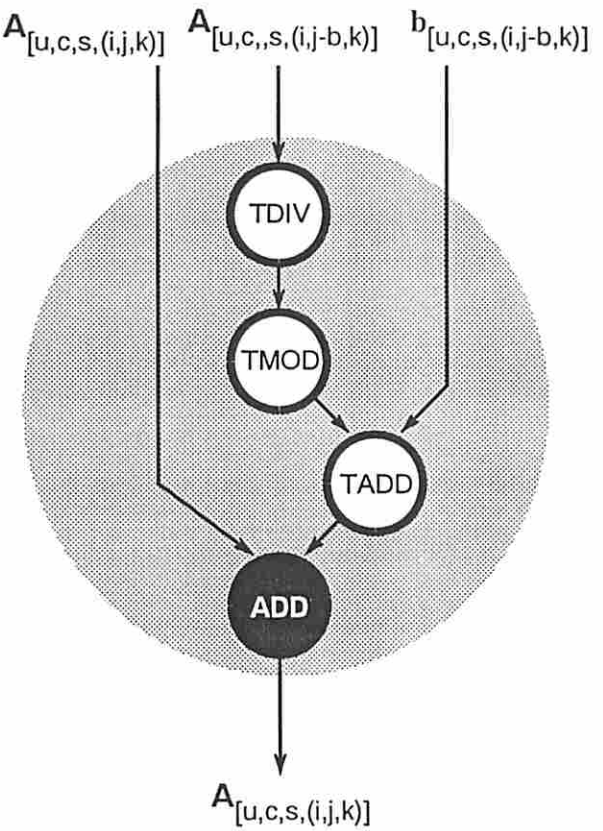

Figure 4.1: A Macro Actor by Lumping Sequential Actors

four sequential instructions. In Fig.4.2, another sequential lumping on a more complex example is demonstrated. The corresponding SISAL code, which shows that how elements in an array are dispatched to four destinations to perform different functions, is as follows:

```
for j in 1, h
  returns array of
    if (mod(i,2k) = 0 & mod (j,2k) = 0)
      then  Function1()          ;i and j are even
      elseif (mod(i,2k) = 0 & mod (j,2k) ~= 0)
        then  Function2()        ;i is even, j is odd
        elseif (mod(i,2k) ~= 0 & mod (j,2k) = 0)
         then  Function3()       ;i is odd, j is even
         else  Function4()       ;i and j are odd
    end if
end for
```

In the above examples, the total execution time can be actually reduced by shortening the length of the sequential path in this macro-actor formatting scheme. For example, if we have a processing unit which contains 4-stage instruction pipelines and if each actor takes 1 time units for each stage, the graph in Fig.4.2, where the longest sequential path has a length of 7, needs a total of 28 time units to execute the sequential actors. On the other hand, it only takes 10 time units (7 time units for ALU, 3 time units for other stages) to execute a macro-actor that contains 7 sequential operations in Fig.4.2. This means that the execution time in the macro-actor execution mode is significantly reduced. In general, to execute sequential actors with a length of n in a PE which has an m-stage instruction pipeline, it only takes $n + m - 1$ time units in the macro execution mode, while it takes $m \times n$ time units in the micro execution mode.

Therefore, it is m times faster to execute actors in the macro execution mode than in the micro execution mode for large n's.

The partitioning process of lumping sequential actors can also be evaluated by certain *cost* functions. In these functions, parameters are assigned with values to represent computation time, communication time, and other overhead. After the graphs are examined by the process of partitioning, macro actors will be generated by the partitioning process as soon as the output value of the cost functions reaches a threshold value.

4.2.2.2 Functional Actors Lumping

The scheme that lumps sequential actors for creating macro actors in the actor-based partitioning can be extended and applied to more complex statements in SISAL programs. Under this partitioning scheme, some form of parallelism will be reduced by lumping independent actors that are in the same statement. However, the locality of data-tokens will increase and hence communication overhead will be reduced due to the macro execution model. For example, the *Damped Jacobi Relaxation* in many numerical iterative solvers can be expressed as

$$U_{i,j}^{(k+1)} = (1-w) \times U_{i,j}^{(K)} + w \times [(U_{i+1,j}^{(k)} + U_{i-1,j}^{(k)} + U_{i,j+1}^{(k)} + U_{i,j-1}^{(k)} + h^2 f_{i,j})/4], \quad (4.1)$$

where the new value must be evaluated by adding together the values of four neighbor grid points, boundary conditions ($h^2 f_{i,j}$), and a weighted (1-w) value

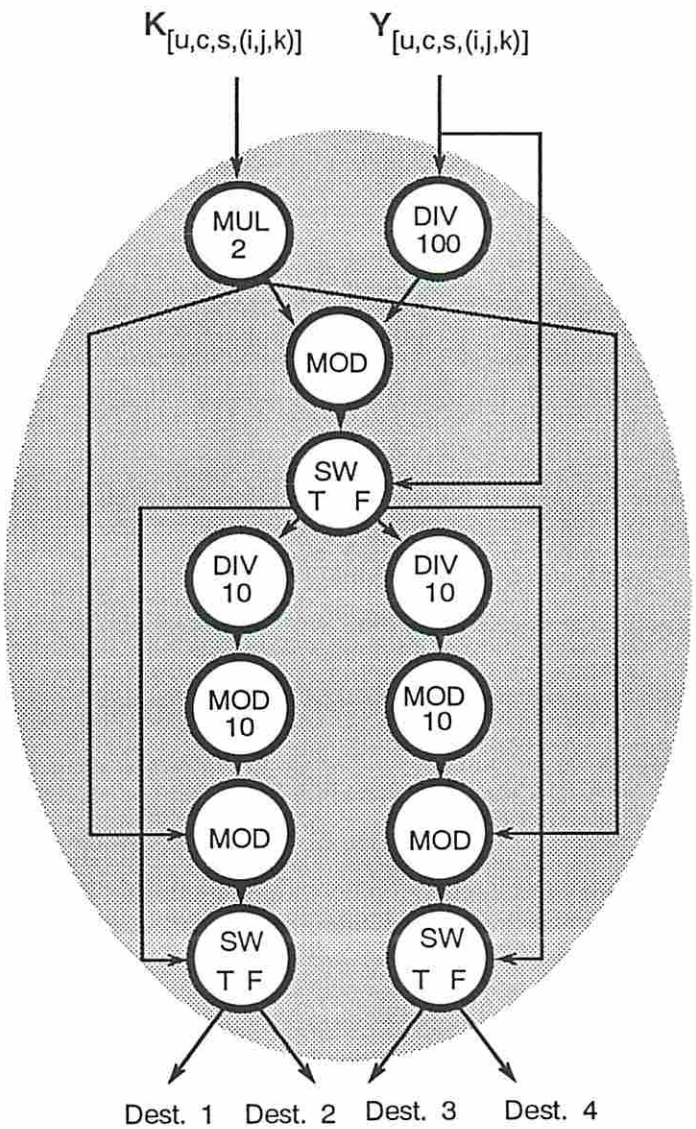

Figure 4.2: A Macro Actor Formed by Actor-based Partitioning

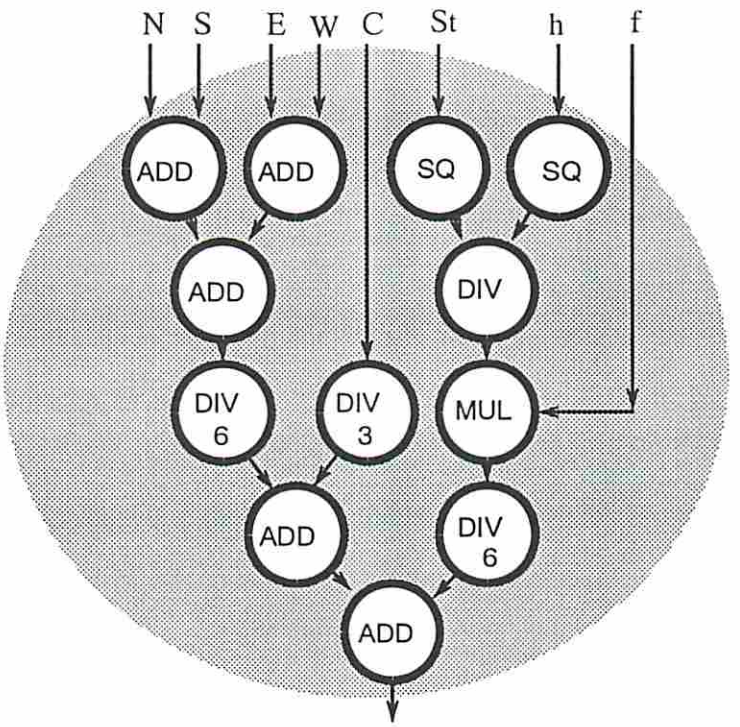

Figure 4.3: A Macro Actor for Damped Jacobi Relaxation

of the point itself. The above equation can also be simplified as: (we choose $w = 2/3$)

$$Center = \frac{Center}{3} + \frac{(North + South + West + East)}{6} + f \times \frac{st \times st}{h \times h \times 6} \quad (4.2)$$

where st and h will calculate the new h in different grid levels. In SISAL, it can be stated as:

```
x[m][i][j]= x[m][i][j]/3
            +(x[m][i-1][j]+x[m][i+1][j]
            +x[m][i][j-1]+x[m][i][j+1])/6
            +(f*st*st)/(h*h*6)
```

The corresponding fine-grain data-flow graph is shown in Fig.4.3, where 12 fine-grain actors perform the relaxation function. According to the criteria in this phase of partitioning, we therefore create a macro actor to perform this function.

Residual Error is another function that is often used in numerical mathematics. The residual error of a linear system $(A \times x = f)$ can be defined as:

$$Residual\ Error = f - A \times x', where\ x'\ is\ an\ iterative\ solution \quad (4.3)$$

In numerical PDE solvers, the residual error(R) can usually be written as:

$$R = f - (North + South + West + East - 4 \times Center) \times \frac{h \times h}{st \times st} \quad (4.4)$$

In SISAL, this function can be translated as:

```
R[m][i][j]= f - (x[m][i-1][j]+x[m][i+1][j]
              + x[m][i][j-1]+x[m][i][j+1]
              - 4*x[m][i][j])*(h*h)/(st*st)
```

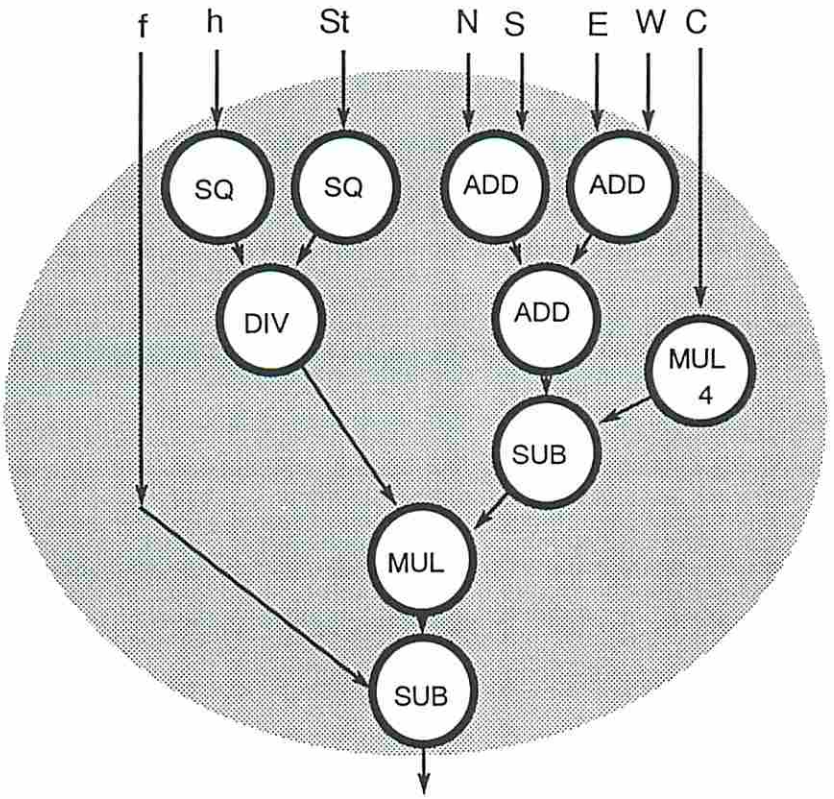

Figure 4.4: A Macro Actor for Calculating Residual Error

According to the criteria in the actor-based partitioning, a macro-actor is then created to perform this function instead of several micro-operations (Fig.4.4).

As another example, summation is used in many numerical applications. The function of summation in SISAL can be written as:

```
for i in 1, h cross j in 1, h
     returns value of sum A[i][j]
end for ;
```

To perform this function, as shown in Fig.4.5, one must add an incoming token with the partial result and check the tag of the token to decide whether the computation is complete. If the computation is not complete, the tag of the partial result must be increased in order to match the next token. From the graph, we find that the sequential path of the function consists of 11 sequential actors. Hence, we form a macro-actor to perform this function.

It should be noted that once new *functional* macro actors which perform unique functions are generated, the subgraphs that represent macro actors can be saved in the system library for later use.

4.2.3 Tag-based Partitioning

In tag-based partitioning, tag-manipulation operations are the main targets to form macro actors. The criteria of this partitioning is to form simple macro actors which will handle complex tags operations. The purpose of creating macro actors for tag-manipulation operations is to avoid multiple copies of data elements which are needed to be delivered to several resources. Hence, after this partitioning, communication overhead will be reduced and redundant operations to extract

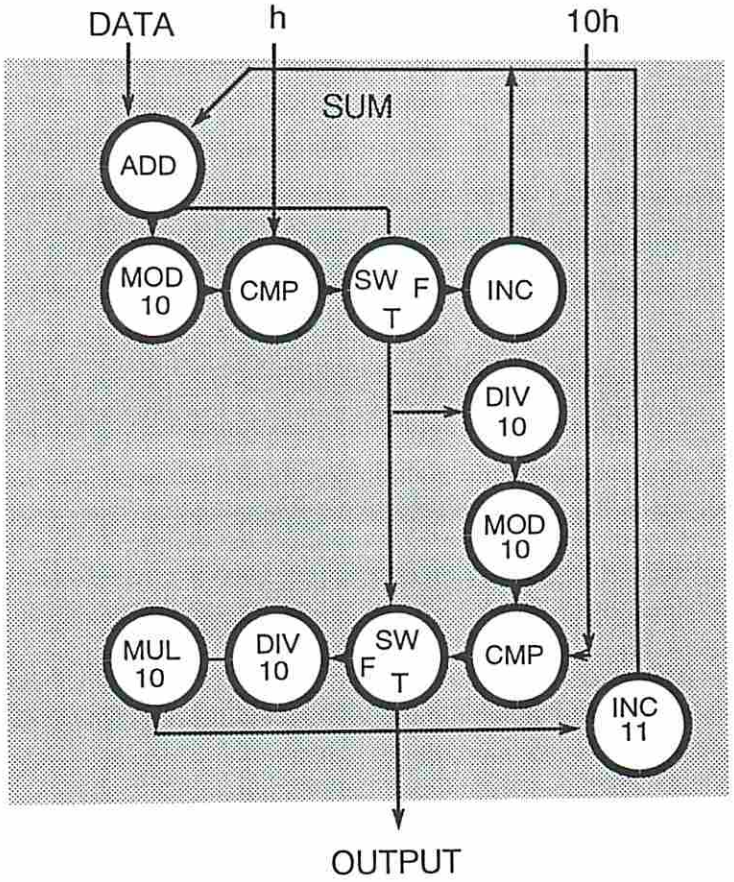

Figure 4.5: A Macro Actor for Summation

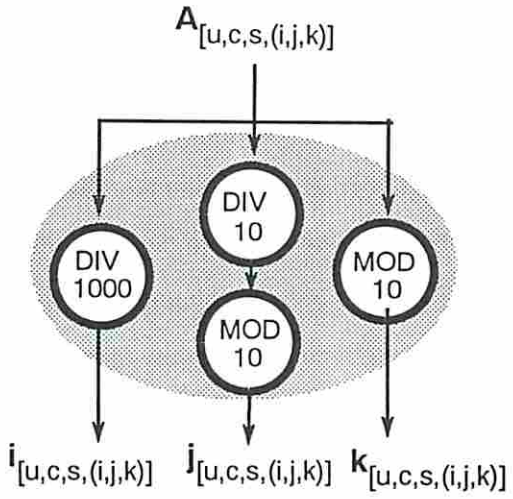

Figure 4.6: A Macro Actor for Tag Manipulation

tags will be eliminated. Fig.4.6 shows how each field in the iteration tag(i) can be obtained by using 4 micro actors. In the tag-based partitioning, we therefore form the four operations as a macro actor to generate each field in the i tag. Besides the above simple tag-operations, the same scheme to form macro actors can also be applied to the function that modifies tags with arbitrary values. In Fig.4.7, an element $A_{[u,c,s,(i,j,k)]}$ is added and subtracted by st in the i and the j fields. The new macro-actor hence lumps all the micro actors and generate the same results: $A_{[u,c,s,(i+st,j,k)]}$, $A_{[u,c,s,(i-st,j,k)]}$, $A_{[u,c,s,(i,j+st,k)]}$, and $A_{[u,c,s,(i,j-st,k)]}$.

From the above examples, the new macro actor instructions can then be defined as:

```
RTGI(i,j,k)        ; read fields i, j, and k in
                     the iteration number tag
```

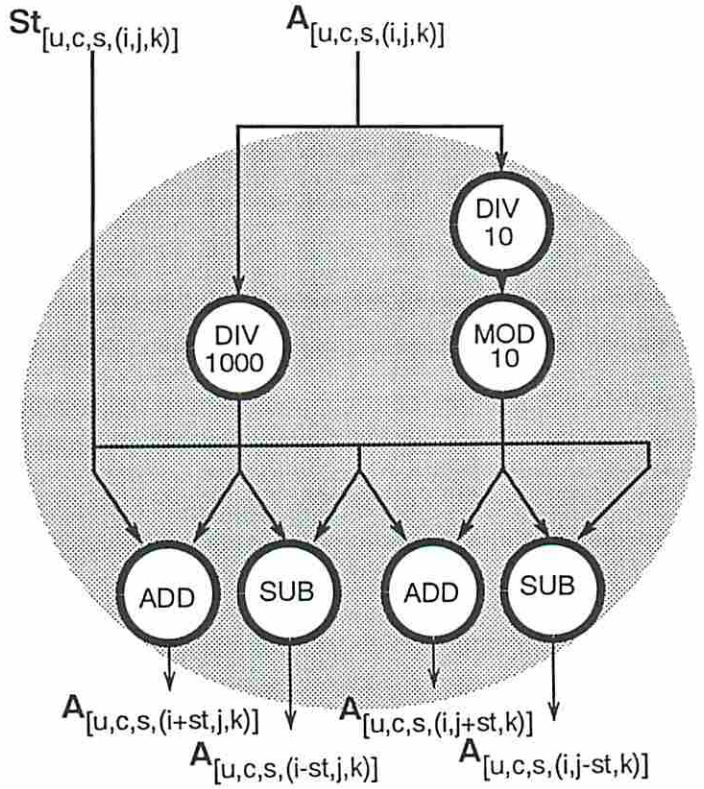

Figure 4.7: A Macro Actor Formed from Tag-based Partitioning

```
WTGI(i,j)          ; add and subtract value into i, and j
                     in the iteration number tag
```

One should note that in this phase of the partitioning, the parallelism in data-flow graphs will be reduced mainly because the tag-manipulation macro actors contain independent operations on different fields of the tag. However, the new macro actors will simplify tags representation, eliminate broadcasting of data and hence reduce communication overhead.

4.2.4 Token-based Partitioning

In this phase of partitioning, not only actors are lumped into macro actors but also data-tokens are collected into fat tokens. Similar to *user's directed partitioning*, the primary benefit from this partitioning is that *Vectorization* techniques can be easily employed to execute macro actors. In the actual implementation, vectors will be formed first, and the streams of tokens will then be sent into the arithmetic unit and executed. As a result, the execution time in the arithmetic unit can be reduced due to the pipelined execution of data. However, in order to vectorize data-tokens, new constructs in high level languages must be developed. Here, we propose the **Vector-for** operation to represent vector operations in SISAL. The new construct will allow its inside function to be concurrently evaluated in the forms of vectors. While the **for** construct in SISAL only allows every index value to be executed in parallel, the Vector-for construct will generate vectors according to the index in the construct and allow each vector to be executed in parallel.

For example, in Fig.4.3, the function of *Relaxation* can be vectorized. The *Vectorized Relaxation* in SISAL can then be coded as:

```
Vector-for i in 1, p cross j in 1, q
  returns array of
     x[m][i][j]= x[m][i][j]/3
                 +(x[m][i-1][j]+x[m][i+1][j]
                 +x[m][i][j-1]+x[m][i][j+1])/6
                 +(f*st*st)/(h*h*6)
  end for;
```

In the above codes, the *for* construct is replaced by the new *Vector-for* construct in the first line to perform vector operations. According to the above codes, the compiler will generate p vectors where each vector has a length q for a two-dimensional array. In order to execute vectorized data-flow graphs, additional actors must be inserted in data-flow graphs for forming vectors and distributing the results: First, in order to collect each individual data-token to form vectors, *Accumulation* (ACC) actors with an *Initial* (SEED) actor must be inserted at the top of vector macro actors to accumulate all the arriving single tokens. When vectors have been formed, the streams are then sent to macro actors and executed. Second, in order to distribute elements after vector operations to different destinations, *Distribution* (SCA) actors must be used to scatter tokens. After vector computations complete, the results will be sent to the SCA actor and distributed to different destinations. Fig.4.8 shows the final graph of the above codes which includes ACC, SEED, SCA, and vector operations in macro actors. Another example of inserting vector operations is demonstrated in calculating the residual error:

```
Vector-for i in 1, h cross j in 1, h
```

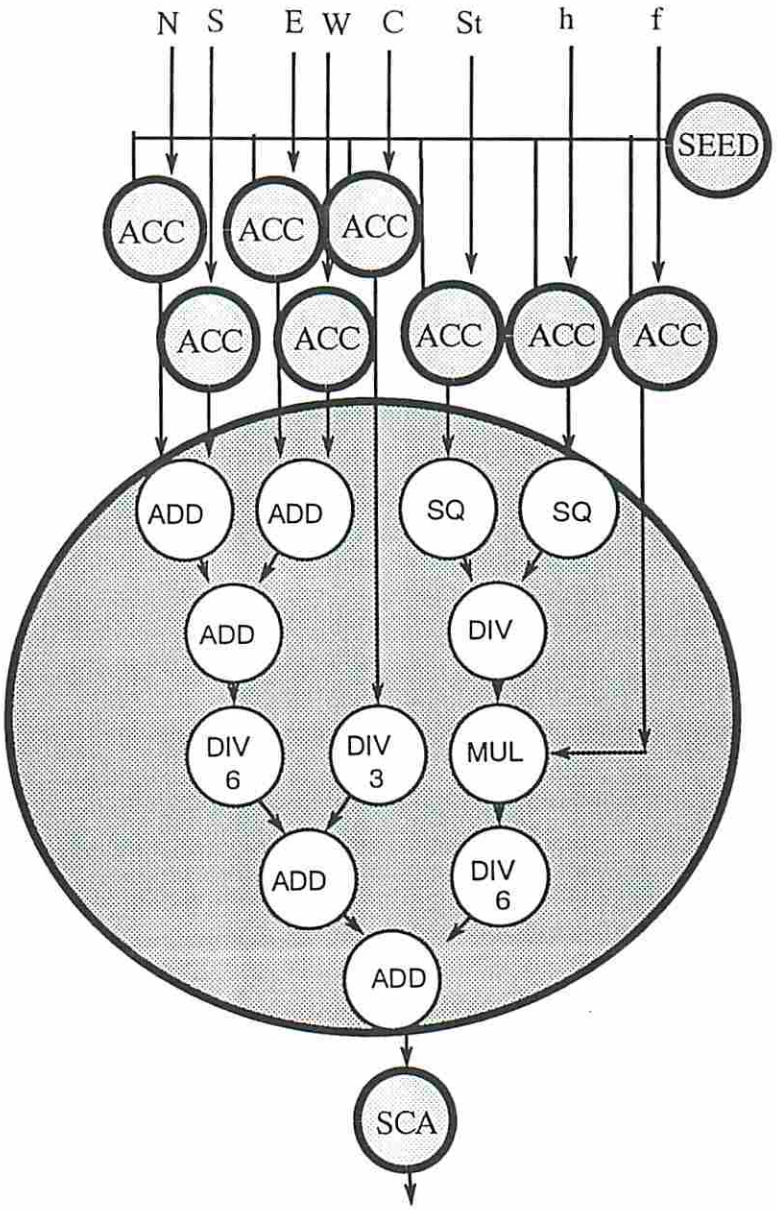

Figure 4.8: Vectorized Macro Actors for Damped Jacobi Relaxation

```
returns array of
   R[m][i][j]= f - (x[m][i-1][j]+x[m][i+1][j]
                 + x[m][i][j-1]+x[m][i][j+1]
                 - 4*x[m][i][j])*(h*h)/(st*st)
end for;
```

The corresponding macro graph is shown in Fig.4.9, where the tokens are collected by ACC actors and distributed by SCA after the execution of macro actors which calculate residual errors.

It should be noted that two kinds of overhead will be introduced in the token-based partitioning. First, the overhead actors (ACC, SEED, and SCA) will adhere to vector operations. However, in the efficient vector operations generated by this partitioning, the time spent in overhead actors will be negligible. Second, when vectorized macro actors are applied to complex structures, extra tag-manipulation macro actors are usually needed to generate proper tags. For example, if a vector operation is applied to a stream with non-continuous indices (A[i][j][k], A[i][j+st][k], A[i][j+2st][k],...). After the vector operations, the output (a stream with continuous indices from SCA), will require tag-manipulation macro actors to generate accurate tags for each token (Fig.4.10) .

4.3 A Case Study and Simulation Results

In this section, the principles of the three-phase partitioning scheme developed in the previous section are applied to numerical multigrid PDE solvers [26]. Multigrid algorithms are known as one of the most efficient algorithms to solve PDEs. Besides the complexity of the algorithms has been proved to be independent of

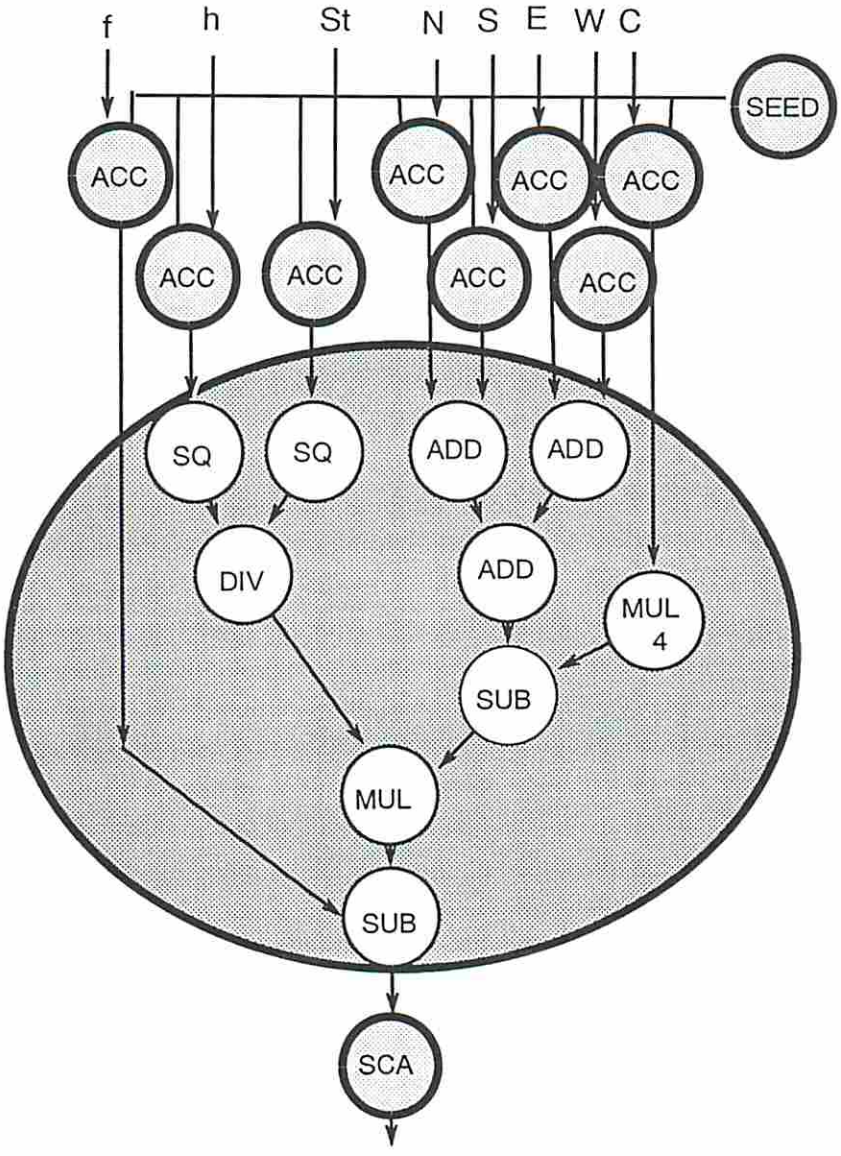

Figure 4.9: Vectorized Macro Actors for Calculating Residual Error

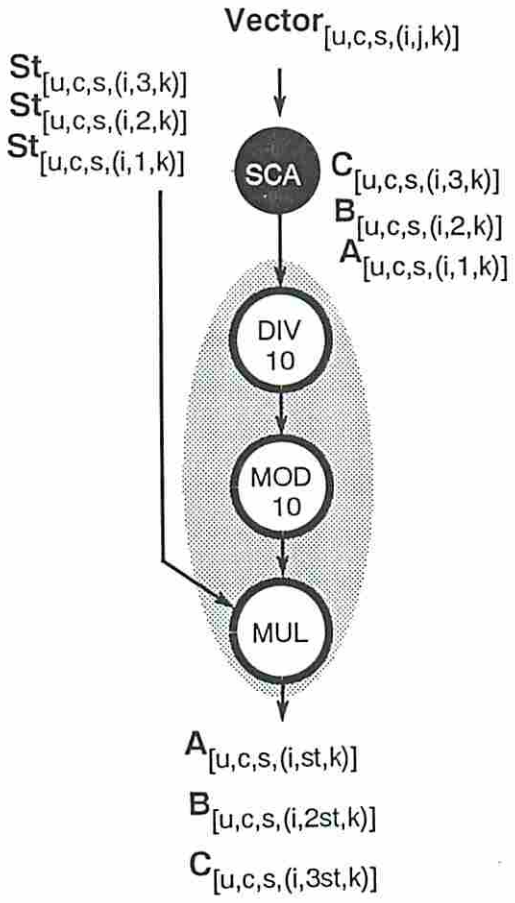

Figure 4.10: A Macro Actor to Support Vectorization

the problem size, its regular grid structure is suited for multiprocessor systems. We will start from the description of multigrid algorithms for solving Elliptic PDE in both SISAL and C high level languages and translate the SISAL programs into assembly codes as well as machine codes. Several data-flow graphs will then be partitioned into variable-grain graphs. Finally, the execution of the partitioned graphs are verified by a deterministic variable-grain data-flow graph simulator.

4.3.1 Problem Definition

A general V-cycle multigrid(GVM) algorithm consists of three main steps:

1. Relaxation: it refers to some basic iterative methods, like weighted Jacobi, which can reduce the error on each grid level.

2. Restriction: this operation forms an error correction term from the fine grid level to the coarse grid level, when the fine grid level has completed the relaxations.

3. Interpolation: it projects the correction term of error from the coarse grid level back to the fine grid level, when the coarse grid level has corrected the error.

There are many strategies to perform each of these three main operations. In general, the GVM algorithm on two grid levels can be expressed as the following procedure:

1. Relax n_1 times on the fine grid level with the initial guess.

2. Compute the residual error on the fine grid level.

3. Restrict the error term into the coarse grid level.

4. Relax n_2 times on the coarse grid level with the initial guess.

5. Interpolate the correction term back to the fine grid level.

6. Relax n_1 times on the fine grid level.

7. Check the stopping criterion and decide either to do another V-cycle or to stop the execution.

From the above, it transpires that the major computation demands in the GVM algorithm come from the relaxations on each grid level and the calculation of the residual error.

4.3.1.1 Poisson's Equation Solvers

In this research, the multigrid algorithms for solving Poisson's equation on the unit square (0,1)×(0,1), with Dirichlet boundary conditions, and different load functions $f(x, y)$:

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = f(x, y), \qquad 0 < x < 1,\ 0 < y < 1. \tag{4.5}$$

has been chosen for evaluating the performance of partitioning schemes. In the experiments, we choose the damped Jacobi method as the relaxation scheme with a fixed relaxation parameter w=2/3 :

$$U_{i,j}^{(k+1)} = (1-w) \times U_{i,j}^{(K)} + w \times [(U_{i+1,j}^{(k)} + U_{i-1,j}^{(k)} + U_{i,j+1}^{(k)} + U_{i,j-1}^{(k)} + h^2 f_{i,j})/4], \quad (4.6)$$

the full weighting restriction (Fig.3.2), bilinear interpolation (Fig.3.3) have also been chosen in the experiments.

4.3.2 High Level Language Programming

In this study, we have programmed the described multigrid PDE solvers in SISAL. The programs consist of the main functions in the algorithms which include Relaxation, Restriction, Interpolation, and convergence check. A piece of code to describe the Damped Jacobi Relaxation is demonstrated in Fig. 4.11 and 4.12.

In order to compare the efficiency between applicative and imperative languages, the algorithms are also coded in a conventional high level language C. By comparing the two programs that describe the same algorithm, we found that the program size in SISAL is about four times greater than that in C language. This is because of the single assignment rules in SISAL and the concise expressions in C language. For example, the codes for the Damped Jacobi Relaxation in C (Fig. 4.13) is relatively short compared to SISAL codes. Indeed, under single assignment rules, extra code is needed in SISAL programs to concatenate the updated

```
function Jacobi (h, level, m : integer ; x, y, f : ThreeDim
                 returns  ThreeDim )
  let
    st := h/exp (2,m) ;
    dh := h / st - 1 ;
    eqone :=   if m = 1 then true
                        else false
               end if ;
    eqlevel := if m = level then true
                            else false
               end if
  in
    if eqone then
                cateY ( m, h, dh, st, 1, 1, x, y, f )
                ||
                for k in 2, level
                  returns array of y[k]
                end for
            elseif eqlevel
              then
                for k in 1, m-1
                  returns array of y[k]
                end for
                ||
                cateY ( m, h, dh, st, m, m, x, y, f )
              else
                for k in 1, m-1
                  returns array of y[k]
                end for
                ||
                cateY ( m, h, dh, st, m, m, x, y, f )
                ||
                for k in m+1, level
                  returns array of y[k]
                end for
     end if
   end let
 end function % Jacobi
```

Figure 4.11: A SISAL Program for the Damped Jacobi Method

```
function cateY ( m, h, dh, st, start, final : integer ;
                 x, y, f : ThreeDim  returns ThreeDim )

  for k in start, final
    returns array of
      for i in 0, 0 cross j in 0, h
        returns array of y[m][i][j]
      end for
      ||
      for i in 1, dh
        returns array of
          for j in 0,0
            returns array of y[m][i][0]
          end for
          ||
          for j in 1, dh
            returns array of
               x[m][i][j]/3.0 +( x[m][i-1][j]+x[m][i+1][j]
                              +x[m][i][j-1]+x[m][i][j+1]
                              -f[m][i][j]* real(st*st)
                              /real(h*h))/6.0
          end for
          ||
          for j in dh+1, h
            returns array of y[m][i][j]
          end for
        end for
        ||
        for i in dh+1, h cross j in 0, h
          returns array of y[m][i][j]
        end for
    end for

end function % cateY
```

Figure 4.12: A SISAL Program for the Damped Jacobi Method (cont.)

```
jacobi(m,n)
int m,n;
{ int i,j,k,st;
  st=h/ power(2,m);
  if(n==0)
     { for (i= 1 ; i<=h/st-1 ; i++)
       for (j= 1 ; j<=h/st-1 ; j++)
                      x[m][i][j]=0;}

  for (i= 1 ; i<=h/st-1 ; i++)
  for (j= 1 ; j<=h/st-1 ; j++)
        y[m][i][j]= x[m][i][j]/3
                    +( x[m][i-1][j]+x[m][i+1][j]
                    +x[m][i][j-1]+x[m][i][j+1]
                    -f[m][i][j]*st*st/(h*h))/6;
 }
```

Figure 4.13: A C Program for the Damped Jacobi Method

elements in an array with other elements. However, due to the functionality in SISAL, it is much easier to translate SISAL programs into data-flow graphs.

4.3.3 Assembly and Machine Codes

Once a SISAL program is translated into a data-flow graph, a data-flow assembly language can be used to describe the structure of graphs. The assembly language described in [9], an actor in a graph can be represented as:

```
A actor_name (input_port_name) -> output_port_name :
  micro_instruction ;
```

and a function can be represented as:

```
DEFINE function_name (IN input_parameter
                      OUT output_parameter)
  CONST x,y,z
  BEGIN
  A actor_name1 (input_port_name) -> output_port_name :
    micro_instruction_1 ;
    micro_instruction_2 ;

  A actor_name2 (input_port_name) -> output_port_name :
    micro_instruction ;

  A actor_name3 (input_port_name) -> output_port_name :
    micro_instruction_1 ;
    micro_instruction_2 ;
    micro_instruction_3 ;
  END
```

The assembly codes that describe graphs are then converted to machine codes by assembler. In [35], the actor, function, input-port, and output-port names are replaced by pseudo codes and the instructions are represented by machine codes.

4.3.4 Simulation Assumptions

The macro data-flow model that contains the von Neumann execution mode to execute macro actors has been adopted for the simulator. It consists of a multiprocessor system which has a Matching Store Unit, an Execution Unit, and an I-structure Memory Unit [5] in each processor. The processors are then interconnected by a packet-switching network. In order to gather reasonable performance statistics on the tested programs, we have made appropriate assumptions on the various hardware and software delays: we assumed that all functional

units (Matching Store Unit, Execution Unit, and I-structure) as well as communication network required a single unit of delay to perform their functions. The program graphs were constructed by using elementary actors and macro-actors that included normal arithmetic/logic operators, gates, structure operators, and tag operators. 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 vector operations would be proportional to the size of data elements to be executed.

4.3.5 Simulation Results

The executions of the micro-actor (fine gain), actor-based partitioned (the first phase), tag-based partitioned (the second phase) and token-based partitioned (the third phase) graphs are simulated in our simulation environment. The following results are observed:

- *Number of Static Actors*: In Table 4.1, 4.2, 4.3, and 4.4, the number of static actors for the purposes of structure handling, tag manipulation, stream generation and actual computation are counted in micro-actor graphs and the graphs after actor-based partitioning, tag-based partitioning and token-based partitioning.

Actors Count of Static Actors		
Fine Grain Graphs		
	Actors	%
Structure Handling	65	16
Tag Manipulation	120	30
Stream Generation	103	25
Compuation	117	29
Total Actors	405	100

TABLE 4.1

Actors Count of Static Actors		
Actor-based Partitioning		
	Actors	%
Structure Handling	65	25
Tag Manipulation	65	25
Stream Generation	94	35
Compuation	41	15
Total Actors	265	100

TABLE 4.2

Actors Count of Static Actors		
Tag-based Partitioning		
	Actors	%
Structure Handling	65	29
Tag Manipulation	26	11
Stream Generation	94	42
Compuation	41	18
Total Actors	226	100

TABLE 4.3

Actors Count of Static Actors		
Token-based Partitioning		
	Actors	%
Structure Handling	65	24
Tag Manipulation	30	11
Stream Generation	136	50
Compuation	41	15
Total Actors	272	100

TABLE 4.4

The results in Table 4.2 show that after actor-based partition, the number of actors has been significantly reduced from 405 to 265. This is mainly due to the *sequential-actor-lumping* which has reduced tag-manipulational actors from 120 to 65 and the *functional-actor-lumping* which has reduced computational actors from 117 to 41. In Table 4.3, it shows that after tag-based partitioning, the number of tag-manipulational actors has been reduced from 65 to 26. However, in token-based partitioning, due to overhead actors are introduced to accumulate individual tokens and to distribute elements of vectors to different processors, it requires more stream generation actors

Comparison of Static Actors Counts		
	Actors	%
Fine Grain Graphs	405	
Actor-based Partitioning	265	65
Tag-based Partitioning	226	56
Token-based Partitioning	272	67

$$Percentage = \frac{No.\ of\ actors\ after\ partition}{No.\ of\ actors\ in\ fine\ grain\ graphs} \times 100\%$$

TABLE 4.5

for proper execution. Hence, in Table 4.4, the number of stream generation actors has increased from 94 to 136 after token-based partitioning.

In Table 4.5, and Fig.4.14 the total number of static actors after each phase of partitioning has been compared. It concludes that the number of static actors can be totally reduced to 56%-67% when compared with the number of actors in fine grain data-flow graphs.

- *Number of Dynamic Actors* : In order to compare the total instructions that are needed to execute multigrid algorithms under different partitioning schemes, the number of dynamic actors have been counted. To fairly compare different partitioning schemes, we have simulated one iteration on each partitioned graphs with different problem sizes.

In Table 4.6, and Fig.4.15 the results show that it requires only 60% of instructions after actor-based partitioning, 40% of instructions after tag-based partitioning, and 50% of instructions after token-based partitioning to execute the same algorithm, when compared with fine grain computation.

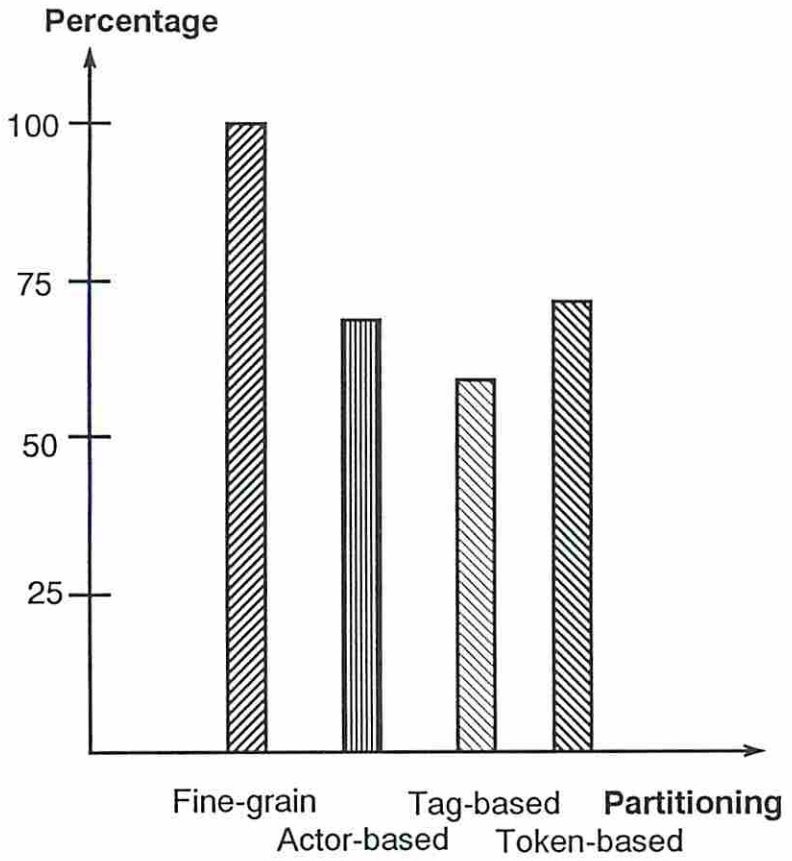

Figure 4.14: A Comparison of Static Actors Counts

Comparison of Dynamic Actors Counts				
Problem Size	3 levels		4 levels	
	Actors	%	Actors	%
Fine Grain Graphs	11,382		52,813	
Actor-based Partitioning	6,702	60	30,253	57
Tag-based Partitioning	4,581	40	20,650	39
Token-based Partitioning	5,864	51	26,558	50

$$Percentage = \frac{No.\ of\ actors\ after\ partition}{No.\ of\ actors\ in\ fine\ grain\ graphs} \times 100\%$$

TABLE 4.6

Comparison of Execution Time in Graph Simulator				
Problem Size	3 levels		4 levels	
	Time	%	Time	%
Fine Grain Graphs	390		1,352	
Actor-based Partitioning	262	67	670	49
Tag-based Partitioning	267	68	675	50
Token-based Partitioning	317	81	551	40

$$Percentage = \frac{Exe.\ time\ of\ partitioned\ graphs}{Exe.\ time\ of\ fine\ grain\ graphs} \times 100\%$$

TABLE 4.7

This results also demonstrate that it will require less percentage of instructions (compared with fine-grain graphs) to perform multigrid algorithms in actual computation than in static instruction count.

- *Execution Time* : In this experiment, the execution time of each partitioned graph has been simulated. Here, we assume that a system of unlimited number of processing units is available in order to exclude machine-dependent resource management problems. Under this assumption, the effect of reducing parallelism in each partitioned graph will be shown by the simulated execution time.

Table 4.7 shows that the execution time of tag-based partitioned graphs does not increase when compared with that of actor-based partitioned graphs. This verifies that tag-manipulational macro actors indeed eliminate broadcasting data overhead and hence reduce communication time, while tag-based partitioning scheme has decreased the available parallelism in the partitioned graphs. The results also show that it is desirable to have

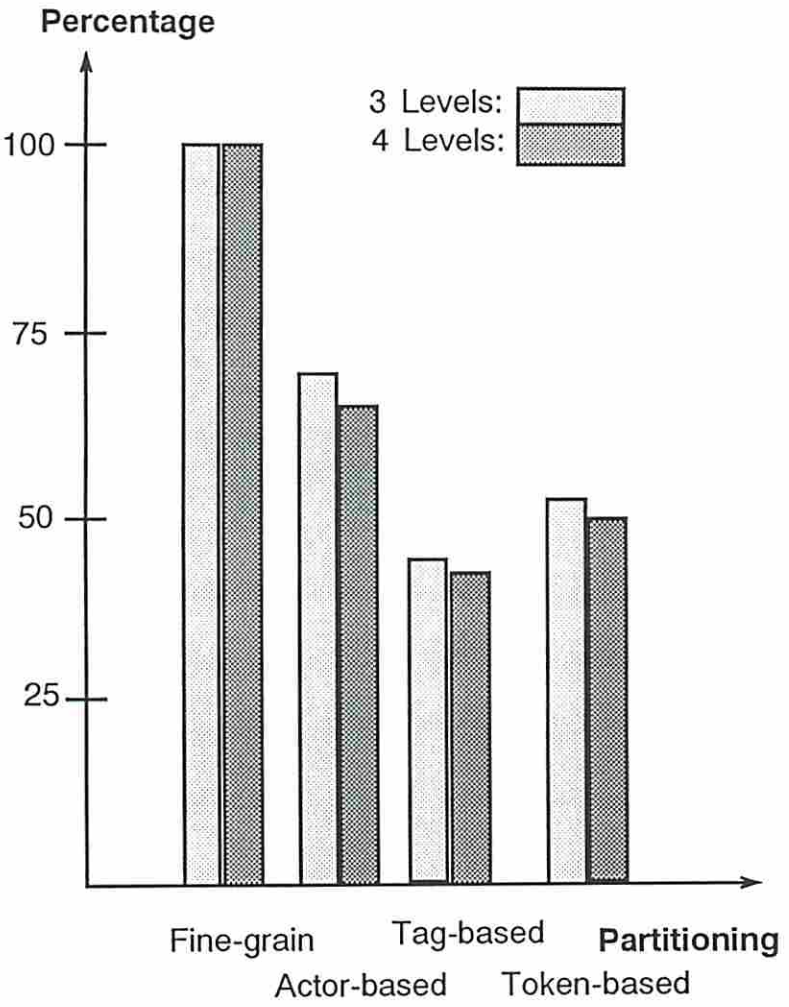

Figure 4.15: A Comparison of Dynamic Actors Counts

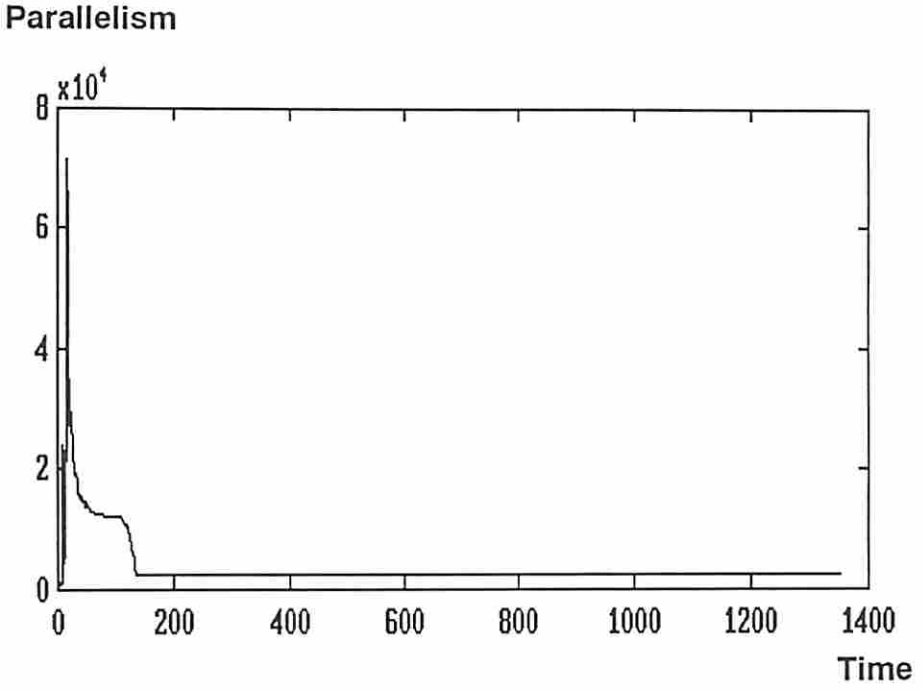

Figure 4.16: The Amount of Parallelism in Micro Actor Graphs

a longer vector in token-based partitioning scheme. For example, with a problem size of 3 levels multigrid (which has a vector length of 7), the token-based partitioned graphs can reduce only 20% of execution time, while it can reduce 60% of execution time in a 4 levels multigrid (which has a vector length of 15).

- *Parallelism :* The existing parallelism for each execution of the partitioned graphs is demonstrated in Fig. 4.16, 4.17, 4.18 and 4.19. By comparing Fig. 4.17 with 4.16, we observe that the amount of parallelism has been

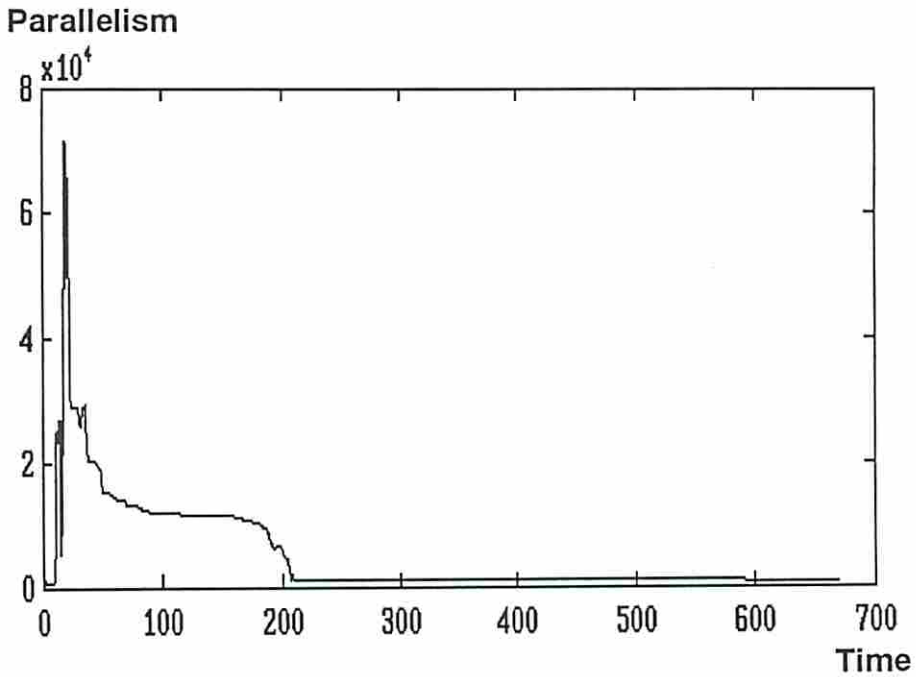

Figure 4.17: The Amount of Parallelism after Actor-based Partitioning

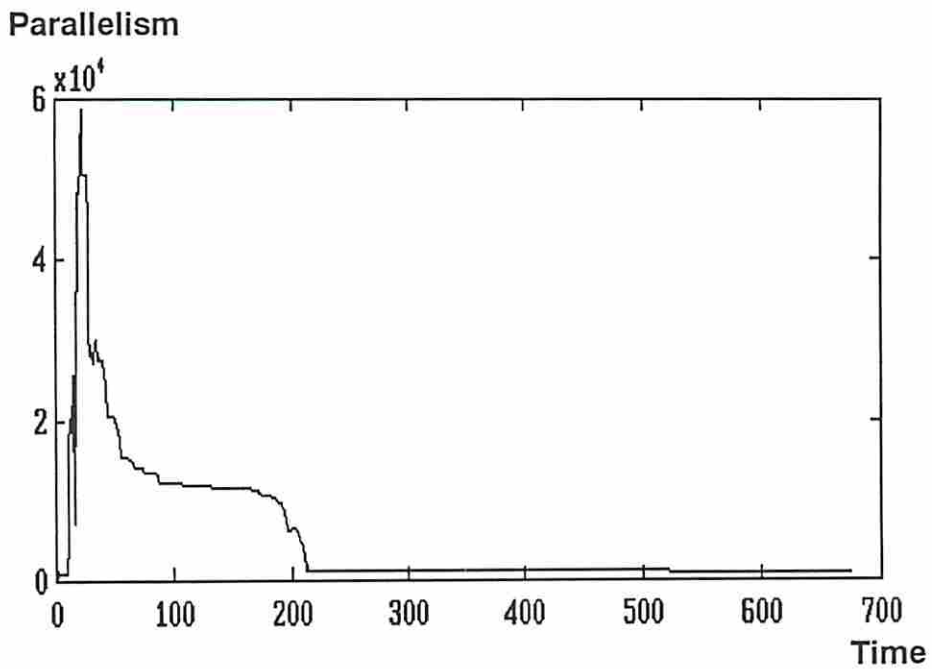

Figure 4.18: The Amount of Parallelism after Tag-based Partitioning

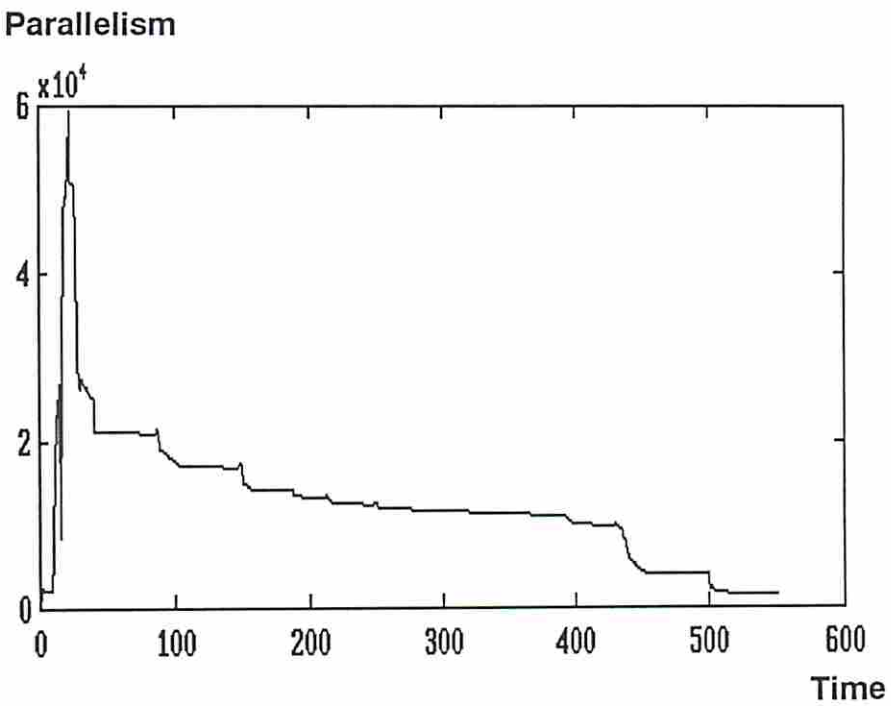

Figure 4.19: The Amount of Parallelism after Token-based Partitioning

indeed increased by the actor-based partitioning scheme which lumps the sequential and functional actors into macro actors and hence shortens the length of sequential paths. Fig. 4.17 and 4.18 show that the peak parallelism has been reduced from 7×10^4 to 6×10^4 by the tag-based partitioning scheme which lumps those independent tag-manipulation actors. This result indicates that fewer resources (less memory, fewer processing elements) will be required in the coarse-grain data-flow computing. In other words, it will be more efficient in the coarse-grain data-flow systems. Fig. 4.19 shows the average parallelism has been increased by the token-based partitioning scheme, while the execution time has been reduced by the vector operations in this scheme.

Finally, in Fig. 4.20, we compare the amount of parallelism and execution time in the fine-grain data-flow graphs with those in the coarse-grain partitioned data-flow graphs. It shows that the three-phase partitioning schemes actually can enhance the performance by increasing the average parallelism and reducing the execution time.

- *Ideal Speed-up* : The ideal speed-up of the partitioned graphs can be obtained from the comparison of the execution time in one processing unit with that in an infinitive numbers of processing units.

Table 4.8, 4.9, and Fig.4.21 show that the ideal speed-ups of the execution in partitioned graphs are better than those in fine-grain computations. For example, in Table 4.9, after token-based partitioning, the ideal speed-up is

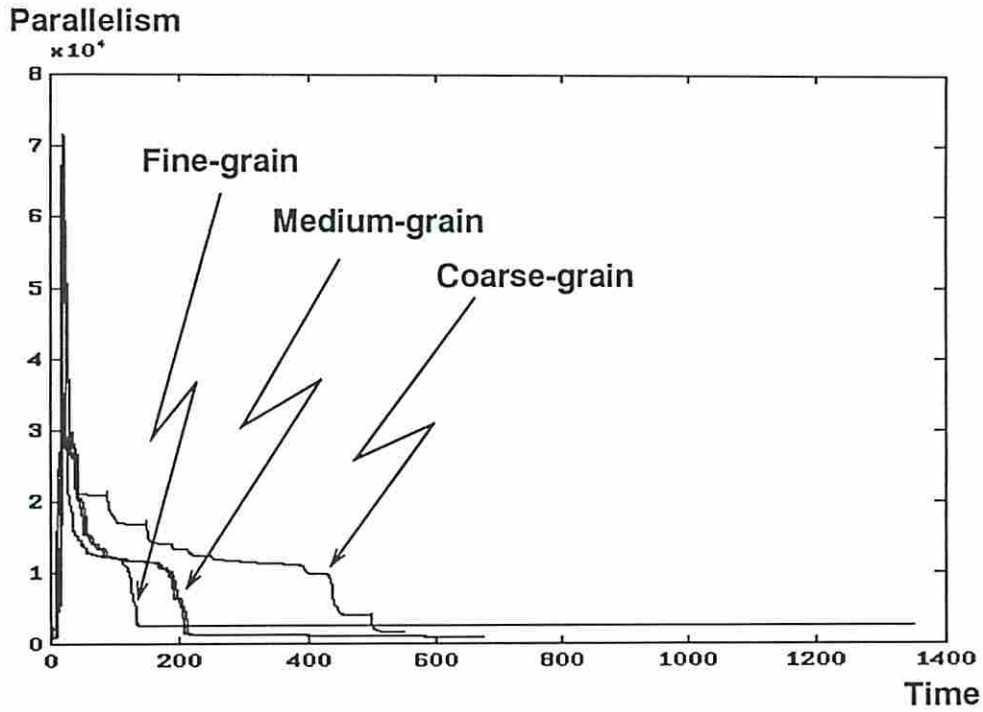

Figure 4.20: Parallelism in the Fine-grain and the Coarse-grain Execution

Ideal Speed-ups			
Problem Size	3 levels		
Execution Time	in 1 PE	in ∞ PEs	Speed-up
Fine Grain Graphs	11,387	390	29
Actor-based Partitioning	13,882	262	53
Tag-based Partitioning	12,010	267	45
Token-based Partitioning	11,976	317	38

$$Speedup = \frac{Exe.\ time\ in\ one\ PE}{Exe.\ time\ in\ N\ PE}$$

TABLE 4.8

Ideal Speed-ups			
Problem Size	4 levels		
Execution Time	in 1 PE	in ∞ PEs	Speed-up
Fine Grain Graphs	52,818	1,352	39
Actor-based Partitioning	65,079	670	97
Tag-based Partitioning	56,650	675	84
Token-based Partitioning	55,146	551	100

$$Speedup = \frac{Exe.\ time\ in\ one\ PE}{Exe.\ time\ in\ N\ PE}$$

TABLE 4.9

100 while it is only 39 in fine grain computation. This demonstrates that macro data-flow graphs are more suitable for multiprocessor systems.

From the simulation results, we have observed that the three-phase partitioning schemes can actually reduce the number of instructions and execution time: In the actor-based partitioning, by lumping the sequential and functional actors, the tag-manipulational and computational actors were reduced. While the tag-based partitioning reduced the amount of parallelism, it indeed eliminated the broadcasting and redundant actors for tag operations. In the token-based partitioning, the vectorized data-flow graphs which exploited the vectorization techniques from the conventional vector processing have enhanced the performance.

4.4 Discussion

In this chapter, we have demonstrated how macro data-flow graphs could be generated. By applying our partitioning schemes on numerical applications, we have also shown how macro actors could be formed and how vectorized data-tokens could be gathered under our three-phase partitioning schemes.

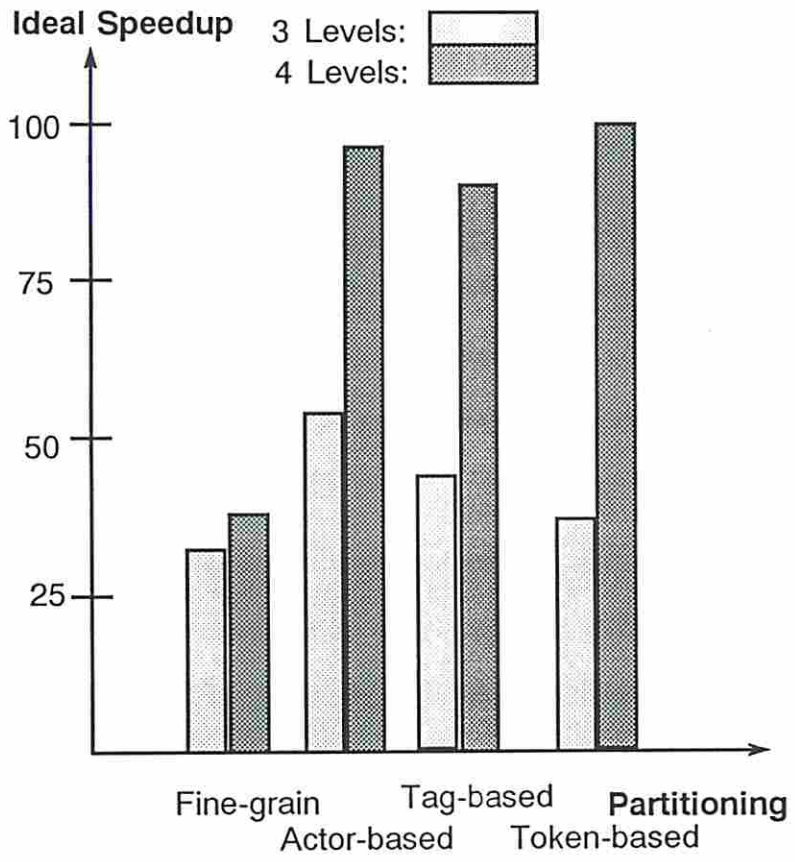

Figure 4.21: Totential Speedups with Different Problem Sizes

We started from the theoretical, static, dynamic, and user-directive partitioning schemes. Then, we concentrated on the partitioning of the fine-grain data-flow graphs, and evaluated of the final performance. In order to reduce execution time, the actor-based partitioning schemes were applied to the sequential and computational actors in the fine grain data-flow graphs. In the second-phase of partitioning, the tag-based partitioning schemes were taken to simplify tag-manipulational operations. In the final phase of partitioning, the token-based partitioning schemes have exploited *vectorization* techniques to achieve efficiency in macro data-flow computing.

In this chapter, we have also conducted a deterministic simulation to verify the partitioning schemes on the graphs of multigrid algorithms for solving Poisson's equations. To this end, we have described the algorithms in SISAL and compared them with the same algorithms in C. We have also translated the programs into data-flow graphs that are described in assembly language and executed the graphs in our simulator. From the results, we have demonstrated that the number of instructions and execution time in the partitioned graphs can be actually reduced by partitioning the actors and data-tokens in the macro data-flow graphs. The results also show that high potential speedups can be achieved in the execution of the partitioned graphs in data-flow systems.

Chapter 5

Data-flow Architectures For Asynchronous Algorithms

Linear systems play an important role in many applications such as PDE solvers. There are basically two approaches to solve linear systems: synchronous [34] and asynchronous methods [13]. While the synchronous methods are easy to implement, they do not yield acceptable levels of performance for complex problems, mainly because of the synchronization barriers among processes. On the other hand, the asynchronous approach has been found by many researchers [10, 13, 20, 25] to exploit efficiently runtime parallelism. In the asynchronous approach, communication between processes is achieved by reading the dynamically updated variables while each process continues its execution to update shared variables. Therefore, the chaotic behavior of data in an asynchronous algorithm is recognized to be very complex. While on asynchronous method can be effective in parallel machines and can deliver high performance, it is difficult to implement due to the chaotic behavior of the method itself. From the software

perspective, language constructs must be defined to specify the asynchronous method, thereby parallelizing the algorithm. From the hardware point of view, special architecture schemes dedicated to the algorithm need to be developed.

In this chapter, we take a coarse-grain approach in the Variable-grain Tagged-token Data-flow (VTD) system to implement asynchronous methods for solving linear systems as opposed to the fine-grain data-flow systems [18, 29]. The objectives of this paper are therefore to evaluate the efficiency of the VTD system for computationally intensive numerical applications in both fine-grain and coarse-grain execution models and to compare the performance of asynchronous algorithms with that of synchronous algorithms. To these ends, we have concentrated on chaotic relaxation for solving linear systems and compared it with the conventional Jacobi method. Our study is therefore aimed in the following convergent directions: specification of the algorithms in a high-level data-flow language, translation into low-level data-flow graphs, partitioning and allocation of the resulting graphs, design of the appropriate architectural mechanisms to support the low-level constructs, and evaluation of the final performance.

We will first introduce special high-level data-flow language constructs (*Async-Repeat* and *Async-For*) for the chaotic behavior in asynchronous algorithms. The scheme to form coarse-grain (macro-actor) data-flow graphs and a specific firing rule in the *Matching Store with Locks* of processors will also be introduced in order to execute correctly computations in the asynchronous algorithms. In this chapter, we are also interested in measuring and comparing the performance of algorithms as well as our VTD system: First, to evaluate the

performance of the architecture, the conventional *Speedup* measurement will be taken to depict the trend of the performance with larger machine configurations. Second, to estimate the amount of the growing parallelism within an algorithm when the algorithm's complexity has been increased, a new measurement, called *Growth Factor*, will be defined to show how suitable the algorithm is for multiprocessor systems. Third, to measure how efficient of parallel systems in executing parallel algorithms, we will introduce a new measurement, called *Scalability Factor*, to demonstrate the scalability property of the systems. Finally, we will define the *Robustness* measure to indicate the potential performance of the systems.

We will then conduct a deterministic simulation of the VTD system and show the system performance in the presence of large degrees of parallelism. We finally observe the simulation results which show that chaotic relaxation executes faster and achieves a higher speedup than the Jacobi method in various machine configurations. Furthermore, the results demonstrate that chaotic relaxation is more tolerant to interprocessor communication delays than the Jacobi method and imposes less computation overhead. In our simulation, the convergence of chaotic relaxation is also verified by the accuracy of the solutions. In both algorithms, the simulation results indicate that the coarse-grain (macro-actor) execution mode outperforms the fine-grain (micro-actor) execution mode in execution time, speedup, and overhead. The scalability factor and robustness of the VTD system are also verified from the simulation results.

5.1 Linear Systems Solvers

We describe in general terms the computational problems posed by solving a linear system and introduce several approaches for numerical solutions.

5.1.0.1 Iterative Methods

In the evaluation of basic iterative methods [34], assume that A is a nonsingular $n \times n$ (sparse or dense) matrix, and that its diagonal entries a_{ii} are all nonzero. Let b be a column vector, then the system of linear equations can be expressed as $Ax = b$:

$$\sum_{j=1}^{n} a_{ij}x_j = b_i \quad for\ 1 \leq i \leq n \tag{5.1}$$

There are several iterative methods to solve the above equation such as Jacobi, Gauss-Seidel, and Successive Over-Relaxation, etc. The process of iteration (also called relaxation) is to evaluate a grid point x_j by referencing the values of its neighbor points x_i, $i = 1, ..., n.\ i \neq j$. In general, iterative methods are suited for execution in a multiprocessor environment because parallelism is inherent in the relaxation processes and a minimum amount of interprocessor communication is needed among relaxation processes [25].

We will concentrate on the Jacobi method and a new class of asynchronous solutions, namely chaotic relaxation. The main difference between synchronous and asynchronous algorithms is the model of execution. In synchronous algorithms,

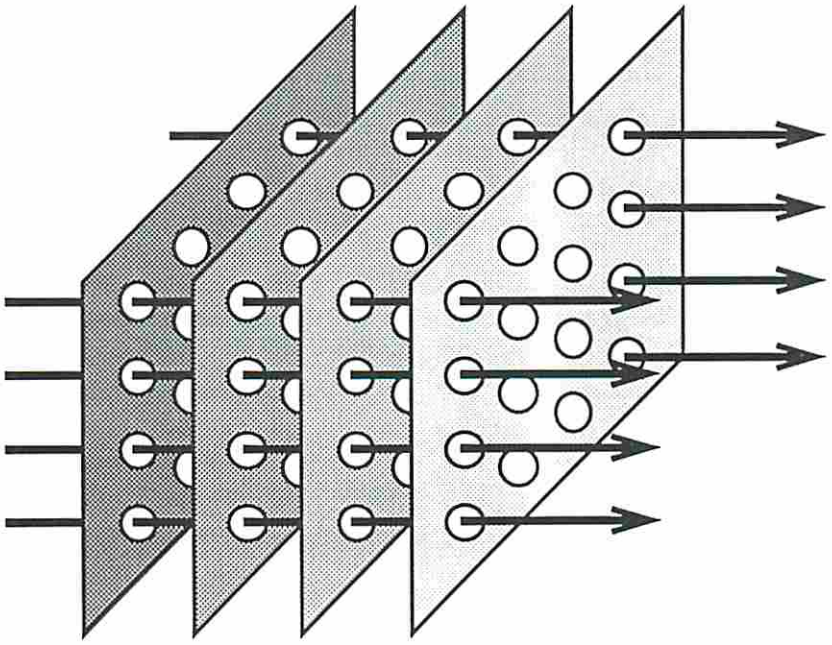

Figure 5.1: Synchronous Execution on two-dimensional grids.

synchronization between processes is required in order to update grid points accordingly. In other words, it is a step-by-step execution (Fig. 5.1). On the other hand, asynchronous algorithms are executed in a "*chaotic*" style: each process is executed without any constraint and grid points can be updated independently as soon as new values have been produced (Fig. 5.2).

5.1.0.2 Jacobi Method

The Jacobi iterative method can be derived from equation (1) as:

$$x_i^{(k+1)} = \frac{-\sum_{j \neq i, j=1}^{n} a_{ij} x_j^{(k)} + b_i}{a_{ii}} \quad for\ i = 1, \ldots, n\ and\ k \geq 0 \qquad (5.2)$$

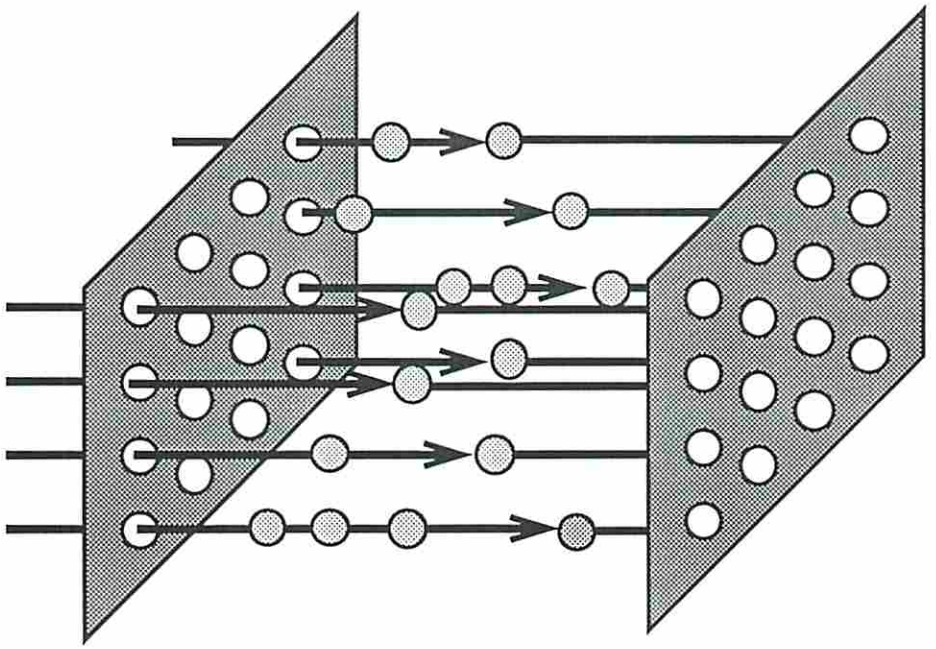

Figure 5.2: Asynchronous Execution on two-dimensional grids.

where the $x_i^{(0)}$'s are initial estimates of the components of the solution x. By examining the Jacobi iterative method shown above, it can be seen that all the components of the previous (*old*) vector $x^{(k)}$ must be saved before the components of the next (new) vector $x^{(k+1)}$ are computed. Therefore, in this algorithm, an iterative sequence of approximations $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$ will be sequentially computed.

To terminate the iterative process, a certain prescribed accuracy ϵ is chosen as the termination criterion. The iterations will terminate when the condition is satisfied with $||x^{(k)} - x^{(k-1)}|| < \epsilon$ and the solution $x^{(k)}$ will be accepted as the answer.

5.1.0.3 Chaotic Relaxation

In the asynchronous approach, each process continues execution to update the elements in $x_{(i)}$ and communication between processes has been achieved by reading the dynamically updated variables. A subset of asynchronous methods, called *chaotic relaxation schemes*, was introduced by Chazan and Miranker [13] to solve linear systems. In a chaotic relaxation scheme, practical constraints on the asynchronous behavior are imposed. In the following, we shall use Baudet's model [10] to describe the asynchronous and chaotic relaxation methods:

> **Definition**: Let **F** be an operator from R^n to R^n. An asynchronous iteration corresponding to the operator **F** and starting with a given vector $x(0)$ is a sequence $x(j)$, $j = 0, 1, \ldots$ of vectors of R^n defined recursively by:
>
> $$x_i(j) = \begin{cases} x_i(j-1) & \text{if } i \notin J_j \\ f_i(x_1(s_1(j)), \ldots, x_n(s_n(j))) & \text{if } i \in J_j \end{cases} \tag{5.3}$$
>
> where J_j is a sequence of nonempty subsets of 1, ... ,n and $s_i(j)$, for $i = 1, 2, \ldots, n$, is a sequence of elements in N^n.

In addition, J_j and $s_i(j)$ are subject to the following conditions, for each $i =$ 1, ... ,n:

1. $s_i(j) \leq j - 1, \quad for\ j = 1,2, \ldots$;

2. $\lim_{j \to \infty} s_i(j) = \infty$;

3. i occurs infinitely often in the sets J_j, $j = 1, 2, \ldots$.

An asynchronous iteration corresponding to $\mathbf{F}$, starting with $x(0)$, will be denoted by $(\mathbf{F}, x(0), \mathbf{J}, \mathbf{S})$, where $\mathbf{J} = \{J_j \mid j = 1, 2, \ldots\}$ and $\mathbf{S} = \{(s_1(j), \ldots s_n(j)) \mid j = 1, 2, \ldots\}$. Based on this model, we describe the following methods on a square grid:

- *Conventional relaxation*: Intuitively, it can be said that in conventional relaxation, a grid point is updated when some central controller has determined that the new value (current iteration) of the neighboring grid points was available. Processing proceeds in this *lockstep* fashion.

- *Asynchronous relaxation*: In this method, each grid point proceeds independently of the status of its neighbors. Each grid point may be trying to calculate its value for iteration k, while the other available values correspond to entirely different iterations (k_1 for the North neighbor, k_2 for the East, k_3 on the West, and k_4 on the South).

- *Chaotic relaxation*: This varies slightly from the previous definition in that, while an asynchronous algorithm imposes no restriction on how "old" a value may be (*i.e.*, how many iterations ago it was produced), chaotic relaxation requires that the updated value of a point be received within a fixed amount of time. Hence, in the above definition, $s_i(j)$ only references the latest updated value. This changes the condition in 2. as:

2': There exists a fixed integer t such that $j - s_i(j) \leq t$ for $j = 1, 2, \ldots$ and $i = 1, \ldots, n$.

In this sense, the chaotic relaxation method is more satisfactory in practical applications although the definition of asynchronous methods is mathematically more rigorous.

To decide the termination criterion of undeterministic behavior, such as in a chaotic relaxation, is a difficult problem. Indeed, in a chaotic relaxation, the number and the sequence of iterations at each grid point are different from each other due to the asynchronous execution of relaxation at each grid point.

5.2 From Algorithms to Data-flow Graphs

We have now established the two categories of algorithms for linear system solvers that we will implement and evaluate on our VTD system. The expression of these algorithms in a high-level data-flow language will now be undertaken and translated into data-flow graphs. While there are many synchronization constructs defined in SISAL, a new class of synchronization constructs have to be introduced in order to describe the chaotic behavior in asynchronous algorithms. In the following, the two approaches (*synchronous* and *asynchronous*) will be expressed by SISAL and new language constructs will be proposed.

5.2.1 The Jacobi Method and Synchronous Constructs

The Jacobi method discussed in the previous section can be described by a block diagram and translated into SISAL.

5.2.1.1 The Block Diagram

In the block diagram shown in Fig. 5.3, a matrix $A[i,j]$ with a size of $(n \times n)$, a column vector $B[i]$, and an initial column vector $X[i]$ are first accepted as the input data. Each function block can be described as follows:

- *Multiplication* : Streams of one and two-dimensional data tokens are first passed through the multiplication procedure which generates $A[i,j] \times X[j]$, for $j = 1$ *to* N

- *Summation* : The output data from the multiplication procedure are then summed up to prepare the generation of the new values of $X[i]$.

- *Generate New* $X[i]$: These data are forwarded along with the $B[i]$ vector data to generate the new vector $X[i]$ by using equation (2).

- *Generate Error Norm* : This stage computes the error vector from the old and the new vectors $X[i]$ (*i.e.*, $X^{(i)}$ and $X^{(i+1)}$).

- *Convergence Check* : The block of convergence check evaluates whether a solution has been reached by checking the output from the block of the error norm against the prescribed accuracy ϵ and makes a decision of whether to loop the tokens back or to route them to the output module.

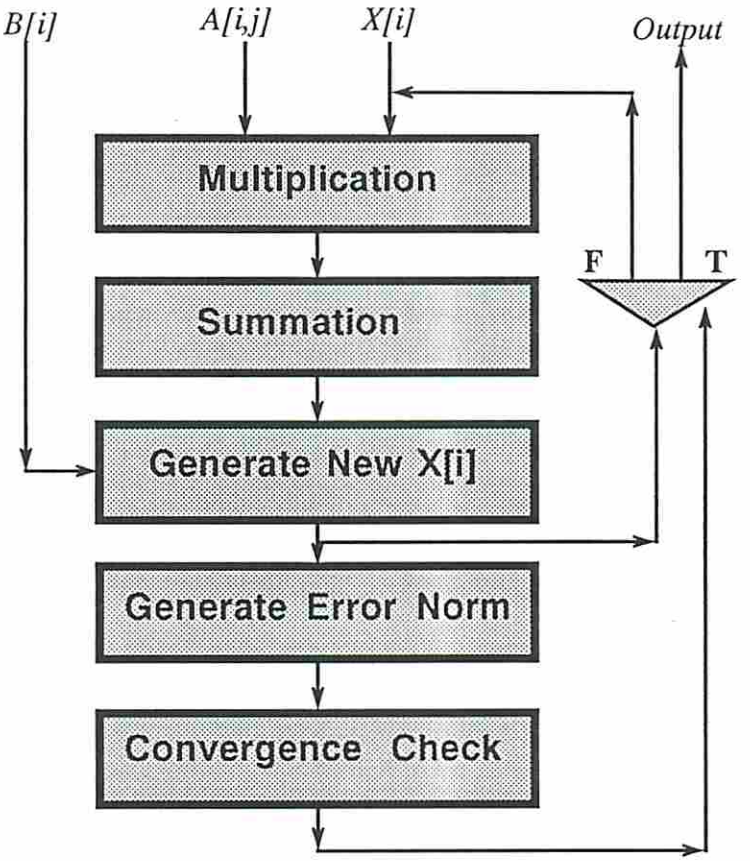

Figure 5.3: The Block diagram for Jacobi Methods ($A \times X = B$).

Once the new $X[i]$ is accepted, the output would be the solution vector $X[i]$ with required accuracy ϵ. It should be noted that each block in the diagram is iteratively performed and the major computation in this algorithm is required by repeatedly computing the new values of vector $X[i]$ and evaluating the termination criterion.

5.2.1.2 The SISAL Programs

The block diagram described in above can be translated into SISAL programs as shown in Fig. 5.4. In the programs, the computation starts from line 4 to decide whether relaxation is needed to be proceeded. The *Relaxation* procedure (from line 5 to 15) performs the relaxation for all of the elements in $X[i]$ and generate new values of vector $X[i]$. The *Convergence Check* procedure (from line 16 to 22) checks through all the elements and generates a termination signal back to line 4. Here, we use a stopping criterion evaluated by an L_∞ norm.

In the SISAL program, one should note that the relaxation on each elements $X[i]$ under the *for* constructs (line 5 and 6) can be executed in parallel mainly due to the definition of language constructs. In the same way, the convergence check on each elements of $X[i]$ and *old* $X[i]$ can also be executed in parallel under the *for* constructs in line 16. However, the algorithm itself will be executed in a synchronous manner. In other words, a *step-by-step* iteration process will sequentially proceed.

5.2.1.3 The Data-flow Graphs

Once the high-level SISAL programs are written, the corresponding data-flow graph can be obtained by compiling the SISAL programs into a fine-grain graph. Here, the fine-grain graph refers to a data-flow graph that contains a single operation in each actor.

A summation ($\sum_{i=1}^{n} x_i$), shown in Fig. 5.5, is an example which demonstrates how a program can be compiled into a fine-grain data-flow graph. In this graph,

```
define main, jacobi
type OneDim = array[real];
type TwoDim = array[OneDim] ;
function jacobi ( A : TwoDim ; B : OneDim ; N : integer ;
                  returns OneDim )
(1) for initial
        Err := 0 ;
        X := array [1: 0.0, 0.0, 0.0, 0.0] ;
(4)     while Err < N repeat          % convergence check
(5)         X := for i in 0, N        % relaxation on X
                temp1 := for j in 1, N
                    temp2 :=
                    if i ≠ j
                        then A[i,j] * old X[j]
(10)                    else 0.0
                    end if ;
                    returns value of sum temp2
                end for;
                returns array of ( B[i] -temp1 ) / A[i,i]
(15)        end for;                  % generate new X
            Err := for i in 1, N      % generate error norm
                temp3:= if abs(X[i] - old X[i]) < ε
                    then 0
                    else 1
(20)            end if ;
                returns value of sum temp3
            end for ;
            returns value of X
        end for
    end function
```

Figure 5.4: A SISAL program for Jacobi methods.

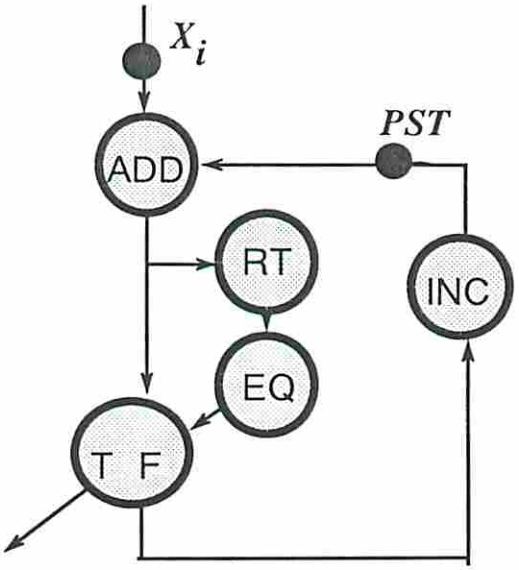

Figure 5.5: An Example of Fine-grain Data-flow Graphs.

when a data-token reaches the Addition (ADD) actor, it will match with the Partial Sum Token (PST) which is set to 0 at the beginning. The result of the addition will then be sent to the Read Tag (RT) actor and the True-False (TF) actor. After the RT actor reads the tag of the token, it sends the tag to the EQ (equal) actor to check whether the tag is equal to n. If the tag is equal to n, then the result is produced from the True gate of the TF actor. If not, the tag of the PST token will be increased by the INC actor and sent back to the ADD actor.

After the compiling process that translates SISAL programs into fine-grain data-flow graphs, the graphs can then be partitioned into coarse-grain data-flow graphs where macro-actors are formed by collecting several micro-actors. We will discuss the partition scheme at the end of this section.

5.2.2 Chaotic Relaxation and Asynchronous Constructs

Chaotic relaxation is another approach particularly successful in parallel environments. However, new language constructs must be introduced in order to describe the algorithm.

5.2.2.1 The Block Diagram

As discussed, the main difference between the two algorithms lies in the execution model. Therefore, the block diagram of chaotic relaxation is similar to the block diagram of the Jacobi method (see Fig. 5.3). In the block diagram of chaotic relaxation, in order to describe it correctly, two groups for the five blocks in Fig. 5.3 are formed: the first group includes *Multiplication*, *Summation*, and *Generate New X* and the second group includes *Generate Error Norm* and *Convergence Check*. When execution begins, both groups will be executed in parallel and there is no dependency between these two groups. In addition to the concurrent execution of these two groups, there also exist many individual indices in the first group that should be calculated asynchronously in parallel. More precise description of how chaotic relaxation works will be discussed with the SISAL programs which contain asynchronous language constructs.

5.2.2.2 The Asynchronous Constructs

Asynchronous computations cannot be easily implemented by traditional high-level programming constructs. Therefore, we designed the *async-repeat* and the *async-for* operations to represent the asynchronous behavior. The main idea of

the new async-repeat and async-for constructs is to release the synchronization constraints from the *repeat* and *for* constructs in SISAL that inherently create synchronization points in the body of the constructs.

1. **Async-repeat** : This construct allows the procedures inside it to be concurrently evaluated without any synchronization between one another. For example, in the following program, statement (1) and statement (2) can be executed simultaneously and repeatedly as long as the condition $c \leq 100$ remains true:

```
for initial
while c ≤ 100 asyn-repeat
        a:= old a + 1 ; - - - - (1)
        c:= a + b ; - - - - - - - (2)
return value of c;
end for
```

   In the above program, under the asynchronous construct, statement (2) could be executed and generate its result before the completion of the execution of statement (1). In other words, under independent execution of statement (1) and (2), c may be already larger than 100 (assume $b = 101$ and $a = 0$) and terminate the process before a is increased by statement (1). Note that the execution model described above will not be allowed in the conventional *repeat* construct which only executes the two statements sequentially due to the synchronization point imposed by the language construct. Besides, the asyn-repeat constructs will be particular successful when some procedure inside the construct is an infinite loop. For example,

in the following program, if procedure_one is an infinite loop that generates results for procedure_two, then only the results of procedure_two can satisfy the condition and terminate the whole process. Therefore, the asyn-repeat construct must be imposed to break the synchronization point between procedure_one and procedure_two.

```
for initial
while condition asyn-repeat
        Procedure_one ;
        Procedure_two ;
end for
```

2. **Asyn-for** : While the conventional *for* construct 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 index values in parallel. For example, in the following program, the return values of array X does not need to wait until all the new values of each index i become available. Instead, each new value of index i can be updated asynchronously as soon as the value is available and the next computation can be started.

```
for initial
X := asyn-for i in 0, N
        temp := Y[i] × old X[i] ;
        returns value of temp × temp
end for;
```

While the asyn-for construct allows each index i to be evaluated asynchronously, it should be noted that the operations within the construct is

an infinite loop that the computation will be proceeded until other process (outer loop) terminates the whole processes. The following program is an example to show that the process under the asyn-for construct will be terminated by the process that under the asyn-repeat construct.

```
for initial
while condition asyn-repeat
    X := asyn-for i in 0, N
        temp := A[i] × old X[i] ;
        returns value of temp × temp
    end for;
    Procedure_two ;
    Procedure_three ;
end for;
```

Overall, under the bodies of the new async-repeat and async-for constructs, synchronization constraints can be released while the repeat and for constructs create synchronization points inside the bodies of the constructs. However, in general, the asyn-for constructs will require the asyn-repeat to co-exist in a program. This is because only the asyn-repeat construct can terminate the process under the asyn-for constructs.

5.2.2.3 The SISAL Programs

The chaotic relaxation can be expressed in SISAL by using the new constructs.

First, in order to allow several procedures to be executed asynchronously in parallel, the *repeat* must be replaced by *async-repeat* at the outer loop of these procedures. In Fig. 5.4, in line 4, the *repeat* should be replaced by *async-repeat.* Therefore, under the *async-repeat* construct, both the *Relaxation* procedure (from

line 5 to 15) and the *Termination Check* (from line 16 to 23) can be executed concurrently without any dependency between each other.

Second, in order to allow each index value, which is under the *for* construct, to be executed asynchronously in parallel, the *for* must be replaced by *async-repeat* at the beginning of the procedure. In line 5, we replace *for* by *async-for*. Therefore, inside the *async-for* construct, each index value can concurrently proceed the execution of the next computation without waiting for other index values which are executing the same function.

To conclude the discussion of asynchronous methods, two remarks should be made: First, the chaotic relaxation requires every process to proceed without waiting for other processes. Second, this chaotic behavior can be represented by the unsynchronized two groups of the block diagram which indeed perform chaotic relaxation and can be described in SISAL programs by exploiting asynchronous language constructs.

5.2.3 Coarse-grain Data-flow Graphs

In fine grain data-flow computation, due to much overhead to respect the functionality in execution, it will result in poor performance for low levels of parallelism. Indeed, to execute fine grain graphs (where each actor represents a single instruction), overhead must be enforced to associate with actual computational actors to insure the correct execution. Therefore, overhead problems will be inevitably created and degrade performance.

When the grain size of a graph is increased, in other words, an actor now represents several operations instead of one single operation in graphs, the overhead problem can be easily alleviated. The concept of *macro-actors* (several operations are grouped into a single actor) described by Gaudiot and Ercegovac [19], is shown to improve performance in a fine grain computation model. 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 actors (with larger granularity) may reduce the available parallelism in programs and increase memory and communication latency. Hence, forming macro-actors from fine grain micro actors is a trade-off between latency and parallelism.

In the course of this research, a macro-actor formation scheme based on lumping the sequential execution paths has been selected for the two iterative methods. We first inspect the data-flow graph of each algorithm and then lump the sequential portions of the graph into macro-actors. The primary benefit derived from this approach is that one can easily reduce the inter-PE and intra-PE communication overhead without losing parallelism at execution time. In this scheme, fat tokens, which contain several data elements in a token, can also be avoided and hence no extra latency will be introduced. In the following, we will demonstrate how this scheme can be applied to data-flow graphs:

for i in 1, h-1 cross j in 1, h-1
return array of $A[i,j,k]$
    $A[i,j,k] + A[i,j-b,k]$
end for;

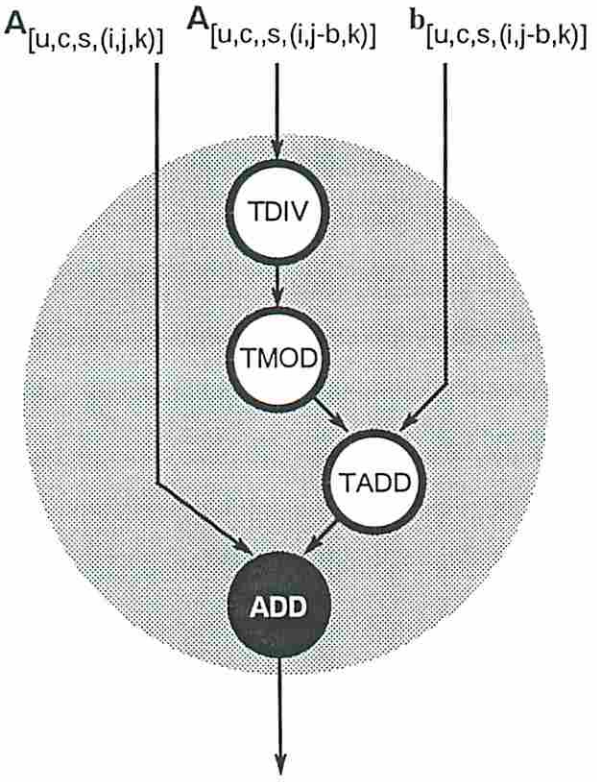

Figure 5.6: Macro-actors Formatting Scheme.

For example, to perform the above SISAL statements whose data-flow graphs are shown in Fig.5.6, the value of $A[i, j-b, k]$ must be obtained and then the index(tag) of the value must be changed in order to add with the value of $A[i, j, k]$. From the graph, we find that a sequential path which consists of four sequential actors exists in this function. Hence, we form a macro-actor to perform the four sequential instructions.

The scheme discussed above can be applied to both algorithms. For example, in Fig. 5.3, we find that, the major sequential portion of the algorithms is the summations in the Summation Block and the Convergence Check Block. Hence, we form a macro-actor to perform the five sequential instructions. A simplified data-flow graph of the Jacobi method with micro and macro actors is demonstrated in Fig. 5.7.

5.3 The VTD System

While the macro-actor concept has reduced the overhead in the fine-gain computation, the architectures must be able to execute actors that contain various-size. Our Variable-grain Tagged-token Data-flow (VTD) system has therefore been designed for this purpose. A new firing rule in the VTD system is also proposed to guarantee the proper behavior of chaotic relaxation and to achieve efficient computations.

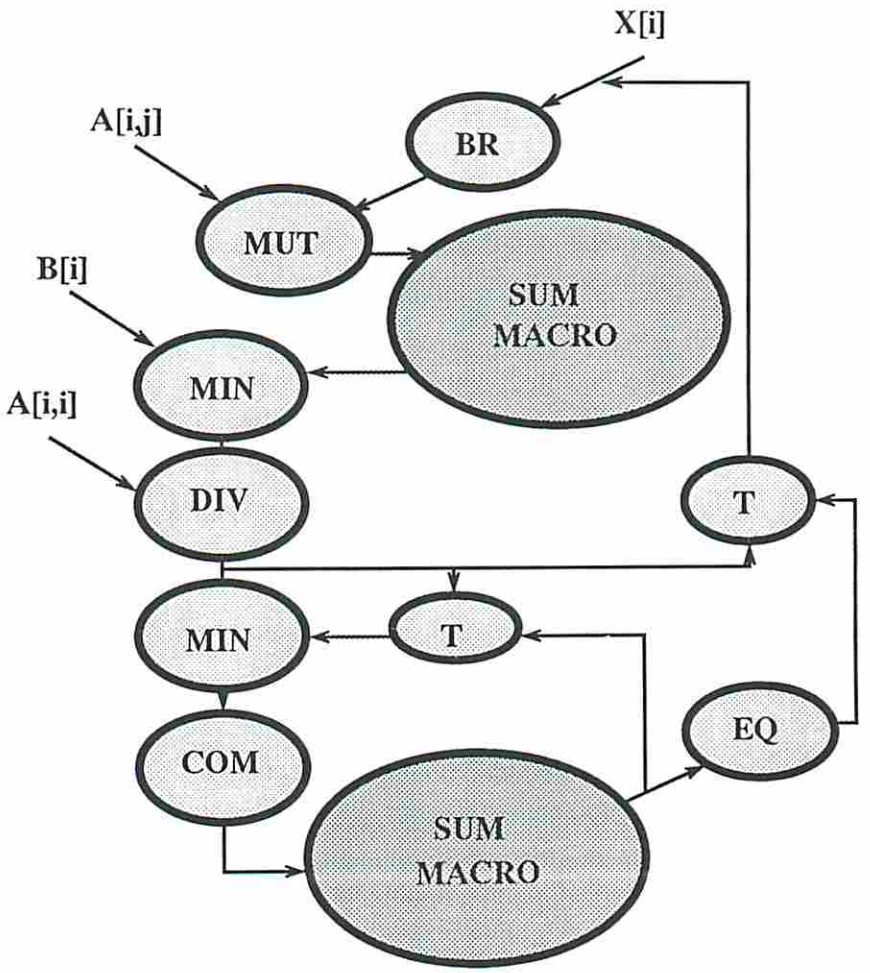

Figure 5.7: An Example of Coarse-grain Data-flow Graphs

5.3.1 Executional Models

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. Indeed, 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.

In a variable-grain data-flow system, at the graph level, the program is represented by a collection of various resolution of actors under the dynamic data-flow model. At the execution level, several execution modes can be considered inside a macro-actor:

1. *von Neumann Model* : This is a hybrid of the data-driven and control-driven models. This model allows the micro-instructions in a macro-actor to be treated as conventional instructions and be executed with a program counter in a processor.

2. *Static Data-flow Model*: This model adapts the *data-forwarding* technique, which has been developed in pipelining multiprocessor systems, to execute the micro-instructions in a macro-actor. For example, the micro-instructions in a macro-actor can be executed under the Acknowledge scheme in a processor.

3. *Dynamic Data-flow Model* : This is the most complex model; it allows a processor to execute those micro-instructions in a macro-actor as in the tagged-tokens data-flow model.

5.3.2 The VTD System

We have selected the first execution model in our VTD system. The VTD system consists of a set of identical Processing Elements (PEs) connected by a hypercube (message-passing) communication network. The structure of a single PE is shown in Fig. 5.8 and consists of 4 units :

- *Matching Store Unit* : In the Matching Store Unit, the tag of the incoming token is associatively checked against that of previously arrived tokens to determine whether it is the last token of the same set to arrive at given instructions. The other tokens should be stored in the associative memory of the Matching Store Unit and held until the last token of the same set arrives. For the last token, the corresponding instance of the instructions can be activated by sending an *argument packet* to the next unit.

- *Instruction Fetch Unit* : The Instruction Fetch Unit receives this packet and fetches the parameters of the instructions. Note that the template contains not only the opcodes but also pointers to the destination actors to which the result of the operations should be sent. A complete *instruction-ready packet* can be formed and sent for execution to the ALU.

- *ALU* : The von Neumann execution model is applied to the packet from the previous stage. The ALU executes the operations with a program counter indicated by the incoming packet and produces result tokens which are received by the Token Formatting Unit.

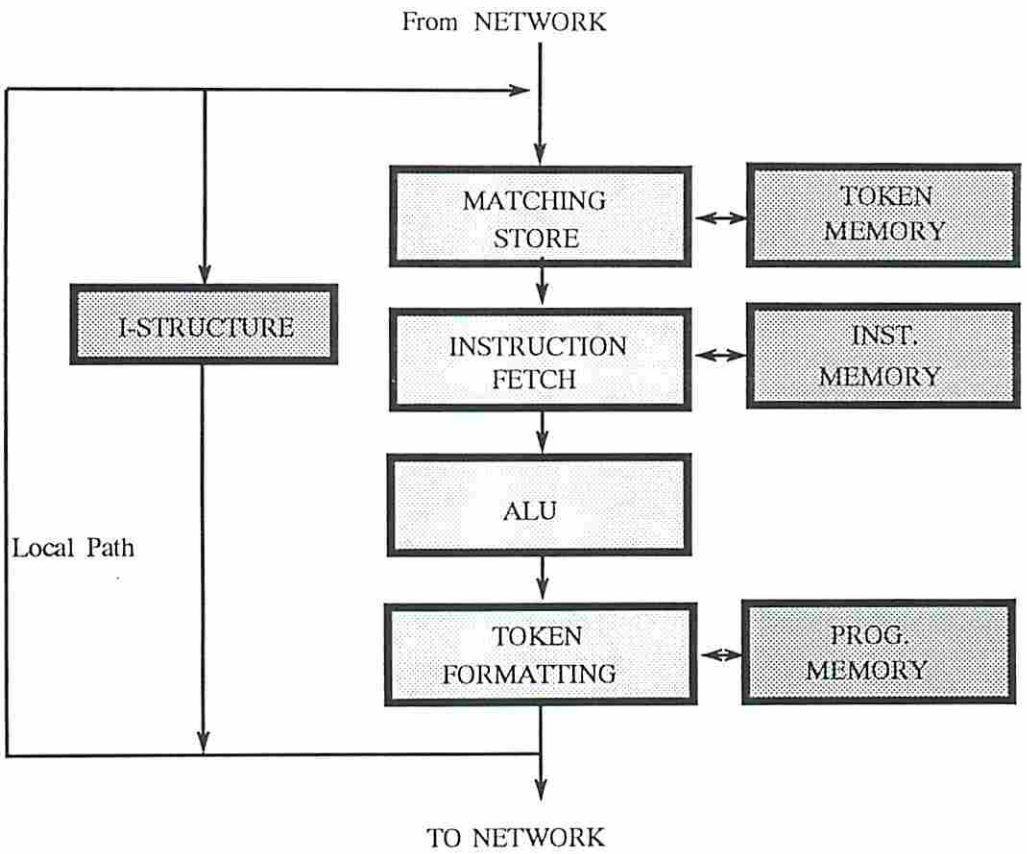

Figure 5.8: A Simplified Structure of a PE

- *Token Formatting Unit* : The TFU receives tokens which have been produced by the ALU. These tokens comprise several fields: the tag associated with the operation (after modification if the operation was a tag modifying operation), the data itself, as well as an *allocation function* field. This field is used by the TFU to determine the destination PE of the token.

5.3.3 The Matching Store with Locks

In chaotic relaxation, due to the asynchronous iterations at each grid point, the value of each grid point must be saved for the relaxation of other grid points. In order to guarantee the proper behavior of the chaotic actors, 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 that 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. 5.9 shows step-by-step the operation of the new firing rule of an actor along with the *matching store with locks*:

1. Initially, when either token A or token B (A and B have the same tags) comes into the actor F, it will be *locked* inside the actor.

2. When the partner token arrives, actor F will be fired and will produce 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 C 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.

It is easy to find that the new firing rule not only is suited for a chaotic relaxation but also can efficiently execute those *"updatable constants"* operations in a general data-flow graph. For example, in the Jacobi method, the matrix A can always be *locked* in the matching store and wait for the new values of vector X after multiplication with the old vector of X. Similarly, the vector b can be *locked* in the Matching Store for generating every new value of vector X. However, one should note that garbage collection and more computation power are required due to the existence of locks.

The idea of the *locks* in the above discussion can also be easily implemented in an indirect memory access scheme (*e.g.* I-structures). In this case, we can simply declare that the block which contains the locks in I-structure storage is a common memory area that can be read and updated as needed. However, it has been shown in [21] that the direct memory access scheme (*e.g.* Token Relabeling) performs better than the indirect memory access scheme in most numerical computations. Hence, we will only study a VTD system with direct memory access scheme.

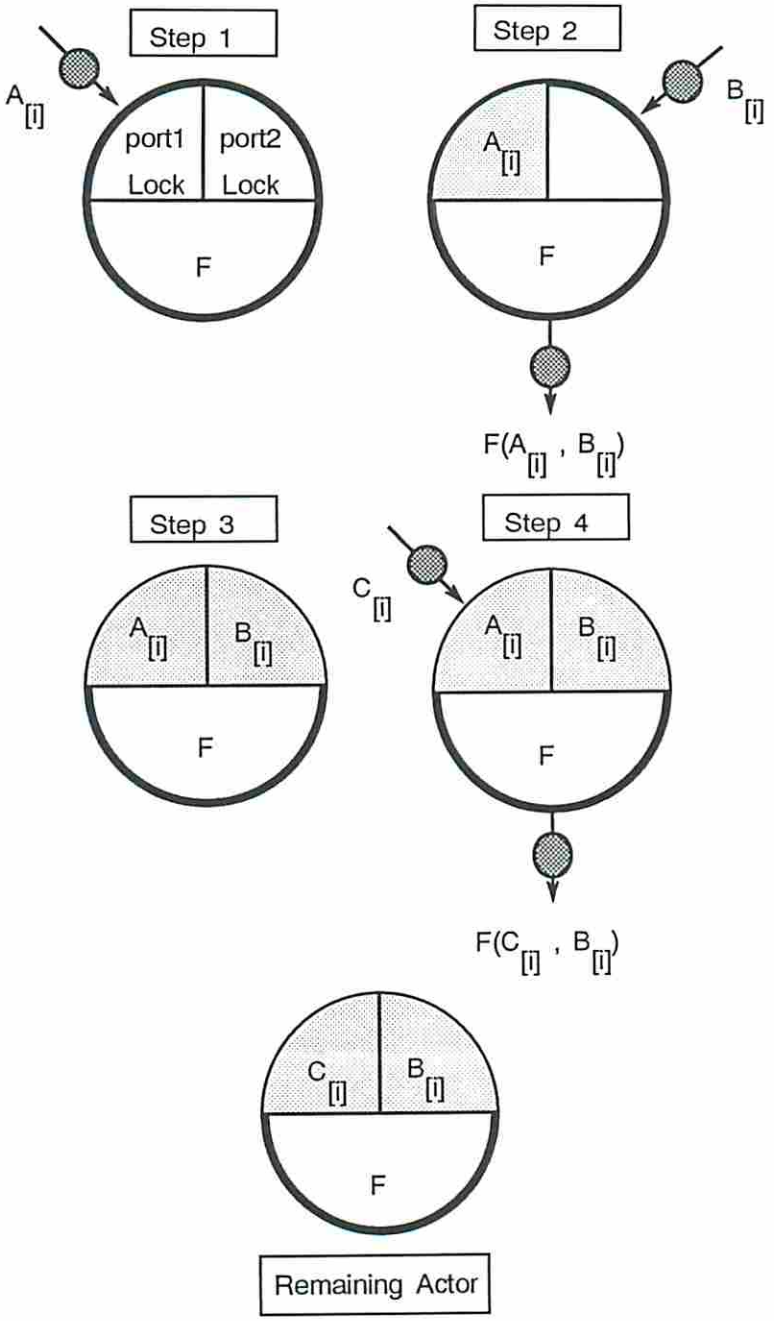

Figure 5.9: A New Firing Rule with locks in Matching Store.

5.4 Performance Measurements

To characterize the performance in the VTD system, new performance measurements are defined along with the conventional performance measurements.

5.4.1 Conventional Measurements

Many system performance measurements, such as speedup and system utilization, have been used to evaluate multiprocessor systems in the past. However, these measurements do not clearly indicate the effectiveness of architectures as well as application programs because there is no indication of how much of the performance is due to the architectures and how much of it is due to the applications. Indeed, the speedup should be measured by scaling the problem to the number of processors, not by fixing problem size. An explanation of misuses of Amdahl's speedup formula has been demonstrated in [24]. Therefore, when multiprocessor systems are evaluated, both parallel algorithms and parallel architectures are mutually required to achieve high performance. For instance, a parallel machine cannot deliver high efficiency in executing a sequential program due to the lack of parallelism within the program. On the other hand, a parallel algorithm cannot guarantee high performance in multiprocessor system if the system cannot exploit the parallelism involved in the program. Clearly, what we need are better performance measurements to reflect the degree of exploited parallelism resulting from algorithms as well as the ability of the architectures to utilize such parallelism.

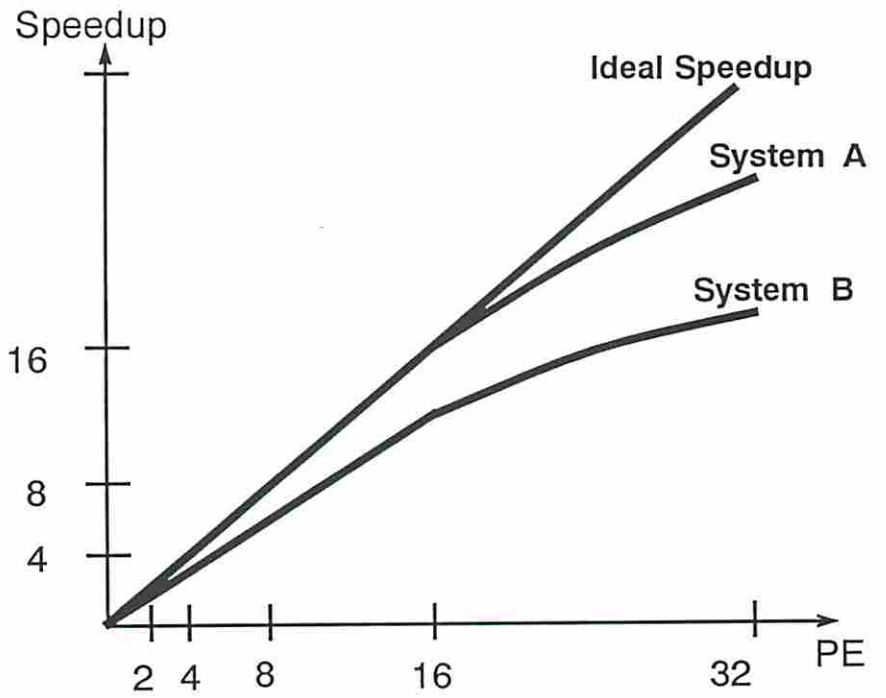

Figure 5.10: Speedups with Various PEs.

We are interested in measuring and comparing the performance of algorithms as well as our VTD system: First, to evaluate the performance of the architecture, the conventional *Speedup* measurement is taken to depict the trend of the performance with larger machine configurations. Second, to estimate the amount of the growing parallelism within an algorithm when the algorithm's complexity has been increased, a new measurement, called *Growth Factor*, is defined to show how suitable of an algorithm is for multiprocessor systems. Third, to measure how efficient of parallel systems in executing parallel algorithms, we introduce a new measurement, called *Scalability Factor*, to demonstrate the scalability property of the systems. Finally, we define the *Robustness* to indicate the potential performance of the systems.

5.4.2 Speedup

The speedup has been conventionally defined as the ratio of the execution time of an Application (AP) on N Processing Elements (PEs) to the execution time of the same application on a single PE :

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

Under this definition, an *ideal* speedup of an architecture is equal to N when there are N PEs in the system. In other words, if a speedup curve is closer to the line of ideal speedup, then the architecture is considered a better parallel system. For instance, in Fig. 5.10, it shows that system "A" performs better than system "B" in term of system "A" having a speedup curve closer to the line of ideal speedup. However, in this definition, it only shows how much of the execution time can be

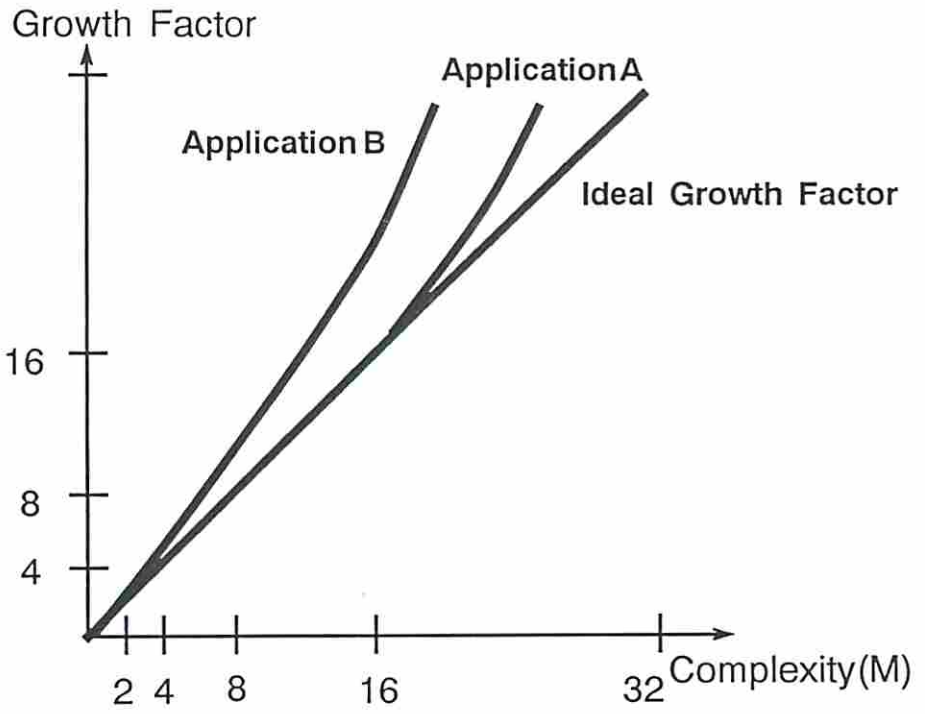

Figure 5.11: Growth Factors with Various Complexity (M).

reduced in various system configurations while the amount of complexity in the application remains unchanged. And this does not show how well an algorithm could be implemented on multiprocessor systems. In other words, the speedup curves only demonstrate the machine domain performance without considering the application aspect.

5.4.3 Growth Factor

Before the speedup in a system is measured, how well an application can perform in parallel systems must be studied. The growth factor is used to show how much of the amount of parallelism changes when the complexity of an algorithm is changed. Here, the complexity of an algorithm refers to the number of operations needed to execute the algorithm. For example, the inner product of vectors $V(a_1, a_2, a_3, ..., a_m)$ and $U(b_1, b_2, b_3, ..., b_m)$ has a complexity of $O(m)$. The growth factor therefore is defined as the ratio of the execution time of an Application (AP) with a complexity $(M \times m)$ on a fixed number of n PEs to the execution time of the same application with a complexity of m on the n PEs.

$$Growth\ Factor\ (AP(Mm),\ PE(n)) = \frac{Exe.\ time\ of\ Mm\ AP\ on\ n\ PEs}{Exe.\ time\ of\ m\ AP\ on\ n\ PEs}$$

Ideally, a perfectly parallel algorithm should have a growth factor proportional to the increasing rate of its complexity (M). For example, a vector to vector multiplication is a perfectly parallel statement in that the amount of parallelism increases at the same rate as the vector length (complexity). Therefore, if an application has a curve of growth factor close to the line of ideal growth factor, it should be considered a better parallel application. For example, in Fig. 5.11, application "A" is a better parallel application than application "B" because the curve of growth factor in "A" is closer to the line of ideal growth factor.

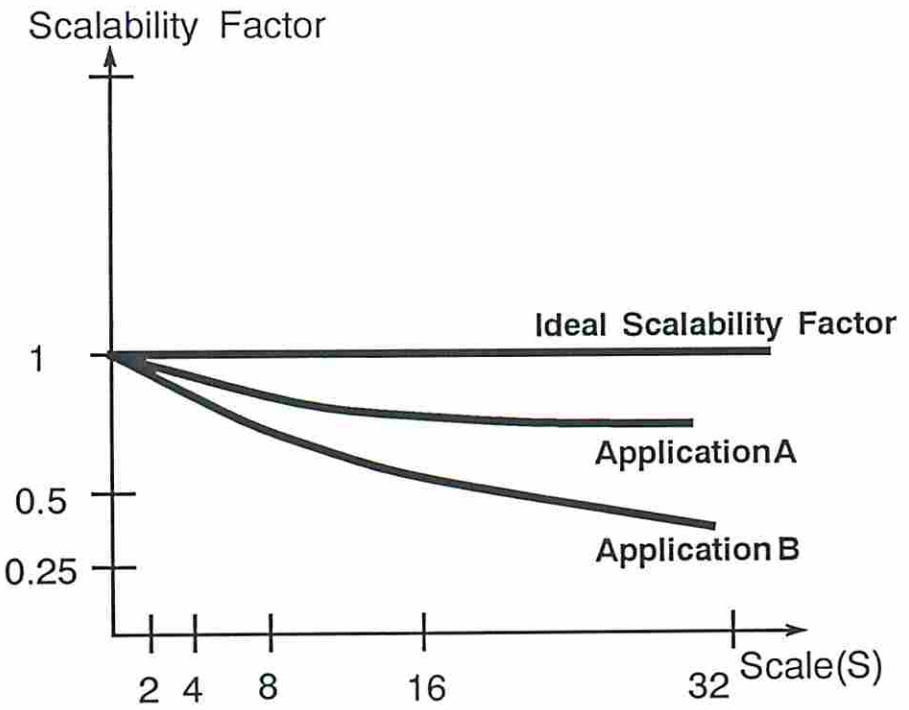

Figure 5.12: Scalability Factors with Various Scales (S).

5.4.4 Scalability Factor

In the course of parallel processing, multiprocessor systems are designed for solving large scale applications. The performance of multiprocessor systems should also be measured by comparing the execution time of solving large problems with that of solving small problems on single processor systems. In other words, the complexity in the applications should be increased while the size of the machine configuration is increased. The scalability factor is defined as the ratio of the execution time of an Application (AP) with a complexity (m) on n PEs to the execution time of the same application with a complexity of $S \times m$ on the $S \times n$ PEs.

$$Scalability\ Factor\ (AP(Sm),\ PE(Sn)) = \frac{Exe.\ time\ of\ m\ AP\ on\ n\ PEs}{Exe.\ time\ of\ Sm\ AP\ on\ Sn\ PEs}$$

If an algorithm has an ideal growth factor and a system has an ideal speedup, then the scalability factor should remain a constant for various values of N. In other words, a perfectly parallel algorithm with a large complexity on a large perfectly parallel system configuration should require the same execution time as it would with a small complexity on a small system configuration. However, due to the fact that most algorithms and systems are not perfectly parallelized, the actual scalability factors will fall below the line of ideal scalability factor. Fig. 5.12 shows that the closer the curve to the ideal line, the higher the scalability of the application and system configuration will be.

5.4.5 Robustness

The robustness property of a system can actually indicate its potential performance [21]. The robustness is defined as the ratio of the execution time of an Application (AP) with a complexity ($R \times m$) on one PE to the execution time of the same application with a complexity of $R \times m$ on the $R \times n$ PEs.

$$Robustness\ (AP(Rm),\ PE(Rn)) = \frac{Exe.\ time\ of\ Rm\ AP\ on\ one\ PE}{Exe.\ time\ of\ Rm\ AP\ on\ Rn\ PEs}$$

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, one of the most important parameters in evaluating a multiprocessor system is to observe the system performance with various problem sizes. We thus express the performance of an architecture by showing the robustness with a large number of PEs.

5.5 Simulation Results

Once the Jacobi method and chaotic relaxation have been programmed and compiled into data-flow graphs, the execution of the graphs in the VTD system can be verified by a deterministic simulation in both the micro-actor (fine-grain) and macro-actor (coarse-grain) execution models.

5.5.1 Simulation Assumptions

The architecture model of a VTD system within the von Neumann execution model 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 linear systems solvers, 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. The program graphs were constructed by using elementary actors and macro-actors including normal arithmetic/logic operators, gates, structure operators, and tag operators. We also assumed that each single elementary actor would take a single unit of delay in the ALU, while the execution time of a macro-actor is equal to the total execution time of micro-instructions inside the macro-actor.

5.5.2 Simulation Results

The execution of the Jacobi method and chaotic relaxation to solve various sizes of linear systems with the termination criterion $||x^{(k)} - x^{(k-1)}||_\infty < 10^{-3}$ have been simulated. An *ad hoc* scheme to terminate a chaotic relaxation has also been chosen:

1. First, we generate a termination process that collects the two most recently updated values from each grid point.

2. Then, we compute the termination criterion as: $||x^{(k)} - x^{(k-1)}||_\infty < \epsilon$ in the process, where the value of k at one grid point may be different from that at another.

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 5.1 : Execution Time and Speedup in Chaotic Relaxation.

From the simulation results, several statistics and observations have been obtained:

- **Execution time** : The execution times of chaotic relaxation in various system configurations with micro-actors and with macro-actors are reported in Table 5.1. The execution times of the Jacobi method in various system configurations with micro-actors and with macro-actors are reported in Table 5.2.

  Observation: The results show that the coarse-grain (macro-actor) graph needs less execution time than the fine-grain (micro-actor) graph in both chaotic relaxation and Jacobi method. In a single processor environment, we find that chaotic relaxation needs more time to execute than the Jacobi method. For example, it takes 108,291 time units for chaotic relaxation and only 79,924 time units for the Jacobi method. In a multiprocessor system, on the contrary, it can be seen that chaotic relaxation reduces the

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 5.2 : Execution Time and Speedup in the Jacobi Method.

execution time much more than the Jacobi method. This can be verified by comparing the execution time in one PE with that in 16 PEs (8,050 time units for chaotic relaxation and 9,759 time units for the Jacobi method in 16 PEs). These results demonstrate that chaotic relaxation is more suitable to multiprocessor systems.

- **Speedup** : The measurement of the speedup is defined in the previous section. The reports of the speedups in various system sizes for both chaotic relaxation and the Jacobi method are attached in Tables 5.1 and 5.2, while Fig. 5.13 shows the trend of the speedups with increasing PEs for the two different relaxation methods.

Observation: The results indicate that the speedup in chaotic relaxation is better than the speedup of the Jacobi method in both macro and micro execution modes. In chaotic relaxation, a superlinear speedup can be sometimes observed due to the nondeterministic property of the algorithm

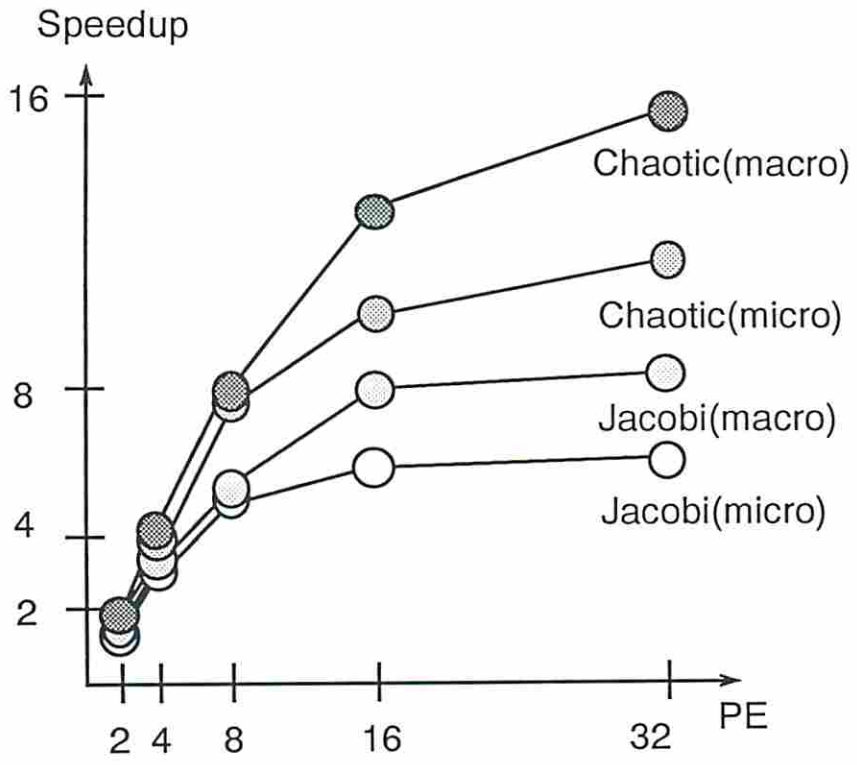

Figure 5.13: Speedup with Problem Size : 16 × 16.

itself. Indeed, the random sequence of relaxations may lead to a faster convergence in multiprocessor systems. This feature is confirmed in Table 5.1: the speedups in a 4 PE system for both macro and micro execution of chaotic relaxation can be as high as 4.01

- **Cross Speedup** : In order to compare the effect of the macro-actor execution mode with that of the micro-actor execution mode, *"cross speedup"* is calculated. The cross speedup is defined by comparing the execution time on N PEs in the macro-actor execution mode with the execution time on one PE in the micro-actor execution mode.

$$Cross\ Speedup\ (N) = \frac{Exe.\ time\ in\ one\ PE\ with\ micro\ execution\ mode}{Exe.\ time\ in\ N\ PEs\ with\ macro\ execution\ mode}$$

The reports of the cross speedups in various system sizes for both chaotic relaxation and the Jacobi method are shown in Table 5.3 and Fig. 5.14 show the trend of the speedups with increasing PEs in both relaxation methods.

Observation: The cross speedup, as defined in the above, indicates how much of the execution time can be improved from the worst case (in one PE) to the best case (in many available PEs). The results show that chaotic relaxation is more likely to achieve high performance in a multiprocessor system

- **Accuracy** : To check the precision of the two iterative methods, we have chosen the absolute L_2 norm (absolute error) to evaluate the accuracy of

Problem Size = 16 × 16		
System Size	Chaotic	Jacobi
number of PEs	Cross Speedup	Cross Speedup
1 PE	1.006	1.15
2 PEs	1.92	2.18
4 PEs	4.03	3.98
8 PEs	8.04	6.75
16 PEs	13.53	9.44
32 PEs	15.87	9.97

TABLE 5.3 : Cross Speedup in Chaotic and Jacobi Methods

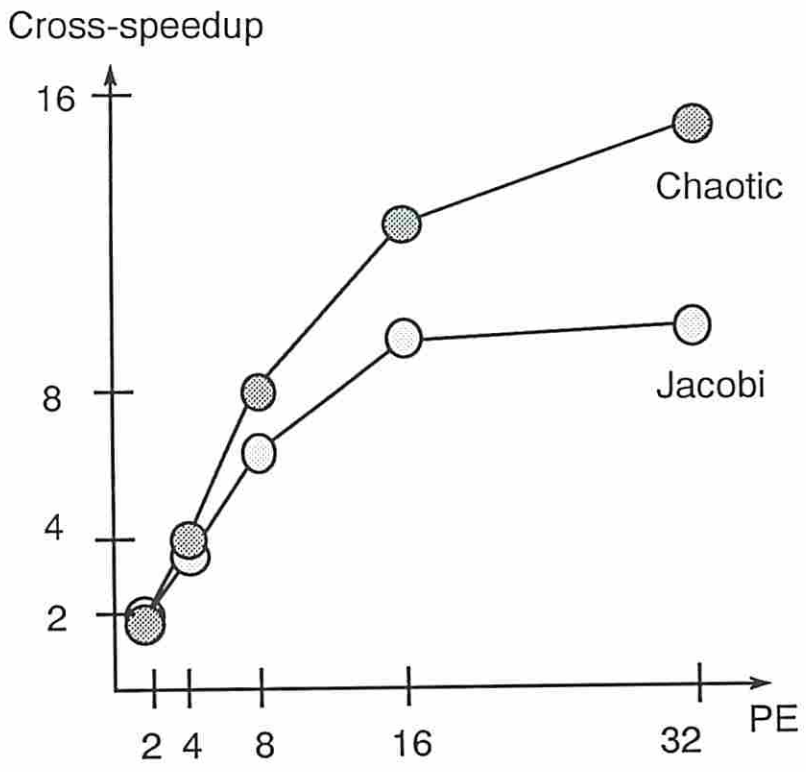

Figure 5.14: Cross Speedup with Problem Size : 16 × 16.

Absolute Errors of Various Methods				
Problem Size : 16 × 16				
System Size : 16 PEs				
Methods	Chaotic (Macro)	Chaotic (Micro)	Jacobi (Macro)	Jacobi (Micro)
L_2 Norm	1.30×10^{-4}	1.33×10^{-4}	4.26×10^{-4}	4.26×10^{-4}

TABLE 5.4 : Accuracy in Chaotic and Jacobi Methods.

the two methods in both the macro and micro execution modes. The results are shown in Table 5.4.

Observation: The results shows that the accuracy of chaotic relaxation has the same order as the Jacobi method. This insures that the intuitive termination criterion proposed in section 2 is an acceptable method for the convergence check in asynchronous algorithms.

- **System tune-up** : In order to observe the effect of network communication time we modified the time delay in the simulated communication network to verify the original assumptions. The execution times of both methods in macro-actor execution mode are compared in Table 5.5, while that in the micro-actor execution mode is reported in Table 5.6 and results of a larger problem size are reported in Table 5.7. The ratio of how the execution time changes in different time delay systems is shown in Fig. 5.15, 5.16, and 5.17. The sensitivity is defined as:

$$Sensitivity = (\frac{Exe.\ time\ on\ long\ delay\ systems}{Exe.\ time\ on\ short\ delay\ systems} - 1) \times 100\%$$

Problem Size = 16 × 16						
⋆ Network=1, Match=1, Fetch=1, ALU=1, Router=1 ♡ Network=10, Match=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	108291	0	79924	79924	0
2 PEs	56690	53746	-5	42112	47877	13
4 PEs	26999	33869	25	23109	36119	56
8 PEs	13548	20592	52	13640	28378	108
16 PEs	8050	17066	112	9759	24239	148
32 PEs	6867	19339	181	9244	30822	233

$$Percentage = (\frac{Exe.\ time\ on\ long\ delay\ systems}{Exe.\ time\ on\ short\ delay\ systems} - 1) \times 100\%$$

TABLE 5.5 : System Tune-up for Macro Execution.

Problem Size = 16 × 16						
⋆ Network=1, Match=1, Fetch=1, ALU=1, Router=1 ♡ Network=10, Match=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	0	92203	92203	0
2 PEs	54430	57478	5	49399	55164	11
4 PEs	27174	33498	23	27901	40806	46
8 PEs	14840	23794	60	18219	33072	81
16 PEs	10538	15198	44	14470	28856	99
32 PEs	9708	21671	123	13971	35393	253

$$Percentage = (\frac{Exe.\ time\ on\ long\ delay\ systems}{Exe.\ time\ on\ short\ delay\ systems} - 1) \times 100\%$$

TABLE 5.6 : System Tune-up for Micro Execution.

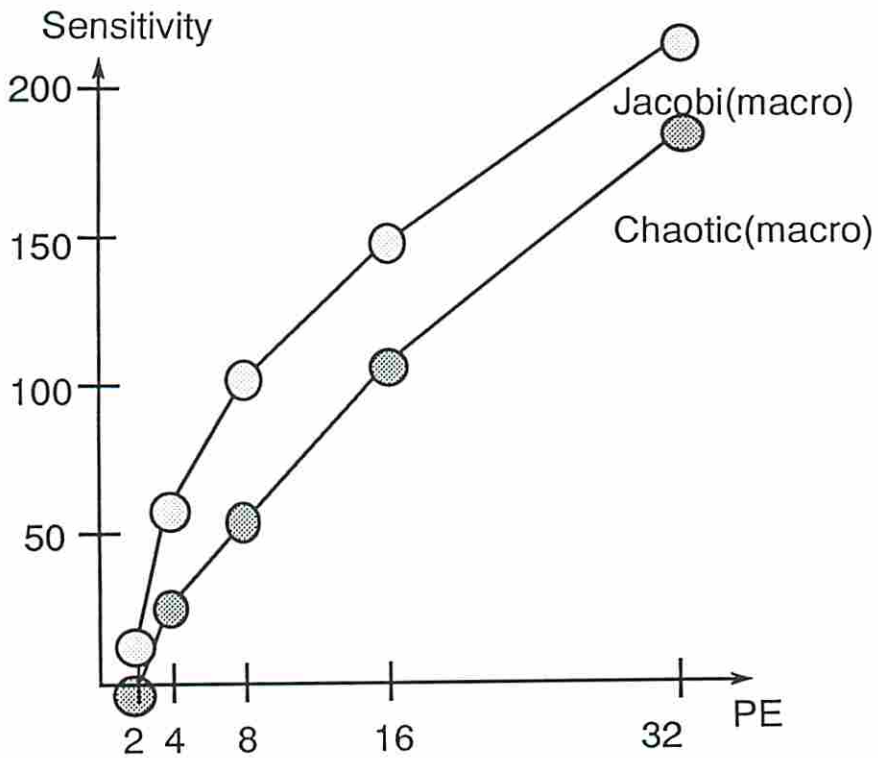

Figure 5.15: Effects of communication-delay on chaotic and Jacobi methods.

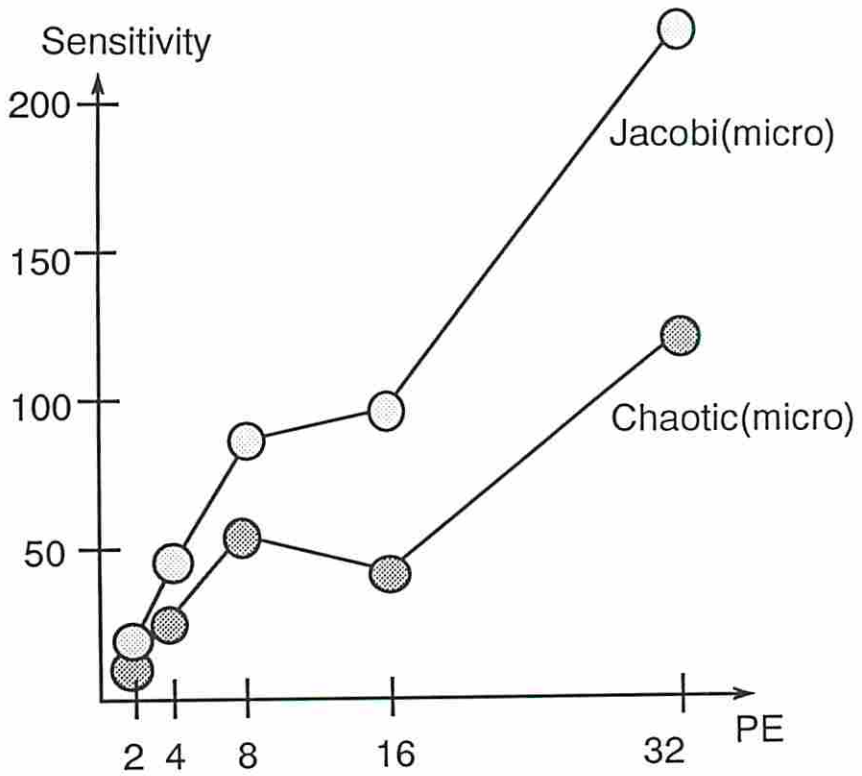

Figure 5.16: Effects of communication-delay on chaotic and Jacobi methods.

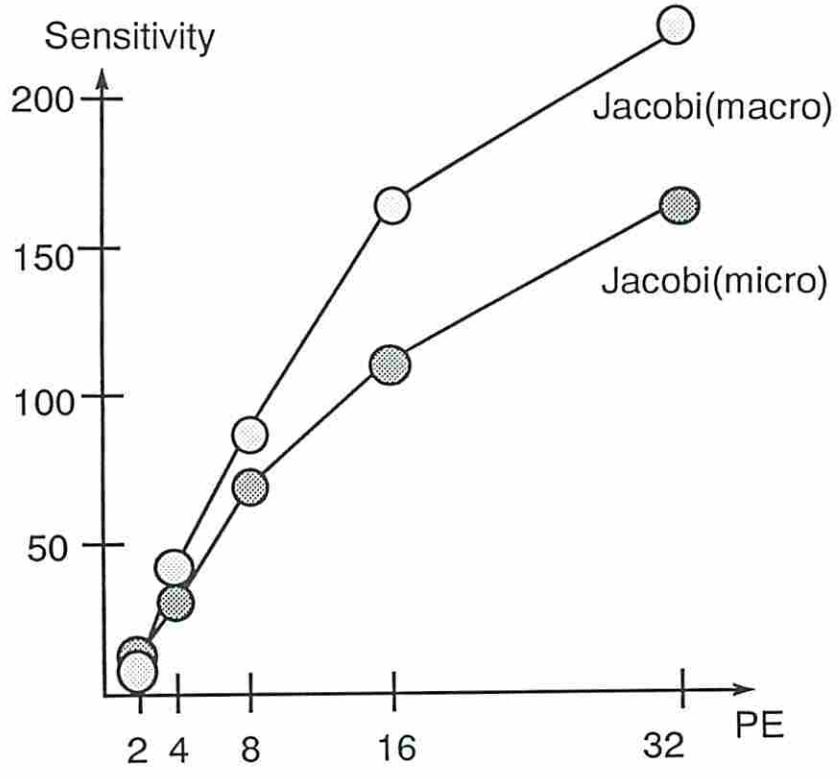

Figure 5.17: Effects of communication-delay for Problem Size = 32 × 32.

Problem Size = 32 × 32						
⋆ Network=1, Match=1, Fetch=1, ALU=1, Router=1 ♡ Network=10, Match=1, Fetch=1, ALU=1, Router=1						
System Size	Jacobi(Micro)			Jacobi(Macro)		
number of PEs	⋆ exe. time	♡ exe. time	%	⋆ exe. time	♡ exe. time	%
1 PE	290008	290008	0	252320	252320	0
2 PEs	150444	160151	6	129652	139359	7
4 PEs	80497	109838	36	68153	97721	43
8 PEs	45533	78169	71	37413	69916	86
16 PEs	29852	60163	101	22096	52201	136
32 PEs	23746	49949	110	15719	42183	168
64 PEs	22819	61044	167	14762	53233	260

$$Percentage = (\frac{Exe.\ time\ on\ long\ delay\ systems}{Exe.\ time\ on\ short\ delay\ systems} - 1) \times 100\%$$

TABLE 5.7 : System Tune-up for Problem Size = 32 × 32.

Observation: The inter-PE communication delay may significantly weaken the performance of a multiprocessor system. Tables 5.5 & 5.6, and Fig. 5.15 & 5.16 show that the Jacobi method is twice as sensitive to this kind of time delay than chaotic relaxation. For example, in Table 5.6, with 2 PEs, the Jacobi method needs 11 % more time in a slow communication network than in a faster network, while chaotic relaxation only needs 5 % more time in the same configuration. Table 5.6, 5.7 and Fig. 5.16, 5.17 show the inter-PE communication delay may have less effect on some machine configurations, for example, 16 PEs for problem size of 16 × 16 and 32 PEs for 32 × 32.

- **Overhead Analysis**: Overhead and non-overhead actors are defined as:

Problem Size = 8 × 8								
number of PEs		1	2	4	8	16	32	64
Jacobi	static actors	61						
	dynamic actors	5276						
	overhead actors	4550						
	percentage	86.3						
Chaotic	static actors	59						
	dynamic actors	3512	4067	3907	4065	3828	3354	3512
	overhead actors	2846	3273	3151	3273	3090	2724	2846
	percentage	81	80.4	80.6	80.5	80.7	81.2	81

$$Percentage = \frac{Number\ of\ overhead\ actors}{Number\ of\ dynamic\ actors} \times 100\%$$

TABLE 5.8 : Actor Traces

1. *"Overhead"* actors: Actors in data-flow graphs that are not involved directly in the algorithm itself but instead participating only in operations related to the U-interpretation model such as production of iteration indices with the appropriate tags, etc.
2. *"Non-overhead"* actors: Actors in data-flow graphs that are directly involved in the algorithm.

The results in Table 5.8 was obtained by counting the execution of certain *marked actors*. The table displays a dynamic count of the actors executed during the processing of an 8 × 8 problem in both chaotic and the Jacobi relaxation modes for various machine configurations.

Observation: The overhead analysis shows that in a fine-grain computation model, even highly parallel applications can entail a large amount of overhead processing (up to 80 %). In addition, Table 5.8 shows that this overhead is a major hindrance in the computations. From our results, the

Scalability Factors					
Number of PEs	Problem Size	Chaotic (Macro)	Chaotic (Micro)	Jacobi (Macro)	Jacobi (Micro)
8 PE	8 × 8	1	1	1	1
16 PEs	16 × 16	0.416	0.409	0.383	0.372
32 PEs	32 × 32	0.278	0.262	0.238	0.226
64 PEs	64 × 64	0.153	0.132	0.121	0.114

TABLE 5.9 : Scalability Factors in the VTD System with Differents Algorithms.

creation of larger computing entities (such as macro-actors) has reduced overhead actors and hence ensured better resource utilization and speedup.

- **Scalability Factor**: The scalability factor of a system is defined in the previous section. We exploit the trend of scalability factors in different problem sizes with various system configurations. We start with a matrix size equal to 8×8 and a machine size equal 8 PE, then 16×16 in 16 PEs, 32×32 in 32 PEs, and 64×64 in 64 PEs. The report is shown in Table 5.9 and the curves are shown in Fig. 5.18.

  Observation: The results show that the chaotic relaxation in the macro execution mode of the VTD system has the best scalability factor while the Jacobi methods in the micro execution mode has the worst scalability factor. However, one should note that the increasing rate of the machine size from 8 PEs to 16 PEs does not equal to the increasing rate of the complexity of the algorithms with a matrix size from 8×8 to 16×16. Therefore, we only compare the relative performance of different algorithms in various execution modes, instead of comparing it with the ideal scalability factor.

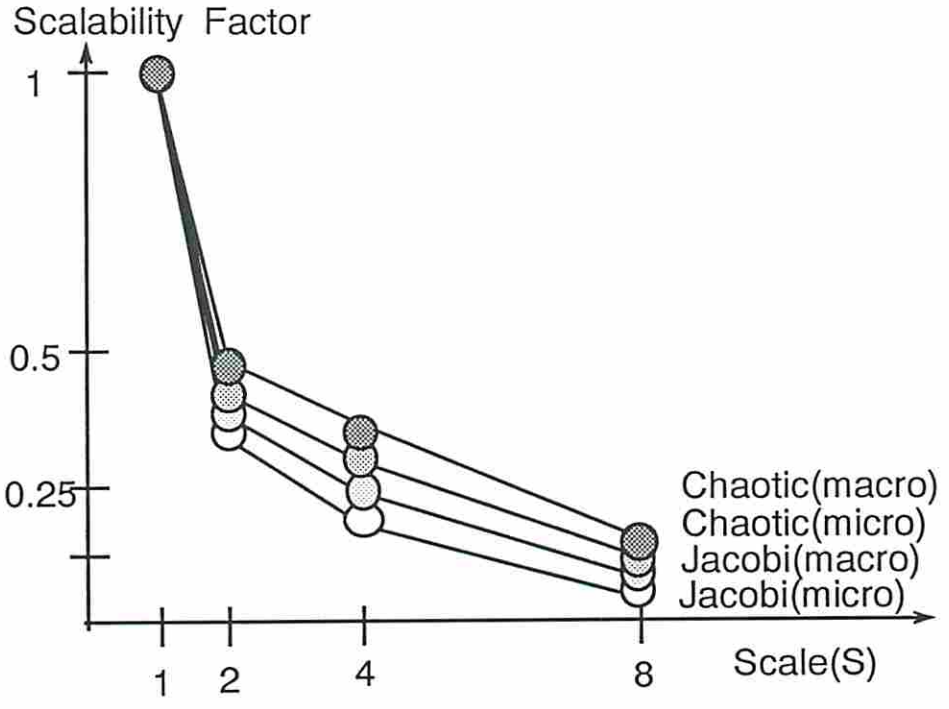

Figure 5.18: Scalability Factors in the VTD System.

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 5.10 : Robustness in Data-flow Architectures.

- **Robustness**: The robustness of a system is defined in the previous section. We exploit the trend of speedups 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 5.10 and the curves are shown in Fig. 5.19.

  Observation: In the results, we know that the speedup curves are almost linear 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 5.10, the speedup of chaotic relaxation for 64×64 problem size can reach up to 52 in a 64 PEs system with the macro execution mode.

5.6 Performance Evaluation

In the previous section, we have shown that the macro execution mode can indeed deliver higher efficiency and higher speedup in data-driven multiprocessor

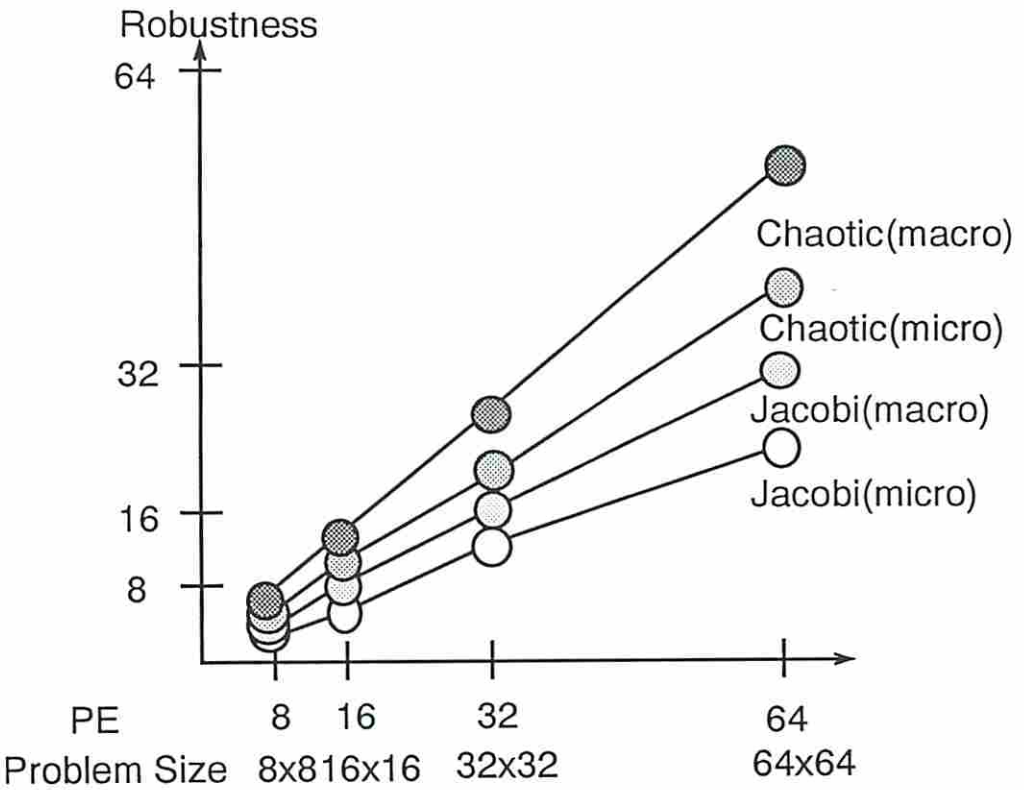

Figure 5.19: Robustness curves in Data-flow Architectures.

systems, while the micro execution mode needs more time to execute the sequential operations in a data-flow graph. In this section, we will extend Amdahl's law and obtain a simple analytic model to verify this phenomenon and explain this behavior.

5.6.1 Extended Amdahl's Law and an Analytic Model

Our data-flow multiprocessor is in fact a collection of pipelined processors (since each data-flow processor (Fig. 5.8) 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:

1. *A-type Parallelism*: It is "spatial" parallelism that refers to many independent entities in a program which can be easily distributed among PEs. In a pipelined-processor system, the A-type parallelism can be allocated to all PEs in a balanced fashion so as to achieve high performance, due to the independency among each entity.

2. *B-type Parallelism*: It is "temporal" parallelism that refers to several independent sequential paths in a program. When the B-type parallelism is executed by a multi-stage pipeline processor, the processor can reach a high utilization because the pipe is saturated by the operations from different sequential paths that take turn to feed in the pipeline of the processor. When the number of processors increases, naturally, the sequential paths will be allocated to different processors. For every processor, in this case,

there will be no longer enough operations to feed in the pipe because of the sequential execution of the operations in the sequential path. Hence, the system performance will degrade due to the unsaturation of the pipeline of every processor.

A matrix multiplied by a vector is the example we use to show how the A-type parallelism and the B-type parallelism affect performance :

Assume we have a matrix **A** which is multiplied by a vector **B**, where **A** has size $n \times n$, **B** has size n, and the result is **C**. For every element c_i in **C**, $c_i = \sum_{k=1}^{n}(a_{i,k} \times b_k)$. According to the above definitions, we know that the multiplications in this equation can provide A-type parallelism because all the multiplications of two elements are independent of one another for a given i. Likewise, the summations for each c_i are of B-type parallelism because there exits a sequential path in each summation function unless the *"tree-reduction"* scheme is used. Hence, we conclude that a total of n^2 A-type parallel operations (the multiplications) and $n \times (n-1)$ of B-type parallel operations (the summations) can be found. If we use n four-stage pipelined processors to execute the function, we will find that the multiplications can easily saturate the pipe of each processor while the summations can hardly saturate the pipe. A space-time diagram of n=64 in Fig. 5.20 clearly shows the situation of the saturated execution of multiplications $(M_{1,1}, M_{1,2}, ...M_{1,64})$ and unsaturated execution of summations $(SUM_{1,1}, SUM_{1,2}, ...SUM_{1,63})$ in a processor.

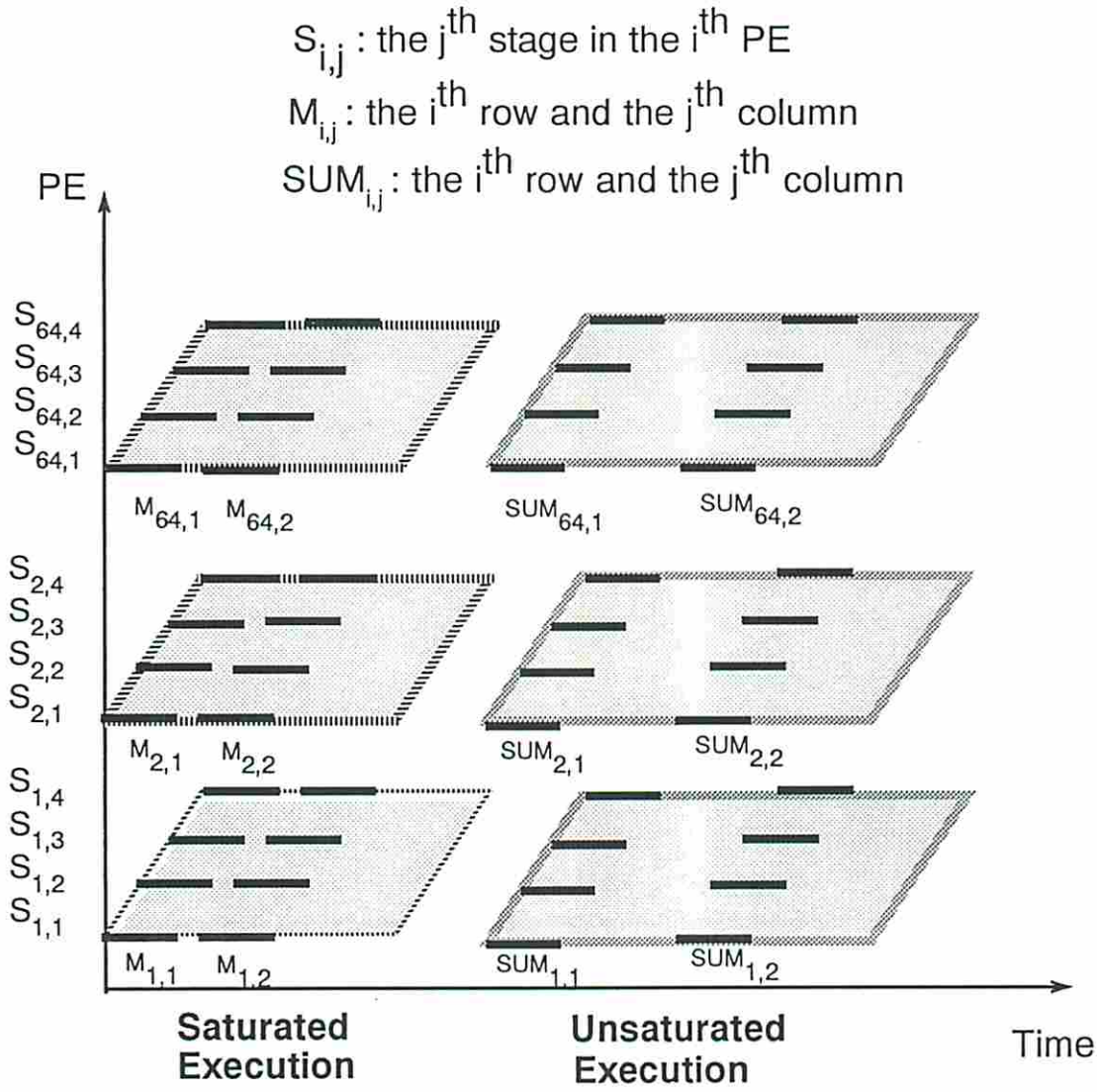

Figure 5.20: A Space-Time Diagram for n=64

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

$$ET(1) = (Sequential\ part) + (Parallel\ part) \quad with\ 1\ PE \tag{5.4}$$

$$ET(P) = (Sequential\ part) + (Parallel\ part)/P \quad with\ P\ PEs \tag{5.5}$$

According to our above notation, the total execution time can be rewritten as:

$$ET(1) = (Sequential\ part) + (A\ type\ part) + (B\ type\ part) \tag{5.6}$$

$$\begin{aligned} ET(P) &= (Sequential\ part) + (B\ type\ part/P) \times T(P) \\ &+ (A - type\ part/P) \end{aligned} \tag{5.7}$$

when P : *number of PEs*

$T(P)$: *interoutput time* $= 1/throughput$,

for each PE of P PEs

Since speedup has been defined as:

$$S_p(P) = \frac{ET(1)}{ET(P)} \tag{5.8}$$

in equation (7), the value of T(P) should be 1 in the ideal case which means that an output is generated at each pipeline cycle. In fact, T(P) will increase when the number of processors (P) increases. This is due to insufficient B-type parallelism in the algorithm and longer communication delay in the system when

many PEs are available. Consequently, it results in the pipeline stages becoming unsaturated and that resource utilization becoming low.

The effect of unsaturated pipes in each PE for both micro and macro execution modes is shown in Fig. 5.21: When there is sufficient parallelism, each pipe in a PE is saturated (in Saturated area) and a linear speedup can be expected. Contrarily, when there is insufficient parallelism, each pipe in a PE is unsaturated (in Unsaturated area) and the speedup is limited. Note that the separation point between the Saturated and Unsaturated areas can be selected according to the application. For instance, it can seen from Fig. 5.20, where we have 64 summations, that a 4-stage pipelined PE will be saturated with 4 summations. Therefore, 16 PEs will be saturated with 64 summations. More PEs would have us enter the "unsaturated area" already seen in Fig. 5.21.

5.6.2 Analysis of The Macro Execution Scheme

From the previous section, the simulation results show that the macro-actor formatting scheme (lump sequential actors into macro-actors) has actually reduced the total execution time by reducing the execution overhead in each PE. In the extreme case, when many PEs are available, a PE takes 20 time units to execute the summation function in Fig. 5.5 (each actor takes 4 time units for execution). But, for a macro-actor that contains the operations of a summation, it takes only 8 time units to execute this function (5 time units for ALU, 3 time units for Matching Store, Instruction Fetch, and Token Formatting Units). This

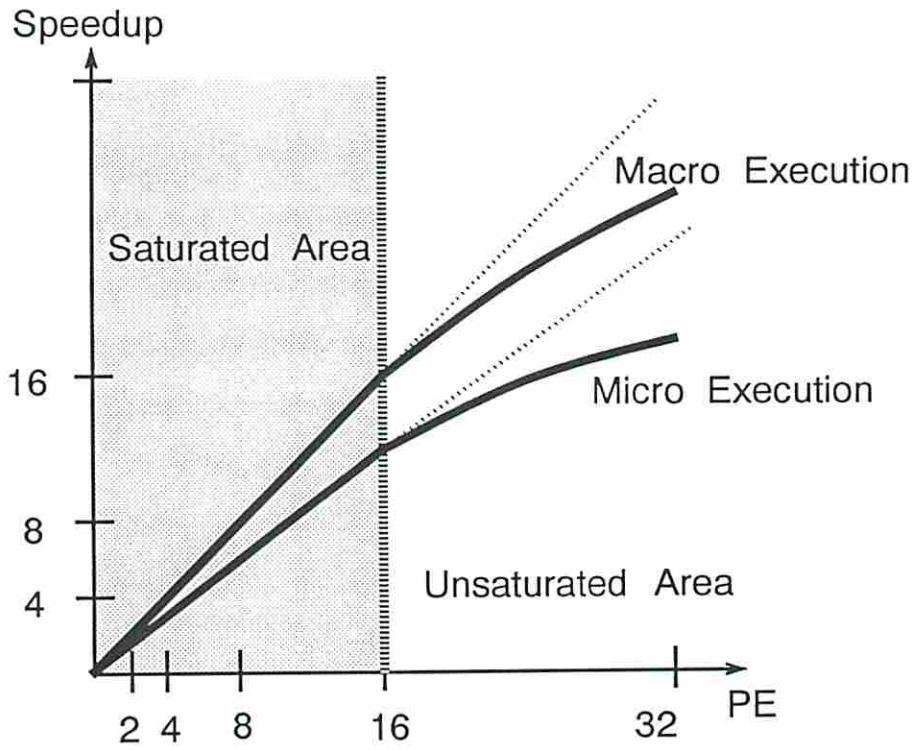

Figure 5.21: Analytical Curves of Macro vs. Micro Execution

means that the execution time is significantly reduced in the macro-actor execution mode by reducing sequential execution overhead. In general, when there is a sequential path of actors with length n which are executed in a PE, it takes $(n+3)$ time units in the macro execution mode, while it takes $4 \times n$ time units in the micro execution mode. Therefore, it is about 4 times faster to execute a sequential path of actors in the macro execution mode.

Hence, without losing the available parallelism, we can conclude that the macro-actor scheme is better than the purely fine grain execution model. However, it should be noted that the above analysis only shows the effect of different granularities on a sequential path. In the actual computation, one should also consider other effects such as the relative parallelism comparing with the number of PEs and the effect between the communication cost and the available parallelism when we choose the sizes of macro-actors.

5.7 Discussion

In this chapter, we have demonstrated how synchronous and asynchronous linear systems solvers could be described in a high level data-flow language (SISAL) and implemented on the Variable-grain Tagged-token Data-flow (VTD) multiprocessor system in both micro and macro execution models. The conventional Jacobi method and the chaotic relaxation were chosen for their known inherent parallelism of execution. While the "*conventional*" principles of the U-interpreter were used in the graph construction of the Jacobi method, chaotic behavior could not be easily realized in this model of interpretation. We therefore proposed a

new scheme for the implementation of chaotic relaxation: the "*Matching Store with Locks*" scheme proceeds with the execution to detect any change 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. Besides the traditional *speedup* performance measurement in multiprocessor systems, we have defined new performance measures, called *Growth Factor*, *Scalability Factor*, and *Robustness*, from the point of view of machines and applications domains to characterize the system performance. An extensive simulation of the execution of these two algorithms on a simulated VTD machine was carried out.

In summary, it can be said that this chapter 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 linear systems 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 the low sensitivity to communication and matching store delays, this type of algorithms will be more efficient in large-scale, distributed multiprocessors.

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 the program graphs. Our new firing rule has also shown easy and efficient of implementation of these two algorithms in a data-driven system.

4. The large amount of overhead is the major hindrance in a multiprocessor system for the fine-grain execution model. The creation of larger computing entities must be undertaken in order to ensure better resource utilization.

5. While the conventional "speedup" has been used to evaluate the performance of multiprocessor systems, new performance measurements which should cover the application domain as well as the machine domain must be developed. Indeed, the proposed performance factors will enhance the evaluation of the performance of multiprocessor systems.

6. 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.

Chapter 6

Conclusions and Future Research

6.1 Conclusions

In this dissertation, we have studied the issues that arise in designing parallel PDE solvers, developing macro data-flow graphs, and implementing numerical algorithms on data-driven architectures. We have shown that with the advent of parallel computing, these three areas should be mutually coordinated and tightly associated.

In this research, we have first presented a highly parallel multigrid algorithm (PVM) for multiprocessor systems. While the conventional multigrid algorithm can not be easily executed in parallel, faster convergence rate and higher system utilization of the PVM have been demonstrated. Indeed, we have proved that the PVM is a simple, general, efficient, and parallel algorithm by theoretical analysis as well as experiments. Therefore, with these characteristics, it can be said that the PVM algorithm can be efficiently implemented on multiprocessor systems.

The data-driven execution model is very attractive for a multiprocessor system because of the massive parallelism it can deliver at the runtime, and requiring no explicit synchronization mechanisms for the processes. We have demonstrated how macro data-flow graphs can be generated. By applying our partitioning schemes on numerical applications, we have shown how macro actors can be formed and how vectorized data-tokens can be gathered through our three-phase partitioning schemes. We concentrated on the partitioning of the fine-grain data-flow graphs, and evaluated the final performance. In order to reduce execution time, the actor-based partitioning schemes were applied to the sequential and computational actors in the fine grain data-flow graphs. In the second-phase of partitioning, the tag-based partitioning schemes were taken to simplify tag-manipulation operations. In the final phase of partitioning, the token-based partitioning schemes have exploited vectorization techniques to achieve efficiency in macro data-flow computing.

In the system design, we have demonstrated how synchronous and asynchronous linear systems solvers could be described in a high level data-flow language (SISAL) and implemented on the VTD multiprocessor system in both micro and macro execution models. The conventional Jacobi method and the chaotic relaxation were chosen for their known inherent parallelism of execution. We have also proposed a new scheme for the implementation of chaotic relaxation: the Matching Store with Locks scheme proceeds with the execution to detect any change on the input arcs, instead of allowing execution upon arrival of a matched token set. This 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 and executed in the VTD system.

Besides the conventional speedup performance measure of multiprocessor systems, we have also defined new performance measures, Growth Factor, Scalability Factor, and Robustness, from the viewpoints of machines and applications domains in order to characterize the system performance more precisely.

In summary, it can be said that data-flow principles of execution can be used to provide high-programmability efficiently in the numerical evaluation of highly concurrent numerical algorithms. The large amount of overhead is the major hindrance in a multiprocessor system for the fine-grain execution model. The creation of larger computing entities must be undertaken in order to ensure better resource utilization. New performance measures which should evaluate application as well as machine domains must be developed in oreder to appraise the accurate performance of multiprocessor systems.

6.2 Suggestions for Future Research

Some suggested research directions are proposed in this section. These include the issues in developing parallel numerical algorithms, data-flow compiler, and asynchronous applications.

6.2.1 Theoretical Analysis of the Operators in PVM

We have presented a new highly parallel multigrid algorithm in Chapter 3. However, variants of the relaxation, restriction, interpolation operators in the PVM algorithm could be further investigated to compare the performance in different operators. The two-grid analysis of the spectral radius of the iteration operator of the PVM algorithm can also be extended to a multi-level analysis for the actual implementation.

6.2.2 Resource Management in the VTD System

In this research, we have studied the partitioning issue at the compile time. Another key issue in data-flow computation is resource management. Allocating processes to available resources can be done in either compile or execution time. The relative merits of each approach could be further studied in order to enhance the VTD system.

6.2.3 Asynchronous Applications on the VTD Systems

Asynchronous algorithms are efficient methods in solving scientific and engineering problems. Much research has also been devoted to the study of asynchronous algorithms in different areas. For example, in the logic circuit simulation, communication network, and artificial neural network. Further research should include the study of the applications of such asynchronous algorithms on these areas as well as the implementation of these asynchronous algorithms on multiprocessor systems.

Appendix A

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Numercial Multigrid Algorithms for PDEs      %
%                in   SISAL                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                              %
% This software package is to exploit multigrid %
% algorithms to solve the Poisson's equation on %
% teh unit square (0,1) x (0,1) with Dirichlet  %
% boundary conditions and different load       %
% functions f(x,y):                            %
%             U''/x'' + U''/y'' = f(x,y)       %
%                  0 < x < 1                   %
%                  0 < y < 1                   %
%                                              %
% Programmer: Chih-Ming Lin                    %
%                                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

define  initialize, Conv, cateX, cateY, cateR, cateF, cateXP,
        Jacobi0, Jacobi1, Res, Interp, downJ, upJ, main

type OneDim = array[real];
type TwoDim = array[OneDim];
type ThreeDim = array[TwoDim];


  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % Main Function:                              %
  %     input:                                  %
  %         a. level: define the step size      %
  %                   on the finest grid level  %
  %                   the step size = 1 / level %
  %         b. fip  : define the value of       %
  %                   f(x,y) on each grid point %
  %                   of the finest grid level  %
  %         c. exip : the exact solution on each %
  %                   grid point.               %
  %     output: the iterative solution of U(x,y)%
```

```
        %               based on multigrid algorithms.  %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function main ( level : integer ; fip,exip : ThreeDim
returns ThreeDim )
    for initial
      f, ex := fip, exip ;
      h := exp ( 2, level ) ;
      r, x := initialize ( h, level, ex ) ;
      y := x ;
      notfinish := true
    while notfinish repeat
      r, f, nextx, nexty :=
      downJ ( h, level, old x, old y, old r, old f ) ;
      xp, y := upJ ( h, level, nextx, nexty, r, f ) ;

      notfinish := Conv ( h, xp[1], y[1] ) ;
      x := for k in 1,1
             returns array of
             for i in 0, h cross j in 0, h
               returns array of y[1][i][j]
             end for
           end for
           ||
           for  k in 2, level
             returns array of
             for i in 0, h cross j in 0, h
               returns array of xp[k][i][j]
             end for
           end for
    returns value of y
    end for
end function % main


        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Initialization:                               %
        %       This function gives the initial guess of %
        %       all the grid points on ecah grid level   %
        %       and set the exact solution for checking  %
```

```
        %     the convergence.                                    %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function initialize ( h, level : integer ; ex : ThreeDim
returns ThreeDim, ThreeDim )
  for k in 1, level
    returns array of
      for i in 0, h cross j in 0, h
        returns array of 0.0
      end for
  end for,

  for k in 1, level
    returns array of
      for i in 0, h cross j in 0, h
        returns array of
           if (k=level) & (i=0 | i=exp (2,k) | j=0 | j=exp(2,k))
             then      ex[k][i][j]
             else      0.0
           end if
      end for
  end for

end function % initialize


        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Convergence check:                                     %
        %     This function compare the consecutive              %
        %     iterative solutions and decide if futher           %
        %     iteration is necessary.                            %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function Conv ( h : integer; x, y : TwoDim
returns boolean )

  let
    s :=
    for i in 1, h-1 cross j in 1, h-1
      returns value of sum y[i][j]*y[i][j]
```

```
    end for ;

    diff :=
    for i in 1, h-1 cross j in 1, h-1
      returns value of sum (y[i][j]- x[i][j])*(y[i][j]- x[i][j])
    end for
  in
    if (diff/s < 1.0E12) then false
                         else true
    end if
  end let
end function % Conv


        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Concatenate Array X:                                   %
        %     This function concatenate the old value            %
        %     of U(x,y) after iteration with the                 %
        %     boundary condition to check the                    %
        %     solution for next iteration.                       %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function cateX ( m, h, dh, start, final : integer ; x : ThreeDim
returns ThreeDim )

  for k in start, final
    returns array of
      for i in 0, 0 cross j in 0, h
        returns array of x[m][i][j]
      end for
      ||
      for i in 1, dh
        returns array of
          for j in 0, 0
            returns array of x[m][i][0]
          end for
          ||
          for j in 1, dh
            returns array of 0.0
          end for
```

```
        ||
        for j in dh+1, h
          returns array of x[m][i][j]
        end for
      end for
      ||
      for i in dh+1, h cross j in 0, h
        returns array of x[m][i][j]
      end for
  end for

end function % cateX


    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Concatenate Array Y:                          %
    %      This function concatenate the new value  %
    %      of U(x,y) after iteration with the       %
    %      boundary condition to update the         %
    %      new value for next iteration.            %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function cateY ( m, h, dh, st, start, final : integer ;
x, y, f : ThreeDim returns ThreeDim )

  for k in start, final
    returns array of
      for i in 0, 0 cross j in 0, h
        returns array of y[m][i][j]
      end for
      ||
      for i in 1, dh
        returns array of
          for j in 0,0
            returns array of y[m][i][0]
          end for
          ||
          for j in 1, dh
            returns array of
               x[m][i][j]/3.0 +( x[m][i-1][j]+x[m][i+1][j]
```

```
                        +x[m][i][j-1]+x[m][i][j+1]
                        -f[m][i][j]* real(st*st)/real(h*h))/6.0
          end for
          ||
          for j in dh+1, h
            returns array of y[m][i][j]
          end for
        end for
        ||
        for i in dh+1, h cross j in 0, h
          returns array of y[m][i][j]
        end for
    end for

end function % cateY


        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Calculate Residual Error:                              %
        %      This function computes the residual error%
        %      of U(x,y) after iteration for                     %
        %      transmitting the error to next                    %
        %      coarse grid level.                                %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function cateR ( m, h, st, dh, start, final : integer ;
y, r, f : ThreeDim returns ThreeDim )

  for k in start, final
    returns array of
      for i in 0, 0 cross j in 0, h
        returns array of r[m][i][j]
      end for
      ||
      for i in 1, dh
        returns array of
          for j in 0, 0
            returns array of r[m][i][0]
          end for
          ||
```

```
        for j in 1, dh
          returns array of
            f[m][i][j] - ( y[m][i-1][j] + y[m][i+1][j]
                         + y[m][i][j-1] + y[m][i][j+1]
                         - 4.0*y[m][i][j])*real(h*h)/real(st*st)
        end for
        ||
        for j in dh+1, h
          returns array of r[m][i][j]
        end for
      end for
      ||
      for i in dh+1, h cross j in 0, h
        returns array of r[m][i][j]
      end for
    end for

end function % cateR


        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Caculate  F:                                 %
        %     This function computes the values in     %
        %     f(x,y) for the next coarser grid         %
        %     level after iteration.                   %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function cateF ( m, h, dh, start, final : integer ;
r, f : ThreeDim returns ThreeDim )

  for k in start, final
    returns array of
      for i in 0, 0 cross j in 0, h
        returns array of f[m][i][j]
      end for
      ||
      for i in 1, dh
        returns array of
          for j in 0, 0
            returns array of f[m][i][0]
```

```
        end for
        ||
        for j in 1, dh
          returns array of
                (2.0*r[m][i*2-1][j*2] + 2.0*r[m][i*2+1][j*2]
                +2.0*r[m][i*2][j*2-1] + 2.0*r[m][i*2][j*2+1]
                +4.0*r[m][i*2][j*2]
                +r[m][i*2-1][j*2-1] + r[m][i*2-1][j*2+1]
                +r[m][i*2+1][j*2-1] + r[m][i*2+1][j*2+1])/16.0
        end for
        ||
        for j in dh+1, h
          returns array of f[m][i][j]
        end for
      end for
      ||
      for i in dh+1, h cross j in 0, h
        returns array of f[m][i][j]
      end for
  end for

end function % cateF


        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Interpolation :                              %
        %     This function interpolates the values    %
        %     in U(x,y) for the upper finer grid       %
        %     level after the coarser grid relaxation. %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function cateXP ( m, h, dh, start, final : integer ;
x, y : ThreeDim returns ThreeDim )

  for k in start, final
    returns array of
      for i in 0, 0 cross j in 0, h
        returns array of x[m+1][i][j]
      end for
      ||
```

```
      for i in 1, dh
        returns array of
          for j in 0, 0
            returns array of x[m+1][i][0]
          end for
          ||
          for j in 1, dh
            returns array of
              if ( mod(i,2) = 0 & mod (j,2) = 0)
                then  y[m+1][i][j]+ y [m][i/2][j/2]
                elseif ( mod(i,2) = 0 & mod (j,2) ~= 0)
                 then
                   y[m+1][i][j]+(y[m][i/2][(j-1)/2]
                                +y[m][i/2][(j+1)/2])/2.0
                 elseif (mod(i,2) ~= 0 & mod (j,2) = 0)
                   then
                     y[m+1][i][j]+(y[m][(i-1)/2][j/2]
                                  +y[m][(i+1)/2][j/2])/2.0
                   else
                      y[m+1][i][j]
                      + (y[m][(i-1)/2][(j-1)/2]
                      + y[m][(i-1)/2][(j+1)/2]
                      + y[m][(i+1)/2][(j-1)/2]
                      + y[m][(i+1)/2][(j+1)/2])/4.0
               end if
          end for
          ||
          for j in dh+1, h
            returns array of x[m+1][i][j]
          end for
        end for
        ||
        for i in dh+1, h cross j in 0, h
          returns array of x[m+1][i][j]
        end for
    end for

end function % cateXP
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Weighted Jacobi for coarse-fine grid :       %
%     This function relaxes the values in      %
%      U(x,y) in the current grid level.       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function Jacobi1 (h, level, m : integer ; x, y, f : ThreeDim
returns  ThreeDim )

  let
    st := h/exp (2,m) ;
    dh := h / st - 1 ;
    eqone :=   if m = 1 then true
                         else false
               end if ;
    eqlevel := if m = level then true
                             else false
               end if
  in
    if eqone then
                cateY ( m, h, dh, st, 1, 1, x, y, f )
                ||
                for k in 2, level
                  returns array of y[k]
                end for

            elseif eqlevel
              then
                 for k in 1, m-1
                   returns array of y[k]
                 end for
                 ||
                 cateY ( m, h, dh, st, m, m, x, y, f )

              else
                 for k in 1, m-1
                   returns array of y[k]
                 end for
                 ||
                 cateY ( m, h, dh, st, m, m, x, y, f )
```

```
                ||
                for k in m+1, level
                  returns array of y[k]
                end for
    end if
  end let
end function % Jacobi1


        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Weighted Jacobi for fine-coarse grid :       %
        %     This function relaxes the values         %
        %     in U(x,y) in the current grid level.     %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function Jacobi0 (h, level, m : integer ; x : ThreeDim
returns  ThreeDim )

  let
    st := h/exp (2,m) ;
    dh := h / st - 1 ;
    eqone :=   if m = 1 then true
                         else false
               end if ;
    eqlevel := if m = level then true
                             else false
               end if
  in
    if eqone then
                cateX ( m, h, dh, 1, 1, x )
                ||
                for k in 2, level
                  returns array of x[k]
                end for

           elseif eqlevel
             then
                for k in 1, m-1
                  returns array of x[k]
                end for
```

```
                ||
                cateX ( m, h, dh, m, m, x )

             else
                for k in 1, m-1
                  returns array of x[k]
                end for
                ||
                cateX ( m, h, dh, m, m, x )
                ||
                for k in m+1, level
                  returns array of x[k]
                end for
    end if
  end let
end function % Jocobi0


        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Restriction :                               %
        %     This function compute the error from the %
        %     coarse grid to fine grid.               %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function Res( h, level, m : integer; x, y, r, f : ThreeDim
returns ThreeDim, ThreeDim )
  let
    st := h/exp (2,m) ;
    dh := h / st  - 1 ;
    ddh := h/(2*st) - 1 ;
    eqtwo :=   if m = 2 then true
                        else false
               end if ;

    eqlevel := if m = level then true
                              else false
               end if ;
    rp :=
      if eqtwo then
                for k in 1, 1
```

```
            returns array of r[k]
          end for
          ||
          cateR ( m, h, st, dh, 2, 2, y, r, f )
          ||
          for k in 3, level
            returns array of r[k]
          end for

      elseif eqlevel
        then
          for k in 1, m-1
            returns array of r[k]
          end for
          ||
          cateR ( m, h, st, dh, m, m, y, r, f )

        else
          for k in 1, m-1
            returns array of r[k]
          end for
          ||
          cateR ( m, h, st, dh, m, m, y, r, f )
          ||
          for k in m+1, level
            returns array of r[k]
          end for
  end if
in

if eqtwo then
          cateF ( m, h, ddh, 1, 1, rp, f )
          ||
          for k in 2, level
            returns array of f[k]
          end for

      elseif eqlevel
        then
          for k in 1, m-2
```

```
              returns array of f[k]
            end for
            ||
            cateF ( m, h, ddh, m-1, m-1, rp, f )
            ||
            for k in m, m
              returns array of f[k]
            end for

         else
            for k in 1, m-2
              returns array of f[k]
            end for
            ||
            cateF ( m, h, ddh, m-1, m-1, rp, f )
            ||
            for k in m, level
              returns array of f[k]
            end for
    end if, rp
  end let
end function % Res


        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Interpolation :                              %
        %     This function compute the error from the %
        %     fine grid to coarse grid.                %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function Interp( h, level, m : integer ; x, y : ThreeDim
returns ThreeDim )

  let
    st := h/exp (2,m) ;
    dh := 2 * h / st - 1 ;
    eqone :=   if m = 1 then true
                         else false
               end if ;
    levelm1 := if m = level-1 then true
```

```
                                  else false
                end if
  in
    if eqone then
                for k in 1,1
                  returns array of x[k]
                end for
                ||
                cateXP ( m, h, dh, 2, 2, x, y )
                ||
                for k in 3, level
                  returns array of x[k]
                end for

           elseif levelm1
             then
                for k in 1, m
                  returns array of x[k]
                end for
                ||
                cateXP ( m, h, dh, level, level, x, y )

             else
                for k in 1, m
                  returns array of x[k]
                end for
                ||
                cateXP ( m, h, dh, m+1, m+1, x, y )
                ||
                for k in m+2, level
                  returns array of x[k]
                end for
    end if
  end let
end function % Interp


        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Relaxation from the finest level :            %
        %     This function do half of the V-cycle to   %
```

```
        %     implement the multigrid algorithms       %
        %     form fine to coarse grid level.          %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function downJ (h,level : integer ; xip,yip,rip,fip : ThreeDim
returns ThreeDim, ThreeDim, ThreeDim, ThreeDim )
      for initial
        k := level ;
        x, r, f := xip, rip, fip ;
        y := Jacobi1( h, level, level, x, yip, f ) ;
      while k > 1 repeat
        f, r :=
        Res( h, level, old k, old x, old y, old r, old f ) ;
        k := old k - 1 ;
        x := Jacobi0( h, level, k, old x ) ;
        y := Jacobi1( h, level, k, x, old y, f ) ;
      returns value of r
              value of f
              value of x
              value of y
      end for
end function % downJ


        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Relaxation from the finest level :             %
        %     This function do half of the V-cycle to    %
        %     implement the multigrid algorithms         %
        %     form fine to coarse grid level.            %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function upJ ( h, level : integer ; xip, yip, r, f : ThreeDim
returns  ThreeDim, ThreeDim )

      for initial
        k := 1 ;
        x, y := xip, yip
      while k < level  repeat
        x := Interp( h, level, old k, old x, old y ) ;
        k := old k + 1 ;
```

```
        y := Jacobi1( h, level, k, x, old y, f )

      returns value of x
              value of y
      end for
end function % upJ
```

Bibliography

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

[2] Advanced Topics in data-flow computing. Edited by J-L. Gaudiot and L. Bic, Prentice Hall, 1990.

[3] D. Abramson. Using a dataflow multiprocessor for functional logic simulation. In *Proceeding of the Third International Conference on Supercomputing*, 1988.

[4] H. Ahmed. A vector data flow architecture. *Technical Report* TRITA-TCS-8905, The royal Institute of Technology, Sep. 1989.

[5] Arvind and K.P Gostelow. The U-interpreter. *IEEE Computer*, pages 42–49, February 1982.

[6] Arvind and R.A. Iannucci. Two fundamental issues in multiprocessors: the data-flow solution. *Technical Report* LCS/TM-241, Laboratory for Computer Science, MIT, September 1983.

[7] Arvind, V. Kathail, and K. Pingali. A data-flow architecture with tagged tokens. *Technical Report* TM-174, Laboratory for Computer Science, MIT, September 1980.

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

[9] M. Ayed and J-L. Gaudiot. The USC Macro Data-Flow Assembly Language. *Technical report* CENG-89-28, USC, Oct. 1898.

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

[11] Tony F. Chan and R. Schreiber. Parallel networks for mutigrid algorithms: Architecture and complexity. *SIAM J. Sci. Stat. Comput.*, 6, July 1985.

[12] Tony F. Chan and Ray Tuminaro. A survey of parallel multigrid algorithms. *Technical Report* RIACS 87.22, NASA Ames Research Center, Moffett Field, CA, Aug 1987.

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

[14] J.B. Dennis. First version of a data flow procedure language. In *Programming Symp: Proc. Colloque sur la Programmation, Paris, France, LNCS 19*, pages 362–376, 1984.

[15] P. Frederickson and O. McBryan. Parall superconvergent multigrid. In *Proceedings of the Third Copper Mountain Conference on Multigrid Methods*, 1987.

[16] D. Gannon and J. van Rosendale. On the structure of parallelism in a highly concurrent PDE solver. *Journal of Parallel and Distributed Computing*, pages:106–135, 1986.

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

[18] J-L. Gaudiot. Data-driven multicomputers in digital signal processing. *Proceedings of the IEEE*, Sep. 1987.

[19] 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.

[20] J-L. Gaudiot and C.M. Lin. Performance of asynchronous algorithms in multi-level data-driven systems. In *Proc. of IFIP WG 10.3 Working Conference on Decentralized Systems, Lyon, France*, December 1989.

[21] J-L. Gaudiot and Y.H. Wei. Token relabeling in a tagged token data-flow architecture. *IEEE Transactions on Computers*, Sep., 1989.

[22] I. Gottlieb. SDF-the structured dataflow model of computing and its archtecture. In *Proceedings of the first Int'l Conference on Supercomputing*, 1985.

[23] A. Greenbaum. A multigrid method for multiprocessors. In *Applied Mathematics and Computation*, pages 75–88, 1985.

[24] J. Gustafson. Reevaluating Amdahl's law. *Communication of the ACM*, pages 532–533, May 1988.

[25] L. Hart and S. McCormick. Asynchronous multilevel adaptive methods for solving partial differential equations on multiprocessors. *Technical Report*, Computational Mathematics Group, The University of Colorado at Denver, 1987.

[26] Stephen McCormick. *Multigrid Methods: Theory, Applications, and Supercomputing.* Marcel Dekker Inc., 1988.

[27] J.R. McGraw and S.K. Skedzielewski. SISAL: Streams and iteration in a single assignment language, language reference manual, version 1.2. *Technical Report* M-146, Lawrence Livermore National Laboratory, March 1985.

[28] W. Najjar and J-L. Gaudiot. A hierarchical data-driven model for multigrid problem solving. In *Proceedings of the International Symposium on High Performance Computer Systems, Paris*, 1987.

[29] W. Najjar and J-L. Gaudiot. Multi-level execution in data-flow architectures. In *Proceedings of the International Conference on Parallel Processing, St.Charles,IL*, 1987.

[30] V. Sarkar. Partitioning and scheduling parallel programs for multiprocessors. *Technical Report* CSL-TR-87-328, Computer Systems Lab., Stanford Univ., April 1987.

[31] V.P. Srini. An architectural comparison of dataflow systems. *IEEE Computer*, 19(3):68–88, March 1986.

[32] S.Sekiguchi, K.Hiraki, and T.Shimada. Efficient vector processing on a dataflow supercomputer sigma-1. In *Proceedings of the 1988 International Conference on Parallel Processing*, 1988.

[33] K. Stüben and U. Trottenberg. Multigrid methods : fundamental algorithms, model problem analysis, and applications. *Multigrid Methods*, pages ed. by W. Hackbusch and U. Trottenberg, Springer–Verlag, New York, 1–176, 1982.

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

[35] N. Yoo and J-L. Gaudiot. The USC macro-data-flow simulator. *Technical Report* CENG-89-27, USC, October 1989.