

AUTOMATIC PROGRAM PARALLELIZATION
USING STATELESS PARALLEL PROCESSING ARCHITECTURE

A Dissertation

Submitted to

The Temple University Graduate Board

in Partial Fulfillment

of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

by

Feijian Sun

April 2004

ABSTRACT

Parallel programming is one of the best-known open problems in parallel processing research. The primary difficulty is in automatic dependency analysis – a critical factor for program partitioning. From the very early attempts in decomposing a program for multiple processors to existing modern parallelizing compilers, they all fall short to solving non-linear, indirect and conditional dependencies. The use of scalable distributed-memory supercomputers makes programming even more difficult due to the lack of a uniform memory space.

This thesis investigates a new approach for automatic sequential-to-parallel program translation by leveraging a dataflow computation model. It is well-known that dataflow computation models allow automatic dependency uncovering without complex static dependency analysis. This dissertation focuses on a coarse-grain dataflow model supported by a stateless parallel processing architecture for distributed-memory supercomputers. Under this model, the user discovers an overall dependency pattern. This pattern is used to direct program partition and data distribution strategies. Non-linear, indirect and conditional dependencies are automatically uncovered at runtime. In contrast, other approaches, such as the PARADIGM project at UIUC, have encountered insurmountable difficulties when attempting to solve nonlinear and other dependencies at compile time.

In this thesis, the dataflow computation model is provided by the Synergy

system – a preliminary implementation of a stateless parallel processing architecture using multiple networked computers.

The proposed technical approach includes a new Parallelization Markup Language (PML). It is used to describe an overall dependency pattern by marking sequential program segments. A PML compiler translates the marked sequential program into multiple parallel programs using the extracted dependency pattern. Based on XML technique, PML is portable and extensible. It is completely independent from programming languages. The PML tags are also powerful enough to describe very complex dependency patterns (thus data partition strategies).

Theoretically, the proposed methodology is applicable to all types of iterative compute-intensive applications. In this thesis, we have chosen four well-recognized numerical applications to illustrate the practical utility and efficiency of the proposed method: Matrix Multiplication, Laplacian Solver using Gauss-Siedel iterations, Ion Generation Simulator and Block LU Factorization. We show that performance measurements from generated programs compare favorably against manually written parallel programs.

The envisioned contributions of this research include

1. Design of Parallelization Markup Language (PML).
2. Design and implementation of PML compiler.
3. Extended tuple space support for simplifying parallel program generation and optimizing runtime parallel performance.

ACKNOWLEDGMENTS

I would like to thank my adviser, Professor Yuan Shi, for his constant support and advice throughout the course of this research. His trust and encouragement helped me tremendously, especially in the early difficult years of this research when progress came painfully slow as the original approach had to be abandoned and numerous alternatives were tried without success. He has provided important direction when I most needed it. I also greatly appreciate his many efforts on my behalf, especially in reviewing and revising the many versions of this thesis.

I would like to thank the members of my committee: Professors James Korsh, Charles Kapps, Henry Sendaula, and Guang R. Gao, for reviewing this thesis and for their valuable feedback. I would especially like to thank Professor Charles Kapps for providing the sequential source code and references of Ion Generation Simulator project.

I would like to thank my fellow student Weijun He for his valuable comments and suggestions on my work, for providing the sequential and manually written parallel codes of Block LU Factorization application, as well as for helping further my understanding of the application.

Most of all, I would like to thank my entire family for all of their love, encouragement, patience, and support throughout my studies.

TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION	1
1.1 PROPOSED APPROACH	1
1.2 ENVISIONED CONTRIBUTIONS	3
1.3 THESIS ORGANIZATION	4
CHAPTER 2. BACKGROUND	5
2.1 PARALLEL PROGRAMMING MODELS	5
2.1.1 <i>Programming Language Extensions</i>	5
2.1.2 <i>Parallel Programming Libraries</i>	9
2.1.3 <i>Summary</i>	12
2.2 STATELESS PARALLEL PROCESSING (SPP) ARCHITECTURE AND SYNERGY SYSTEM	13
2.2.1 <i>The SPP Architecture</i>	13
2.2.2 <i>Synergy – A Preliminary Implementation Of SPP</i>	19
2.3 XML (EXTENSIBLE MARKUP LANGUAGE)	23
2.3.1 <i>What Does XML Look Like?</i>	23
2.3.2 <i>XML Is Extensible</i>	25
2.3.3 <i>XML Is Structured</i>	27
2.3.4 <i>XML Validation</i>	28
2.3.5 <i>Summary</i>	29
CHAPTER 3. THE PARALLELIZATION MARKUP LANGUAGE (PML)	30
3.1 PML TAG SPECIFICATION	30
3.1.1 <i>The Target Tag</i>	32
3.1.2 <i>Communication Tags</i>	33
3.1.3 <i>Code Blocking Tags</i>	36
3.1.4 <i>The Token Tag</i>	38
3.2 PML TAG VALIDATION	39
3.3 A PML-TAGGING EXAMPLE	40
3.3.1 <i>The Original Sequential Program</i>	40
3.3.2 <i>PML-tagging Strategies</i>	41
3.4 SUMMARY	44
CHAPTER 4. THE SOURCE-TO-SOURCE PARALLELIZING COMPILER	45
4.1 PML TAG PARSING	46
4.2 TREE TRANSFORMATION	47
4.3 CODE GENERATION	49
4.4 ADDITIONAL ACTIVITIES	50
4.5 SUMMARY	51

CHAPTER 5.	GENERATED PARALLEL PROGRAMS	52
5.1	GENERATED MASTER PROGRAM	52
5.1.1	<i>A Master Program Example</i>	54
5.2	GENERATED WORKER PROGRAM	55
5.2.1	<i>A Worker Program Example</i>	57
5.3	THE WORKFLOW AT RUNTIME	60
5.4	SUMMARY	61
CHAPTER 6.	EXTENDED TUPLE SPACE SUPPORTS	62
6.1	WORKLOAD PARTITION	62
6.2	DATA PARTITION AND RESULT ASSEMBLY	64
6.3	SCALAR MANIPULATION	67
6.4	THE API CALLS	67
6.5	SUMMARY	68
CHAPTER 7.	EXPERIMENTAL RESULTS	69
7.1	MATRIX MULTIPLICATION	69
7.1.1	<i>The Sequential Solution</i>	69
7.1.2	<i>PML-tagged Sequential Program</i>	71
7.1.3	<i>Parallel Program Workflow</i>	73
7.1.4	<i>Parallel Processing Performance</i>	74
7.2	LAPLACIAN SOLVER USING GAUSS-SEIDEL ITERATION	75
7.2.1	<i>The Sequential Solution</i>	75
7.2.2	<i>PML-tagged Sequential Program</i>	78
7.2.3	<i>Parallel Program Workflow</i>	82
7.2.4	<i>Parallel Processing Performance</i>	83
7.3	ION GENERATION SIMULATOR	85
7.3.1	<i>The Sequential Solution</i>	85
7.3.2	<i>PML-tagged Sequential Program</i>	90
7.3.3	<i>Parallel Program Workflow</i>	94
7.3.4	<i>Parallel Processing Performance</i>	95
7.4	BLOCK LU FACTORIZATION	98
7.4.1	<i>The Sequential Solution</i>	99
7.4.2	<i>PML-tagged Sequential Program</i>	103
7.4.3	<i>Parallel Program Workflow</i>	107
7.4.4	<i>Parallel Processing Performance</i>	108
CHAPTER 8.	CONCLUSION	109
8.1	SUMMARY OF CONTRIBUTIONS	110

8.2	FUTURE WORK.....	111
Appendix A	- Parallel Programs For The Example In Section 3.3.....	113
Appendix B	- Parallel Programs For Matrix Multiplication.....	124
Appendix C	- Parallel Laplacian Solver Using Gauss-Seidel Iteration.	129
Appendix D	- Parallel Programs For Ion Generation Simulator.....	137
Appendix E	- Parallel Programs For Block LU Factorization.....	147
REFERENCES	155

LIST OF FIGURES

Figure 2.1	A Simple HPF Program	7
Figure 2.2	Tuple Space	11
Figure 2.3	The Stateless Parallel Processing (SPP) Architecture	14
Figure 2.4	Crossbar Switch.....	14
Figure 2.5	Unidirectional Virtual Ring.....	16
Figure 2.6	Traditional Dataflow Model(a) and the SPP Architecture(b)	17
Figure 2.7	The SPP Architectural Support	18
Figure 2.8	A Sample XML Document.....	25
Figure 2.9	The Sample DTD File.....	28
Figure 3.1	Default DataFlow Direction.....	33
Figure 3.2	DataFlow Direction With XCHG Option	34
Figure 4.1	The Structure of The Parallelizing Compiler.....	45
Figure 4.2	The Tag-tree Of The Simple Sequential Program	47
Figure 4.3	The Tag-tree Of The Master Program	48
Figure 4.4	The Tag-tree Of The Worker Program	49
Figure 5.1	The Structure of Master Program	53
Figure 5.2	The Structure of Worker Program.....	56
Figure 5.3	Workflow For The Example Programs	60
Figure 6.1	An Example of Workload Partition in Tuple Space.....	63
Figure 6.2	Read Entire Matrix	64
Figure 6.3	Send Entire Matrix.....	65
Figure 6.4	Read A Sub-range Of Matrix	65
Figure 6.5	Send A Sub-range Of Matrix.....	66
Figure 7.1	Workflow For Matrix Multiplication	73
Figure 7.2	A 2-D Grid of Values and a 4-Point Stencil.....	76
Figure 7.3	The Dependency Pattern For Laplacian Solver	79
Figure 7.4	Diagonal Wavefront Of Computation	79
Figure 7.5	Workflow For Laplacian Solver.....	82
Figure 7.6	Workflow For Ion Generation Simulator.....	94
Figure 7.7	Block LU factorization of the partitioned matrix A.	98
Figure 7.8	From Stage k To $k+1$	99
Figure 7.9	Workflow For Block LU Factorization.....	107

LIST OF TABLES

Table 2.1	Important MPI Function Calls	10
Table 7.1	The Performance Comparison For Matrix Multiplication	74
Table 7.2	The Performance Comparison For Laplacian Solver	83
Table 7.3	The Performance Comparison For Ion Generation Simulator	95
Table 7.4	The Performance Comparison For Block LU Factorization	108

CHAPTER 1. INTRODUCTION

Rapid advances in computer hardware and communication technologies have shown that distributed-memory supercomputers is the least expensive to build while promising the most potential for high performance and reliability. Today, according to the web site “www.top500.org”, over 40% of the top 500 supercomputers are clusters of computers.

The best-known difficulty in parallel processing research is parallel programming. The primary open problem is in automatic dependency analysis – a critical factor for program partitioning. All existing parallelizing compilers fall short to solving non-linear, indirect and conditional dependencies. Consequently, the user must manually control the program partition and coordination even for shared-memory supercomputers. The use of scalable distributed-memory supercomputers makes programming even more difficult due to the lack of a uniform memory space. For distributed-memory supercomputers, the user has to deal with additional missions, such as data partition/distribution and inter-process communication/synchronization.

1.1 Proposed Approach

This thesis investigates a new approach for automatic sequential-to-parallel program translation by leveraging a dataflow computation model. The dataflow computation model allows automatic dependency uncovering without complex static dependency

analysis. In particular, this thesis focuses on a coarse-grain dataflow model supported by a stateless parallel processing architecture that promises both high performance and high availability at the same time.

Under this model, as long as the user can describe an overall dependency pattern of a sequential program, this pattern can be used to direct program partition and data distribution strategies for parallel program generation. In other words, the parallel code generator will always generate codes for each sequential program with a described dependency pattern. The parallel performance, however, will depend on the tagging strategy. The worst-case is that the workflow of parallel execution is serialized producing zero or negative performance gain. A fundamental departing point from other approaches is that all these can be accomplished without concerning any non-linear, indirect and conditional dependencies (as they are automatically uncovered at runtime).

In contrast, approaches without using the dataflow computation model, such as the PARADIGM project at UIUC, have encountered insurmountable difficulties when attempting to solve nonlinear and other dependencies at compile time.

In this thesis, the dataflow computation model is provided by the Synergy system – a preliminary implementation of a stateless parallel processing architecture using multiple networked computers. Our technical approach includes a new Parallelization Markup Language (PML) – a special purpose XML (Extensible Markup Language). It is used to describe an overall dependency pattern by marking sequential program segments. A PML

compiler translates the marked sequential program into multiple parallel programs using the extracted dependency pattern. Based on XML technique, PML is portable and extensible. It is completely independent from programming languages. The PML tags are also powerful enough to describe very complex dependency patterns (thus data partition strategies).

Theoretically, the proposed methodology is applicable to all types of iterative compute-intensive applications. In this thesis, we have chosen four well-recognized numerical applications to illustrate the practical utility and efficiency of the proposed method: Matrix Multiplication, Laplacian Solver using Gauss-Siedel iterations, Ion Generation Simulator and Block LU Factorization. We show that performance measurements from generated programs compare favorably against manually written parallel programs.

1.2 Envisioned Contributions

This thesis is the first effort to automatically generate parallel programs for distributed-memory supercomputers without using direct message-passing protocols. The envisioned contributions of this research include:

1. Design of Parallelization Markup Language (PML).
2. Design and implementation of PML compiler.
3. Extended tuple space support for simplifying parallel program generation and optimizing runtime parallel performance.

1.3 Thesis Organization

This dissertation is organized as follows. Chapter-2 provides the background information. Chapter-3 describes the PML tag set. Chapter-4 describes PML compiler details. Chapter-5 presents structures of the generated parallel programs. Chapter-6 describes the extended tuple space support. Chapter-7 presents experimental results of the four numerical applications. Lastly, Chapter-8 contains the final conclusion.

CHAPTER 2. BACKGROUND

This chapter presents background information related to the proposed research.

2.1 Parallel Programming Models

Parallel programming models can be roughly categorized into two classes: (1) programming language extensions; and (2) parallel programming libraries. This section presents an overview of these models.

2.1.1 Programming Language Extensions

Programming language extensions involve the design of parallel processing constructs for a host sequential programming language, and typically target vectorizing or shared-memory supercomputers. The parallel constructs are integrated into the host language as language extensions. When writing parallel programs, a programmer applies parallel constructs to specify parallelizable portions in a sequential program. The compiler generates specialized instructions to control multiple processes running across multiple processors simultaneously. A well-known example is High Performance Fortran.

- High Performance Fortran (HPF)

Based on earlier significant efforts, such as Fortran-D [13] and CM-Fortran [22, 32, 38], HPF (High Performance Fortran [17]) is one of the best-known parallel compiler systems. HPF provides an efficient environment for programmers writing parallel applications in Fortran language for vectorizing or shared-memory supercomputers. HPF contains a set of machine-independent directives and parallel processing constructs as the extensions to Fortran-90. Examples include the **PURE** procedure and the **FORALL** statement.

A **PURE** procedure has no execution side effects other than through its parameters. Programmers guarantee that a **PURE** procedure can execute simultaneously across multiple processors with no ill effects. This allows HPF to assume that it will only operate on local data and does not need any data communication during the execution.

A **FORALL** construct resembles a **DO** loop construct, but its evaluation rules really treat the statements within the construct body as indexed parallel operations, in which the execution of each statement for all selected index values is distributed to multiple processors, and a statement is executed for all selected index values before the next statement is executed. All the procedures inside a **FORALL** construct body must be **PURE**. Figure 2.1 shows a simple HPF program containing a **FORALL** construct.

Figure 2.1 A Simple HPF Program

```
FORALL (I=2:N-1, J=2:N-1)
  A(I, J) = (A(I+1, J)+A(I-1, J)+A(I, J+1)+A(I, J-1))/4.0
  B(I, J) = 1.0/A(I+1, J+1)
END FORALL
```

For an assignment statement in this example, all expressions on the right hand side are evaluated for all selected index values and the evaluations may occur in any order of the selected index values. After all of the evaluations have been performed, the assignments for the statement may occur in any order. Thus in the first assignment statement, it is always the original values of the elements in array A that participate in the calculation. In the second assignment statement, it is the new values of the elements of array A that determine the values of the elements of array B .

- The PARADIGM Project

The PARADIGM (PARAllelizing compiler for Distributed-memory General-purpose Multicomputers) project [4, 15] at University of Illinois uses information expressed in HPF directives and parallel constructs for automatically compiling HPF programs into efficient message-passing programs for distributed-memory supercomputers.

In the PARADIGM project, programmers write HPF directives and parallel constructs to assist the compiler with the process of efficiently compiling code, as well as providing information that is impossible to determine at compile-time or efficiently at run-time [18].

Using HPF directives and parallel constructs, however, the project ties itself to the host programming language FORTRAN.

- The OpenMP Specification

OpenMP [26] is an Application Programming Interface (API) specification that can be used to specify shared-memory parallelism in Fortran and C/C++ programs. The MP in OpenMP stands for Multi Processing. OpenMP consists of three primary API components: compiler directives, runtime library routines, and environment variables. The API is specified for Fortran and C/C++ programs and implemented for multiple platforms including most Unix platforms and Windows NT.

OpenMP is the latest effort to standardize shared-memory parallel programming directives in the industry [37]. Since the early 90's, vendors of shared-memory supercomputers have each provided a unique set of directives, very similar in syntax and semantics. A developer would augment a sequential program with directives specifying which loops were to be parallelized, and the compiler would be responsible for automatically parallelizing such loops across the SMP processors. Implementations were all functionally similar, but were diverging (as usual). An earlier standardization effort, the draft for ANSI X3H5 in 1994, was never formally adopted, largely due to waning interest as distributed-memory supercomputers became popular. OpenMP consolidates these different directive sets into a single syntax and semantics, and finally delivers the long-awaited promise of single source portability for shared-memory parallelism.

OpenMP also addresses the inability of previous shared-memory directive sets to deal with coarse grain parallelism [37]. In the past, limited support for coarse grain work has led to developers thinking that shared-memory parallel programming was inherently limited to fine-grain parallelism -- this isn't the case with OpenMP. The “orphaned” directives in OpenMP offer the features necessary to represent coarse-grained algorithms.

OpenMP was designed to exploit certain characteristics of shared-memory architectures [37]. The ability to directly access memory throughout the system (with minimum latency and no explicit address mapping) combined with very fast shared memory locks, makes shared-memory architectures best suited for supporting OpenMP. Systems that don't fit the classic shared-memory architecture may provide hardware or software layers that present the appearance of a shared-memory system, but often at the cost of higher latencies or special limitations. For example, OpenMP could be implemented for a distributed-memory system on top of MPI, so OpenMP's latencies would be greater than that of MPI (whereas typically the reverse is the case on a shared-memory system).

2.1.2 Parallel Programming Libraries

Parallel programming libraries involve the design of parallel processing libraries for some sequential programming languages, and typically target distributed-memory supercomputers. A programmer first partitions a sequential program into multiple

subprograms that can run in parallel, and then uses procedural calls for inter-program communication/synchronization. The resulting parallel application typically includes a “master” program that partitions the total computation into multiple pieces and distributes them to multiple “worker” programs. Parallel processing starts when multiple “worker” programs compute at the same time.

Well-known examples include MPI [27], PVM [5], Linda [7, 8] and Synergy [36]. In particular, MPI and PVM are message passing parallel systems while Linda and Synergy use a tuple space mechanism.

- Message Passing

A message passing system, such as MPI (Message Passing Interface) or PVM (Parallel Virtual Machine), provides a library of message passing procedure calls. Parallel programs communicate with each other synchronously and directly by sending messages.

Table 2.1 shows 6 most important calls out of total 125 calls in MPI library:

Table 2.1 **Important MPI Function Calls**

<i>MPI Name</i>	<i>Functionality</i>
MPI_INIT	Intializes API
MPI_COMM_SIZE	Finds out how many processors are active
MPI_COMM_RANK	Returns the number of the calling processes
MPI_SEND	Sends a message
MPI_RECV	Receives a message
MPI_FINALIZE	Terminates MPI

In MPI system, an message buffer is defined as a triplet (*address, count, data type*), where the *address* specifies the start of the buffer; *count* specifies the number of data units contained in the buffer; and *data type* specifies the data type of the information being transferred. There are two basic message operations: *send* and *receive*. Each message is marked with a unique tag that is used by the receiver for rearranging message sequence if messages arrive out of order. The tags are allocated at runtime and no wildcard tag matching is allowed in message receiving.

- *Tuple Space*

A tuple space system, such as Linda or Synergy, provides a library of function calls for tuple operations. Parallel programs communicate with each other through one or more tuple spaces asynchronously, as shown in Figure 2.2. Tuple operations are the only means for communication/synchronization among parallel programs.

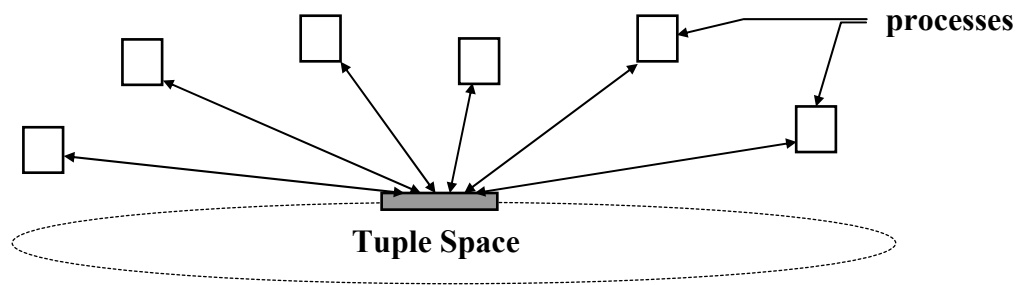


Figure 2.2 **Tuple Space**

A tuple space is an abstract space defined for holding nothing but tuples. In Synergy, tuples are stored in FIFO order. Each tuple is a named object with the format (*tuple_name*, *data*). Tuple names must be unique in a tuple space. Pattern-matching rules are used for updating or removing tuples, rather than addressing directly. There are three tuple operations: *put*, *read* and *get*. Each tuple space has only one entry port through which only one tuple transaction can be performed at any given time. Distributing multiple tuple spaces across multiple computers can minimize tuple transactions' latency and provide tuple matching load balance.

In comparison to message passing systems, the tuple space mechanism provides critical flexibility in communication/synchronization patterns required for dynamic load balancing and fault tolerance in scalable parallel processing systems.

2.1.3 Summary

Existing parallel programming models use language extensions and parallel libraries. After more than a decade of practice, it is generally agreed that it is quite difficult to write an efficient parallel program by hand, using either language extensions or parallel libraries. The programming models are tied to particular host programming languages and specific parallel processing architectures. When writing a parallel program, a programmer must not only learn language extensions or parallel libraries, but also be familiar with underlying architectures. Such difficulties make automatic program parallelization approaches highly desirable [1, 24, 28, 30].

2.2 Stateless Parallel Processing (SPP) Architecture And Synergy System

This thesis investigates automatic parallel program generation for the Stateless Parallel Processing (SPP) architecture [34]. This section presents an overview of the SPP architecture and its preliminary implementation - the Synergy system.

2.2.1 The SPP Architecture

- SPP Architecture Concepts

In the SPP conceptual architecture, fully configured computers (or nodes) are interconnected together by *multiple redundant switching networks* and a *unidirectional virtual ring network*, as shown in Figure 2.3.

It is designed that the collective multiple switching network has a relative constant delay. A typical implementation of a network switch is a crossbar switch. A crossbar switch is a grid of logic gates; adding more nodes increases the capacitive loading, but the delay is increased only slightly as long as packaging boundaries aren't exceeded, e.g., an intra-board connection becomes an inter-board connection [23]. Figure 2.4 shows the inside of a crossbar in a schematic way.

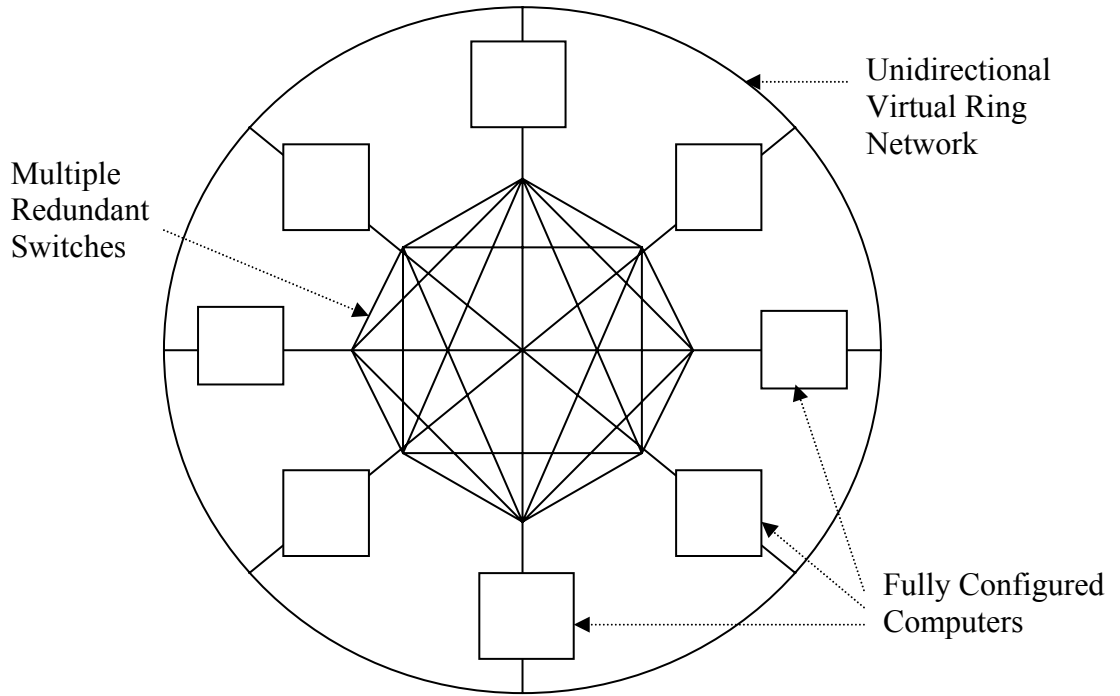


Figure 2.3 The Stateless Parallel Processing (SPP) Architecture

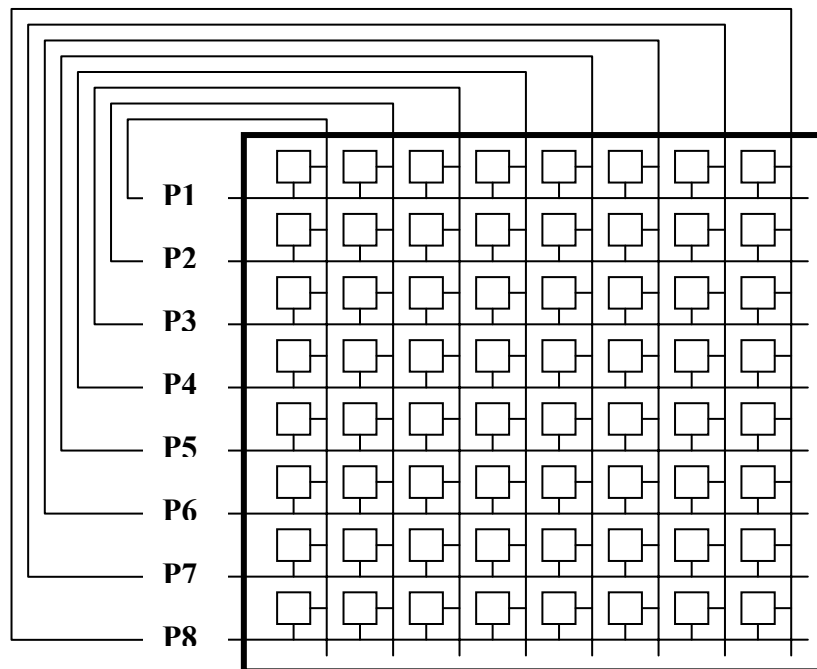


Figure 2.4 Crossbar Switch

The collective multiple switching network provides multiple direct paths from every node to every others, and the redundant paths are used for scalable performance and fault tolerance purposes. These properties promise both high performance and high availability at the same time.

In the unidirectional virtual ring network, each node maintains a list of nodes participating on the ring and can automatically detect and isolate faulty nodes. In the SPP architecture, the virtual ring is only responsible for tuple queries and SPP backbone management. Actual tuple transmissions are direct point-to-point. The unidirectional virtual ring network also provides a reliable multicast capability in that any message sent on the network can be accessed by any node, as shown in Figure 2.5. In addition, the SPP backbone can provide full bandwidth support for multiple simultaneous multicasts.

The use of tuple space programming model allows parallel programs uncoupled in time [9]. The parallel programs allow automatic formation of SIMD, MIMD and pipeline processor clusters at runtime and automatic uncovering of any non-linear, indirect and conditional dependencies.

This design promises extremely high performance leveraging multiple redundant computing nodes and communication switches while providing resilience to multiple computing node and communication switch failures.

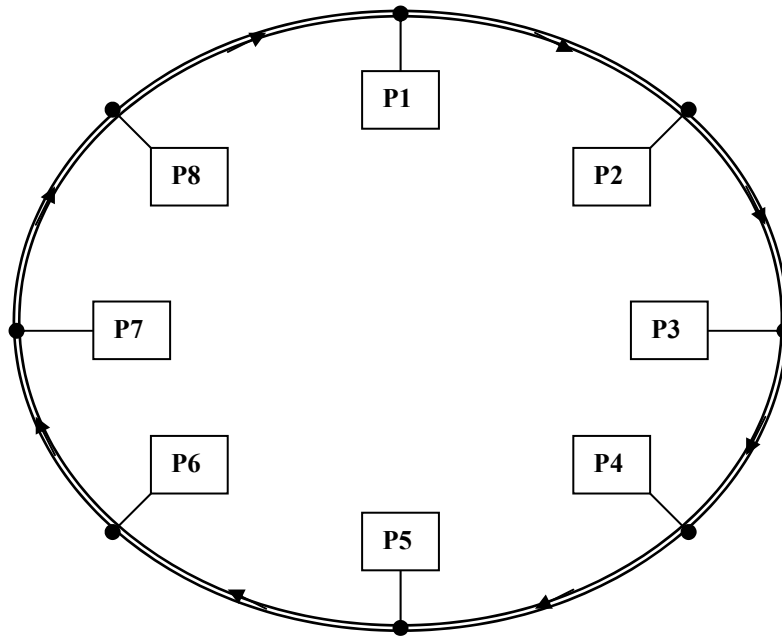


Figure 2.5 Unidirectional Virtual Ring

The SPP architecture supports a coarse-grain *dataflow computation model* [40, 11].

- SPP Computation Model

Traditional dataflow computation model [2, 16, 20], shown in Figure 2.6(a), has an implicit emphasis on very-fine grain parallelism (i.e. instruction-level granularity). The basic idea is that data flow from instruction to instruction, and rather than a central control mechanism, instructions may execute at any time after the arrival of the data they need. Although its theoretical potential of extracting the maximum amount of parallelism from a sequential program is fairly intuitive, the chief concern is the overhead involved in data matching and instruction firing that leads to excessive communication overheads.

This in fact was the major reason for its performance failures.

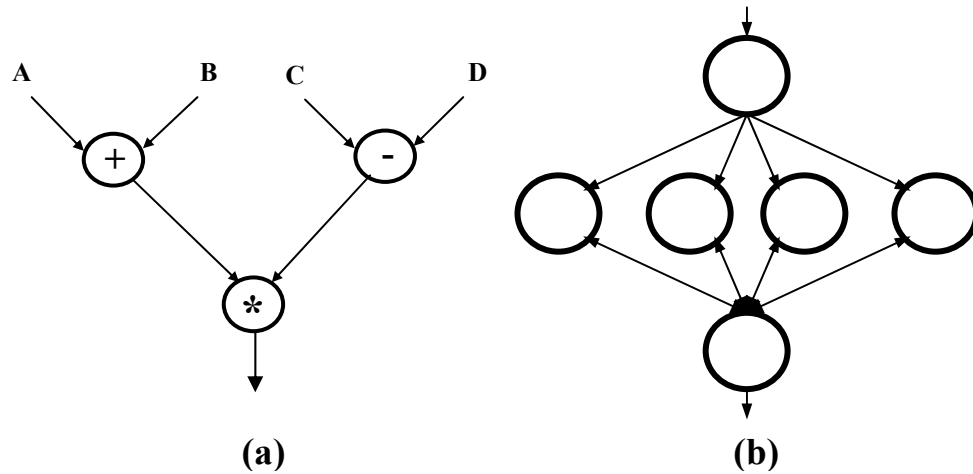


Figure 2.6 **Traditional Dataflow Model(a) and the SPP Architecture(b)**

In the SPP architecture, shown in Figure 2.6(b), tuples flow from process to process. Each SPP process is responsible for a part of the overall work, and is a pure tuple-driven program without explicit global state controls (i.e. “*stateless*”); and it may “fire” only after arrival of the tuple it needs. Thus, distributed processing significantly offsets the overhead involved in tuple matching and process firing.

Parallel applications in SPP involve multiple Master and Worker programs. A Master program represents the central control of a collection of working units. It includes distribution of assignments and assembly of results. A Worker program represents an agent that repeatedly processes assignments until termination. This is also known as the “scatter-and-gather” parallel processing model.

The SPP architecture and dataflow computation model can automatically uncover all parallelisms embedded in any set of properly partitioned programs.

- SPP Parallel Processing Example

Using the SPP architecture, the master program along with the tuple spaces can be placed at any computing node while worker programs are started in a way of one on each node, as shown in Figure 2.7.

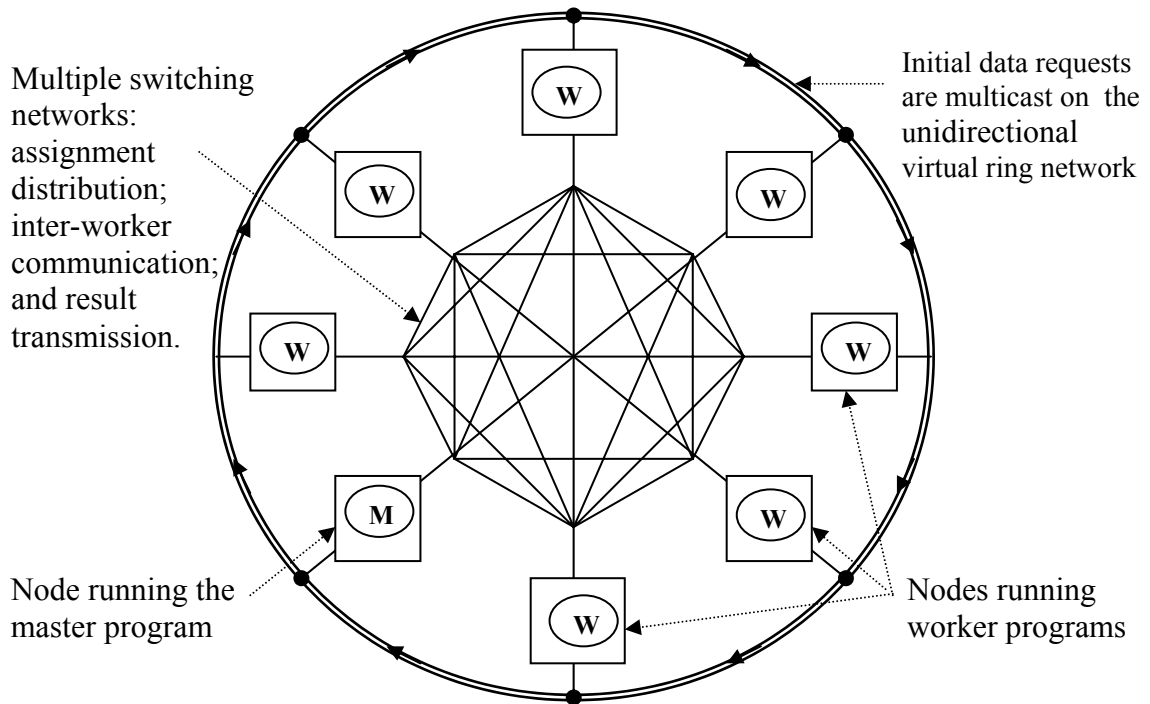


Figure 2.7 The SPP Architectural Support

All SPP programs are pure tuple-driven processes without explicit global state controls (i.e. “stateless”). They may start processing an assignment only after matching their

tuple queries. The collective multiple switching network supports massive concurrent data transmissions with high performance. Assignments are automatically distributed to worker programs and results are sent back to the master program.

The SPP computation model allows automatic uncovering of all types of dependencies at runtime. Thus, it can automatically schedule parallel programs to form SIMD, MIMD and pipeline clusters at runtime. It allows dynamic system reconfiguration without downtime. For generation of parallel programs from sequential code, automatic uncovering of dependencies paves a critical path to success.

2.2.2 Synergy – A Preliminary Implementation Of SPP

Synergy uses passive objects for inter-process communication. Supported communication mechanisms include tuple space, file and database. As a simplified SPP prototype implementation that lacks an efficient tuple matching mechanism and full fault tolerance support. Synergy runs on clusters of computers. It offers programming ease, load balancing and some fault tolerance benefits.

- Parallel Programming With Synergy

Synergy API (application programming interface) is a small set of operators defined in a parallel programming library, named Synergy Language Injection Library (LIL), which contains the procedure calls for all three supported inter-process communication

mechanisms. Synergy API blends well into the conventional sequential programs. It is particularly helpful for re-engineering legacy applications.

Programmers use the API to compose parallel programs for Synergy. These parallel programs use a conventional open-manipulate-close sequence for each passive object. Each parallel program is individually compiled using a conventional compiler and the parallel library LIL.

- Synergy Services Components

Synergy contains the following service components:

- 1) Two memory resident service daemons (*PMD* and *CID*). These daemons resolve network references and are responsible for remote process/object execution and management.
- 2) Two dynamic object daemons (*TSH* and *FAH*). These daemons are launched before every parallel application begins and are removed after the application terminates. They implement the defined semantics of LIL operators.
- 3) A customized Distributed Application Controller (*DAC*). This program actually synthesizes a multiprocessor application. It conducts processor binding and records relevant information about all processes involved in the application until completion. DAC represents a customized virtual multiprocessor for each application.
- 4) Synergy shell: (*prun* and *pcheck*). These programs are Synergy runtime user interface. Prun launches a parallel application. Pcheck is a runtime monitor for managing

multiple parallel applications and processes.

- Synergy At Runtime

Synergy can execute multiple parallel applications on the same cluster at the same time.

A parallel application is synthesized through its configuration file and an automatic processor-binding algorithm. The configuration file is written in CSL (Configuration Specification Language). For an application to run, the user must specify the inter-process communication mechanism and the process coordination in the file.

Synergy's automatic processor binding algorithm is extremely simple: unless specifically designated, it binds all tuple space objects, one master and one worker to a single processor. Other processors run the worker-type (with repeatable logic) processes, one for each. Since network is the bottleneck, this binding algorithm minimizes network traffic thus promising good performance for most applications using the current tuple matching engine.

At runtime, the application's dynamic dataflow is converted to a static, bipartite IPC (Inter-Program Communication) graph, which is used for automatically mapping parallel programs onto a set of networked computers.

The fault tolerance feature in Synergy covers two possible cases: processor failures and worker process failures. Processor failures discovered before a run are automatically

isolated. Worker process failures during a parallel execution is treated by a "tuple shadowing" technique. With the feature, Synergy can automatically recover the lost data from a lost worker with little overhead. The feature brings the availability of a multiprocessor application to be equal to that of a single processor and is completely transparent to application programs.

Synergy provides the basis for automatic load balancing. Optimal load balancing requires adjusting tuple sizes. The adjustments can adapt guided self-scheduling [31], factoring [19] or fixed chunking with the theory of optimal granule size for load balancing [33].

- Synergy History And Future

At the time of this writing, the latest Synergy version, V3.0, runs on SunOS 5.9 and RedHat Linux 9 (*Strike*). It is an enhancement to Synergy V2.0 (released in early 1994). Earlier versions of the same system appeared in the literature under the names of Configurator (1982), ZEUS (1986), MT (1989) and Synergy V1.0 (1992) respectively.

Next generation of Synergy will be the Stateless Machine (SLM), a full implementation of the SPP architecture. In addition to all benefits that Synergy offers, SLM will have a distributed tuple matching engine that promises to fulfill a wider range of performance requirements. It will also offer a non-stop computing platform with total fault tolerance. In SLM, the static IPC graph will be implemented via a self-healing backbone.

2.3 XML (Extensible Markup Language)

Derived from SGML (Standard Generalized Markup Language), XML (Extensible Markup Language) is a cross-platform, software and hardware independent markup tool [39]. The main benefits of XML are that it is extensible, it is structured, and it is validating.

Much like HTML (HyperText Markup Language), XML uses `<`, `>` and `&` to create markup tags (element and attribute structures). But There are two key differences between XML and HTML. First, unlike HTML designed for displaying information, XML is designed for describing information. Second, HTML tags are predefined, but XML tags are not. While authors can only use the tags defined in the HTML standard for marking HTML documents, XML allows authors to define their own tags.

The approach in this thesis includes an XML-based Parallelization Markup Language (PML), designed for describing an overall dependency pattern by marking sequential program segments. This section presents an overview of XML.

2.3.1 What Does XML Look Like?

All XML documents must satisfy a number of rules to be considered “well-formed”. Programs only know how to deal with well-formed documents. Basically, a well-formed document is one that is properly tagged. Every tag must have a corresponding closing tag

unless the tag is empty (and there is a short-hand notation for that). If a tag contains attributes, then the value of each attribute must be enclosed in quotations.

Nesting is also important. The last opened tag must be the first to close. Here is a simple demonstration:

```
<header1>  
  <header2>  
    <header3>  
    </header3>  
  </header2>  
</header1>
```

The indentation doesn't mean anything as far as XML is concerned. It is merely for enhancing readability.

If a tag is empty (i.e., it contains nothing between the opening and closing tags), then XML allows a short-hand notation. The following:

```
<tagname/>  
is equivalent to  
<tagname></tagname>
```

Figure 2.8 shows a sample XML document for describing FAQs [14],

Figure 2.8 A Sample XML Document

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE FAQ SYSTEM "FAQ.DTD">
<FAQ>
  <INFO>
    <SUBJECT>XML</SUBJECT>
    <AUTHOR>Lars Marius Garshol</AUTHOR>
    <VERSION>1.0</VERSION>
    <DATE>20.jun.97</DATE>
  </INFO>
  <PART NO="1">
    <Q NO="1">
      <QTEXT>What is XML?</QTEXT>
      <A>SGML light.</A>
    </Q>

    <Q NO="2">
      <QTEXT>What can I use it for?</QTEXT>
      <A>Anything.</A>
    </Q>
  </PART>
</FAQ>
```

XML documents are text-based and tend to be quite verbose, making them highly readable and portable. XML parsing is well-defined and widely-implemented on different platforms, making it simple to describe and retrieve complex information in XML documents in a variety of environments.

2.3.2 XML Is Extensible

As its name suggests, XML is extensible. XML allows authors of XML documents to define their own markup tags. As shown in above example, the tags, such as <FAQ> and

<INFO>, are not defined in any XML standard, but are “invented” by the author of the XML document. The extensibility offers authors more options for defining meanings in their XML documents.

By defining their own markup tags, authors can encode the information of their XML documents much more precisely. As a result, programs processing these documents can "understand" them much better and, therefore, process the information in ways that are impossible with other markup tools (or ordinary text processor).

- XML Applications

Creating new markup tags makes XML a markup language that can create other markup languages, including HTML. The extensibility immediately opens up unlimited possibilities in that it allows an expert in a particular field to define the standard for that field. Many industries have already started efforts to define their own standards. Such standards are called XML applications.

Some XML applications that are either complete or under development are:

- Bio-informatic Sequence Markup Language (BSML) - a standard for specifying biological characteristics, such as DNA sequences.
- Information and Content Exchange (ICE) - to facilitate content exchange among web sites.

- Mathematical Markup Language (MathML) - to markup various mathematical entities and notations.
- Open Financial Exchange (OFX) - a standard for exchange of financial information.
- Open Software Description (OSD) - a standard for specifying various parts of a software package.
- Open Trading Protocol (OTP) – to provide an interoperable framework for internet commerce.
- Tutorial Markup Language (TML) - a standard for marking a tutorial.
- vCard (Electronic Business Card) - a standard for specifying business card-like data.
- Weather Observation Markup Language Format (OMF) - a standard for weather related information.

2.3.3 XML Is Structured

XML adds structure to the information it describes. XML not only provides meaningful tags to explain the information, but also holds a certain structure. In above example, we see a “containment” relationship where the tags, <INFO> and <PART>, are elements contained in tag <FAQ>, and the tags, <SUBJECT>, <AUTHOR>, <VERSION> and <DATE> are contained in tag <INFO>. The inherent structure of XML documents is very important when it comes to manipulation and interaction between a programming language and XML documents.

2.3.4 XML Validation

XML is self-descriptive with a Document Type Definition (DTD) or an XML Schema. Using a DTD, developers can create a set of rules that will validate their documents. Below is the DTD for the markup tags shown in above example [14],

Figure 2.9 The Sample DTD File

```
<!ELEMENT FAQ      (INFO, PART+)>
<!ELEMENT INFO     (SUBJECT, AUTHOR, VERSION?, DATE?)>
<!ELEMENT SUBJECT  (#PCDATA)>
<!ELEMENT AUTHOR   (#PCDATA)>
<!ELEMENT VERSION  (#PCDATA)>
<!ELEMENT DATE     (#PCDATA)>
<!ELEMENT PART     (Q+)>
<!ELEMENT Q        (QTEXT, A)>
<!ELEMENT QTEXT    (#PCDATA)>
<!ELEMENT A        (#PCDATA)>
<!ATTLIST PART     NO      CDATA #IMPLIED
                  TITLE  CDATA #IMPLIED>
<!ATTLIST Q        NO      CDATA #IMPLIED>
```

<!ELEMENT> is used to define elements like this: **<!ELEMENT NAME CONTENTS>**, where *NAME* gives the element name, and *CONTENTS* describes which elements are allowed inside this element. ‘?’ after an element means that it can be skipped, ‘+’ means that it must be included one or more times, and ‘*’ means that it can be skipped or included one or more times. **#PCDATA** means ordinary text without markup.

<!ATTLIST> defines the attributes of an element. In above DTD file, both *PART* and *Q* have an attribute, called *NO*, which contains ordinary text and can be skipped. In addition to *NO*, *PART* has another attribute, called *TITLE*, which also contains ordinary text and can be skipped.

The DTD provides a standard mechanism to specify a grammar. That means, if a program can interpret a DTD, it can interpret a wide variety of rules specified by the DTD. The specification of XML rules via a DTD also means that a document that claims to be compliant to a specific XML application can easily be checked. This means that the program itself does not have to worry about validity of a given document. This is important since it guarantees information uniformity. Programs can be written so they focus on what the information is rather than what the information should be. If the information is verified before processing occurs, then the program has to deal less with exceptions and “special” cases. In actual fact, the DTD cannot capture every kind of rule, so the program must still do some error checking.

2.3.5 Summary

XML is an extensible, structured and validating markup language, designed for describing information. XML documents are text-based and tend to be quite verbose, making them highly readable and portable. With XML, developers can create new markup tags for their particular projects. In this research, we propose an XML-based Parallelization Markup Language (PML), for describing an overall dependency pattern by marking sequential program segments. To the best knowledge of the author, PML is the first attempt of applying the XML technique for parallel processing.

CHAPTER 3. THE PARALLELIZATION MARKUP LANGUAGE (PML)

In automatic program parallelization approaches, there are always the information that the parallelizing compiler needs for the process of efficiently compiling code but can hardly determine by itself. Programmers must provide it with the information in some way. While using HPF directives and parallel constructs for this purpose, the PARADIGM project ties itself to the programming language, FORTRAN. In this research, we propose an XML-based Parallelization Markup Language (PML), which is completely independent from programming languages.

This chapter presents the specification of PML in detail. We will first illustrate PML tags. We then present the DTD document for PML. Finally, we mark a sample sequential program to illustrate how to use PML for generating efficient parallel programs.

3.1 PML Tag Specification

There are altogether eight tags organized in two types: a) manually inserted tags and b) automatically inserted tags.

Manual tags include the following:

- Code blocking tags (*parallel tag*, *master tag*, *worker tag* and *reference tag*),
- A loop partition tag (*target tag*),

- Communication tags (*send tag* and *read tag*).

The manual tags specify

- How to split the code into master and worker(s)?
- Which loop need to be parallelized?
- What data need to be communicated between master and workers?
- How to partition data?

The PML manual tags can be considered as the API (Application Programming Interface) for using the SPP parallel processor.

Automatic tags include

- A token tag.
- XML standard header and trailer tags (these are NOT considered as PML tags).

Automatic tags are transparent to users. The *token tag* is for distributing tasks. It is generated based on the information collected from *target tags*. The XML header tags are automatically inserted in order to conform to XML standard for parsing the marked sequential program as a “well-formed” XML document.

3.1.1 The *Target Tag*

A *target tag* identifies the loop to be parallelized. Each tag identifies one loop. Target tags can be used to mark nested loops. Target tags cannot be nested itself.

```
<target      index="<loop_index_variable_name>"  
              limits="(<start_expr>, <stop_expr>, <step_expr>)"  
              [chunk="<grain_size>" | jump="<jump_size>"]  
              order="<order_number>">
```

Attribute ‘index’ contains the name of the loop’s index variable. Attribute ‘limits’ contains the boundary information of the loop. Attribute ‘chunk’ defines the grain-size of the partition. Attribute ‘order’ specifies the priority number for this loop when there are multiple-nested loops involved.

- *Example 1*

```
<target index="i" limits="(0,N,1)" chunk="G" order="1">  
for (int i = 0; i < N; I++)  
</target>  
{  
    // Loop body.  
}
```

- *Example 2*

```
<target index="i" limits="(0,N,1)" chunk="G" order="1">  
for (int i = 0; i < N; i++)  
</target>  
{  
    // Loop body1.  
    <target index="j" limits="(0,N,1)" chunk="G" order="2">  
    for (int j = 0; j < N; j++)  
    </target>  
    {  
        // Loop body2.  
    }  
}
```

The first example illustrates how to position the target tag for a loop. The second example illustrates the case for two-nested loops. The values of the ‘order’ attributes allow the user to specify processing order of the targeting loops. It enables PML to describe complex program partitioning strategies, such as the wavefront method [25]. In this example, the outer loop will be partitioned and processed first.

3.1.2 Communication Tags

Communication tags include <send> and <read> tags. The actual effects of these tags transfer data between the parallel programs. Using SPP tuple space, we only have to focus on how to send and read named tuples via the space. By default, as shown in Figure 3.1, <send> tags in master program send data to a tuple space, named “distributor”, while <read> tags in master program read data from a tuple space, named “constructor”; And <send> tags in worker program send data to “constructor” tuple space while <read> tags in worker program read data from “distributor” tuple space.

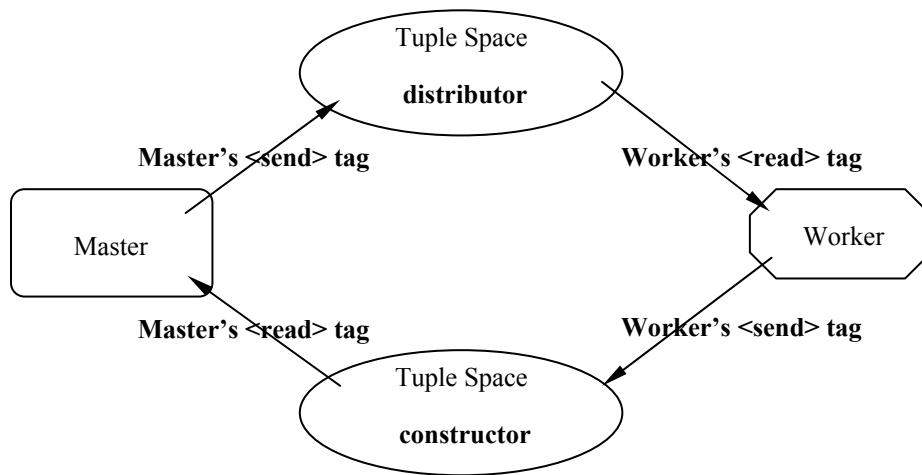


Figure 3.1 Default DataFlow Direction

```

<send var="<var_name>" type="<var_type>" [opt="ONCE|XCHG|_MAX|_MIN"]/>
<read var="<var_name>" type="<var_type>" [opt="ONCE|XCHG|_MAX|_MIN"]/>

```

The values of attributes “var” and “type” are the name and type of the variable to be transferred. The optional attribute ‘opt’ is used for special data transferring requirements. The meanings of the four options are explained below. The “ONCE” option tells the compiler that the data communication should happen only once, for data initialization, in the parallel programs; the XCHG option tells the compiler to switch the default data transfer directions (see Figure 3.2); the “_MAX” option tells tuple space to keep the maximum value for a tuple when being updated. The “_MIN” option tells tuple space to keep the minimum value.

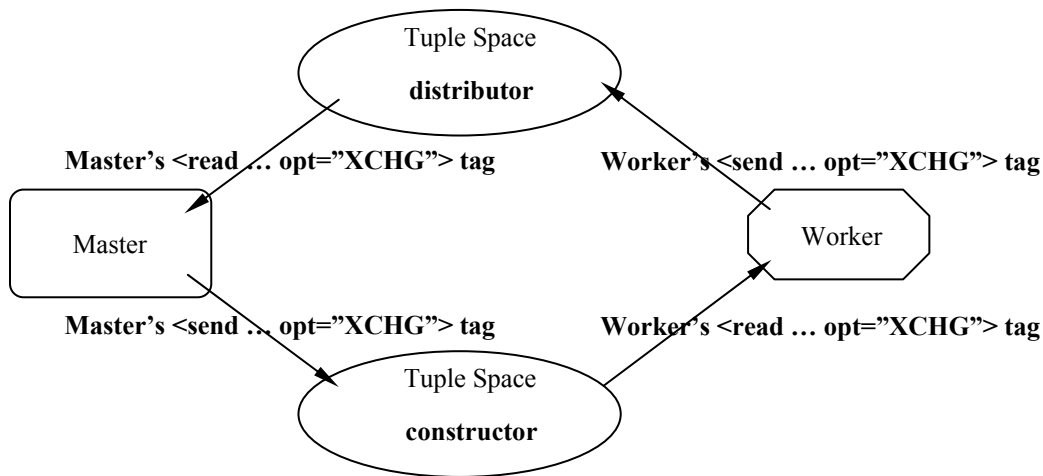


Figure 3.2 DataFlow Direction With XCHG Option

The following examples show the usage of <send> and <read> tags,

- Example 1: `<send var="v" type="int" opt="_MAX">`

This tag sends an integer variable, v, to a tuple space.

- *Example 2:* `<read var="A" type="double [N] [N]">`

This tag reads an entire $N \times N$ matrix, A.

If the data being transferred needs to be partitioned, the information of partitioning strategy may be contained in the formats of attribute “type”, as shown below.

Format-1. `type="double [N (x~y)] [N]"`

This format tells that the data being transferred is a sub-range of a matrix partitioned along the 1st dimension, from row **x** to row **y** (not included). Similarly, formats, “`double [N] [N (x~y)]`” and “`double [N (x1~y1)] [N (x2~y2)]`”, are used for the 2nd and two dimensional partitions respectively.

Format-2. `type="double [N (i)] [N]"`

This format tells that the data being transferred is a sub-range of a matrix partitioned along the 1st dimension in the same way as defined in a *target tag* for partitioning a loop with index **i**. Similarly, formats, “`double [N] [N (j)]`” and “`double [N (i)] [N (j)]`”, are used for the 2nd and two dimensional partitions respectively.

Format-3. `type="double [N (i : $L-x)] [N]"`

This format is similar to the format-2 except that the lower bound of each block from the partitioned matrix is shifted toward the **lower** end by **x** rows. This is for applications that require “boarder elements” located in the lower **x** rows in addition to the main body of the calculation. Alternatively, formats, “`double [N] [N (j : $L-y)]`”

and `"double[N(i:$L-x)][N(j:$L-y)]"`, specify the need for the 2nd dimension and both dimension boarder elements.

Format-4. `type="double[N(i:$U+x)][N]"`

This format is similar to the format-2 except that the upper bound of each block from the partitioned matrix is shifted toward the **upper** end by **x** rows. This is for applications that require “boarder elements” located in the upper **x** rows in addition to the main body of the calculation. Alternatively, formats, `"double[N][N(j:$U+y)]"` and `"double[N(i:$U+x)][N(j:$U+y)]"`, specify the need for 2nd dimension and both dimension boarder elements.

Above formats can be mixed in the same “type” attribute, where each format applies to a dimension, representing a complex data partitioning strategy.

3.1.3 Code Blocking Tags

Code blocking tags include `<parallel>`, `<master>`, `<worker>` and `<reference>`. They instruct the compiler to split the sequential code into master and worker parallel programs.

```
<parallel appname="<parallel_application_name>">
</parallel>
```

The `<parallel>` tag is the outmost wrapper. Each sequential program can have only one `<parallel>` tag, which wraps up the entire execution body of the sequential program. The

value of ‘appname’ attribute is used by the compiler as the part of the generated master and worker program names. Each <parallel> tag must contain at least one <master> tag.

```
<master id="<parallel_code_id">
</master>
```

The <master> tag is used for wrapping a code block suitable for a work distribution center (a parallel master). The <master> tag must be inside of a <parallel> tag. The value of ‘id’ attribute is used together with the value of ‘appname’ attribute in <parallel> tag to create the name of generated worker program. Each <master> tag must contain one <worker> tag.

```
<worker>
</worker>
```

The <worker> tag is used for wrapping the code block suitable for a worker program. This is the core calculating piece of the sequential program. The <worker> tag is inside a <master> tag. Each <worker> tag must contain at least one *target tag* for loop partition.

Both <master> and <worker> tags may contain many communication tags.

```
<reference>
</reference>
```

The <reference> tag is an optional tag for wrapping up additional code pieces, such as macro definitions and variable declarations that should be copied directly to the generated

worker program. The `<reference>` tag is the only tag that can be used outside `<parallel>` tag.

In summary all seven manual tags should be organized as follows,

```
<reference>
</reference>
<parallel>
  <reference>
  </reference>
  <master>
    <send> or <read>
    <worker>
      <send> or <read>
      <target>
        the loop to be parallelized
      </target>
      <send> or <read>
    </worker>
    <send> or <read>
  </master>
</parallel>
```

3.1.4 The Token Tag

The *token tag* is created by the PML parallelizing compiler automatically using the information gathered from the *target tags*. Token distribution code is generated for the *token tag*. The token sent by the master describes the total workload to be processed and the grain size, while each token received by a worker specifies a task (i.e. a piece of the total workload) for processing. For example, if a loop of 1000 iterations is partitioned in every 100 iterations (or grain size), then the total workload is 1000 iterations and each task is 100 iterations. Both workload and task are defined by their boundary information, such as the ‘start’, ‘stop’ and ‘step’ values.

```
<token action="[SET|GET]" idxset="(<loop_index_variable_name> (...)...">
```

The attribute ‘action’ equals “SET” for a master, while it equals “GET” for a worker. Attribute ‘idxset’ defines the list of index variables, corresponding to the loops to be parallelized, separated by parenthesis. For example, if there is only one loop (index variable *i*) to be parallelized, then attribute ‘idxset’ would be ”(i)”; if there are two loops (index variables, *i* and *j*) to be parallelized, then attribute ‘idxset’ would be ”(i)(j)”.

3.2 PML Tag Validation

The grammar of PML is shown in the following DTD document, which is used by the parallelizing compiler for identifying and validating every PML tag in a sequential program.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!ELEMENT root (#PCDATA | reference | parallel)*>
<!ELEMENT reference (#PCDATA)>
  <!ATTLIST reference id CDATA #IMPLIED >
<!ELEMENT parallel (#PCDATA | reference | master)*>
  <!ATTLIST parallel appname CDATA #REQUIRED >
<!ELEMENT master (#PCDATA | send | read | worker)*>
  <!ATTLIST master id CDATA #REQUIRED >
<!ELEMENT worker (#PCDATA | read | send | target)*>
<!ELEMENT send EMPTY>
  <!ATTLIST send var CDATA #REQUIRED
                 type CDATA #REQUIRED
                 opt CDATA #IMPLIED >
<!ELEMENT read EMPTY>
  <!ATTLIST read var CDATA #REQUIRED
                 type CDATA #REQUIRED
                 opt CDATA #IMPLIED >
```

```

<!ELEMENT target (#PCDATA)>
  <!ATTLIST target      index  CDATA  #REQUIRED
                        order  CDATA  #REQUIRED
                        limits CDATA  #REQUIRED
                        jump   CDATA  #IMPLIED
                        chunk   CDATA  #IMPLIED >

```

Note that these rules define the syntax of PML. They do NOT define the syntax of the attributes in each tag. The PLM compiler supplies additional syntax checking for the attributes, such as attributes ‘type’ and ‘opt’ in <send> and <read> tags, and attribute ‘limits’ in <target> tag.

3.3 A PML-tagging Example

The sample program initializes a two-dimensional matrix. It will be used in this section for illustration of the usage of PML, and in latter chapters for description of generated parallel programs.

3.3.1 The Original Sequential Program

The sequential program, as shown below, has two-nested loops that is going through the two-dimensional matrix row by row and from left to right at each row, and setting the value of each element to be a constant.

```

/* Constant, N, is defined in the header file */
#include "init.h"

main()

```

```

{
    double A[N][N];
    double v;
    int i, j;

    v = 99.9;

    for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
        {
            A[i, j] = v;
        }
    }
}

```

3.3.2 PML-tagging Strategies

There are three ways to tag this simple sequential program: horizontal partitioning (loop index i), vertically partitioning (loop index j), and partitioning along both dimensions (or tiling). We show three marked versions below.

- Partitioning Horizontally

```

/* <reference> */
/* Constant, N, and the grain size, G, are defined in the header file */
#include "init.h"
/* </reference> */

/* <parallel appname="init1"> */
int main()
{
    /* <reference> */
    double A[N][N];
    double v;
    int i, j;
    /* </reference> */

    v = 99.9;

    /* <master id="123"> */
    /* <send var="v" type="double" opt="ONCE"/> */
    /* <send var="A" type="double[N][N]"/> */

    /* <worker> */

```

```

/* <read var="v" type="double" opt="ONCE"/> */
/* <read var="A" type="double[N(i)][N]"/> */

/* <target index="i" limits="(0,N,1)" chunk="G" order="1"> */
for (i = 0; i < N; i++)
/* </target> */
{
    for (j = 0; j < N; j++)
    {
        A[i, j] = W;
    }
}

/* <send var="A" type="double[N(i)][N]"/> */
/* </worker> */

/* <read var="A" type="double[N][N]"/> */
/* </master> */
}
/* </parallel> */

```

- Partitioning Vertically

```

/* <reference> */
/* Constant, N, and the grain size, G, are defined in the header file */
#include "init.h"
/* </reference> */

/* <parallel appname="init2"> */
int main()
{
    /* <reference> */
    double A[N][N];
    double v;
    int i, j;
    /* </reference> */

    v = 99.9;

    /* <master id="123"> */
    /* <send var="v" type="double" opt="ONCE"/> */
    /* <send var="A" type="double[N][N]"/> */

    /* <worker> */
    /* <read var="v" type="double" opt="ONCE"/> */
    /* <read var="A" type="double[N][N(j)]"/> */

    for (i = 0; i < N; i++)
    {
        /* <target index="j" limits="(0,N,1)" chunk="G" order="1">
        */
        for (j = 0; j < N; j++)
        /* </target> */

```

```

        {
            A[i, j] = v;
        }
    }

    /* <send var="A" type="double[N][N(j)]"/> */
    /* </worker> */

    /* <read var="A" type="double[N][N]"/> */
    /* </master> */
}
/* </parallel> */

```

- Partitioning On Both Dimensions

```

/* <reference> */
/* Constant, N, and the grain size, G, are defined in the header file */
#include "init.h"
/* </reference> */

/* <parallel appname="init3"> */
int main()
{
    /* <reference> */
    double A[N][N];
    double v;
    int i, j;
    /* </reference> */

    v = 99.9;

    /* <master id="123"> */
    /* <send var="v" type="double" opt="ONCE"/> */
    /* <send var="A" type="double[N][N]"/> */

    /* <worker> */
    /* <read var="v" type="double" opt="ONCE"/> */
    /* <read var="A" type="double[N(i)][N(j)]"/> */

    /* <target index="i" limits="(0,N,1)" chunk="G" order="1"> */
    for (i = 0; i < N; i++)
    /* </target> */
    {
        /* <target index="j" limits="(0,N,1)" chunk="G" order="2">
        */
        for (j = 0; j < N; j++)
        /* </target> */
        {
            A[i, j] = v;
        }
    }

    /* <send var="A" type="double[N(i)][N(j)]"/> */

```

```
    /* </worker> */  
    /* <read var="A" type="double [N] [N]" /> */  
    /* </master> */  
}  
/* </parallel> */
```

Chapter 4 illustrates the parse-tree generated by the PML compiler. Chapter 5 illustrates the generated programs.

3.4 Summary

This chapter presents a Parallelization Markup Language (PML) designed for marking sequential programs for parallel processing using distributed-memory supercomputers. The critical assumption is the use of dataflow computation model implemented by two tuple space objects. This is supported directly by the Synergy parallel processing system – a preliminary implementation of the SPP architecture.

To the best knowledge of the author, PML is the first attempt of applying the XML technique for parallel processing. Compared with other approaches using directives and parallel constructs, PML is more readable, portable and extensible. As demonstrated in later chapters, PML is also powerful enough to implement very complex data partition/distribution patterns.

CHAPTER 4. THE SOURCE-TO-SOURCE PARALLELIZING COMPILER

The source-to-source parallelizing compiler contains three major phases, PML tag parsing, tree transformation and parallel code generation, as shown in Figure 4.1.

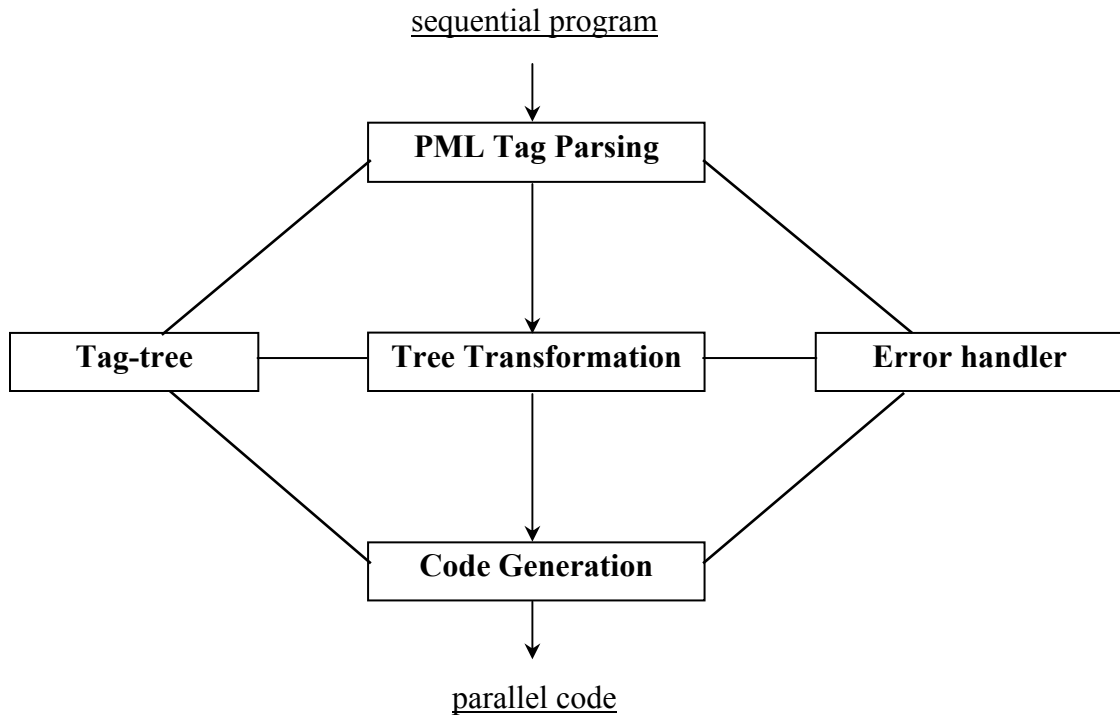


Figure 4.1 **The Structure of The Parallelizing Compiler**

The PML tag parser extracts PML-tags in a sequential program and translates them into a tag-tree. The Tree Transformer splits the tag-tree into master tree(s) and worker tree(s). It also inserts new nodes into both trees to facilitate parallel code generation. Finally, the Parallel Code Generator traverses the master and worker trees, and generates parallel programs. This chapter illustrates each of these phases in detail.

4.1 PML Tag Parsing

The core is a PML parser. It must first validate the embedded PML tags against the DTD for PML. It extracts the PML tags in the sequential code and creates corresponding nodes in a parse tree. The untagged code segments are not parsed but stored as verbatim text nodes. They are linked to the associated PML tag nodes in the tree.

Since the parser focuses only on the PML tags in the sequential program and does not parse untagged programming code, in theory, the proposed system can be used to process sequential programs in any language.

Figure 4.2 shows the tag-tree for the simple sequential program in Section 3.3.2 for partitioning in both dimensions (tiling).

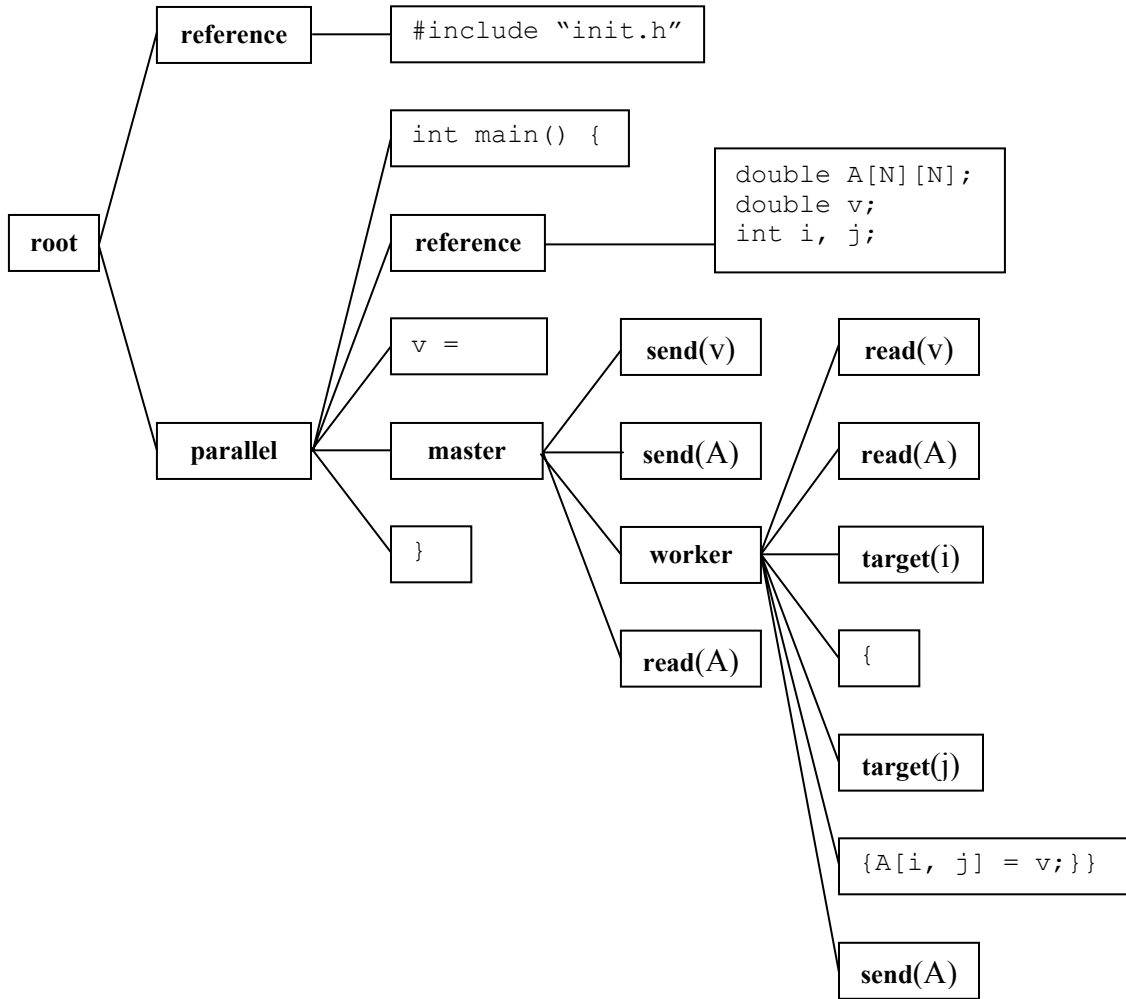


Figure 4.2 The Tag-tree Of The Simple Sequential Program

4.2 Tree Transformation

At the tree transformation phase, the transformer first splits the worker sub-tree off the original tree, and then repairs each tree into an independent parse-tree.

For each removed worker node, the compiler inserts a *token node* in its place in the

original tree. The tree then becomes the template for generating the master program. In order to make worker tree complete, the transformer creates a new root node, copies the *reference* and *parallel* nodes from master tree and adds them under the new *root* node. Finally, it adds the worker sub-tree under the *parallel* node.

In addition, the transformer also inserts some text nodes under parallel and worker nodes as their children. These text nodes contain the untagged sequential program segments, such as the ‘main’ function construct, the ‘while’ loop construct and the ‘exit’ statement. These treatments facilitate the generation of correct worker programs.

Figure 4.3 represents the tag-tree of the Master program for the tagging example in Section 3.3.2 using two dimensional partitioning.

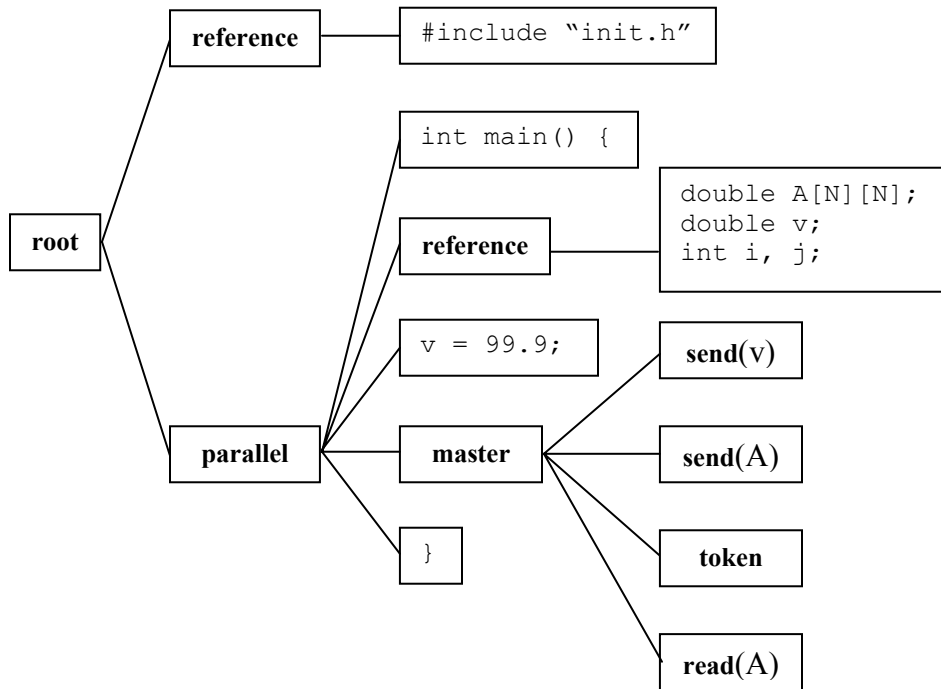


Figure 4.3 The Tag-tree Of The Master Program

Figure 4.4 presents the Worker program tag-tree for the sample.

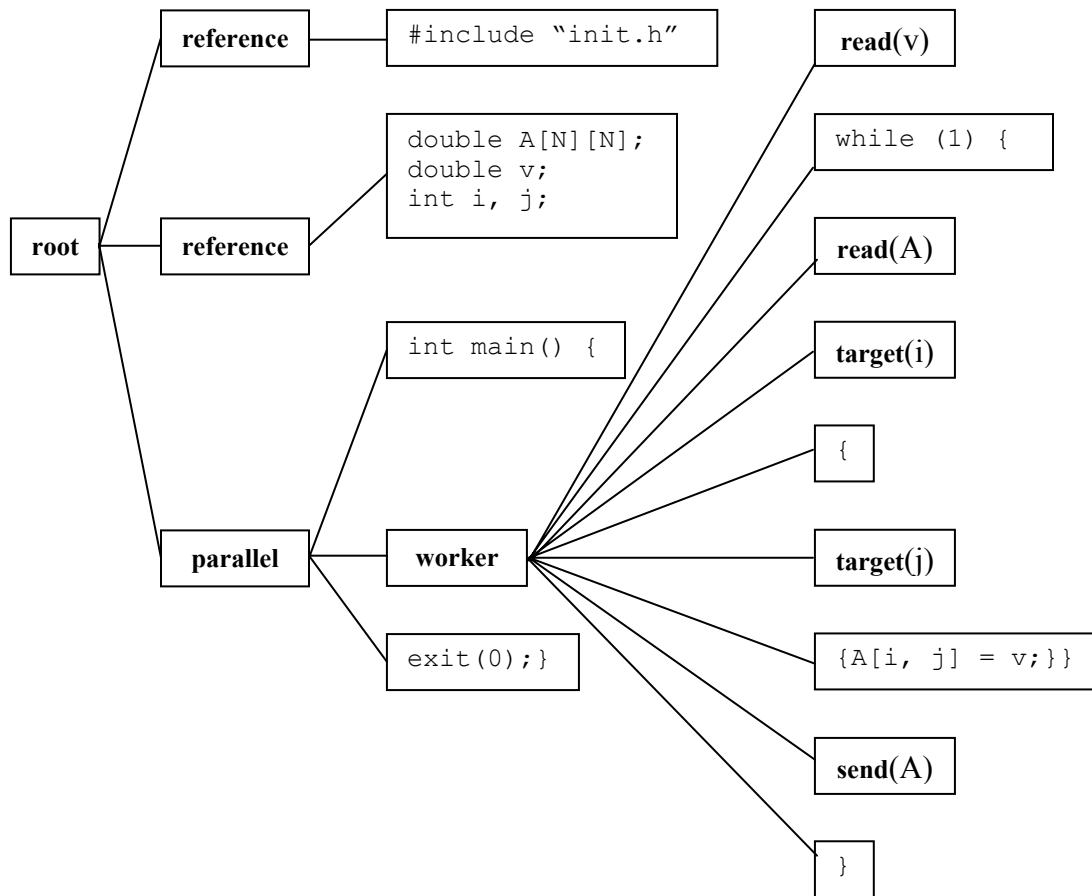


Figure 4.4 The Tag-tree Of The Worker Program

4.3 Code Generation

The parallel code generator (PCG) produces parallel master and worker programs by traversing the corresponding master and worker trees.

For *text* nodes, PCG simply outputs the untagged code. For *parallel* nodes, it first creates

declarations for PML compiler-created variables. For *master* and *worker* nodes, it generates statements for opening, cleaning up (master node only), and closing tuple spaces. For data *communication* nodes, it inserts proper statements for tuple operations. For *target* nodes, it copies original loop statements but replaces the boundary expressions with PML compiler-created variables.

It is worth mentioning that PCG relies on an extended tuple space support in order to simplify generated parallel programs and improve performance (Chapter 6). The detailed structure and workflow of the generated master and worker programs, as well as the extended tuple space support, will be described in the next chapter using the same two-dimensional example presented in Section 3.3.2.

4.4 Additional Activities

Strictly speaking, the PML-tagged sequential program is not a complete XML document. Additional XML tags, such as the PML header tag, need to be inserted into the document before sending it to the parser. These additional tags are transparent to user but are essential for the parser to validate and parse the document.

Another special processing is to deal with XML-sensitive characters, such as “&”, “<”, “>”, etc., in the source code. These characters have special meanings and must be converted according to XML guideline. At parallel code generation phase, of course, these characters must be converted back.

4.5 Summary

This chapter has presented the structure of the PML compiler and the details of its major phases. The key design philosophy is to maintain the high portability and extensibility of PML. The current implementation of meta-constructs (such as tuple space operations) uses the C programming language. This approach allows processing sequential programs in all languages with C interface. Languages without C interface will need a different PML code generator.

CHAPTER 5. GENERATED PARALLEL PROGRAMS

The PML compiler generates two types of parallel programs for each PML-tagged sequential program: *Master* and *Worker*. Both programs contain properly mixed sequential source code and PML compiler-created code. The PML compiler-created code is responsible for task distribution and data communication through tuple spaces.

In this chapter, we present the structures of the generated parallel programs in detail. Further, we describe their workflow at runtime.

5.1 Generated Master Program

The generated master program contains the original sequential source code with the following changes:

1. The sections wrapped in `<worker>` and `</worker>` tags are removed;
2. Declarations of PML compiler-created variables are inserted;
3. Initialization of parallel processing environment is inserted;
4. Code for sending a token is inserted;
5. Code for each data communication tag inside the `<master>` and `</master>` tags is inserted;
6. Termination of parallel processing environment is inserted.

The structure of master program is shown in Figure 5.1.

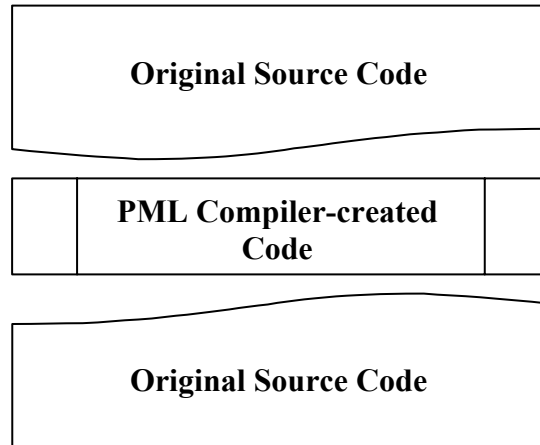


Figure 5.1 **The Structure of Master Program**

A typical flow-chart of master program is as follows,

1. Start and initialize “distributor” and “constructor” tuple spaces;
2. Send a token to “distributor” using the information contained in *target* tags;
3. Send data to “distributor” using the information in *send* tags;
4. Read results from “constructor” using the information in *read* tags;
5. Stop “distributor” and “constructor” tuple spaces.

The token sent by the master contains the meta information about the loop-block to be partitioned (i.e. the total workload). It includes the loop index variable(s), the loop boundaries, the grain size(s) and the partitioning order (in case of partitioning multiple-nested loops). The PML compiler obtains the meta information from *target* tags and generates the code for sending a token automatically.

5.1.1 A Master Program Example

Following is the master program generated from the two-dimensional partition example in Section 3.3.2. The sections in *Italic* format are PML compiler-created code.

```
#include "parallel.h"
/* <reference> */
/* Constant, N, and the grain size, G, are defined in the header file */
#include "init.h"
/* </reference> */

/* <parallel appname="init3"> */
double * _tp_v_123;
double * _tp_A_123;
int _x0_123;
int _x1_123;

int main()
{
    /* <reference> */
    double A[N][N];
    double v;
    int i, j;
    /* </reference> */

    v = 99.9;

    /* <master id="123"> */
    _distributor = _open_space("distributor", 0, "123");
    _constructor = _open_space("constructor", 0, "123");
    _cleanup_space(_distributor, "123");
    _cleanup_space(_constructor, "123");

    /* <send var="v" type="double" opt="ONCE"/> */
    sprintf(_tp_name, "double:v#%s", "123");
    _tp_size = sizeof(double);
    _tp_v_123 = &v;
    _status =
    _send_data(_distributor, _tp_name, (char *)_tp_v_123, _tp_size);
    if (_status < 0) exit(-1);

    /* <token action="SET" idxset="(i)(j)"/> */
    sprintf(_tp_name, "token#%s", "123");
    sprintf(_tp_token, "!(i:%d~%d,%d:#%d)(j:%d~%d,%d:#%d)",
            0, N, 1, G, 0, N, 1, G);

```

Declarations of PML compiler-generated variables

Initialization of parallel processing environment

```

    _tp_size = sizeof(_tp_token);
    _tokens =
    _set_token(_distributor, _tp_name, (char *)_tp_token, _tp_size);
    if (_tokens < 0) exit(-1);

    /* <send var="A" type="double[N][N]"/> */
    sprintf(_tp_name, "double(%d)(%d):A#%s", (N), (N), "123");
    _tp_size = ((N) * (N)) * sizeof(double);
    _tp_A_123 = (double *)malloc(_tp_size);
    for (_x0_123 = 0; _x0_123 < (N); _x0_123 +=1) {
        for (_x1_123 = 0; _x1_123 < (N); _x1_123 +=1) {
            _tp_A_123[_x0_123 * (N) + _x1_123] = A[_x0_123][_x1_123];
        }
    }
    _status =
    _send_data(_distributor, _tp_name, (char *)_tp_A_123, _tp_size);
    if (_status < 0) exit(-1);
    free(_tp_A_123);

    /* <read var="A" type="double[N][N]"/> */
    sprintf(_tp_name, "double(%d)(%d):A#%s", (N), (N), "123");
    _tp_size = ((N) * (N)) * sizeof(double);
    _tp_A_123 = (double *)malloc(_tp_size);
    _tp_size =
    _read_data(_constructor, _tp_name, (char *)_tp_A_123, _tp_size);
    if (_tp_size < 0) exit(-1);
    for (_x0_123 = 0; _x0_123 < (N); _x0_123 +=1) {
        for (_x1_123 = 0; _x1_123 < (N); _x1_123 +=1) {
            A[_x0_123][_x1_123] = _tp_A_123[_x0_123 * (N) + _x1_123];
        }
    }
    free(_tp_A_123);

    _close_space(_constructor, "123", 1);
    _close_space(_distributor, "123", 1);
    /* </master> */
}
/* </parallel> */

```

Termination of
parallel processing
environment

5.2 Generated Worker Program

The generated worker program essentially contains the pieces of original source code wrapped in <worker> and </worker> tags. The following is also added:

1. The sections wrapped in <reference> and </reference> tags;
2. Variable declarations for those generated by the PML compiler;

3. Code for opening and closing tuple spaces;
4. Code implementing each data communication tag inside the <worker> and </worker> tags;
5. Code for receiving tokens.

The structure of worker program is shown in Figure 5.2.

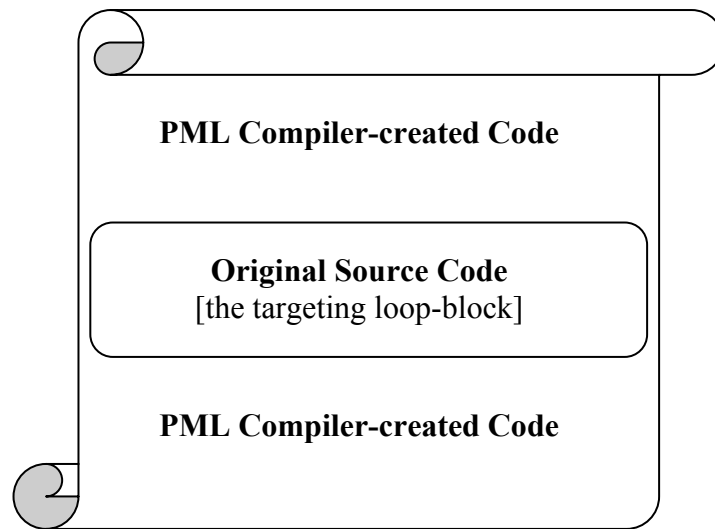


Figure 5.2 **The Structure of Worker Program**

A typical flow-chart of worker program is as follows:

1. Open “distributor” and “constructor” tuple spaces;
2. Read data from “distributor” using the information in *read* tags where the attribute ‘opt’ = ”ONCE” (i.e. data initialization);
3. Read a token from “distributor”;
4. If the token is an *exit* token, then jump to step-9;

5. Read data from “distributor” using the information in *read* tags;
6. Run the targeting loop-block using the information in the token;
7. Send results to “constructor” using the information in *send* tags;
8. Back to step-3;
9. Close “distributor” and “constructor” tuple spaces.

Each token received by a worker contains the meta information about a piece of the partitioned loop-block (i.e. a task). It includes the loop index variable(s), and the boundaries for the piece. The boundary information represents the workload instruction for a receiving worker. It is used by the worker for reading data and sending results. The PML compiler inserts the code for receiving tokens automatically.

5.2.1 A Worker Program Example

Following is the worker program generated from the two-dimensional partition example in Section 3.3.2. The sections in *Italic* format are PML compiler-created code.

```
#include "parallel.h"
/* <reference> */
/* Constant, N, and the grain size, G, are defined in the header file */
#include "init.h"
/* </reference> */

/* <reference> */
double A[N][N];
double v;
int i, j;
/* </reference> */

/* <parallel appname="init3"> */
double * _tp_v_123;
```

```

double *_tp_A_123;
int _x0_123;
int _y0_123;
int _x1_123;
int _y1_123;
int _i_start = 0;
int _i_stop = 0;
int _i_step = 0;
int _j_start = 0;
int _j_stop = 0;
int _j_step = 0;

```

Variable declarations for those generated by the PML compiler

```

main(int argc, char **argv[])
{

```

```

    /* <worker id="123"> */

```

```

    _distributor = _open_space("distributor", 0, "123");
    _constructor = _open_space("constructor", 0, "123");

```

```

    /* <read var="v" type="double" opt="ONCE"/> */

```

```

    sprintf(_tp_name, "double:v#%s", "123");
    _tp_size = sizeof(double);
    _tp_v_123 = &v;
    _tp_size =
    _read_data(_distributor, _tp_name, (char *)_tp_v_123, _tp_size);
    if (_tp_size < 0) exit(-1);

```

```

while (1)
{

```

```


```

```

    /* <token action="GET" idxset="(i) (j)"/> */

```

```

    sprintf(_tp_name, "token#%s", "123");
    _tp_size = 0;
    _tp_size =
    _get_token(_distributor, _tp_name, (char *)_tp_token, _tp_size);
    if (_tp_size < 0) exit(-1);
    if (_tp_token[0] == '!') break;
    sscanf(_tp_token, "%d@(i:%d~%d,%d) (j:%d~%d,%d)",
           &tokens,
           &_i_start, &_i_stop, &_i_step,
           &_j_start, &_j_stop, &_j_step);

```

```

    /* <read var="A" type="double[N(i)][N(j)]"/> */

```

```

    sprintf(_tp_name, "double(%d) (%d):A#%s[%d~%d,%d] [%d~%d,%d]@%d",
           (N), (N), "123",
           _i_start, _i_stop, 1, _j_start, _j_stop, 1,
           sizeof(double));
    _tp_size =
    (((_i_stop - _i_start - 1) / 1 + 1) *
     ((_j_stop - _j_start - 1) / 1 + 1)) * sizeof(double);
    _tp_A_123 = (double *)malloc(_tp_size);
    _tp_size =
    _read_data(_distributor, _tp_name, (char *)_tp_A_123, _tp_size);
    if (_tp_size < 0) exit(-1);
    for (_x0_123 = _i_start, _y0_123 = 0; _x0_123 < _i_stop;
         _x0_123 += 1, _y0_123 += 1) {
        for (_x1_123 = _j_start, _y1_123 = 0; _x1_123 < _j_stop;

```

```

        _x1_123 +=1, _y1_123 ++) {
            A[_x0_123][_x1_123] =
                _tp_A_123[_y0_123 * ((_j_stop - _j_start - 1) / 1 + 1) +
                _y1_123];
        }
    }
    free(_tp_A_123);

```

```

/*<target index="i" order="1" limits="(0,N,1)" chunk="G">*/
for (i = _i_start; i < _i_stop; i += _i_step)
/*</target>*/
{
    /*<target index="j" order="2" limits="(0,N,1)" chunk="G">*/
    for (j = _j_start; j < _j_stop; j += _j_step)
    /*</target>*/
    {
        A[i, j] = v;
    }
}

```

Run the targeting loop-
block using the information
in the token

```

/* <send var="A" type="double[N(i)][N(j)]" /> */
sprintf(_tp_name, "double(%d) (%d) :A#%s[%d~%d,%d] [%d~%d,%d]@%d",
        (N), (N), "123",
        _i_start, _i_stop, 1,
        _j_start, _j_stop, 1,
        sizeof(double));
_tp_size =
    ((_i_stop - _i_start - 1) / 1 + 1) *
    ((_j_stop - _j_start - 1) / 1 + 1) * sizeof(double);
_tp_A_123 = (double *)malloc(_tp_size);
for (_x0_123 = _i_start, _y0_123 =0; _x0_123 < _i_stop;
    _x0_123 +=1, _y0_123 ++) {
    for (_x1_123 = _j_start, _y1_123 =0; _x1_123 < _j_stop;
        _x1_123 +=1, _y1_123 ++) {
        _tp_A_123[_y0_123 * ((_j_stop - _j_start - 1) / 1 + 1) +
        _y1_123] =
            A[_x0_123][_x1_123];
    }
}
_status =
_send_data(_constructor, _tp_name, (char *)_tp_A_123, _tp_size);
if (_status < 0) exit(-1);
free(_tp_A_123);
}

```

```

    close_space(_constructor, "123", 0);
    close_space(_distributor, "123", 0);
/* </worker> */
exit(0);
}

```

Closing tuple spaces

```

/* </parallel> */

```

5.3 The Workflow At Runtime

Figure 5.3 describes how the example master and worker programs cooperate through tuple spaces at runtime.

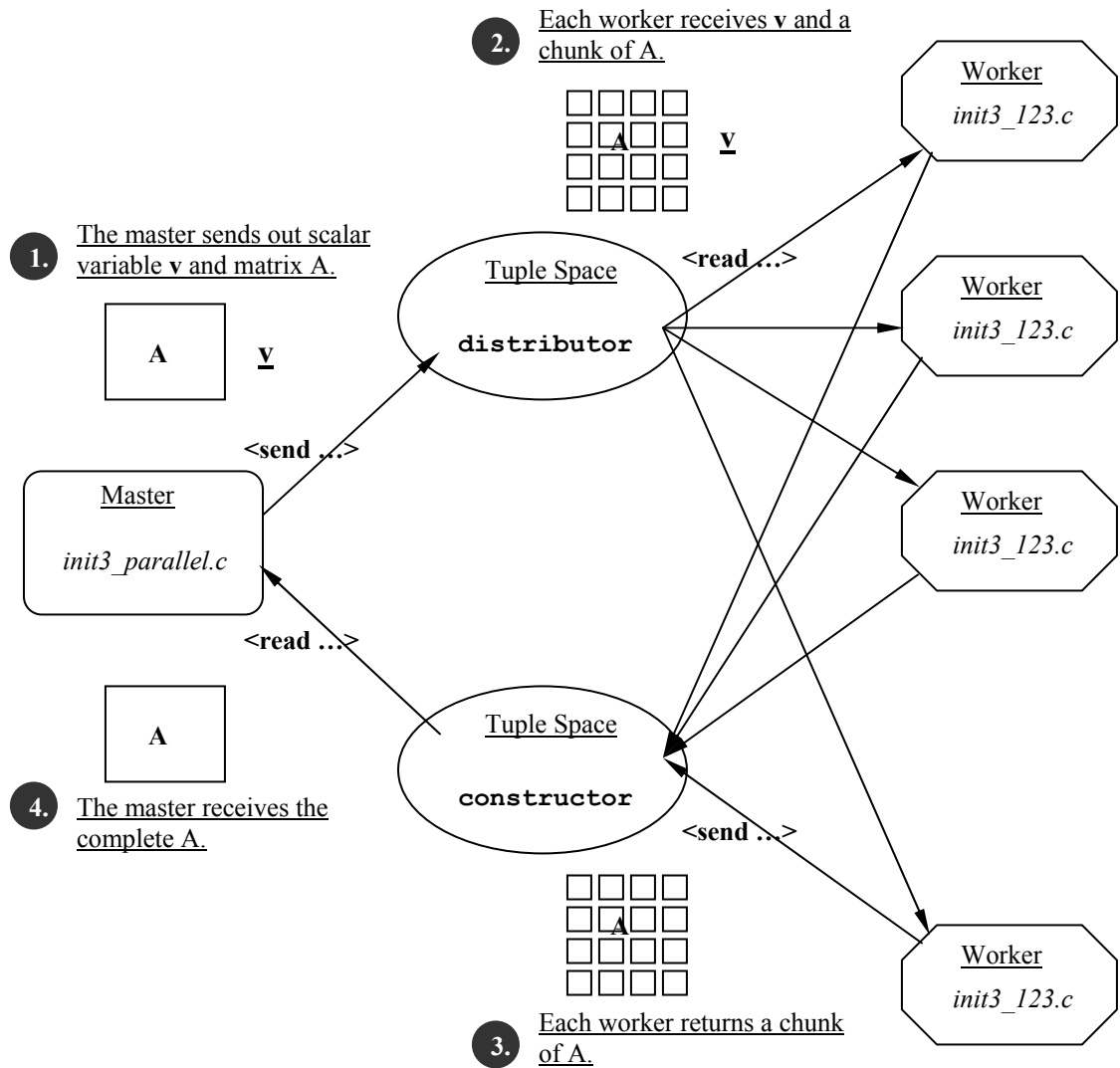


Figure 5.3 Workflow For The Example Programs

The master first sends the matrix A to “distributor” tuple space, and then reads the updated matrix A from “constructor” tuple space. Note that neither data partition nor result assembly does the master. On the other side, workers first read chunks of matrix A from “distributor”, initialize them, and then send updated chunks of matrix A to “constructor”, using their boundary information.

5.4 Summary

This chapter has presented the structures of the generated master and worker programs in detail. It has also described their workflow at runtime. Both programs are combinations of sequential source code and PML compiler-created code. Token distribution and data communication are implemented through tuple spaces by the PML compiler-created code. Workload partition, data partition and result assembly are not performed in the generated parallel programs. As we will describe in the next chapter, they are handled by the enhanced tuple spaces.

CHAPTER 6. EXTENDED TUPLE SPACE SUPPORTS

Tuple space is the central mechanism for inter-process communication. In order to simplify the structures of the generated master and worker programs, the following duties have been shifted from the parallel programs into tuple space.

- Workload partition
- Data partition
- Result assembly
- Scalar manipulation.

As a result, tuple space has been enhanced with more meta-operations. In this chapter, we describe the extended tuple space supports in detail.

6.1 Workload Partition

In the generated parallel programs, as we presented in previous chapters, a master sends to tuple space a token describing the total workload and the grain size, while from tuple space workers receive tokens, each of which specifies a task (i.e. a piece of the total workload). The actual workload partition is performed by tuple space.

Tuple space first parses the information inside the token sent by the master for the total workload, the grain size and the partitioning order (in case of partitioning multiple-nested

loops). Then, it partitions the total workload by the grain size into tasks, and creates a group of new tokens, one for each task. The new tokens will be distributed to workers. The task in each new token represents the workload instruction for a receiving worker. Note that in case of partitioning multiple-nested loops, the new tokens are created and distributed in the partitioning order.

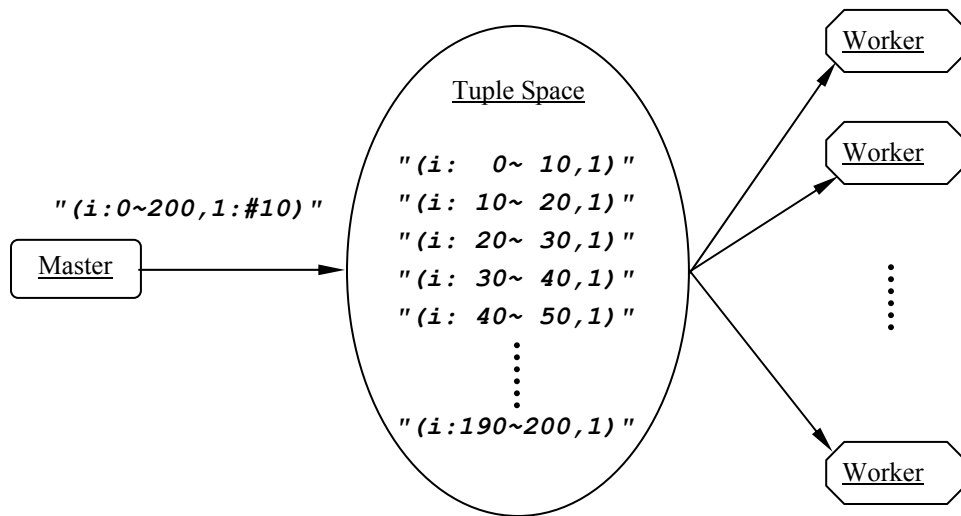


Figure 6.1 An Example of Workload Partition in Tuple Space

Figure 6.1 shows an example of workload partition in tuple space. The token sent by the master has the format $(loop_index:start\sim stop,step:\#grain_size)$, where $start$, $stop$ and $step$ are the loop boundaries. The token describes the total workload as a loop of total 200 iterations and the grain size as 10. Tuple space partitions the loop into twenty tasks, each of which is a piece of 10 iterations. Twenty new tokens are created for workers. Their format is $(loop_index:start\sim stop,step)$, where the $start$, $stop$ and $step$ are the boundaries for the piece.

6.2 Data Partition And Result Assembly

In the parallel programs' workflow, as we presented in previous chapters, the master first sends an matrix to "distributor" tuple space; workers read chunks of the matrix from "distributor", process them in parallel, and send updated chunks of the matrix to "constructor" tuple space; the master then read an updated matrix from "constructor". The actual data partition and result assembly is performed by tuple space.

When master or worker *reads* an entire matrix, tuple space seeks all tuples containing either the entire matrix or relevant sub-range data. If found a tuple containing the entire matrix, tuple space returns it immediately. Otherwise, as shown in Figure 6.2, tuple space will exactly place relevant sub-range data into the matrix, and returns it when it is complete.

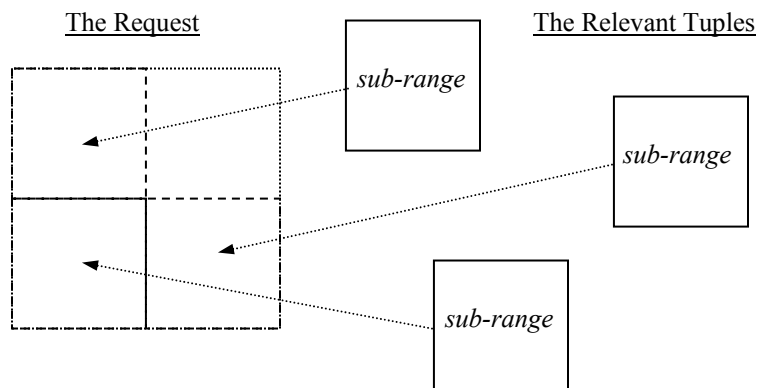


Figure 6.2 **Read Entire Matrix**

When master or worker *sends* an entire matrix, tuple space seeks all requests for either the entire matrix or relevant sub-range data. For requests for the entire matrix, tuple space fulfils them immediately. For others, as shown in Figure 6.3, tuple space will exactly extract relevant sub-range data from the matrix, and fulfils them.

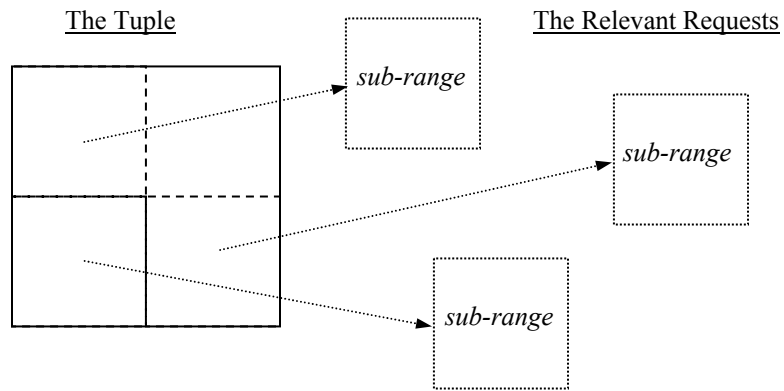


Figure 6.3 **Send Entire Matrix**

When master or worker *reads* a sub-range of a matrix, tuple space first seeks any tuple containing either the entire matrix or a larger sub-range data covering the smaller sub-range data being requested. If such tuple exists, as shown in Figure 6.4, tuple space will exactly extract the sub-range data from the tuple, and returns it.

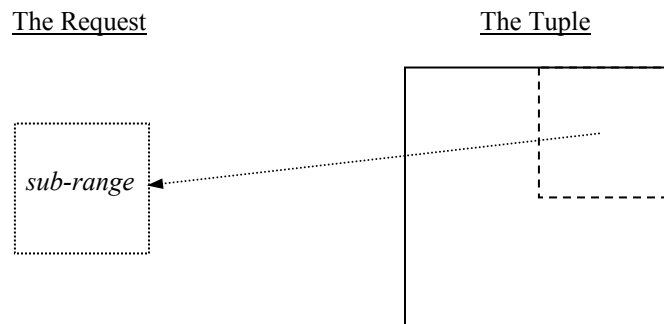


Figure 6.4 **Read A Sub-range Of Matrix**

When master or worker *sends* a sub-range of a matrix, tuple space first seeks any request for either the entire matrix or a larger sub-range data covering the smaller sub-range data being received. If such requests exist, as shown in Figure 6.5, tuple space will exactly place the sub-range data into the requests, and fulfils them when they are complete.

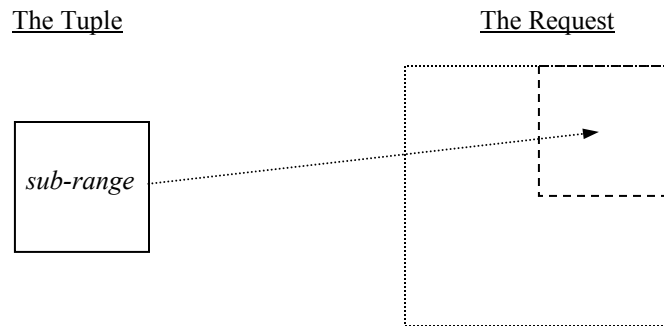


Figure 6.5 **Send A Sub-range Of Matrix**

Shifting the duties of data partition and result assembly from the parallel programs into tuple space not only simplifies the structures of the generated parallel programs, but also improves parallel processing performance by reducing the number of communication sessions. Assuming a matrix is partitioned into 100 pieces for parallel processing, for example, if data partition and result assembly stay in the master program, then at the beginning the master needs 100 communication sessions, one for sending each piece, and at the end it needs another 100 communication sessions, one for receiving each result. With the enhancement, however, the master needs only two communication sessions, the first for sending the entire matrix and the second for receiving the updated matrix.

6.3 Scalar Manipulation

Tuple space matches tuples with their names. By default, an old tuple gets overwritten by a new tuple with the same name. In order to deal with complex scalar manipulations in some applications, such as Solving Laplace's Equation with Gauss-Seidel Iteration, two additional functions, "max" and "min", are implemented in the enhanced tuple space for updating tuples containing scalar variables.

Other than the default "overwrite" semantics, "max" is to keep the maximum value for a tuple being updated, while "min" is to keep the minimum value. The two semantics can be activated per "send" or "read" operation. As we presented in Section 3.1.2, the optional attribute "opt" in <send> and <read> tags has options, "_MAX" and "_MIN", which are used for the activation of the two semantics respectively.

6.4 The API Calls

The API calls for the enhancements are shown below:

```
int _open_space(char *space_name, char *mode, char *workerId);
int _close_space(int id, char *workerId, int cleanup);
int _cleanup_space(int id, char *workerId);
int _set_token(int id, char *tpname, char *tpvalue, int tpsize);
int _get_token(int id, char *tpname, char *tpvalue, int tpsize);
int _send_data(int id, char *tpname, char *tpvalue, int tpsize);
int _read_data(int id, char *tpname, char *tpvalue, int tpsize);
```

`_set_token` and `_get_token` instruct tuple space for workload partition. Data partition, result assembly, and scalar manipulation are specified through `_send_data` and `_read_data`. While `_get_token` removes tokens in tuple space, `_read_data` does not delete tuples. Lastly, `_cleanup_space` deletes all tuples matching `workerId`.

6.5 Summary

This chapter has described extended tuple space supports in detail. With the enhancements in tuple space, the structures of the generated parallel programs are greatly simplified. Further, the parallel processing performance is also improved, due to reducing the number of communication sessions.

CHAPTER 7. EXPERIMENTAL RESULTS

The four chosen applications are: matrix multiplication, Laplacian Solver using Gauss-Seidel iteration, Ion Generation Simulator and Block LU Factorization. They represent a wide-range of numerical programs of typical dependency patterns.

7.1 Matrix Multiplication

Matrix multiplication is a very useful operation in mathematics. The product C of two matrices, A and B , is defined by

$$\begin{pmatrix} c_{11} & c_{12} & \dots & c_{1p} \\ c_{21} & c_{22} & \dots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n1} & \dots & c_{np} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n1} & \dots & a_{nm} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1p} \\ b_{21} & b_{22} & \dots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m1} & \dots & b_{mp} \end{pmatrix}$$

where $c_{ik} = a_{ij}b_{jk}$ and j is summed over for possible values of i and k . The rows of A must have the same length as the columns of B . Otherwise, the product is undefined.

7.1.1 The Sequential Solution

For simplicity, the application uses a simple three-nested loop implementation. The sequential program for Matrix Multiplication is shown below. The loop block in **Bold** faced segments will be marked as the target for parallelization.

```

/* =====
 * Sequential Matrix Multiplication, "matrix_seq.c".
 */
#include "matrix.h"

main(int argc, char **argv[])
{
    int i, j, k;

    for (i = 0; i < N ; i++)
    {
        for (j = 0; j < N; j++)
        {
            A[i][j] = (double) i * j ;
            B[i][j] = (double) i * j ;
            C[i][j] = 0;
        }
    }

    for (i = 0; i < N; i++)
    {
        for (k = 0; k < N; k++)
        {
            for (j = 0; j < N; j++)
            {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }

    /*
    for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
        {
            printf("%8.1f ", C[i][j]);
        }
        printf("\n");
    }
    */
    exit(0);
}

/* =====
 * The Header File, "matrix.h".
 */
#include <stdio.h>

#define N 100
#define G 20

double A[N][N], B[N][N], C[N][N];

```

7.1.2 PML-tagged Sequential Program

Below is the PML-tagged sequential program for Matrix Multiplication. The generated parallel programs are shown in Appendix B.

```
/* =====
 * Sequential Matrix Multiplication, "matrix_seq.c".
 */
/* <reference> */
#include "matrix.h"
/* </reference> */

/* <parallel appname="matrix"> */
main(int argc, char **argv[])
{
    /* <reference id="123"> */
    int i, j, k;
    /* </reference> */
    for (i = 0; i < N ; i++)
    {
        for (j = 0; j < N; j++)
        {
            A[i][j] = (double) i * j ;
            B[i][j] = (double) i * j ;
            C[i][j] = 0;
        }
    }

    /* <master id="123"> */
    /* <send var="B" type="double[N][N]" opt="ONCE"/> */
    /* <send var="A" type="double[N][N]"/>*/

    /* <worker> */
    /* <read var="B" type="double[N][N]" opt="ONCE"/> */
    /* <read var="A" type="double[N(i)][N]"/> */

    /* <target index="i" limits="(0,N,1)" chunk="G" order="1"> */
    for (i = 0; i < N; i++)
    /* </target> */
    {
        for (k = 0; k < N; k++)
        {
            for (j = 0; j < N; j++)
            {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}
```

```
/* <send var="C" type="double[N(i)][N]" /> */
/* </worker> */

/* <read var="C" type="double[N][N]" /> */
/* </master> */

for (i = 0; i < N; i++)
{
    for (j = 0; j < N; j++)
    {
        printf("%8.1f ", C[i][j]);
    }
    printf("\n");
}
exit(0);
}
/* </parallel> */
```

7.1.3 Parallel Program Workflow

Figure 7.1 shows the runtime workflow of the generated parallel programs for Matrix Multiplication.

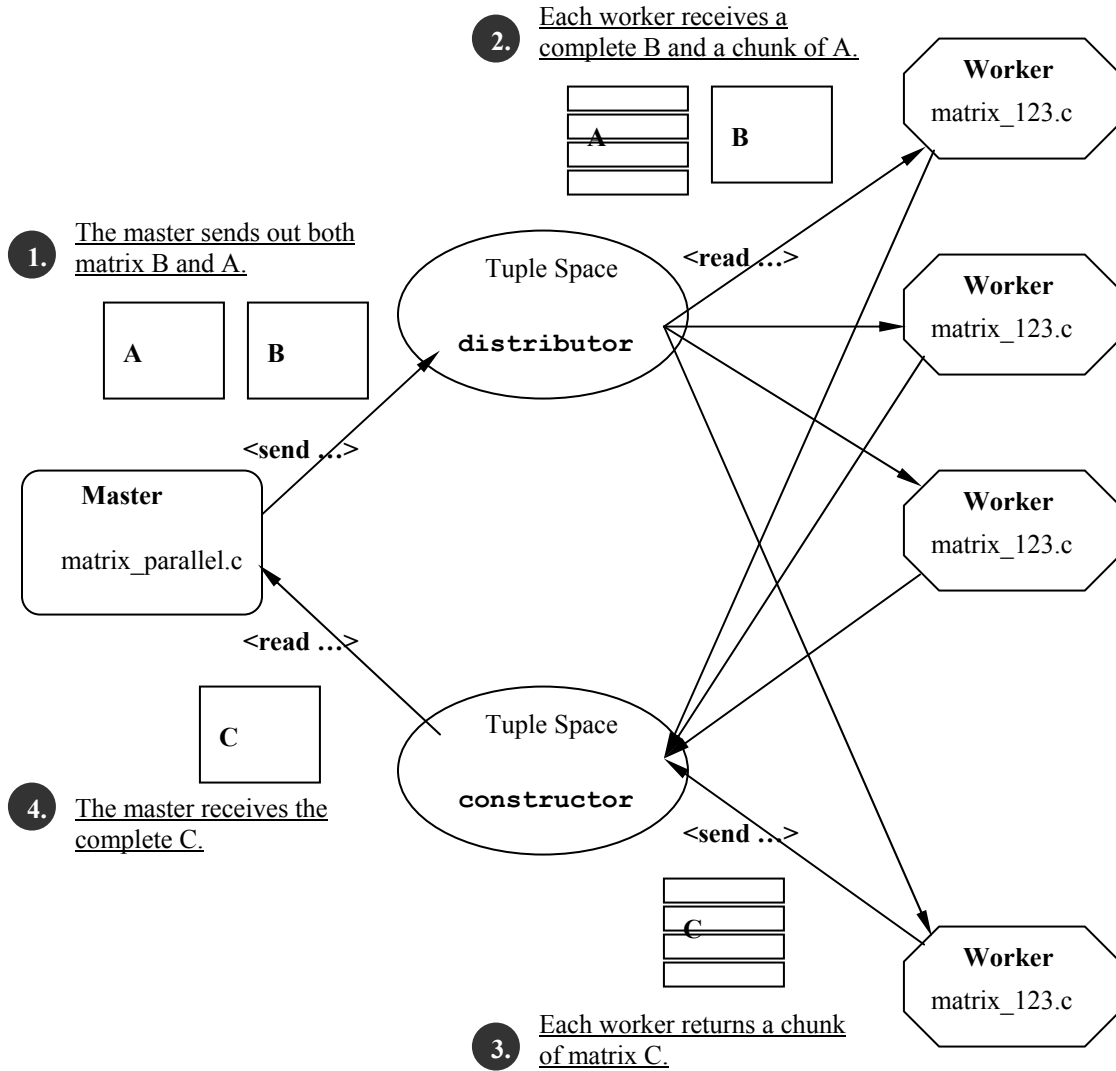


Figure 7.1 Workflow For Matrix Multiplication

7.1.4 Parallel Processing Performance

Table 7.1 lists the performance testing results of the generated parallel programs for Matrix Multiplication.

Table 7.1 **The Performance Comparison For Matrix Multiplication**

<u>Processors</u>	<u>Matrix Size</u>	<u>Automatic Generated (seconds)</u>	<u>Manually Created (seconds)</u>
2	600 X 600	18	20
2	800 X 800	28	35
2	1000 X 1000	58	67
2	1600 X 1600	162	218
2	2000 X 2000	322	407
4	600 X 600	21	24
4	800 X 800	33	45
4	1000 X 1000	64	60
4	1600 X 1600	160	178
4	2000 X 2000	302	310

Due to the extended tuple space supports, the automatically generated parallel programs out-performed its manually crafted counterpart.

7.2 Laplacian Solver Using Gauss-Seidel Iteration

Laplace's equation in two dimensions is:

$$\partial^2 f / \partial x^2 + \partial^2 f / \partial y^2 = 0$$

where x and y represent coordinates in space, and f is the value we're computing. If we approximate the second derivatives in X and Y using a difference method, we obtain:

$$f_{[x, y]} = (f_{[x-1, y]} + f_{[x+1, y]} + f_{[x, y-1]} + f_{[x, y+1]}) / 4$$

This formulation gives us a way to solve the equation numerically: given any initial guess F_0 , we can calculate a new guess F_1 by using F_0 on the right hand side of the equation above. We can then use the calculated F_1 to calculate a new guess F_2 , and so on. Unfortunately, as the iterative process is carried out, values change, so that it must be repeated to converge on a final solution.

7.2.1 The Sequential Solution

The equation is illustrated in Figure 7.2 as a two-dimensional grid of data values [29]. The values along the outer edges are kept constant and known as the boundary values or boundary conditions; these edge values will be the input to the calculation. All the interior values are set in such a way that each value is the average of its neighbors; these interior values will be the output. The neighbors of a value is the ones immediately above, below, left and right of it, which defines what's called a 4-point stencil, also illustrated in Figure 7.2: the four points shown contribute to the average.

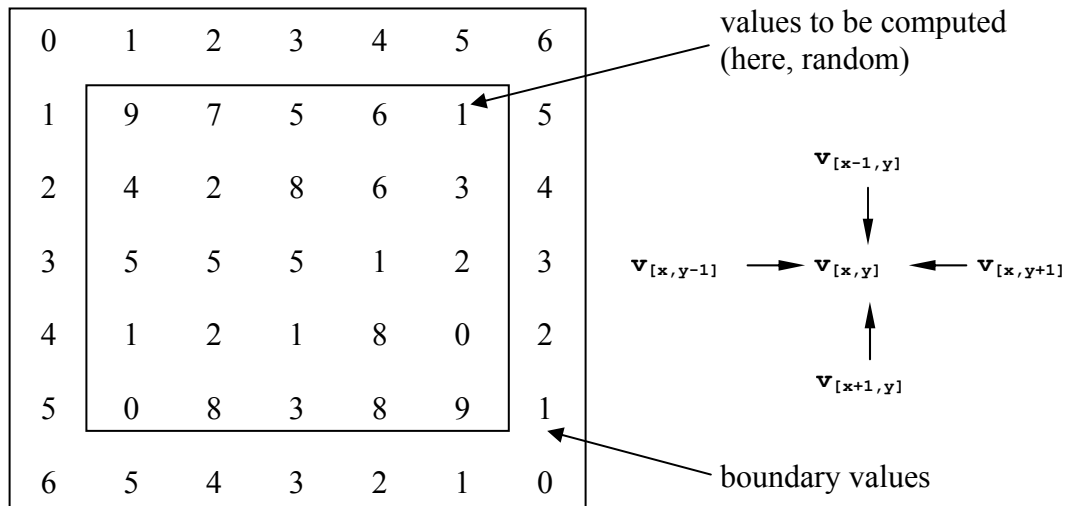


Figure 7.2 A 2-D Grid of Values and a 4-Point Stencil

Using finite differences, Gauss-Seidel iterative solver repeatedly iterates calculating each interior value until the solution differences between two iterations are less than a pre-set value. In other words, the calculation stops when every calculated value, every value on the interior of the grid, is close enough to the average of its neighbors. A value for “close_enough” is another input.

The sequential program for Laplacian Solver using Gauss-Seidel Iteration [6] is shown below. The loop block in **Bold** faced segments will be marked as the target for parallelization.

```

/* =====
 * Sequential Laplacian Solver/Gauss-Seidel Iteration, "gauss_seq.c".
 */
#include "gauss.h"

```

```

int main()
{
    rdim = DIM;
    inita();

    for (iterno= 1; iterno <= MAXITER; iterno++)
    {
        resid = 0.0;

        for (i = 1; i < rdim+1; i++)
        {
            for (j = 1; j < DIM+1; j++)
            {
                Atmp      = A[i][j];
                A[i][j]   = 0.25 * (A[i+1][j] + A[i-1][j] +
                                   A[i][j+1] + A[i][j-1]);

                residtmp =
                (Atmp!=0.0? fabs(fabs(A[i][j]-Atmp)/Atmp) : (1.0+EPS));
                if (residtmp > resid) resid = residtmp;
            }
        }

        if (resid < EPS) break;      /* termination of iterations */
    }

    /*
for (i = 0; i < DIM+2; i++)
{
    for (j = 0; j < DIM+2; j++)
    {
        printf("%8.1f ", A[i][j]);
    }
    printf("\n");
}
*/
return 0;
}

void inita()
{
    for (i = 0; i < rdim+2; i++) /* initial guess */
        for (j = 0; j < DIM+2; j++)
            A[i][j] = 0.0;
    for (i = 0; i < rdim+2; i++) /* set boundaries */
    {
        A[i][0]      = LFTBOUND;
        A[i][DIM+1] = RITBOUND;
    }
    for (j = 0; j < DIM+2; j++)
        A[0][j]      = TOPBOUND;
    for (j = 0; j < DIM+2; j++)
        A[rdim+1][j] = BOTBOUND;
}

```

```

/* =====
 * The Header File, "gauss.h".
 */
#include <stdio.h>
#include <math.h>

#define DIM          3          /* problem domain size in each direction*/
#define MAXITER     2          /* maximum permitted iterations */
#define EPS         0.1        /* maximum allowed relative precision */
#define TOPBOUND    200.0      /* Top B.C. */
#define BOTBOUND    0.0        /* Bot B.C. */
#define LFTBOUND    0.0        /* Lft B.C. */
#define RITBOUND    0.0        /* Rit B.C. */
#define DEBUG       1          /* For initial debugging 1, 0 o.w. */
#define LOG         0          /* 1 to output data to profile, 0 o.w.*/
#define GRAIN       2

int    rdim;                /* row dimension of the domain */
double A[DIM+2][DIM+2];    /* maxm. size of the domain with boundaryd */
int    rowsize,            /* number of bytes in each row */
       iterno,            /* current iteration number */
       i,                 /* loop indices */
       j;                 /* loop indices */
double resid,             /* residual */
       residtmp,         /* temp */
       Atmp;             /* temp */

void inita();              /* initilizes A with initial guess */

```

7.2.2 PML-tagged Sequential Program

Through the analysis on the targeting loop-block in above sequential code, we can obtain an extremely tied-up dependency pattern presented on the iteration matrix of the targeting loop-block, as shown in Figure 7.3. The pattern tells that the computation in iteration (i, j) is dependent on the results from two previous iterations, $(i-1, j)$ and $(i, j-1)$. With such type of dependency pattern, parallelism exists only among the iterations along the same diagonal lines in the iteration matrix, also shown in Figure 7.3.

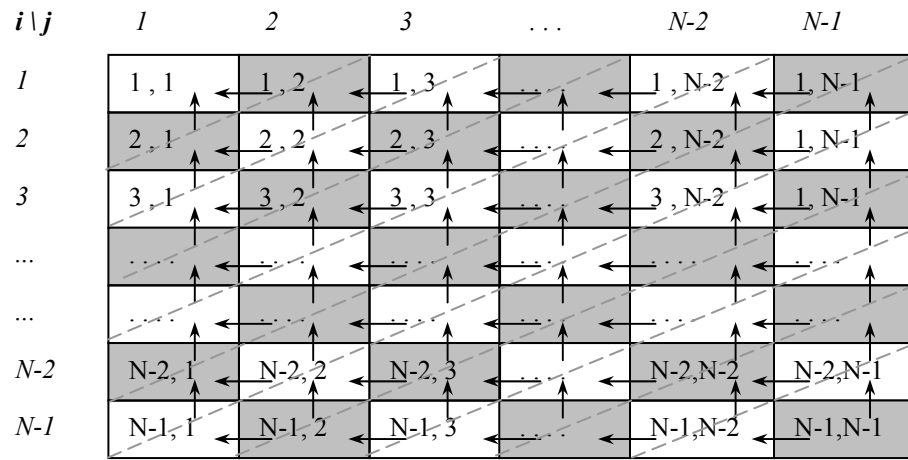


Figure 7.3 The Dependency Pattern For Laplacian Solver

In order to get exactly the same result as the original program and keep its good mathematical properties, we can run a diagonal wave of computation through the iteration matrix, i.e., doing everything on the “wavefront” in parallel [29], as shown in Figure 7.4. The wavefront method [25] is described as a hyperplane through the space of indices traversed by the loops.

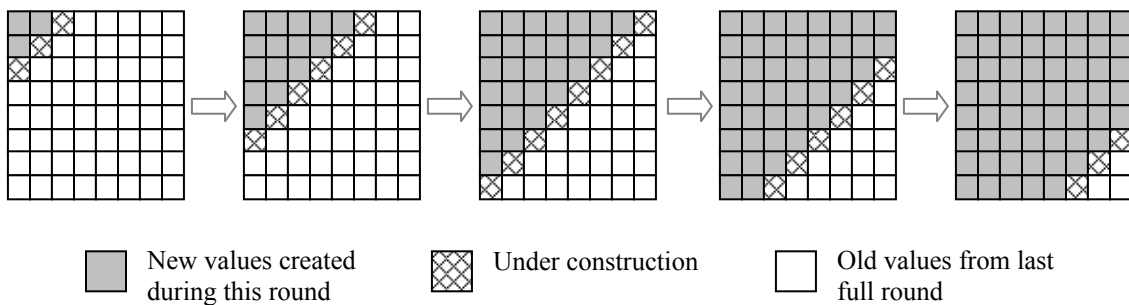


Figure 7.4 Diagonal Wavefront Of Computation

As shown in Figure 7.4, the parallelism available with wavefront method is not uniform.

It starts small, then increases to a maximum across the diagonal of the iteration matrix, then decreases again. It is in general difficult to take advantage of such irregular parallelism, even by manually written parallel programs. We show that PML is powerful enough to describe such irregular parallelism and automatically generate the parallel programs from the sequential code.

Below is the PML-tagged sequential program for Laplacian Solver using Gauss-Seidel Iteration. The generated parallel programs are shown in Appendix C.

```

/* =====
 * Sequential Laplacian Solver/Gauss-Seidel Iteration, "gauss_seq.c".
 */
/* <reference> */
#include "gauss.h"
/* </reference> */

/* <parallel appname="gauss"> */
int main()
{
    rdim = DIM;

    inita();

    /* <master id="123"> */
    for (iterno= 1; iterno <= MAXITER; iterno++)
    {
        /* <send var="A" type="double [DIM+2] [DIM+2]" opt="ONCE"/> */

        /* <send var="A"
           type="double [DIM+2 (0~1) ] [DIM+2 (0~DIM+1) ]"
           opt="XCHG"/> */
        /* <send var="A"
           type="double [DIM+2 (0~DIM+1) ] [DIM+2 (DIM+1~DIM+2) ]"
           opt="XCHG"/> */
        /* <send var="A"
           type="double [DIM+2 (DIM+1~DIM+2) ] [DIM+2 (1~DIM+2) ]"
           opt="XCHG"/> */
        /* <send var="A"
           type="double [DIM+2 (1~DIM+2) ] [DIM+2 (0~1) ]"
           opt="XCHG"/> */

        /* <worker> */
    }
}

```

```

/* <read var="A" type="double [DIM+2] [DIM+2]"/> */
/* <read var="A"
    type="double [DIM+2 (i:$L-1)] [DIM+2 (j)]"
    opt="XCHG"/> */
/* <read var="A"
    type="double [DIM+2 (i)] [DIM+2 (j:$L-1)]"
    opt="XCHG"/> */

resid = 0.0;

/* <target index="i" limits="(1,rDIM+1,1)" chunk="GRAIN"
    order="1"> */
for (i = 1; i < rDIM+1; i++)
/* </target> */
{
    /* <target index="j" limits="(1,DIM+1,1)" chunk="GRAIN"
        order="1"> */
    for (j = 1; j < DIM+1; j++)
    /* </target> */
    {
        Atmp      = A[i][j];
        A[i][j]  = 0.25 * (A[i+1][j] + A[i-1][j] +
                          A[i][j+1] + A[i][j-1]);

        residtmp =
        (Atmp!=0.0? fabs(fabs(A[i][j]-Atmp)/Atmp) : (1.0+EPS));
        if (residtmp > resid) resid = residtmp;
    }
}

/* <send var="resid" type="double" opt="_MAX"/> */
/* <send var="A" type="double [DIM+2 (i)] [DIM+2 (j)]"/> */

/* </worker> */

/* <read var="A" type="double [DIM+2] [DIM+2]"/> */
/* <read var="resid" type="double" opt="_MAX"/> */

if (resid < EPS) break;    /* termination of iterations */
}
/* </master> */

/*
for (i = 0; i < DIM+2; i++)
{
    for (j = 0; j < DIM+2; j++)
    {
        printf("%8.1f ", A[i][j]);
    }
    printf("\n");
}
*/
return 0;
}
/* </parallel> */

```

7.2.3 Parallel Program Workflow

Figure 7.5 shows the runtime workflow of the generated parallel programs for Laplacian Solver using Gauss-Seidel Iteration.

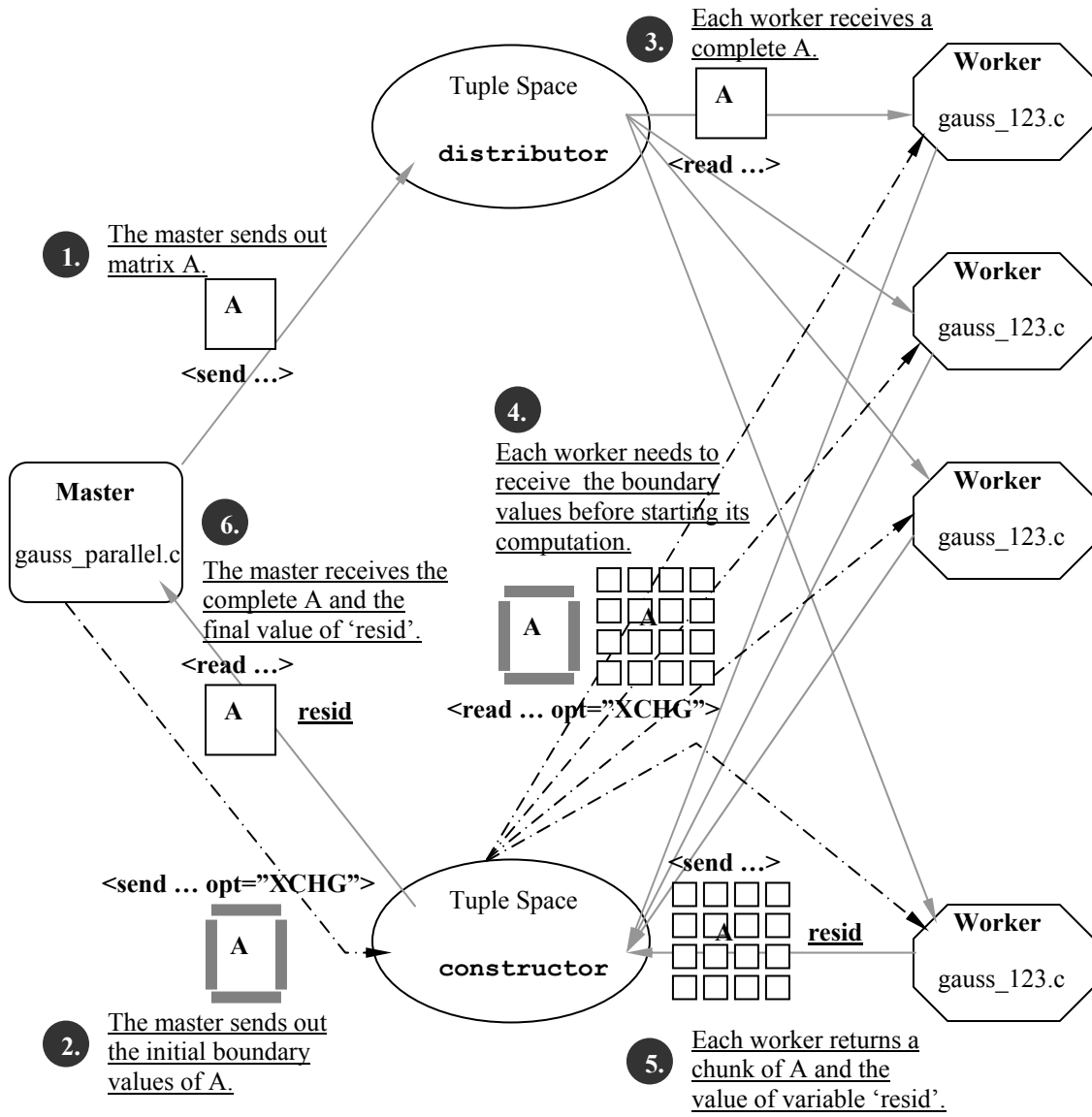


Figure 7.5 Workflow For Laplacian Solver

7.2.4 Parallel Processing Performance

This application has an $O(N^2)$ communication complexity for each $O(N^2)$ computation complexity. This means that unless the network speed can match the aggregate processing speed, no speedup is expected. The following performance figures were obtained on a cluster of four high-speed computers (2.6 GHZ P4 processors) on a slow LAN (10/100 Mbps simplex).

Table 7.2 lists the performance testing results of the generated parallel programs for Laplacian Solver using Gauss-Seidel Iteration.

Table 7.2 **The Performance Comparison For Laplacian Solver**

<u>Processors</u>	<u>Matrix Size</u>	<u>Automatic Generated (seconds)</u>	<u>Sequential Version (seconds)</u>
4	8 X 8	3	0
4	20 X 20	6	0
4	40 X 40	14	0
4	80 X 80	51	1
4	100 X 100	98	2
4	200 X 200	565	14

The following timing model [35] analysis shows that the parallel performance will only improve if the network speed is almost 10 times faster than the single processor speed using four CPU.

Suppose N is the dimension size of matrix, P is the number of processors, W is the processor capability in algorithmic steps per second, δ is the size of each matrix element in bytes, and μ is the network speed in bytes per second. Then we have:

The sequential time is $T_{seq} = \frac{N^2}{W}$, and the parallel time is $T_{par} = \frac{N^2}{PW} + \frac{\delta N^2}{\mu}$,

To make Speedup > 1 , we have: $Sp = \frac{T_{seq}}{T_{par}} = \frac{P\mu}{\varpi + \delta PW} > 1$ (3)

From equation (3), we have $\mu > \frac{\delta PW}{1 - P}$

Therefore, if P is 4 and δ is 8 bytes for *double* type, then to obtain a speedup, the network speed μ must be nearly 10 times faster than the single processor speed W .

7.3 Ion Generation Simulator

The typical technique for generating charged air molecules (ions) is to apply a high voltage (up to 10,000 volts, typically) to a very sharp needle. It produces a very high electrical field near the surface of the needle, which causes a transfer of electrons between the needle and the surrounding air, resulting in the generation of molecules containing a net positive or negative electrical charge [21].

There are two methods for the generation of ions, the A. C. method and the D. C. method. The A. C. method uses a single set of needles. Applying a high-voltage, 60 Hz A. C. to the set of needles causes the system to switch back and forth between positive and negative ion generation as the ionizing voltage reverses polarity. The D. C. method uses two sets of needles charged to equal and opposite D. C. voltages to generate the required positive and negative ions [21].

The application in this experiment is the D. C. method.

7.3.1 The Sequential Solution

Voltage in ionization space can be described using Laplace's equation,

$$\partial^2 V / \partial x^2 + \partial^2 V / \partial y^2 = 0$$

where voltage V is a function of coordinates x and y in space. For a numerical solution, the voltages in space can be represented across a matrix where the element is:

$$V_{[x,y]} = (V_{[x-1,y]} + V_{[x+1,y]} + V_{[x,y-1]} + V_{[x,y+1]}) / 4$$

As we introduced earlier, the equation can be solved by repeating the iterative process of setting each element until the matrix converge on a final solution. To avoid oscillation in the solution, the change in any step is limited to a portion of the desired value. This results in the under-relaxed recursion version:

$$V_{[x,y]} = (V_{[x-1,y]} + V_{[x+1,y]} + V_{[x,y-1]} + V_{[x,y+1]} + UV_{[x,y]}) / (4 + U)$$

The parameter U is the under-relaxation factor and is a positive real number [21].

The sequential program for Ion Generation Simulator is shown below. The loop block in **Bold** faced segments will be marked as the target for parallelization.

```

/* =====
 * Sequential Ion Generation Simulator, "ion_seq.c".
 */
#include "ion.h"

int main()
{
    int I,J,K, COUNT;
    float V[MAXI+1][MAXJ+2];
    float R1S, R2S, R3, A, B, C, URAXP4, RI, RJ;
    float SIGPX, SIGMX, SIGPY, SIGMY;

    long t0, t1;          /* start and end elapsed time */

    char FIELD[MAXI][MAXJ];
    char MARK[20];

    strcpy(MARK, "abcdefghijklmnopqrst");

    for (I=1; I<=MAXI; I++)
    {
        for (J=1; J<=MAXJ; J++)
        {
            R1S=sqrt((float)(pow((IP1-I),2)+pow((JP1-J),2)));
            R2S=sqrt((float)(pow((IP2-I),2)+pow((JP2-J),2)));
            R3=(float)(MAXI-I);
            A=R1S*R2S;
            B=R1S*R3;
        }
    }
}

```

```

        C=R2S*R3;
        V[I][J]=(VP2*B+VP1*C)/(A+B+C);
    }
}

for (J=TLFT; J<=TRGT; J++)
{
    V[TLEV][J] = 0.0;
}

t0 = time ((long*)0);

for (COUNT=1; COUNT<=ITER; COUNT++)
{
    for (J=1; J<=MAXJ; J++)
    {
        V[0][J]=V[1][J];
    }
    for (I=0; I<=MAXI-1; I++)
    {
        V[I][0]=V[I][1];
        V[I][MAXJ+1]=V[I][MAXJ];
    }

    for (I=1; I<=MAXI-1; I++)
    {
        for (J=1; J<=MAXJ; J++)
        {
            if ((I==IP1) && (J==JP1)) || ((I==IP2) && (J==JP2)) ||
                ((I==TLEV) && (J>=TLFT) && (J<=TRGT)))
                continue;

            RI=(float) (I);
            RJ=(float) (J);

            SIGPX=SIGMA_f(RI+0.5,RJ);
            SIGMX=SIGMA_f(RI-0.5,RJ);
            SIGPY=SIGMA_f(RI,RJ+0.5);
            SIGMY=SIGMA_f(RI,RJ-0.5);

            V[I][J]= (URAX*V[I][J]
                + (SIGMY*V[I][J-1]+SIGPY*V[I][J+1]
                +SIGMX*V[I-1][J]+SIGPX*V[I+1][J]))
                / (URAX+SIGPX+SIGMX+SIGPY+SIGMY);
        }
    }
}

for (I=1; I<=MAXI-1; I++)
{
    strcpy(FIELD[I-1] , " ");
    for (J=1; J<=MAXJ; J++)
    {
        for (K=1; K<=NDVP; K++)
        {

```

```

                if (abs(V[I][J]-DVP[K-1]) < abs(DVP[K-1]/20.0))
                    FIELD[I-1][J-1] = MARK[K-1];
            }
        }

FIELD[IP1-1][JP1-1]='*';
FIELD[IP2-1][JP2-1]='*';

strcpy(&FIELD[TLEV-1][TLFT-
1],"1*****");

for (J=1; J<=MAXJ; J++)
{
    FIELD[MAXI-1][J-1]='*';
}

for (I=1; I<=MAXI; I++)
{
    for (J=MAXJ; J>=2; J--)
    {
        if (FIELD[I-1][J-1] != ' ')
            break;
    }

    printf("( %s)\n", FIELD[I-1]);
}

t1 = time((long *)0) - t0;

printf ("\nTime: (%d), MAXI(J): (%d), Iters: (%d)\n",
        t1, MAXI, ITER);

printf("End of Lapsig.c\n");
exit(0);
}

float SIGMA_f(float RI,float RJ)
{
    float RIP1, RJP1, RIP2, RJP2;
    float D1,D2;
    float SIGMA;

    RIP1 = (float)IP1;
    RJP1 = (float)JP1;

    RIP2 = (float)IP2;
    RJP2 = (float)JP2;

    D1=sqrt(pow((RI-RIP1),2)+pow((RJ-RJP1),2));
    D2=sqrt(pow((RI-RIP2),2)+pow((RJ-RJP2),2));
    SIGMA=1.0/max(1.0,min(D1,D2));

    return (SIGMA);
}

```

```

}

float max(float x, float y)
{
    if (x > y) return x;
    return y;
}

float min(float x, float y)
{
    if (x < y) return x;
    return y;
}

/* =====
 * The Header File, "ion.h".
 */
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define MAXI 120
#define MAXJ 120

#define GRAIN 20

float VP1 = 10000;
int IP1 = 1, JP1 = 41;

float VP2 = -10000;
int IP2 = 1, JP2 = 80;

int ITER = 20;

int TLEV = 100, TLFT = 41, TRGT = 80;

float URAX = 4.0;

float DVP[20] = { 100, -100,
                 200, -200,
                 500, -500,
                 1000, -1000,
                 5000, -5000};

int NDVP = 10; /* = the # of elements in DVP */

float SIGMA_f(float RI, float RJ);
float max(float x, float y);
float min(float x, float y);

```

7.3.2 PML-tagged Sequential Program

Ion Generation Simulator is a type of Laplacian Solver application. As we illustrated in previous chapter, parallelism exists only among the iterations along the same diagonal lines in the iteration matrix of the targeting loop-block. We use the wavefront method [25] for parallelizing the application.

Below is the PML-tagged sequential program for Ion Generation Simulator. The generated parallel programs are shown in Appendix D.

```
/* =====  
 * Sequential Ion Generation Simulator, "ion_seq.c".  
 */  
/* <reference> */  
# include "ion.h"  
/* </reference> */  
  
/* <parallel appname="ion"> */  
int main()  
{  
    /* <reference> */  
    int I,J,K, COUNT;  
    float V[MAXI+1][MAXJ+2];  
    float R1S, R2S, R3, A, B, C, URAXP4, RI, RJ;  
    float SIGPX, SIGMX, SIGPY, SIGMY;  
    /* </reference> */  
  
    long t0, t1;          /* start and end elapsed time */  
    char FIELD[MAXI][MAXJ];  
    char MARK[20];  
    strcpy(MARK,"abcdefghijklmnopqrst");  
  
    for (I=1; I<=MAXI; I++)  
    {  
        for (J=1; J<=MAXJ; J++)  
        {  
            R1S=sqrt((float)(pow((IP1-I),2)+pow((JP1-J),2)));  
            R2S=sqrt((float)(pow((IP2-I),2)+pow((JP2-J),2)));  
            R3=(float)(MAXI-I);  
            A=R1S*R2S;  
            B=R1S*R3;
```

```

        C=R2S*R3;
        V[I][J]=(VP2*B+VP1*C)/(A+B+C);
    }
}

for (J=TLFT; J<=TRGT; J++)
{
    V[TLEV][J] = 0.0;
}

t0 = time ((long*)0);

/* <master id="123"> */
for (COUNT=1; COUNT<=ITER; COUNT++)
{
    for (J=1; J<=MAXJ; J++)
    {
        V[0][J]=V[1][J];
    }
    for (I=0; I<=MAXI-1; I++)
    {
        V[I][0]=V[I][1];
        V[I][MAXJ+1]=V[I][MAXJ];
    }

    /* <send var="V" type="float[MAXI+1][MAXJ+2]" opt="ONCE"/> */
    /* <send var="V"
        type="float[MAXI+1(0~1)][MAXJ+2(0~MAXJ+1)]"
        opt="XCHG"/> */
    /* <send var="V"
        type="float[MAXI+1(0~MAXI)][MAXJ+2(MAXJ+1~MAXJ+2)]"
        opt="XCHG"/> */
    /* <send var="V"
        type="float[MAXI+1(MAXI~MAXI+1)][MAXJ+2(1~MAXJ+2)]"
        opt="XCHG"/> */
    /* <send var="V"
        type="float[MAXI+1(1~MAXI+1)][MAXJ+2(0~1)]"
        opt="XCHG"/> */

    /* <worker> */

    /* <read var="V" type="float[MAXI+1][MAXJ+2]"/> */
    /* <read var="V"
        type="float[MAXI+1(I:$L-1)][MAXJ+2(J)]"
        opt="XCHG"/> */
    /* <read var="V"
        type="float[MAXI+1(I)][MAXJ+2(J:$L-1)]"
        opt="XCHG"/> */

    /* <target index="I" limits="(1,MAXI,1)" chunk="GRAIN"
        order="1"> */
    for (I=1; I<=MAXI-1; I++)
    /* </target> */
    {
        /* <target index="J" limits="(1,MAXJ+1,1)"

```

```

        chunk="GRAIN" order="1"> */
for (J=1; J<=MAXJ; J++)
/* </target> */
{
    if (((I==IP1) && (J==JP1)) || ((I==IP2) && (J==JP2)) ||
        ((I==TLEV) && (J>=TLFT) && (J<=TRGT)))
        continue;

    RI=(float) (I);
    RJ=(float) (J);

    SIGPX=SIGMA_f(RI+0.5,RJ);
    SIGMX=SIGMA_f(RI-0.5,RJ);
    SIGPY=SIGMA_f(RI,RJ+0.5);
    SIGMY=SIGMA_f(RI,RJ-0.5);

    V[I][J]= (URAX*V[I][J]
    +(SIGMY*V[I][J-1]+SIGPY*V[I][J+1]
    +SIGMX*V[I-1][J]+SIGPX*V[I+1][J]))
    /(URAX+SIGPX+SIGMX+SIGPY+SIGMY);
}
}

/* <send var="V" type="float[MAXI+1(I)][MAXJ+2(J)]"/> */

/* </worker> */

/* <read var="V" type="float[MAXI+1][MAXJ+2]"/> */
}
/* </master> */

for (I=1; I<=MAXI-1; I++)
{
    strcpy(FIELD[I-1] , " ");
    for (J=1; J<=MAXJ; J++)
    {
        for (K=1; K<=NDVP; K++)
        {
            if (abs(V[I][J]-DVP[K-1]) < abs(DVP[K-1]/20.0))
                FIELD[I-1][J-1] = MARK[K-1];
        }
    }
}

FIELD[IP1-1][JP1-1]='*';
FIELD[IP2-1][JP2-1]='*';

strcpy(&FIELD[TLEV-1][TLFT-
1],"1*****");

for (J=1; J<=MAXJ; J++)
{
    FIELD[MAXI-1][J-1]='*';
}

```

```

for (I=1; I<=MAXI; I++)
{
    for (J=MAXJ; J>=2; J--)
    {
        if (FIELD[I-1][J-1] != ' ')
            break;
    }

    printf("( %s)\n", FIELD[I-1]);

}

t1 = time((long *)0) - t0;
printf ("\nTime: (%d), MAXI(J): (%d), Iters: (%d)\n",
        t1, MAXI, ITER);

printf("End of ion_seq.c\n");
exit(0);
}
/* </parallel> */

/* <reference> */
float SIGMA_f(float RI, float RJ)
{
    float RIP1, RJP1, RIP2, RJP2;
    float D1, D2;
    float SIGMA;

    RIP1 = (float)IP1;
    RJP1 = (float)JP1;

    RIP2 = (float)IP2;
    RJP2 = (float)JP2;

    D1=sqrt(pow((RI-RIP1), 2)+pow((RJ-RJP1), 2));
    D2=sqrt(pow((RI-RIP2), 2)+pow((RJ-RJP2), 2));
    SIGMA=1.0/max(1.0, min(D1, D2));

    return (SIGMA);
}
float max(float x, float y)
{
    if (x > y) return x;
    return y;
}
float min(float x, float y)
{
    if (x < y) return x;
    return y;
}
/* </reference> */

```

7.3.3 Parallel Program Workflow

Figure 7.6 shows the runtime workflow of the generated parallel programs for Ion Generation Simulator.

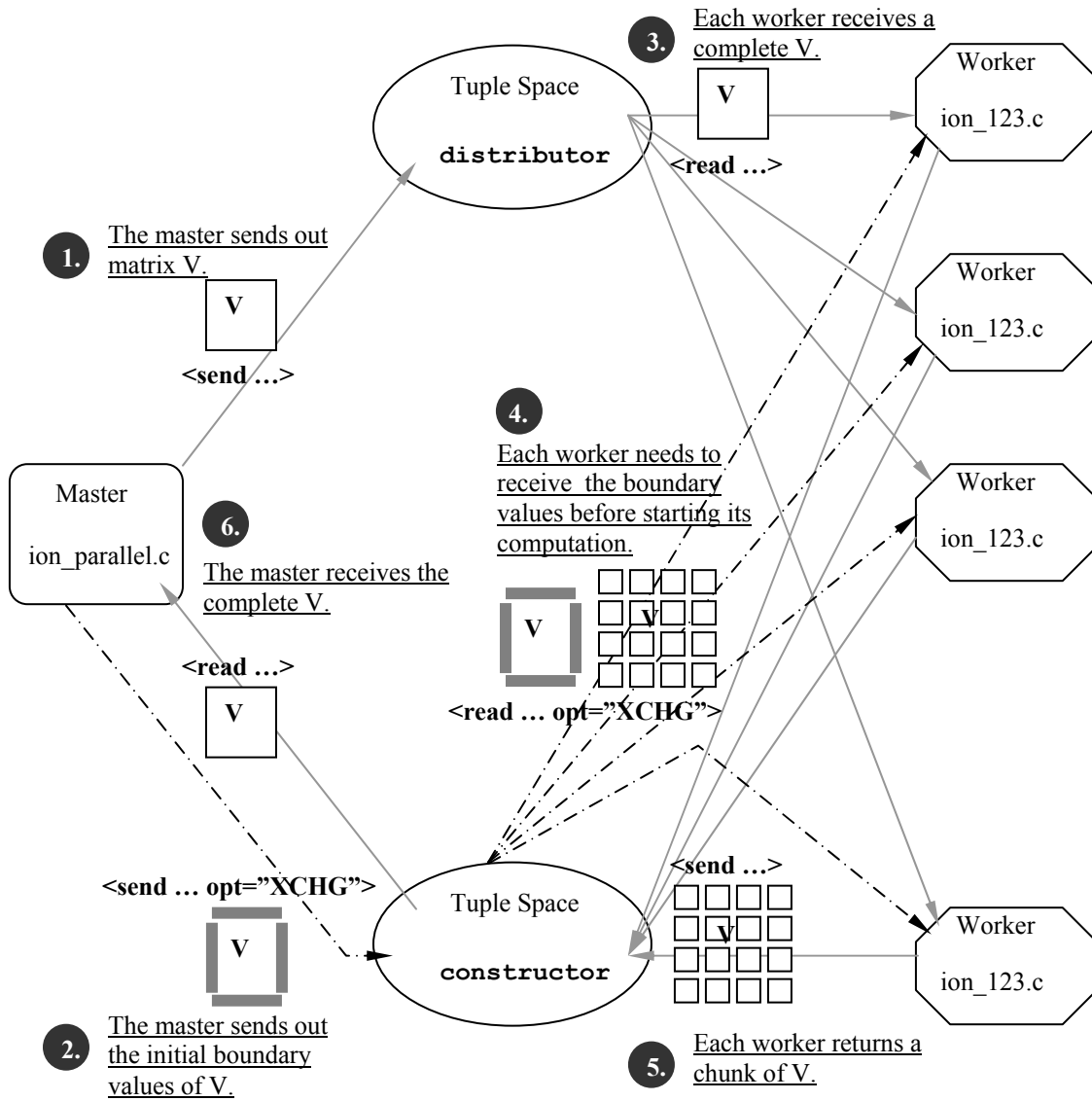


Figure 7.6 Workflow For Ion Generation Simulator

7.3.4 Parallel Processing Performance

Table 7.3 lists the performance testing results of the generated parallel programs for Ion Generation Simulator.

Table 7.3 The Performance Comparison For Ion Generation Simulator

Problem Size MAXI x MAXJ	Grain Size	Iterations	Execution Time (seconds)	
			Sequential	Parallel
120 x 120	40	10	1	1*
120 x 120	40	20	2	2 a)
120 x 120	40	40	5	5 a)
120 x 120	40	80	9	10 a)
240 x 240	60	10	5	7 b)
240 x 240	120	10	5	5 b)
240 x 240	80	10	5	3 b)
240 x 240	80	20	9	6
240 x 240	80	40	20	12
240 x 240	80	80	40	24
480 x 480	120	10	21	16 b)
480 x 480	240	10	21	12 b)
480 x 480	160	10	21	11 b)
480 x 480	160	20	39	20
480 x 480	160	40	80	41
480 x 480	160	80	159	83
960 x 960	240	10	81	55 b)
960 x 960	480	10	81	48 b)
960 x 960	320	10	81	41 b)
960 x 960	320	20	160	82
960 x 960	320	40	317	162
960 x 960	320	80	632	323

- a) The problem size is too small for these entries that the parallel processing setup costs dominate the overall processing time.

b) These entries reflect grain size tuning efforts for the better execution time.

The computation results show that both problem size and number of iteration has no effect on speedup (except when the problem size is too small). Execution times are indeed proportionally to the number of iterations. After the initial setup overhead, the PML compiler-generated codes outperformed the sequential code approximately two times using four processors. Due to the wavefront method, the entire system slows down if there are too many processors involved.

The following Timing Models capture the essential performance characteristics for the ion simulator.

Let

N: Problem size (matrix dimension)

W: Calibrated processor power in arithmetic steps per second

Iter: Number of iterations

μ : Calibrated network speed in bytes per second

δ : Size of data cell in bytes (float = 4)

$$T_{seq} = \frac{Iter \cdot N^2}{W} \quad (1)$$

$$T_{par} = \frac{Iter \cdot N^2}{P \cdot W} + \frac{Iter \cdot \delta \cdot N^2}{\mu} \quad (2)$$

The speed up after parallelization will then be:

$$Sp = \frac{T_{seq}}{T_{par}} = \frac{P}{1 + \frac{\delta \cdot P \cdot W}{\mu}} \quad (3)$$

Equation (3) illustrates the following facts:

- a) The number of iteration has no effect on potential speedup.
- b) The problem size N has no effect on potential speedup.
- c) The network speed will be the largest performance bottleneck.

These facts are effectively confirmed by the experimental results.

7.4 Block LU Factorization

Suppose the $N \times N$ matrix A is partitioned as shown in Figure 7.7, Block LU factorization takes the form, $A = LU$, where the partitioning of L and U is also shown in Figure 7.7.

$$\mathbf{A} = \begin{array}{|c|c|} \hline \mathbf{A}_{00} & \mathbf{A}_{01} \\ \hline \mathbf{A}_{10} & \mathbf{A}_{11} \\ \hline \end{array} = \begin{array}{|c|c|} \hline \mathbf{L}_{00} & \mathbf{0} \\ \hline \mathbf{L}_{10} & \mathbf{L}_{11} \\ \hline \end{array} * \begin{array}{|c|c|} \hline \mathbf{U}_{00} & \mathbf{U}_{01} \\ \hline \mathbf{0} & \mathbf{U}_{11} \\ \hline \end{array} = \mathbf{LU}$$

Figure 7.7 Block LU factorization of the partitioned matrix A .

Then we may write,

$$A_{00} = L_{00} U_{00} \quad (1)$$

$$A_{10} = L_{10} U_{00} \quad (2)$$

$$A_{01} = L_{00} U_{01} \quad (3)$$

$$A_{11} = L_{10} U_{01} + L_{11} U_{11} \quad (4)$$

Where A_{00} is $r \times r$, A_{01} is $r \times (N-r)$, A_{10} is $(N-r) \times r$, and A_{11} is $(N-r) \times (N-r)$. L_{00} and L_{11} are lower triangular matrices with 1's on the main diagonal, and U_{00} and U_{11} are upper triangular matrices [12].

Equations (1) and (2) taken together perform an LU factorization on the first $N \times r$ panel of A (i.e. A_{00} and A_{10}). Once this is completed, the matrices L_{00} , L_{10} , and U_{00} are known, and

the lower triangular system in Eq. (3) can be solved to give U_{01} . Finally, we rearrange Eq. (4) as,

$$A'_{11} = A_{11} - L_{10} U_{01} = L_{11} U_{11} \quad (5)$$

From this equation we see that the problem of finding L_{11} and U_{11} reduces to finding the LU factorization of the $(N-r) \times (N-r)$ matrix A'_{11} . This can be done by applying the steps outlined above to A'_{11} instead of to A . Repeating these steps K times ($K = N/r$), we obtain the LU factorization of the original $N \times N$ matrix A .

7.4.1 The Sequential Solution

Figure 7.8 shows the block LU factorization algorithm from stage k to $k+1$ [12].

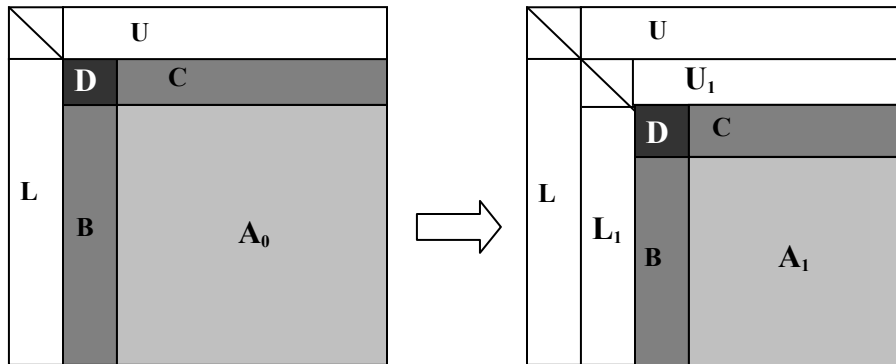


Figure 7.8 From Stage k To $k+1$

After k of these K steps, The trapezoidal submatrices L and U have already been factored. L has kr columns, and U has kr rows. Matrix A has also been updated. Panel D is rxr , B is $(N-kr) \times r$, and C is $r \times (N-kr)$. Step $k+1$ then proceeds as follows,

1. solve the triangular system, $B = L_1 U_0$.

2. solve the triangular system, $C = L_0 U_1$.
3. update on the trailing submatrix, $A'_{11} = A_{11} - L_{10} U_{01}$.

After $k+1$ step, another r columns of LI and r rows of UI are evaluated.

The sequential program for Block LU Factorization is shown below. The loop block in **Bold** faced segments will be marked as the target for parallelization.

```

/* =====
 * Sequential Block LU Factorization, "blu_seq.c".
 */
#include "blu.h"

int main()
{
    int i, j, dist, k1, k2, p1, p2, q1, q2, q3;
    int subdist, rowdist, coldist;
    float inSubMat[M][M], outSubMat[M][M], LU[M][M], L[M][M], U[M][M];

    for (i=0; i<N; i++)
    {
        for (j=0; j<N; j++)
        {
            if (i==j)
            {
                outMat[i][j] = N;
            }
            else
            {
                outMat[i][j] = 1;
            }
        }
    }

    for (i = 0; i < N; i = i + M)
    {
        j = i + M - 1;
        dist = M;
        if (j > N-1)
        {
            j = N-1;
            dist = N - i;
        }

        //LU factors for submatrix
    }
}

```

```

    for (k1 = 0; k1 < dist; k1++)
    {
        for (k2 = 0; k2 < dist; k2++)
        {
            inSubMat[k1][k2] = outMat[i+k1][i+k2];
        }
    }
LUFactor(inSubMat, outSubMat, dist);
//update
for (k1 = 0; k1 < dist; k1++)
{
    for (k2 = 0; k2 < dist; k2++)
    {
        outMat[i+k1][i+k2] = outSubMat[k1][k2];
        LU[k1][k2] = outSubMat[k1][k2];
    }
}

for (k1 = j + 1; k1 < N; k1 = k1 + M)
{
    k2 = k1 + M - 1;
    subdist = M;
    if (k2 > N-1)
    {
        k2 = N - 1;
        subdist = N - k1;
    }

    //Solve LZ
    for (p1 = i; p1 < i+M; p1++)
    {
        for (p2 = k1; p2 <= k2; p2++)
        {
            inSubMat[p1-i][p2-k1] = outMat[p1][p2];
        }
    }
    TriangleSolver(LU, inSubMat, outSubMat, subdist, 1);
    //update
    for (p1 = i; p1 < i+M; p1++)
    {
        for (p2 = k1; p2 <= k2; p2++)
        {
            outMat[p1][p2] = outSubMat[p1-i][p2-k1];
        }
    }

    //Solve WU
    for (p1 = i; p1 < i+M; p1++)
    {
        for (p2 = k1; p2 <= k2; p2++)
        {
            inSubMat[p2-k1][p1-i] = outMat[p2][p1];
        }
    }
    TriangleSolver(LU, inSubMat, outSubMat, subdist, 2);
}

```

```

//update
for (p1 = i; p1 < i+M; p1++)
{
    for (p2 = k1; p2 <= k2; p2++)
    {
        outMat[p2][p1] = outSubMat[p2-k1][p1-i];
    }
}

//A = A - WZ
for (k1 = i + M; k1 < N; k1 = k1 + M)
{
    k2 = k1 + M - 1;
    rowdist = M;
    if (k2 > N-1)
    {
        k2 = N - 1;
        rowdist = N - k1;
    }

    for (p1 = i + M; p1 < N; p1 = p1 + M)
    {
        p2 = p1 + M - 1;
        coldist = M;
        if (p2 > N-1)
        {
            p2 = N - 1;
            coldist = N - p1;
        }

        /*
        * matrix multiplication
        * outMat[k1:k2][p1:p2] = outMat[k1:k2][p1:p2] -
        *   outMat[k1:k2][i:i+M-1]* outMat[i:i+M-1][k1:k2];
        */
        for (q1 = k1; q1 <= k2; q1++)
        {
            for (q2 = p1; q2 <= p2; q2++)
            {
                for (q3 = 0; q3 < M; q3++)
                {
                    outMat[q1][q2] = outMat[q1][q2] -
                        outMat[q1][i+q3]*outMat[i+q3][q2];
                }
            }
        }
    }
}

/*
for (i=0; i<N; i++)
{
    for (j=0; j<N; j++)

```

```

        {
            printf("%6.3f ", outMat[i][j]);
        }
        printf("\n");
    }
    */
    return 0;
}

/* =====
 * The Header File, "blu.h".
 */
#include <stdio.h>

#define N 200
#define M 100

float outMat[N][N];

void LUFactor(float [][], float [][], int);
void TriangleSolver(float [][], float [][], float [][], int, int);

```

7.4.2 PML-tagged Sequential Program

Below is the PML-tagged sequential program for Block LU Factorization. The generated parallel programs are shown in Appendix E.

```

/* =====
 * Sequential Block LU Factorization, "blu.c".
 */
/* <reference> */
#include "blu.h"
/* </reference> */

/* <parallel appname="blu"> */
int main()
{
    /* <reference> */
    int i, j, dist, k1, k2, p1, p2, q1, q2, q3;
    int subdist, rowdist, coldist;
    float inSubMat[M][M], outSubMat[M][M], LU[M][M], L[M][M], U[M][M];
    /* </reference> */

    for (i=0; i<N; i++)
    {

```

```

for (j=0; j<N; j++)
{
    if (i==j)
    {
        outMat[i][j] = N;
    }
    else
    {
        outMat[i][j] = 1;
    }
}
}

/* <master id="123"> */
for (i = 0; i < N; i = i + M)
{
    j = i + M - 1;
    dist = M;
    if (j > N-1)
    {
        j = N-1;
        dist = N - i;
    }

    //LU factors for submatrix
    for (k1 = 0; k1 < dist; k1++)
    {
        for (k2 = 0; k2 < dist; k2++)
        {
            inSubMat[k1][k2] = outMat[i+k1][i+k2];
        }
    }
    LUFactor(inSubMat, outSubMat, dist);
    //update
    for (k1 = 0; k1 < dist; k1++)
    {
        for (k2 = 0; k2 < dist; k2++)
        {
            outMat[i+k1][i+k2] = outSubMat[k1][k2];
            LU[k1][k2] = outSubMat[k1][k2];
        }
    }
}

for (k1 = j + 1; k1 < N; k1 = k1 + M)
{
    k2 = k1 + M - 1;
    subdist = M;
    if (k2 > N-1)
    {
        k2 = N - 1;
        subdist = N - k1;
    }

    //Solve LZ
    for (p1 = i; p1 < i+M; p1++)

```

```

    {
        for (p2 = k1; p2 <= k2; p2++)
        {
            inSubMat[p1-i][p2-k1] = outMat[p1][p2];
        }
    }
    TriangleSolver(LU, inSubMat, outSubMat, subdist, 1);
    //update
    for (p1 = i; p1 < i+M; p1++)
    {
        for (p2 = k1; p2 <= k2; p2++)
        {
            outMat[p1][p2] = outSubMat[p1-i][p2-k1];
        }
    }

    //Solve WU
    for (p1 = i; p1 < i+M; p1++)
    {
        for (p2 = k1; p2 <= k2; p2++)
        {
            inSubMat[p2-k1][p1-i] = outMat[p2][p1];
        }
    }
    TriangleSolver(LU, inSubMat, outSubMat, subdist, 2);
    //update
    for (p1 = i; p1 < i+M; p1++)
    {
        for (p2 = k1; p2 <= k2; p2++)
        {
            outMat[p2][p1] = outSubMat[p2-k1][p1-i];
        }
    }
}

/* <send var="i" type="int"/> */
/* <send var="outMat" type="float[N(i~N)][N(i~N)]"/> */

/* <worker> */
/* <read var="i" type="int"/> */

/* <read var="outMat" type="float[N(i~i+M)][N(i~N)]"/> */
/* <read var="outMat" type="float[N(k1)][N(i~N)]"/> */

//A = A - WZ
/* <target index="k1" limits="(i+M,N,M)" chunk="M" order="1"> */
for (k1 = i + M; k1 < N; k1 = k1 + M)
/* </target> */
{
    k2 = k1 + M - 1;
    rowdist = M;
    if (k2 > N-1)
    {
        k2 = N - 1;
        rowdist = N - k1;
    }
}

```

```

    }

    for (p1 = i + M; p1 < N; p1 = p1 + M)
    {
        p2 = p1 + M - 1;
        coldist = M;
        if (p2 > N-1)
        {
            p2 = N - 1;
            coldist = N - p1;
        }

        /*
        * matrix multiplication
        * outMat[k1:k2][p1:p2] = outMat[k1:k2][p1:p2] -
        *   outMat[k1:k2][i:i+M-1]* outMat[i:i+M-1][k1:k2];
        */
        for (q1 = k1; q1 <= k2; q1++)
        {
            for (q2 = p1; q2 <= p2; q2++)
            {
                for (q3 = 0; q3 < M; q3++)
                {
                    outMat[q1][q2] = outMat[q1][q2] -
                        outMat[q1][i+q3]*outMat[i+q3][q2];
                }
            }
        }
    }

    /* <send var="outMat" type="float[N(k1)][N(i+M~N)]"/> */
    /* </worker> */

    /* <read var="outMat" type="float[N(i+M~N)][N(i+M~N)]"/> */
}
/* </master> */

/*
for (i=0; i<N; i++)
{
    for (j=0; j<N; j++)
    {
        printf("%6.3f ", outMat[i][j]);
    }
    printf("\n");
}
*/
return 0;
}
/* </parallel> */

```

7.4.3 Parallel Program Workflow

Figure 7.9 shows the runtime workflow of the generated parallel programs for Block LU Factorization.

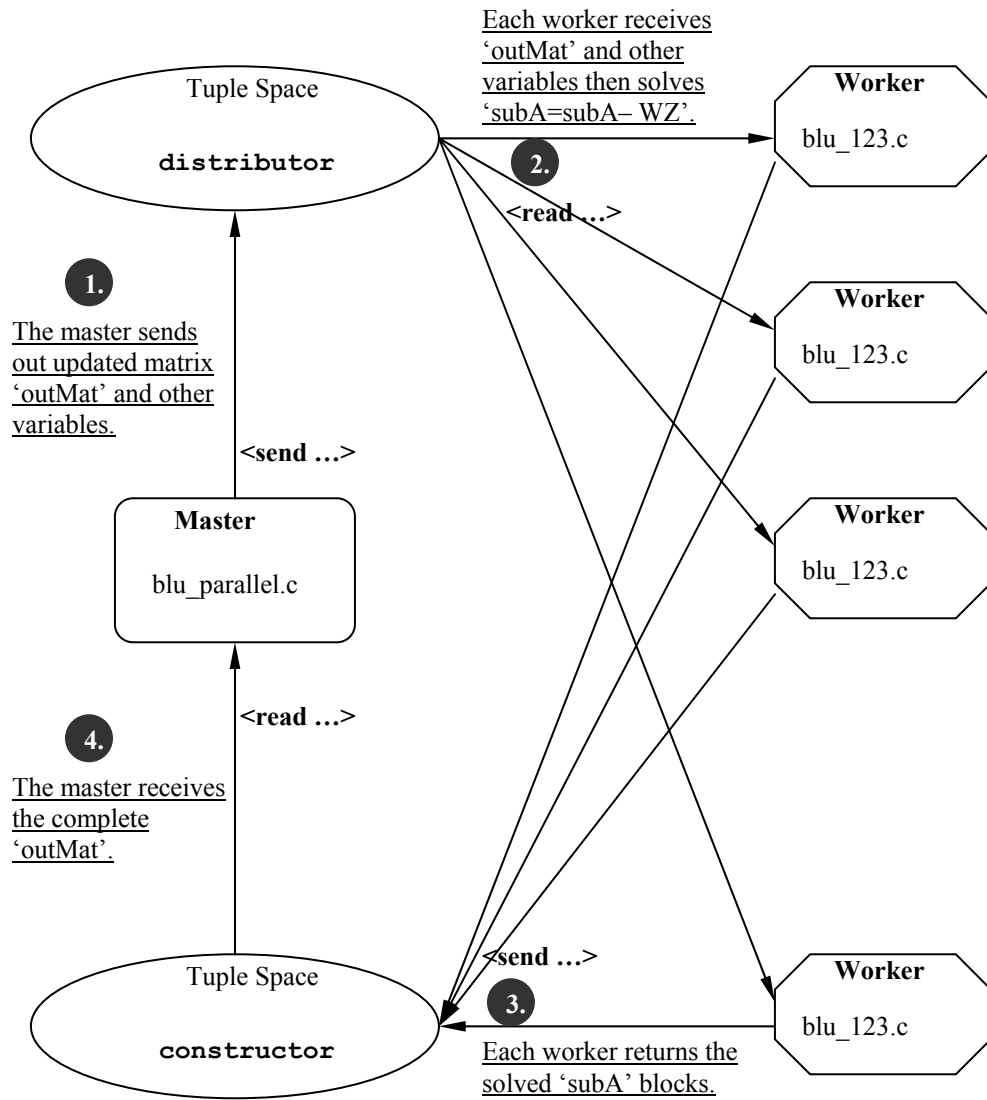


Figure 7.9 Workflow For Block LU Factorization

7.4.4 Parallel Processing Performance

Table 7.4 lists the performance testing results of the generated parallel programs for Block LU Factorization.

Table 7.4 **The Performance Comparison For Block LU Factorization**

<u>Processors</u>	<u>Matrix Size</u>	<u>Automatic Generated (seconds)</u>	<u>Manually Created (seconds)</u>
2	200 x 200	1	1
4	400 x 400	2	8
4	800 x 800	8	30
4	1200 x 1200	22	66
4	1600 x 1600	47	121

Due to the extended tuple space supports, the automatically generated parallel programs out-performed its manually crafted counterpart.

CHAPTER 8. CONCLUSION

This thesis investigates a new approach for automatic sequential-to-parallel program translation by leveraging the coarse-grain dataflow computation model in the SPP architecture. The dataflow computation model allows automatic dependency uncovering without complex static dependency analysis. Using the proposed approach, the user only needs to describe an overall dependency pattern with simple PML tags. This pattern is automatically extracted by the PML compiler and used to generate parallel programs. Non-linear, indirect and conditional dependencies are automatically uncovered at runtime. In other words, the PML compiler correctly implements the described dependency patterns leaving the SPP runtime environment to uncover embedded parallelisms. Therefore, if a strictly sequential program is tagged correctly, the generated parallel code will produce the exact same results with zero or negative performance gain.

The fundamental departing point from other parallelizing compiler approaches is the use of dataflow computation model. Without the SPP architecture, however, the dataflow computation model alone cannot promise high performance delivery. This thesis leverages both prior results in pursue of a higher result. In contrast, approaches without using the dataflow computation model, such as the PARADIGM project at UIUC, will have to overcome insurmountable difficulties when attempting to solve nonlinear and other dependencies at compile time.

We have chosen four well-recognized numerical applications as test cases in order to examine the expressive power of the PML tags and effectiveness of generated parallel programs. Computational experiments were conducted in the dataflow computation environment provided by the Synergy system – a preliminary implementation of the SPP architecture using multiple networked computers. We have shown that generated parallel programs can perform favorably against manually written parallel programs. All experiments are easily repeatable.

Theoretically, the proposed methodology is applicable to all types of iterative compute-intensive applications. This means that it can enable automatic generation of parallel programs for many host programming languages. Practical applications of this technology can potentially unleash a vast amount of productivity gain for scientific and engineering applications.

8.1 Summary of Contributions

This thesis is the first effort to automatically generate parallel programs for distributed-memory supercomputers without using direct message-passing protocols. The envisioned contributions of this research include

- Design of Parallelization Markup Language (PML).
- Design and implementation of PML compiler.
- Extended tuple space support for simplifying parallel program generation and for optimizing runtime parallel performance.

8.2 Future Work

We have shown that the PML design is simple yet powerful enough for describing a wide-range of dependency patterns. Thus it enables automatic generation of a variety of parallel programs fully leveraging the target parallel processing components. However, since the tagging strategy directly affects the parallel processing performance, the use of timing models [35] is highly recommended BEFORE each tagging practice.

This research compliments the SPP research into a powerful high performance, highly available computing tool set that can be used for all mission critical applications. Further researches in this direction include:

- Enhance the PML compiler to automatically discovery of PML tag attributes. The PML tags, such as the *target* tag, *send* and *read* tags, contain much information the user must fill in. It is possible for PML compiler to conduct syntactical analysis to extract some of those information, such as the name and type of the variables being transferred, and the loop index and boundary information.
- Enrich PML tags to further increase their expressive power. Interactive tools may be built to assist sequential program tagging with drag-and-drop clicks.
- Extend host programming language support. Current PML implementation

supports only programming languages with C interface. Since PML is language independent, it should be relatively easy to adapt to different host language(s) by developing different code generators and syntax analyzers.

- Timing model support. It has been shown that timing model analysis can effectively guide PML tagging strategy. It is possible to develop tools that can automate timing model analysis and tie its results with PML tagging strategy.
- For applications that high performance out-weighs reliability, the *token* tuple concept in the enhanced tuple space can be generalized to implement a versatile dependence graph for further performance improvements at the expense of load balancing and fault tolerance. In other words, it is possible to achieve direct communication performance while retaining distributed tuple matching mechanism and automatic code generation.

Appendix A – Parallel Programs For The Example In Section 3.3

1. Partitioning Horizontally

- The Master Program

```
#include "parallel.h"
/* <reference> */
/* Constant, N, and the grain size, G, are defined in the header file */
#include "init.h"
/* </reference> */

/* <parallel appname="init1"> */
double * _tp_v_123;
double * _tp_A_123;
int _x0_123;
int _x1_123;

int main()
{
    /* <reference> */
    double A[N][N];
    double v;
    int i, j;
    /* </reference> */

    v = 99.9;

    /* <master id="123"> */
    _distributor = _open_space("distributor", 0, "123");
    _constructor = _open_space("constructor", 0, "123");
    _cleanup_space(_distributor, "123");
    _cleanup_space(_constructor, "123");

    /* <send var="v" type="double" opt="ONCE"/> */
    sprintf(_tp_name, "double:v#%s", "123");
    _tp_size = sizeof(double);
    _tp_v_123 = &v;
    _status =
    _send_data(_distributor, _tp_name, (char *)_tp_v_123, _tp_size);
    if (_status < 0) exit(-1);

    /* <token action="SET" idxset="(i)"/> */
    sprintf(_tp_name, "token#%s", "123");
    sprintf(_tp_token, "(i:%d~%d,%d:#%d)", 0, N, 1, G);
    _tp_size = sizeof(_tp_token);
    _tokens =
    _set_token(_distributor, _tp_name, (char *)_tp_token, _tp_size);
    if (_tokens < 0) exit(-1);
}
```

```

    /* <send var="A" type="double[N][N]"> */
    sprintf(_tp_name, "double(%d)(%d):A#%s", (N), (N), "123");
    _tp_size = ((N) * (N)) * sizeof(double);
    _tp_A_123 = (double *)malloc(_tp_size);
    for (_x0_123 = 0; _x0_123 < (N); _x0_123 +=1) {
        for (_x1_123 = 0; _x1_123 < (N); _x1_123 +=1) {
            _tp_A_123[_x0_123 * (N) + _x1_123] = A[_x0_123][_x1_123];
        }
    }
    _status =
    _send_data(_distributor, _tp_name, (char *)_tp_A_123, _tp_size);
    if (_status < 0) exit(-1);
    free(_tp_A_123);

    /* <read var="A" type="double[N][N]"> */
    sprintf(_tp_name, "double(%d)(%d):A#%s", (N), (N), "123");
    _tp_size = ((N) * (N)) * sizeof(double);
    _tp_A_123 = (double *)malloc(_tp_size);
    _tp_size =
    _read_data(_constructor, _tp_name, (char *)_tp_A_123, _tp_size);
    if (_tp_size < 0) exit(-1);
    for (_x0_123 = 0; _x0_123 < (N); _x0_123 +=1) {
        for (_x1_123 = 0; _x1_123 < (N); _x1_123 +=1) {
            A[_x0_123][_x1_123] = _tp_A_123[_x0_123 * (N) + _x1_123];
        }
    }
    free(_tp_A_123);

    _close_space(_constructor, "123", 1);
    _close_space(_distributor, "123", 1);
    /* </master> */
}
/* </parallel> */

```

- The Worker Program

```

#include "parallel.h"
/* <reference> */
/* Constant, N, and the grain size, G, are defined in the header file */
#include "init.h"
/* </reference> */

/* <reference> */
double A[N][N];
double v;
int i, j;
/* </reference> */

/* <parallel appname="init1"> */
double * _tp_v_123;
double * _tp_A_123;
int _x0_123;
int _y0_123;

```

```

int _x1_123;
int _y1_123;
int _i_start = 0;
int _i_stop = 0;
int _i_step = 0;

main(int argc, char **argv[])
{
    /* <worker id="123"> */
    _distributor = _open_space("distributor", 0, "123");
    _constructor = _open_space("constructor", 0, "123");

    /* <read var="v" type="double" opt="ONCE"/> */
    sprintf(_tp_name, "double:v#%s", "123");
    _tp_size = sizeof(double);
    _tp_v_123 = &v;
    _tp_size =
    _read_data(_distributor, _tp_name, (char *)_tp_v_123, _tp_size);
    if (_tp_size < 0) exit(-1);

    while (1)
    {
        /* <token action="GET" idxset="(i)"/> */
        sprintf(_tp_name, "token#%s", "123");
        _tp_size = 0;
        _tp_size =
        _get_token(_distributor, _tp_name, (char *)_tp_token, _tp_size);
        if (_tp_size < 0) exit(-1);
        if (_tp_token[0] == '!') break;
        sscanf(_tp_token, "%d@(i:%d~%d,%d)",
            &_tokens, &_i_start, &_i_stop, &_i_step);

        /* <read var="A" type="double[N(i)][N]"/> */
        sprintf(_tp_name, "double(%d)(%d):A#%s[%d~%d,%d][%d~%d,%d]@%d",
            (N), (N), "123", _i_start, _i_stop, 1, 0, (N), 1,
            sizeof(double));
        _tp_size =
        (((_i_stop - _i_start - 1) / 1 + 1) * (N)) * sizeof(double);
        _tp_A_123 = (double *)malloc(_tp_size);
        _tp_size =
        _read_data(_distributor, _tp_name, (char *)_tp_A_123, _tp_size);
        if (_tp_size < 0) exit(-1);
        for (_x0_123 = _i_start, _y0_123 = 0; _x0_123 < _i_stop;
            _x0_123 +=1, _y0_123 ++) {
            for (_x1_123 = 0; _x1_123 < (N); _x1_123 +=1) {
                A[_x0_123][_x1_123] =
                _tp_A_123[_y0_123 * (N) + _x1_123];
            }
        }
        free(_tp_A_123);

        /*<target index="i" order="1" limits="(0,N,1)" chunk="G">*/
        for (i = _i_start; i < _i_stop; i +=_i_step)
        /*</target>*/
        {

```

```

        for (j = 0; j < N; j++)
        {
            A[i, j] = W;
        }
    }

    /* <send var="A" type="double[N(i)][N]" /> */
    sprintf(_tp_name, "double(%d) (%d):A#%s[%d~%d,%d][%d~%d,%d]@%d",
        (N), (N), "123", _i_start, _i_stop, 1, 0, (N), 1,
        sizeof(double));
    _tp_size =
        (((_i_stop - _i_start - 1) / 1 + 1) * (N)) * sizeof(double);
    _tp_A_123 = (double *)malloc(_tp_size);
    for (_x0_123 = _i_start, _y0_123 = 0; _x0_123 < _i_stop;
        _x0_123 += 1, _y0_123 += 1) {
        for (_x1_123 = 0; _x1_123 < (N); _x1_123 += 1) {
            _tp_A_123[_y0_123 * (N) + _x1_123] =
                A[_x0_123][_x1_123];
        }
    }
    _status =
        _send_data(_constructor, _tp_name, (char *)_tp_A_123, _tp_size);
    if (_status < 0) exit(-1);
    free(_tp_A_123);
}

_close_space(_constructor, "123", 0);
_close_space(_distributor, "123", 0);
/* </worker> */
exit(0);
}
/* </parallel> */

```

2. Partitioning Vertically

- The Master Program

```

#include "parallel.h"
/* <reference> */
/* Constant, N, and the grain size, G, are defined in the header file */
#include "init.h"
/* </reference> */

/* <parallel appname="init2"> */
double * _tp_v_123;
double * _tp_A_123;
int _x0_123;
int _x1_123;

int main()
{
    /* <reference> */
    double A[N][N];

```

```

double v;
int i, j;
/* </reference> */

v = 99.9;

/* <master id="123"> */
_distributor = _open_space("distributor", 0, "123");
_constructor = _open_space("constructor", 0, "123");
_cleanup_space(_distributor, "123");
_cleanup_space(_constructor, "123");

/* <send var="v" type="double" opt="ONCE"/> */
sprintf(_tp_name, "double:v#%s", "123");
_tp_size = sizeof(double);
_tp_v_123 = &v;
_status =
_send_data(_distributor, _tp_name, (char *)_tp_v_123, _tp_size);
if (_status < 0) exit(-1);

/* <token action="SET" idxset="(j)"/> */
sprintf(_tp_name, "token#%s", "123");
sprintf(_tp_token, "(j:%d~%d,%d:#%d)", 0, N, 1, G);
_tp_size = sizeof(_tp_token);
_tokens =
_set_token(_distributor, _tp_name, (char *)_tp_token, _tp_size);
if (_tokens < 0) exit(-1);

/* <send var="A" type="double[N][N]"/> */
sprintf(_tp_name, "double(%d)(%d):A#%s", (N), (N), "123");
_tp_size = ((N) * (N)) * sizeof(double);
_tp_A_123 = (double *)malloc(_tp_size);
for (_x0_123 = 0; _x0_123 < (N); _x0_123 +=1) {
    for (_x1_123 = 0; _x1_123 < (N); _x1_123 +=1) {
        _tp_A_123[_x0_123 * (N) + _x1_123] = A[_x0_123][_x1_123];
    }
}
_status =
_send_data(_distributor, _tp_name, (char *)_tp_A_123, _tp_size);
if (_status < 0) exit(-1);
free(_tp_A_123);

/* <read var="A" type="double[N][N]"/> */
sprintf(_tp_name, "double(%d)(%d):A#%s", (N), (N), "123");
_tp_size = ((N) * (N)) * sizeof(double);
_tp_A_123 = (double *)malloc(_tp_size);
_tp_size =
_read_data(_constructor, _tp_name, (char *)_tp_A_123, _tp_size);
if (_tp_size < 0) exit(-1);
for (_x0_123 = 0; _x0_123 < (N); _x0_123 +=1) {
    for (_x1_123 = 0; _x1_123 < (N); _x1_123 +=1) {
        A[_x0_123][_x1_123] = _tp_A_123[_x0_123 * (N) + _x1_123];
    }
}
free(_tp_A_123);

```

```

    _close_space(_constructor, "123", 1);
    _close_space(_distributor, "123", 1);
    /* </master> */
}
/* </parallel> */

```

- The Worker Program

```

#include "parallel.h"
/* <reference> */
/* Constant, N, and the grain size, G, are defined in the header file */
#include "init.h"
/* </reference> */

/* <reference> */
double A[N][N];
double v;
int i, j;
/* </reference> */

/* <parallel appname="init2"> */
double * _tp_v_123;
double * _tp_A_123;
int _x0_123;
int _y0_123;
int _x1_123;
int _y1_123;
int _j_start = 0;
int _j_stop = 0;
int _j_step = 0;

main(int argc, char **argv[])
{
    /* <worker id="123"> */
    _distributor = _open_space("distributor", 0, "123");
    _constructor = _open_space("constructor", 0, "123");

    /* <read var="v" type="double" opt="ONCE"/> */
    sprintf(_tp_name, "double:v#%s", "123");
    _tp_size = sizeof(double);
    _tp_v_123 = &v;
    _tp_size =
    _read_data(_distributor, _tp_name, (char *)_tp_v_123, _tp_size);
    if (_tp_size < 0) exit(-1);

    while (1)
    {
        /* <token action="GET" idxset="(j)"/> */
        sprintf(_tp_name, "token#%s", "123");
        _tp_size = 0;
        _tp_size =
        _get_token(_distributor, _tp_name, (char *)_tp_token, _tp_size);
    }
}

```

```

if (_tp_size < 0) exit(-1);
if (_tp_token[0] == '!') break;
sscanf(_tp_token, "%d@(j:%d~%d,%d)",
        &_tokens, &_j_start, &_j_stop, &_j_step);

/* <read var="A" type="double[N][N(j)]"/> */
sprintf(_tp_name, "double(%d)(%d):A#s[%d~%d,%d][%d~%d,%d]@%d",
        (N), (N), "123", 0, (N), 1, _j_start, _j_stop, 1,
        sizeof(double));
_tp_size =
((N) * ((_j_stop - _j_start - 1) / 1 + 1)) * sizeof(double);
_tp_A_123 = (double *)malloc(_tp_size);
_tp_size =
_read_data(_distributor, _tp_name, (char *)_tp_A_123, _tp_size);
if (_tp_size < 0) exit(-1);
for (_x0_123 = 0; _x0_123 < (N); _x0_123 +=1) {
    for (_x1_123 = _j_start, _y1_123 =0; _x1_123 < _j_stop;
        _x1_123 +=1, _y1_123 ++) {
        A[_x0_123][_x1_123] =
        _tp_A_123[_x0_123 * ((_j_stop - _j_start - 1) / 1 + 1) +
        _y1_123];
    }
}
free(_tp_A_123);

for (i = 0; i < N; i++)
{
    /*<target index="j" order="1" limits="(0,N,1)" chunk="G">*/
    for (j = _j_start; j < _j_stop; j +=_j_step)
    /*</target>*/
    {
        A[i, j] = v;
    }
}

/* <send var="A" type="double[N][N(j)]"/> */
sprintf(_tp_name, "double(%d)(%d):A#s[%d~%d,%d][%d~%d,%d]@%d",
        (N), (N), "123", 0, (N), 1, _j_start, _j_stop, 1,
        sizeof(double));
_tp_size =
((N) * ((_j_stop - _j_start - 1) / 1 + 1)) * sizeof(double);
_tp_A_123 = (double *)malloc(_tp_size);
for (_x0_123 = 0; _x0_123 < (N); _x0_123 +=1) {
    for (_x1_123 = _j_start, _y1_123 =0; _x1_123 < _j_stop;
        _x1_123 +=1, _y1_123 ++) {
        _tp_A_123[_x0_123 * ((_j_stop - _j_start - 1) / 1 + 1) +
        _y1_123] =
        A[_x0_123][_x1_123];
    }
}
_status =
_send_data(_constructor, _tp_name, (char *)_tp_A_123, _tp_size);
if (_status < 0) exit(-1);
free(_tp_A_123);
}

```

```

    _close_space(_constructor, "123", 0);
    _close_space(_distributor, "123", 0);
    /* </worker> */
    exit(0);
}
/* </parallel> */

```

3. Partitioning on Both Dimensions

- The Master Program

```

#include "parallel.h"
/* <reference> */
/* Constant, N, and the grain size, G, are defined in the header file */
#include "init.h"
/* </reference> */

/* <parallel appname="init3"> */
double * _tp_v_123;
double * _tp_A_123;
int _x0_123;
int _x1_123;

int main()
{
    /* <reference> */
    double A[N][N];
    double v;
    int i, j;
    /* </reference> */

    v = 99.9;

    /* <master id="123"> */
    _distributor = _open_space("distributor", 0, "123");
    _constructor = _open_space("constructor", 0, "123");
    _cleanup_space(_distributor, "123");
    _cleanup_space(_constructor, "123");

    /* <send var="v" type="double" opt="ONCE"/> */
    sprintf(_tp_name, "double:v#%s", "123");
    _tp_size = sizeof(double);
    _tp_v_123 = &v;
    _status =
    _send_data(_distributor, _tp_name, (char *)_tp_v_123, _tp_size);
    if (_status < 0) exit(-1);

    /* <token action="SET" idxset="(i) (j)"/> */
    sprintf(_tp_name, "token#%s", "123");
    sprintf(_tp_token, "!(i:%d~%d,%d:#%d) (j:%d~%d,%d:#%d)",
            0, N, 1, G, 0, N, 1, G);
    _tp_size = sizeof(_tp_token);

```

```

    _tokens =
    _set_token(_distributor, _tp_name, (char *)_tp_token, _tp_size);
    if (_tokens < 0) exit(-1);

    /* <send var="A" type="double[N][N]"> */
    sprintf(_tp_name, "double(%d)(%d):A#%s", (N), (N), "123");
    _tp_size = ((N) * (N)) * sizeof(double);
    _tp_A_123 = (double *)malloc(_tp_size);
    for (_x0_123 = 0; _x0_123 < (N); _x0_123 +=1) {
        for (_x1_123 = 0; _x1_123 < (N); _x1_123 +=1) {
            _tp_A_123[_x0_123 * (N) + _x1_123] = A[_x0_123][_x1_123];
        }
    }
    _status =
    _send_data(_distributor, _tp_name, (char *)_tp_A_123, _tp_size);
    if (_status < 0) exit(-1);
    free(_tp_A_123);

    /* <read var="A" type="double[N][N]"> */
    sprintf(_tp_name, "double(%d)(%d):A#%s", (N), (N), "123");
    _tp_size = ((N) * (N)) * sizeof(double);
    _tp_A_123 = (double *)malloc(_tp_size);
    _tp_size =
    _read_data(_constructor, _tp_name, (char *)_tp_A_123, _tp_size);
    if (_tp_size < 0) exit(-1);
    for (_x0_123 = 0; _x0_123 < (N); _x0_123 +=1) {
        for (_x1_123 = 0; _x1_123 < (N); _x1_123 +=1) {
            A[_x0_123][_x1_123] = _tp_A_123[_x0_123 * (N) + _x1_123];
        }
    }
    free(_tp_A_123);

    _close_space(_constructor, "123", 1);
    _close_space(_distributor, "123", 1);
    /* </master> */
}
/* </parallel> */

```

- The Worker Program

```

#include "parallel.h"
/* <reference> */
/* Constant, N, and the grain size, G, are defined in the header file */
#include "init.h"
/* </reference> */

/* <reference> */
double A[N][N];
double v;
int i, j;
/* </reference> */

/* <parallel appname="init3"> */

```

```

double * _tp_v_123;
double * _tp_A_123;
int _x0_123;
int _y0_123;
int _x1_123;
int _y1_123;
int _i_start = 0;
int _i_stop = 0;
int _i_step = 0;
int _j_start = 0;
int _j_stop = 0;
int _j_step = 0;

main(int argc, char **argv[])
{
    /* <worker id="123"> */
    _distributor = _open_space("distributor", 0, "123");
    _constructor = _open_space("constructor", 0, "123");

    /* <read var="v" type="double" opt="ONCE"/> */
    sprintf(_tp_name, "double:v#%s", "123");
    _tp_size = sizeof(double);
    _tp_v_123 = &v;
    _tp_size =
    _read_data(_distributor, _tp_name, (char *)_tp_v_123, _tp_size);
    if (_tp_size < 0) exit(-1);

    while (1)
    {
        /* <token action="GET" idxset="(i)(j)"/> */
        sprintf(_tp_name, "token#%s", "123");
        _tp_size = 0;
        _tp_size =
        _get_token(_distributor, _tp_name, (char *)_tp_token, _tp_size);
        if (_tp_size < 0) exit(-1);
        if (_tp_token[0] == '!') break;
        sscanf(_tp_token, "%d@(i:%d~%d,%d)(j:%d~%d,%d)",
            &_tokens,
            &_i_start, &_i_stop, &_i_step,
            &_j_start, &_j_stop, &_j_step);

        /* <read var="A" type="double[N(i)][N(j)]"/> */
        sprintf(_tp_name, "double(%d)(%d):A#%s[%d~%d,%d][%d~%d,%d]@%d",
            (N), (N), "123",
            _i_start, _i_stop, 1, _j_start, _j_stop, 1,
            sizeof(double));
        _tp_size =
        (((_i_stop - _i_start - 1) / 1 + 1) *
        ((_j_stop - _j_start - 1) / 1 + 1)) * sizeof(double);
        _tp_A_123 = (double *)malloc(_tp_size);
        _tp_size =
        _read_data(_distributor, _tp_name, (char *)_tp_A_123, _tp_size);
        if (_tp_size < 0) exit(-1);
        for (_x0_123 = _i_start, _y0_123 = 0; _x0_123 < _i_stop;
            _x0_123 +=1, _y0_123 ++ ) {

```

```

        for (_x1_123 = _j_start, _y1_123 =0; _x1_123 < _j_stop;
            _x1_123 +=1, _y1_123 ++) {
            A[_x0_123][_x1_123] =
                _tp_A_123[_y0_123 * ((_j_stop - _j_start - 1) / 1 + 1) +
                _y1_123];
        }
    }
    free(_tp_A_123);

    /*<target index="i" order="1" limits="(0,N,1)" chunk="G">*/
    for (i = _i_start; i < _i_stop; i +=_i_step)
    /*</target>*/
    {
        /*<target index="j" order="2" limits="(0,N,1)" chunk="G">*/
        for (j = _j_start; j < _j_stop; j +=_j_step)
        /*</target>*/
        {
            A[i, j] = v;
        }
    }

    /* <send var="A" type="double[N(i)][N(j)]"/> */
    sprintf(_tp_name, "double(%d) (%d) :A#%s[%d~%d,%d] [%d~%d,%d]@%d",
        (N), (N), "123",
        _i_start, _i_stop, 1,
        _j_start, _j_stop, 1,
        sizeof(double));
    _tp_size =
        (((_i_stop - _i_start - 1) / 1 + 1) *
        ((_j_stop - _j_start - 1) / 1 + 1)) * sizeof(double);
    _tp_A_123 = (double *)malloc(_tp_size);
    for (_x0_123 = _i_start, _y0_123 =0; _x0_123 < _i_stop;
        _x0_123 +=1, _y0_123 ++) {
        for (_x1_123 = _j_start, _y1_123 =0; _x1_123 < _j_stop;
            _x1_123 +=1, _y1_123 ++) {
            _tp_A_123[_y0_123 * ((_j_stop - _j_start - 1) / 1 + 1) +
            _y1_123] =
                A[_x0_123][_x1_123];
        }
    }
    _status =
    _send_data(_constructor, _tp_name, (char *)_tp_A_123, _tp_size);
    if (_status < 0) exit(-1);
    free(_tp_A_123);
}

_close_space(_constructor, "123", 0);
_close_space(_distributor, "123", 0);
/* </worker> */
exit(0);
}
/* </parallel> */

```

Appendix B – Parallel Programs For Matrix Multiplication

- The Master Program

```
/* =====
 * Compiler-Generated Master Program, "matrix_parallel.c".
 */
#include "parallel.h"
/* <reference> */
#include "matrix.h"
/* </reference> */

/* <parallel appname="matrix"> */
double * _tp_B_123;
double * _tp_A_123;
double * _tp_C_123;
int _x0_123;
int _x1_123;
int _y0_123;
int _y1_123;

main(int argc, char **argv[])
{
    /* <reference id="123"> */
    int i, j, k;
    /* </reference> */

    for (i = 0; i < N ; i++)
    {
        for (j = 0; j < N; j++)
        {
            A[i][j] = (double) i * j ;
            B[i][j] = (double) i * j ;
            C[i][j] = 0;
        }
    }

    /* <master id="123"> */
    _distributor = _open_space("distributor", 0, "123");
    _constructor = _open_space("constructor", 0, "123");
    _cleanup_space(_distributor, "123");
    _cleanup_space(_constructor, "123");

    /* <send var="B" type="double[N ][N ]" opt="ONCE"/> */
    sprintf(_tp_name, "double(%d) (%d):B#%s", (N), (N), "123");
    _tp_size = ((N) * (N)) * sizeof(double);
    _tp_B_123 = (double *)malloc(_tp_size);
    for (_x0_123 = 0; _x0_123 < (N); _x0_123 +=1) {
        for (_x1_123 = 0; _x1_123 < (N); _x1_123 +=1) {
            _tp_B_123[_x0_123 * (N) + _x1_123] =

```

```

        B[_x0_123][_x1_123];
    }
}
_status =
_send_data(_distributor, _tp_name, (char *)_tp_B_123, _tp_size);
if (_status < 0) exit(-1);
free(_tp_B_123);

/* <token action="SET" idxset="(i)"/> */
sprintf(_tp_name, "token#%s", "123");
sprintf(_tp_token, "(i:%d~%d,%d:#%d)", 0, N, 1, G);
_tp_size = sizeof(_tp_token);
_tokens =
_set_token(_distributor, _tp_name, (char *)_tp_token, _tp_size);
if (_tokens < 0) exit(-1);

/* <send var="A" type="double[N ][N ]"/> */
sprintf(_tp_name, "double(%d)(%d):A#%s", (N), (N), "123");
_tp_size = ((N) * (N)) * sizeof(double);
_tp_A_123 = (double *)malloc(_tp_size);
for (_x0_123 = 0; _x0_123 < (N); _x0_123 +=1) {
    for (_x1_123 = 0; _x1_123 < (N); _x1_123 +=1) {
        _tp_A_123[_x0_123 * (N) + _x1_123] =
            A[_x0_123][_x1_123];
    }
}
_status =
_send_data(_distributor, _tp_name, (char *)_tp_A_123, _tp_size);
if (_status < 0) exit(-1);
free(_tp_A_123);

/* <read var="C" type="double[N ][N ]"/> */
sprintf(_tp_name, "double(%d)(%d):C#%s", (N), (N), "123");
_tp_size = ((N) * (N)) * sizeof(double);
_tp_C_123 = (double *)malloc(_tp_size);
_tp_size =
_read_data(_constructor, _tp_name, (char *)_tp_C_123, _tp_size);
if (_tp_size < 0) exit(-1);
for (_x0_123 = 0; _x0_123 < (N); _x0_123 +=1) {
    for (_x1_123 = 0; _x1_123 < (N); _x1_123 +=1) {
        C[_x0_123][_x1_123] =
            _tp_C_123[_x0_123 * (N) + _x1_123];
    }
}
free(_tp_C_123);

_close_space(_constructor, "123", 1);
_close_space(_distributor, "123", 1);
/* </master> */

/*
for (i = 0; i < N; i++)
{
    for (j = 0; j < N; j++)
    {

```

```

        printf("%8.1f ", C[i][j]);
    }
    printf("\n");
}
*/
exit(0);
}
/* </parallel> */

```

- The Worker Program

```

/* =====
 * Compiler-Generated Worker Program, "matrix_123.c".
 */
#include "parallel.h"
/* <reference> */
#include "matrix.h"
/* </reference> */

/* <reference id="123"> */
int i, j, k;
/* </reference> */

/* <parallel appname="matrix"> */
double * _tp_B_123;
double * _tp_A_123;
double * _tp_C_123;
int _x0_123;
int _x1_123;
int _y0_123;
int _y1_123;
int _i_start = 0;
int _i_stop = 0;
int _i_step = 0;

main(int argc, char **argv[])
{
    /* <worker id="123"> */
    _distributor = _open_space("distributor", 0, "123");
    _constructor = _open_space("constructor", 0, "123");

    /* <read var="B" type="double[N ][N ]" opt="ONCE"/> */
    sprintf(_tp_name, "double(%d)(%d):B#%s", (N), (N), "123");
    _tp_size = ((N) * (N)) * sizeof(double);
    _tp_B_123 = (double *)malloc(_tp_size);
    _tp_size =
    _read_data(_distributor, _tp_name, (char *)_tp_B_123, _tp_size);
    if (_tp_size < 0) exit(-1);
    for (_x0_123 = 0; _x0_123 < (N); _x0_123 +=1) {
        for (_x1_123 = 0; _x1_123 < (N); _x1_123 +=1) {
            B[_x0_123][_x1_123] =
            _tp_B_123[_x0_123 * (N) + _x1_123];
        }
    }
}

```

```

}
free(_tp_B_123);

while (1)
{
    /* <token action="GET" idxset="(i)"/> */
    sprintf(_tp_name, "token#%s", "123");
    _tp_size = 0;
    _tp_size =
    _get_token(_distributor, _tp_name, (char *)_tp_token, _tp_size);
    if (_tp_size < 0) exit(-1);
    if (_tp_token[0] == '!') break;
    sscanf(_tp_token, "%d@(i:%d~%d,%d)",
           &_tokens, &_i_start, &_i_stop, &_i_step);

    /* <read var="A" type="double[N(i)][N ]"/> */
    sprintf(_tp_name, "double(%d) (%d):A#%s[%d~%d,%d] [%d~%d,%d]@%d",
           (N), (N), "123",
           _i_start, _i_stop, _i_step, 0, (N), 1,
           sizeof(double));
    _tp_size =
    (((_i_stop - _i_start - 1) / _i_step + 1) * (N)) *
    sizeof(double);
    _tp_A_123 = (double *)malloc(_tp_size);
    _tp_size =
    _read_data(_distributor, _tp_name, (char *)_tp_A_123,
              _tp_size);
    if (_tp_size < 0) exit(-1);
    for (_x0_123 = _i_start, _y0_123 = 0; _x0_123 < _i_stop;
         _x0_123 += _i_step, _y0_123++) {
        for (_x1_123 = 0; _x1_123 < (N); _x1_123 += 1) {
            A[_x0_123][_x1_123] =
            _tp_A_123[_y0_123 * (N) + _x1_123];
        }
    }
    free(_tp_A_123);

    /*<target index="i" order="1" limits="(0,N,1)" chunk="G">*/
    for (i = _i_start; i < _i_stop; i += _i_step)
    /*</target>*/
    {
        for (k = 0; k < N; k++)
        {
            for (j = 0; j < N; j++)
            {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }

    /* <send var="C" type="double[N(i)][N ]"/> */
    sprintf(_tp_name, "double(%d) (%d):C#%s[%d~%d,%d] [%d~%d,%d]@%d",
           (N), (N), "123",
           _i_start, _i_stop, _i_step, 0, (N), 1,
           sizeof(double));

```

```

    _tp_size =
    (((_i_stop - _i_start - 1) / _i_step + 1) * (N)) *
    sizeof(double);
    _tp_C_123 = (double *)malloc(_tp_size);
    for (_x0_123 = _i_start, _y0_123 = 0; _x0_123 < _i_stop;
        _x0_123 += _i_step, _y0_123++) {
        for (_x1_123 = 0; _x1_123 < (N); _x1_123 += 1) {
            _tp_C_123[_y0_123 * (N) + _x1_123] =
            C[_x0_123][_x1_123];
        }
    }
    _status =
    _send_data(_constructor, _tp_name, (char *)_tp_C_123,
              _tp_size);
    if (_status < 0) exit(-1);
    free(_tp_C_123);
}

_close_space(_constructor, "123", 0);
_close_space(_distributor, "123", 0);
/* </worker> */

exit(0);
}
/* </parallel> */

```

Appendix C – Parallel Laplacian Solver Using Gauss-Seidel Iteration

- The Master Program

```
/* =====
 * Compiler-Generated Master Program, "gauss_parallel.c".
 */
#include "parallel.h"
/* <reference> */
#include "gauss.h"
/* </reference> */

void inita(); /* initilizes A with initial guess */

/* <parallel appname="gauss"> */
double * _tp_A_123;
double * _tp_resid_123;
int _x0_123;
int _x1_123;
int _y0_123;
int _y1_123;

int main()
{
    rdim = DIM;

    inita();

    /* <master id="123"> */
    _distributor = _open_space("distributor", 0, "123");
    _constructor = _open_space("constructor", 0, "123");

    for (iterno= 1; iterno <= MAXITER; iterno++)
    {
        _cleanup_space(_distributor, "123");
        _cleanup_space(_constructor, "123");

        /* <send var="A" type="double[DIM+2][DIM+2]" opt="ONCE"/> */
        sprintf(_tp_name, "double(%d) (%d):A#%s",
            (DIM+2), (DIM+2), "123");
        _tp_size = ((DIM+2) * (DIM+2)) * sizeof(double);
        _tp_A_123 = (double *)malloc(_tp_size);
        for (_x0_123 = 0; _x0_123 < (DIM+2); _x0_123 +=1) {
            for (_x1_123 = 0; _x1_123 < (DIM+2); _x1_123 +=1) {
                _tp_A_123[_x0_123 * (DIM+2) + _x1_123] =
                    A[_x0_123][_x1_123];
            }
        }
        _status =
        _send_data(_distributor, _tp_name, (char *)_tp_A_123,
```

```

        _tp_size);
if (_status < 0) exit(-1);
free(_tp_A_123);

/* <token action="SET" idxset="(i)(j)"> */
sprintf(_tp_name, "token#%s", "123");
sprintf(_tp_token, "(i:%d~%d,%d:#%d)(j:%d~%d,%d:#%d)",
        1, rdim+1, 1, GRAIN, 1, DIM+1, 1, GRAIN);
_tp_size = sizeof(_tp_token);
_tokens =
_set_token(_distributor, _tp_name, (char *)_tp_token, _tp_size);
if (_tokens < 0) exit(-1);

/* <send var="A" type="double[DIM+2(0~1)][DIM+2(0~DIM+1)]"
    opt="XCHG"> */
sprintf(_tp_name, "double(%d)(%d):A#%s[%d~%d,%d][%d~%d,%d]@%d",
        (DIM+2), (DIM+2), "123", (0), (1), 1, (0), (DIM+1), 1,
        sizeof(double));
_tp_size =
(((1) - (0)) * ((DIM+1) - (0))) * sizeof(double);
_tp_A_123 = (double *)malloc(_tp_size);
for (_x0_123 = (0), _y0_123 = 0; _x0_123 < (1);
    _x0_123 += 1, _y0_123 += 1) {
    for (_x1_123 = (0), _y1_123 = 0; _x1_123 < (DIM+1);
        _x1_123 += 1, _y1_123 += 1) {
        _tp_A_123[_y0_123 * ((DIM+1) - (0)) + _y1_123] =
        A[_x0_123][_x1_123];
    }
}
_status =
_send_data(_constructor, _tp_name, (char *)_tp_A_123, _tp_size);
if (_status < 0) exit(-1);
free(_tp_A_123);

/* <send var="A"
    type="double[DIM+2(0~DIM+1)][DIM+2(DIM+1~DIM+2)]"
    opt="XCHG"> */
sprintf(_tp_name, "double(%d)(%d):A#%s[%d~%d,%d][%d~%d,%d]@%d",
        (DIM+2), (DIM+2), "123", (0), (DIM+1), 1, (DIM+1), (DIM+2), 1,
        sizeof(double));
_tp_size =
(((DIM+1) - (0)) * ((DIM+2) - (DIM+1))) * sizeof(double);
_tp_A_123 = (double *)malloc(_tp_size);
for (_x0_123 = (0), _y0_123 = 0; _x0_123 < (DIM+1);
    _x0_123 += 1, _y0_123 += 1) {
    for (_x1_123 = (DIM+1), _y1_123 = 0; _x1_123 < (DIM+2);
        _x1_123 += 1, _y1_123 += 1) {
        _tp_A_123[_y0_123 * ((DIM+2) - (DIM+1)) + _y1_123] =
        A[_x0_123][_x1_123];
    }
}
_status =
_send_data(_constructor, _tp_name, (char *)_tp_A_123, _tp_size);
if (_status < 0) exit(-1);
free(_tp_A_123);

```

```

/* <send var="A"
   type="double [DIM+2 (DIM+1~DIM+2) ] [DIM+2 (1~DIM+2) ]"
   opt="XCHG"/> */
sprintf(_tp_name, "double(%d) (%d):A#%s[%d~%d,%d][%d~%d,%d]@%d",
        (DIM+2), (DIM+2), "123", (DIM+1), (DIM+2), 1, (1), (DIM+2), 1,
        sizeof(double));
_tp_size =
    (((DIM+2) - (DIM+1)) * ((DIM+2) - (1))) * sizeof(double);
_tp_A_123 = (double *)malloc(_tp_size);
for (_x0_123 = (DIM+1), _y0_123 =0; _x0_123 < (DIM+2);
    _x0_123 +=1, _y0_123 ++) {
    for (_x1_123 = (1), _y1_123 =0; _x1_123 < (DIM+2);
        _x1_123 +=1, _y1_123 ++) {
        _tp_A_123[_y0_123 * ((DIM+2) - (1)) + _y1_123] =
            A[_x0_123][_x1_123];
    }
}
_status =
_send_data(_constructor, _tp_name, (char *)_tp_A_123, _tp_size);
if (_status < 0) exit(-1);
free(_tp_A_123);

/* <send var="A" type="double [DIM+2 (1~DIM+2) ] [DIM+2 (0~1) ]"
   opt="XCHG"/> */
sprintf(_tp_name, "double(%d) (%d):A#%s[%d~%d,%d][%d~%d,%d]@%d",
        (DIM+2), (DIM+2), "123", (1), (DIM+2), 1, (0), (1), 1,
        sizeof(double));
_tp_size =
    (((DIM+2) - (1)) * ((1) - (0))) * sizeof(double);
_tp_A_123 = (double *)malloc(_tp_size);
for (_x0_123 = (1), _y0_123 =0; _x0_123 < (DIM+2);
    _x0_123 +=1, _y0_123 ++) {
    for (_x1_123 = (0), _y1_123 =0; _x1_123 < (1);
        _x1_123 +=1, _y1_123 ++) {
        _tp_A_123[_y0_123 * ((1) - (0)) + _y1_123] =
            A[_x0_123][_x1_123];
    }
}
_status =
_send_data(_constructor, _tp_name, (char *)_tp_A_123, _tp_size);
if (_status < 0) exit(-1);
free(_tp_A_123);

/* <read var="A" type="double [DIM+2] [DIM+2]" /> */
sprintf(_tp_name, "double(%d) (%d):A#%s",
        (DIM+2), (DIM+2), "123");
_tp_size = ((DIM+2) * (DIM+2)) * sizeof(double);
_tp_A_123 = (double *)malloc(_tp_size);
_tp_size =
_read_data(_constructor, _tp_name, (char *)_tp_A_123, _tp_size);
if (_tp_size < 0) exit(-1);
for (_x0_123 = 0; _x0_123 < (DIM+2); _x0_123 +=1) {
    for (_x1_123 = 0; _x1_123 < (DIM+2); _x1_123 +=1) {
        A[_x0_123][_x1_123] =

```

```

        _tp_A_123[_x0_123 * (DIM+2) + _x1_123];
    }
}
free(_tp_A_123);

/* <read var="resid" type="double" opt="_MAX"/> */
sprintf(_tp_name, "double:resid#%s?MAX@%d", "123", _tokens);
_tp_size = sizeof(double);
_tp_resid_123 = &resid;
_tp_size =
_read_data(_constructor, _tp_name, (char *)_tp_resid_123,
           _tp_size);
if (_tp_size < 0) exit(-1);

if (resid < EPS) break;    /* termination of iterations */
}

_close_space(_constructor, "123", 1);
_close_space(_distributor, "123", 1);
/* </master> */

/*
for (i = 0; i < DIM+2; i++)
{
    for (j = 0; j < DIM+2; j++)
    {
        printf("%8.1f ", A[i][j]);
    }
    printf("\n");
}
*/
return 0;
}
/* </parallel> */

void inita()
{
    for (i = 0; i < rdim+2; i++)    /* initial guess */
        for (j = 0; j < DIM+2; j++)
            A[i][j] = 0.0;
    for (i = 0; i < rdim+2; i++)    /* set boundaries */
    {
        A[i][0] = LFTBOUND;
        A[i][DIM+1] = RITBOUND;
    }
    for (j = 0; j < DIM+2; j++)
        A[0][j] = TOPBOUND;
    for (j = 0; j < DIM+2; j++)
        A[rdim+1][j] = BOTBOUND;
}

```

- The Worker Program

```

/* =====
 * Compiler-Generated Worker Program, "gauss_123.c".
 */
#include "parallel.h"

/* <reference> */
#include "gauss.h"
/* </reference> */

/* <parallel appname="gauss"> */
double * _tp_A_123;
double * _tp_resid_123;
int _x0_123;
int _x1_123;
int _y0_123;
int _y1_123;
int _i_start = 0;
int _i_stop = 0;
int _i_step = 0;
int _j_start = 0;
int _j_stop = 0;
int _j_step = 0;

main(int argc, char **argv[])
{
    /* <worker id="123"> */
    _distributor = _open_space("distributor", 0, "123");
    _constructor = _open_space("constructor", 0, "123");

    while (1)
    {
        /* <token action="GET" idxset="(i)(j)"> */
        sprintf(_tp_name, "token#%s", "123");
        _tp_size = 0;
        _tp_size =
        _get_token(_distributor, _tp_name, (char *)_tp_token, _tp_size);
        if (_tp_size < 0) exit(-1);
        if (_tp_token[0] == '!') break;
        sscanf(_tp_token, "%d@(i:%d~%d,%d)(j:%d~%d,%d)",
            &_tokens,
            &_i_start, &_i_stop, &_i_step,
            &_j_start, &_j_stop, &_j_step);

        /* <read var="A" type="double[DIM+2][DIM+2]"> */
        sprintf(_tp_name, "double(%d)(%d):A#%s",
            (DIM+2), (DIM+2), "123");
        _tp_size = ((DIM+2) * (DIM+2)) * sizeof(double);
        _tp_A_123 = (double *)malloc(_tp_size);
        _tp_size =
        _read_data(_distributor, _tp_name, (char *)_tp_A_123,
            _tp_size);
        if (_tp_size < 0) exit(-1);
        for (_x0_123 = 0; _x0_123 < (DIM+2); _x0_123 +=1) {
            for (_x1_123 = 0; _x1_123 < (DIM+2); _x1_123 +=1) {
                A[_x0_123][_x1_123] =

```

```

        _tp_A_123[_x0_123 * (DIM+2) + _x1_123];
    }
}
free(_tp_A_123);

/* <read var="A" type="double [DIM+2(i:$L-1)] [DIM+2(j)]"
   opt="XCHG"/> */
sprintf(_tp_name, "double(%d) (%d):A#%s[%d~%d,%d] [%d~%d,%d]@%d",
        (DIM+2), (DIM+2), "123",
        (_i_start-1), _i_start, _i_step,
        _j_start, _j_stop, _j_step,
        sizeof(double));
_tp_size =
(((_i_start - (_i_start-1) - 1) / _i_step + 1) *
 ((_j_stop - _j_start - 1) / _j_step + 1)) *
sizeof(double);
_tp_A_123 = (double *)malloc(_tp_size);
_tp_size =
_read_data(_constructor, _tp_name, (char *)_tp_A_123, _tp_size);
if (_tp_size < 0) exit(-1);
for (_x0_123 = (_i_start-1), _y0_123 = 0; _x0_123 < _i_start;
    _x0_123 += _i_step, _y0_123++) {
    for (_x1_123 = _j_start, _y1_123 = 0; _x1_123 < _j_stop;
        _x1_123 += _j_step, _y1_123++) {
        A[_x0_123][_x1_123] =
        _tp_A_123[_y0_123 *
        ((_j_stop - _j_start - 1) / _j_step + 1) +
        _y1_123];
    }
}
free(_tp_A_123);

/* <read var="A" type="double [DIM+2(i)] [DIM+2(j:$L-1)]"
   opt="XCHG"/> */
sprintf(_tp_name, "double(%d) (%d):A#%s[%d~%d,%d] [%d~%d,%d]@%d",
        (DIM+2), (DIM+2), "123",
        _i_start, _i_stop, _i_step,
        (_j_start-1), _j_start, _j_step,
        sizeof(double));
_tp_size =
((( _i_stop - _i_start - 1) / _i_step + 1) *
 (( _j_start - (_j_start-1) - 1) / _j_step + 1)) *
sizeof(double);
_tp_A_123 = (double *)malloc(_tp_size);
_tp_size =
_read_data(_constructor, _tp_name, (char *)_tp_A_123, _tp_size);
if (_tp_size < 0) exit(-1);
for (_x0_123 = _i_start, _y0_123 = 0; _x0_123 < _i_stop;
    _x0_123 += _i_step, _y0_123++) {
    for (_x1_123 = (_j_start-1), _y1_123 = 0; _x1_123 < _j_start;
        _x1_123 += _j_step, _y1_123++) {
        A[_x0_123][_x1_123] =
        _tp_A_123[_y0_123 *
        ((_j_start - (_j_start-1) - 1) / _j_step + 1) +
        _y1_123];
    }
}

```

```

    }
}
free(_tp_A_123);

resid = 0.0;

/*<target index="i" order="1" limits="(1,rdim+1,1)"
    chunk="GRAIN">*/
for (i = _i_start; i < _i_stop; i +=_i_step)
/*</target>*/
{
    /*<target index="j" order="1" limits="(1,DIM+1,1)"
        chunk="GRAIN">*/
    for (j = _j_start; j < _j_stop; j +=_j_step)
    /*</target>*/
    {
        Atmp      = A[i][j];
        A[i][j]   = 0.25 * (A[i+1][j] + A[i-1][j] +
                           A[i][j+1] + A[i][j-1]);
        residtmp =
            (Atmp!=0.0? fabs(fabs(A[i][j]-Atmp)/Atmp) : (1.0+EPS));
        if (residtmp > resid) resid = residtmp;
    }
}

/* <send var="resid" type="double" opt=" MAX"/> */
sprintf(_tp_name, "double:resid#%s?MAX@%d", "123", _tokens);
_tp_size = sizeof(double);
_tp_resid_123 = &resid;
_status =
_send_data(_constructor, _tp_name, (char *)_tp_resid_123,
           _tp_size);
if (_status < 0) exit(-1);

/* <send var="A" type="double[DIM+2(i) ][DIM+2(j) ]"/> */
sprintf(_tp_name, "double(%d) (%d):A#%s[%d~%d,%d][%d~%d,%d]@%d",
        (DIM+2), (DIM+2), "123",
        _i_start, _i_stop, _i_step,
        _j_start, _j_stop, _j_step,
        sizeof(double));
_tp_size =
(((_i_stop - _i_start - 1) / _i_step + 1) *
 ((_j_stop - _j_start - 1) / _j_step + 1)) *
sizeof(double);
_tp_A_123 = (double *)malloc(_tp_size);
for (_x0_123 = _i_start, _y0_123 =0; _x0_123 < _i_stop;
     _x0_123 +=_i_step, _y0_123 ++) {
    for (_x1_123 = _j_start, _y1_123 =0; _x1_123 < _j_stop;
         _x1_123 +=_j_step, _y1_123 ++) {
        _tp_A_123[_y0_123 *
            ((_j_stop - _j_start - 1) / _j_step + 1) +
            _y1_123] =
            A[_x0_123][_x1_123];
    }
}
}

```

```
        _status =
        _send_data(_constructor, _tp_name, (char *)_tp_A_123, _tp_size);
        if (_status < 0) exit(-1);
        free(_tp_A_123);
    }

    _close_space(_constructor, "123", 0);
    _close_space(_distributor, "123", 0);
    /* </worker> */

    exit(0);
}
/* </parallel> */
```

Appendix D – Parallel Programs For Ion Generation Simulator

- The Master Program

```
/* =====
 * Compiler-Generated Master Program, "ion_parallel.c".
 */
#include "parallel.h"
/* <reference> */
# include "ion.h"
/* </reference> */

/* <parallel appname="ion"> */
float * _tp_v_123;
int _x0_123;
int _x1_123;
int _y0_123;
int _y1_123;
int _I_start = 0;
int _I_stop = 0;
int _I_step = 0;
int _J_start = 0;
int _J_stop = 0;
int _J_step = 0;

int main()
{
    /* <reference> */
    int I,J,K, COUNT;
    float V[MAXI+1][MAXJ+2];
    float R1S, R2S, R3, A, B, C, URAXP4, RI, RJ;
    float SIGPX, SIGMX, SIGPY, SIGMY;
    /* </reference> */

    long t0, t1;                /* start and end elapsed time */

    char FIELD[MAXI][MAXJ];
    char MARK[20];

    strcpy(MARK,"abcdefghijklmnopqrst");

    for (I=1; I<=MAXI; I++)
    {
        for (J=1; J<=MAXJ; J++)
        {
            R1S=sqrt((float)(pow((IP1-I),2)+pow((JP1-J),2)));
            R2S=sqrt((float)(pow((IP2-I),2)+pow((JP2-J),2)));
            R3=(float)(MAXI-I);
            A=R1S*R2S;
            B=R1S*R3;
        }
    }
}
```

```

        C=R2S*R3;
        V[I][J]=(VP2*B+VP1*C)/(A+B+C);
    }
}

for (J=TLFT; J<=TRGT; J++)
{
    V[TLEV][J] = 0.0;
}

t0 = time ((long*)0);

/* <master id="123"> */
_distributor = _open_space("distributor", 0, "123");
_constructor = _open_space("constructor", 0, "123");

for (COUNT=1; COUNT<=ITER; COUNT++)
{
    for (J=1; J<=MAXJ; J++)
    {
        V[0][J]=V[1][J];
    }
    for (I=0; I<=MAXI-1; I++)
    {
        V[I][0]=V[I][1];
        V[I][MAXJ+1]=V[I][MAXJ];
    }

    _cleanup_space(_distributor, "123");
    _cleanup_space(_constructor, "123");

    /* <send var="V" type="float[MAXI+1][MAXJ+2]" opt="ONCE"/> */
    sprintf(_tp_name, "float(%d)(%d):V#%s",
        (MAXI+1), (MAXJ+2), "123");
    _tp_size = ((MAXI+1) * (MAXJ+2)) * sizeof(float);
    _tp_V_123 = (float *)malloc(_tp_size);
    for (_x0_123 = 0; _x0_123 < (MAXI+1); _x0_123 +=1) {
        for (_x1_123 = 0; _x1_123 < (MAXJ+2); _x1_123 +=1) {
            _tp_V_123[_x0_123 * (MAXJ+2) + _x1_123] =
                V[_x0_123][_x1_123];
        }
    }
    _status =
    _send_data(_distributor, _tp_name, (char *)_tp_V_123,
        _tp_size);
    if (_status < 0) exit(-1);
    free(_tp_V_123);

    /* <token action="SET" idxset="(I)(J)"/> */
    sprintf(_tp_name, "token#%s", "123");
    sprintf(_tp_token, "(I:%d~%d,%d:#%d)(J:%d~%d,%d:#%d)",
        1, MAXI, 1, GRAIN, 1, MAXJ+1, 1, GRAIN);
    _tp_size = sizeof(_tp_token);
    _tokens =
    _set_token(_distributor, _tp_name, (char *)_tp_token, _tp_size);
}

```

```

if (_tokens < 0) exit(-1);

/* <send var="V" type="float[MAXI+1(0~1)] [MAXJ+2(0~MAXJ+1)]"
   opt="XCHG"/> */
sprintf(_tp_name, "float(%d) (%d):V#%s[%d~%d,%d][%d~%d,%d]@%d",
        (MAXI+1), (MAXJ+2), "123", (0), (1), 1,
        (0), (MAXJ+1), 1, sizeof(float));
_tp_size =
(((1) - (0) - 1) / 1 + 1) *
(((MAXJ+1) - (0) - 1) / 1 + 1)) *
sizeof(float);
_tp_V_123 = (float *)malloc(_tp_size);
for (_x0_123 = (0), _y0_123 = 0; _x0_123 < (1);
    _x0_123 += 1, _y0_123 += 1) {
    for (_x1_123 = (0), _y1_123 = 0; _x1_123 < (MAXJ+1);
        _x1_123 += 1, _y1_123 += 1) {
        _tp_V_123[_y0_123 *
            (((MAXJ+1) - (0) - 1) / 1 + 1) +
            _y1_123] = V[_x0_123][_x1_123];
    }
}
_status =
_send_data(_constructor, _tp_name, (char *)_tp_V_123, _tp_size);
if (_status < 0) exit(-1);
free(_tp_V_123);

/* <send var="V"
   type="float[MAXI+1(0~MAXI)] [MAXJ+2(MAXJ+1~MAXJ+2)]"
   opt="XCHG"/> */
sprintf(_tp_name, "float(%d) (%d):V#%s[%d~%d,%d][%d~%d,%d]@%d",
        (MAXI+1), (MAXJ+2), "123", (0), (MAXI), 1,
        (MAXJ+1), (MAXJ+2), 1, sizeof(float));
_tp_size =
((((MAXI) - (0) - 1) / 1 + 1) *
(((MAXJ+2) - (MAXJ+1) - 1) / 1 + 1)) *
sizeof(float);
_tp_V_123 = (float *)malloc(_tp_size);
for (_x0_123 = (0), _y0_123 = 0; _x0_123 < (MAXI);
    _x0_123 += 1, _y0_123 += 1) {
    for (_x1_123 = (MAXJ+1), _y1_123 = 0; _x1_123 < (MAXJ+2);
        _x1_123 += 1, _y1_123 += 1) {
        _tp_V_123[_y0_123 *
            (((MAXJ+2) - (MAXJ+1) - 1) / 1 + 1) +
            _y1_123] = V[_x0_123][_x1_123];
    }
}
_status =
_send_data(_constructor, _tp_name, (char *)_tp_V_123, _tp_size);
if (_status < 0) exit(-1);
free(_tp_V_123);

/* <send var="V"
   type="float[MAXI+1(MAXI~MAXI+1)] [MAXJ+2(1~MAXJ+2)]"
   opt="XCHG"/> */
sprintf(_tp_name, "float(%d) (%d):V#%s[%d~%d,%d][%d~%d,%d]@%d",

```

```

        (MAXI+1), (MAXJ+2), "123", (MAXI), (MAXI+1), 1,
        (1), (MAXJ+2), 1, sizeof(float));
    _tp_size =
    (((MAXI+1) - (MAXI) - 1) / 1 + 1) *
    (((MAXJ+2) - (1) - 1) / 1 + 1) *
    sizeof(float);
    _tp_V_123 = (float *)malloc(_tp_size);
    for (_x0_123 = (MAXI), _y0_123 = 0; _x0_123 < (MAXI+1);
        _x0_123 += 1, _y0_123++) {
        for (_x1_123 = (1), _y1_123 = 0; _x1_123 < (MAXJ+2);
            _x1_123 += 1, _y1_123++) {
            _tp_V_123[_y0_123 *
                ((MAXJ+2) - (1) - 1) / 1 + 1) +
                _y1_123] = V[_x0_123][_x1_123];
        }
    }
    _status =
    _send_data(_constructor, _tp_name, (char *)_tp_V_123, _tp_size);
    if (_status < 0) exit(-1);
    free(_tp_V_123);

    /* <send var="V"
        type="float[MAXI+1(1~MAXI+1)][MAXJ+2(0~1)]"
        opt="XCHG"/> */
    sprintf(_tp_name, "float(%d)(%d):V#%s[%d~%d,%d][%d~%d,%d]@%d",
        (MAXI+1), (MAXJ+2), "123", (1), (MAXI+1), 1,
        (0), (1), 1, sizeof(float));
    _tp_size =
    (((MAXI+1) - (1) - 1) / 1 + 1) *
    (((1) - (0) - 1) / 1 + 1) *
    sizeof(float);
    _tp_V_123 = (float *)malloc(_tp_size);
    for (_x0_123 = (1), _y0_123 = 0; _x0_123 < (MAXI+1);
        _x0_123 += 1, _y0_123++) {
        for (_x1_123 = (0), _y1_123 = 0; _x1_123 < (1);
            _x1_123 += 1, _y1_123++) {
            _tp_V_123[_y0_123 *
                (((1) - (0) - 1) / 1 + 1) +
                _y1_123] = V[_x0_123][_x1_123];
        }
    }
    _status =
    _send_data(_constructor, _tp_name, (char *)_tp_V_123, _tp_size);
    if (_status < 0) exit(-1);
    free(_tp_V_123);

    /* <read var="V" type="float[MAXI+1][MAXJ+2]"/> */
    sprintf(_tp_name, "float(%d)(%d):V#%s",
        (MAXI+1), (MAXJ+2), "123");
    _tp_size = ((MAXI+1) * (MAXJ+2)) * sizeof(float);
    _tp_V_123 = (float *)malloc(_tp_size);
    _tp_size =
    _read_data(_constructor, _tp_name, (char *)_tp_V_123, _tp_size);
    if (_tp_size < 0) exit(-1);
    for (_x0_123 = 0; _x0_123 < (MAXI+1); _x0_123 += 1) {

```

```

        for (_x1_123 = 0; _x1_123 < (MAXJ+2); _x1_123 +=1) {
            V[_x0_123][_x1_123] =
                _tp_V_123[_x0_123 * (MAXJ+2) + _x1_123];
        }
    }
    free(_tp_V_123);
}

_close_space(_constructor, "123", 1);
_close_space(_distributor, "123", 1);
/* </master> */

for (I=1; I<=MAXI-1; I++)
{
    strcpy(FIELD[I-1] , " ");
    for (J=1; J<=MAXJ; J++)
    {
        for (K=1; K<=NDVP; K++)
        {
            if (abs(V[I][J]-DVP[K-1]) < abs(DVP[K-1]/20.0))
                FIELD[I-1][J-1] = MARK[K-1];
        }
    }
}

FIELD[IP1-1][JP1-1]='*';
FIELD[IP2-1][JP2-1]='*';

strcpy(&FIELD[TLEV-1][TLFT-
1],"1*****");

for (J=1; J<=MAXJ; J++)
{
    FIELD[MAXI-1][J-1]='*';
}

for (I=1; I<=MAXI; I++)
{
    for (J=MAXJ; J>=2; J--)
    {
        if (FIELD[I-1][J-1] != ' ')
            break;
    }

    printf("( %s)\n", FIELD[I-1]);
}

t1 = time((long *)0) - t0;

printf ("\nTime: (%d), MAXI(J): (%d), Iters: (%d)\n",
        t1, MAXI, ITER);

printf("End of Lapsig.c\n");
exit(0);

```

```

}
/* </parallel> */

/* <reference> */
float SIGMA_f(float RI, float RJ)
{
    float RIP1, RJP1, RIP2, RJP2;
    float D1, D2;
    float SIGMA;

    RIP1 = (float)IP1;
    RJP1 = (float)JP1;

    RIP2 = (float)IP2;
    RJP2 = (float)JP2;

    D1=sqrt(pow((RI-RIP1),2)+pow((RJ-RJP1),2));
    D2=sqrt(pow((RI-RIP2),2)+pow((RJ-RJP2),2));
    SIGMA=1.0/max(1.0,min(D1,D2));

    return (SIGMA);
}

float max(float x, float y)
{
    if (x > y) return x;
    return y;
}

float min(float x, float y)
{
    if (x < y) return x;
    return y;
}
/* </reference> */

```

- The Worker Program

```

/* =====
 * Compiler-Generated Worker Program, "ion_123.c".
 */
#include "parallel.h"

/* <reference> */
# include "ion.h"
/* </reference> */

/* <reference> */
int I, J, K, COUNT;
float V[MAXI+1][MAXJ+2];
float R1S, R2S, R3, A, B, C, URAXP4, RI, RJ;
float SIGPX, SIGMX, SIGPY, SIGMY;
/* </reference> */

```

```

/* <reference> */
float SIGMA_f(float RI,float RJ)
{
    float RIP1, RJP1, RIP2, RJP2;
    float D1,D2;
    float SIGMA;

    RIP1 = (float)IP1;
    RJP1 = (float)JP1;

    RIP2 = (float)IP2;
    RJP2 = (float)JP2;

    D1=sqrt(pow((RI-RIP1),2)+pow((RJ-RJP1),2));
    D2=sqrt(pow((RI-RIP2),2)+pow((RJ-RJP2),2));
    SIGMA=1.0/max(1.0,min(D1,D2));

    return (SIGMA);
}

float max(float x, float y)
{
    if (x > y) return x;
    return y;
}

float min(float x, float y)
{
    if (x < y) return x;
    return y;
}
/* </reference> */

/* <parallel appname="ion"> */
float *_tp_V_123;
int _x0_123;
int _x1_123;
int _y0_123;
int _y1_123;
int _I_start = 0;
int _I_stop = 0;
int _I_step = 0;
int _J_start = 0;
int _J_stop = 0;
int _J_step = 0;

main(int argc, char **argv[])
{
    /* <worker id="123"> */
    _distributor = _open_space("distributor", 0, "123");
    _constructor = _open_space("constructor", 0, "123");

    while (1)
    {

```

```

/* <token action="GET" idxset="(I) (J)"/> */
sprintf(_tp_name, "token#%s", "123");
_tp_size = 0;
_tp_size =
_get_token(_distributor, _tp_name, (char *)_tp_token, _tp_size);
if (_tp_size < 0) exit(-1);
if (_tp_token[0] == '!') break;
sscanf(_tp_token, "%d@(I:%d~%d,%d) (J:%d~%d,%d)",
      &_tokens,
      &_I_start, &_I_stop, &_I_step,
      &_J_start, &_J_stop, &_J_step);

/* <read var="V" type="float[MAXI+1][MAXJ+2]"/> */
sprintf(_tp_name, "float(%d) (%d):V#%s",
      (MAXI+1), (MAXJ+2), "123");
_tp_size = ((MAXI+1) * (MAXJ+2)) * sizeof(float);
_tp_V_123 = (float *)malloc(_tp_size);
_tp_size =
_read_data(_distributor, _tp_name, (char *)_tp_V_123,
      _tp_size);
if (_tp_size < 0) exit(-1);
for (_x0_123 = 0; _x0_123 < (MAXI+1); _x0_123 +=1) {
  for (_x1_123 = 0; _x1_123 < (MAXJ+2); _x1_123 +=1) {
    V[_x0_123][_x1_123] =
      _tp_V_123[_x0_123 * (MAXJ+2) + _x1_123];
  }
}
free(_tp_V_123);

/* <read var="V" type="float[MAXI+1 (I:$L-1)][MAXJ+2 (J)]"
      opt="XCHG"/> */
sprintf(_tp_name, "float(%d) (%d):V#s[%d~%d,%d][%d~%d,%d]@%d",
      (MAXI+1), (MAXJ+2), "123",
      (_I_start-1), _I_start, 1,
      _J_start, _J_stop, 1,
      sizeof(float));
_tp_size =
  (((_I_start - (_I_start-1) - 1) / 1 + 1) *
  ((_J_stop - _J_start - 1) / 1 + 1)) * sizeof(float);
_tp_V_123 = (float *)malloc(_tp_size);
_tp_size =
_read_data(_constructor, _tp_name, (char *)_tp_V_123, _tp_size);
if (_tp_size < 0) exit(-1);
for (_x0_123 = (_I_start-1), _y0_123 = 0; _x0_123 < _I_start;
    _x0_123 +=1, _y0_123 +=1) {
  for (_x1_123 = _J_start, _y1_123 = 0; _x1_123 < _J_stop;
      _x1_123 +=1, _y1_123 +=1) {
    V[_x0_123][_x1_123] = _tp_V_123[_y0_123 *
      ((_J_stop - _J_start - 1) / 1 + 1) +
      _y1_123];
  }
}
free(_tp_V_123);

/* <read var="V" type="float[MAXI+1 (I)][MAXJ+2 (J:$L-1)]"

```

```

        opt="XCHG"/> */
    sprintf(_tp_name, "float(%d) (%d):V#%s[%d~%d,%d][%d~%d,%d]@%d",
        (MAXI+1), (MAXJ+2), "123",
        _I_start, _I_stop, 1,
        (_J_start-1), _J_start, 1,
        sizeof(float));
    _tp_size =
    (((_I_stop - _I_start - 1) / 1 + 1) *
    ((_J_start - (_J_start-1) - 1) / 1 + 1)) *
    sizeof(float);
    _tp_V_123 = (float *)malloc(_tp_size);
    _tp_size =
    _read_data(_constructor, _tp_name, (char *)_tp_V_123, _tp_size);
    if (_tp_size < 0) exit(-1);
    for (_x0_123 = _I_start, _y0_123 = 0; _x0_123 < _I_stop;
        _x0_123 +=1, _y0_123 ++) {
        for (_x1_123 = (_J_start-1), _y1_123 = 0; _x1_123 < _J_start;
            _x1_123 +=1, _y1_123 ++) {
            V[_x0_123][_x1_123] =
            _tp_V_123[_y0_123] *
            ((_J_start - (_J_start-1) - 1) / 1 + 1) +
            _y1_123];
        }
    }
    free(_tp_V_123);

    /* <target index ="I" order ="1" limits="(1,MAXI,1)"
        chunk ="GRAIN"> */
    for (I = _I_start; I < _I_stop; I +=_I_step)
    /* </target> */
    {
        /* <target index ="J" order ="1" limits="(1,MAXJ+1,1)"
            chunk ="GRAIN"> */
        for (J = _J_start; J < _J_stop; J +=_J_step)
        /* </target> */
        {
            if (((I==IP1) && (J==JP1)) || ((I==IP2) && (J==JP2)) ||
                ((I==TLEV) && (J>=TLFT) && (J<=TRGT)))
                continue;

            RI=(float) (I);
            RJ=(float) (J);

            SIGPX=SIGMA_f(RI+0.5,RJ);
            SIGMX=SIGMA_f(RI-0.5,RJ);
            SIGPY=SIGMA_f(RI,RJ+0.5);
            SIGMY=SIGMA_f(RI,RJ-0.5);

            V[I][J]= (URAX*V[I][J]
            +(SIGMY*V[I][J-1]+SIGPY*V[I][J+1]
            +SIGMX*V[I-1][J]+SIGPX*V[I+1][J]))
            / (URAX+SIGPX+SIGMX+SIGPY+SIGMY);
        }
    }
}

```

```

/* <send var="V" type="float[MAXI+1(I)][MAXJ+2(J)]"/> */
sprintf(_tp_name, "float(%d)(%d):V#%s[%d~%d,%d][%d~%d,%d]@%d",
        (MAXI+1), (MAXJ+2), "123",
        _I_start, _I_stop, 1,
        _J_start, _J_stop, 1,
        sizeof(float));
_tp_size =
(((_I_stop - _I_start - 1) / 1 + 1) *
((_J_stop - _J_start - 1) / 1 + 1)) *
sizeof(float);
_tp_V_123 = (float *)malloc(_tp_size);
for (_x0_123 = _I_start, _y0_123 = 0; _x0_123 < _I_stop;
    _x0_123 +=1, _y0_123 +=) {
    for (_x1_123 = _J_start, _y1_123 = 0; _x1_123 < _J_stop;
        _x1_123 +=1, _y1_123 +=) {
        _tp_V_123[_y0_123 *
            ((_J_stop - _J_start - 1) / 1 + 1) +
            _y1_123] =
            V[_x0_123][_x1_123];
    }
}
_status =
_send_data(_constructor, _tp_name, (char *)_tp_V_123, _tp_size);
if (_status < 0) exit(-1);
free(_tp_V_123);
}

_close_space(_constructor, "123", 0);
_close_space(_distributor, "123", 0);
/* </worker> */

exit(0);
}
/* </parallel> */

```

Appendix E – Parallel Programs For Block LU Factorization

- The Master Program

```
/* =====
 * Compiler-Generated Master Program, "blu_parallel.c".
 */
#include "parallel.h"
/* <reference> */
#include "blu.h"
/* </reference> */

/* <parallel appname="blu"> */
int * _tp_i_123;
float * _tp_outMat_123;
int _x0_123;
int _y0_123;
int _x1_123;
int _y1_123;
int _k1_start = 0;
int _k1_stop = 0;
int _k1_step = 0;

int main()
{
    /* <reference> */
    int i, j, dist, k1, k2, p1, p2, q1, q2, q3;
    int subdist, rowdist, coldist;
    float inSubMat[M][M], outSubMat[M][M], LU[M][M], L[M][M], U[M][M];
    /* </reference> */

    for (i=0; i<N; i++)
    {
        for (j=0; j<N; j++)
        {
            if (i==j)
            {
                outMat[i][j] = N;
            }
            else
            {
                outMat[i][j] = 1;
            }
        }
    }

    /* <master id="123"> */
    _distributor = _open_space("distributor", 0, "123");
    _constructor = _open_space("constructor", 0, "123");
}
```

```

for (i = 0; i < N; i = i + M)
{
    j = i + M - 1;
    dist = M;
    if (j > N-1)
    {
        j = N-1;
        dist = N - i;
    }

    // LU factors for submatrix
    for (k1 = 0; k1 < dist; k1++)
    {
        for (k2 = 0; k2 < dist; k2++)
        {
            inSubMat[k1][k2] = outMat[i+k1][i+k2];
        }
    }
    LUFactor(inSubMat, outSubMat, dist);
    //update
    for (k1 = 0; k1 < dist; k1++)
    {
        for (k2 = 0; k2 < dist; k2++)
        {
            outMat[i+k1][i+k2] = outSubMat[k1][k2];
            LU[k1][k2] = outSubMat[k1][k2];
        }
    }

    for (k1 = j + 1; k1 < N; k1 = k1 + M)
    {
        k2 = k1 + M - 1;
        subdist = M;
        if (k2 > N-1)
        {
            k2 = N - 1;
            subdist = N - k1;
        }

        //Solve LZ
        for (p1 = i; p1 < i+M; p1++)
        {
            for (p2 = k1; p2 <= k2; p2++)
            {
                inSubMat[p1-i][p2-k1] = outMat[p1][p2];
            }
        }
        TriangleSolver(LU, inSubMat, outSubMat, subdist, 1);
        //update
        for (p1 = i; p1 < i+M; p1++)
        {
            for (p2 = k1; p2 <= k2; p2++)
            {
                outMat[p1][p2] = outSubMat[p1-i][p2-k1];
            }
        }
    }
}

```

```

    }

    //Solve WU
    for (p1 = i; p1 < i+M; p1++)
    {
        for (p2 = k1; p2 <= k2; p2++)
        {
            inSubMat[p2-k1][p1-i] = outMat[p2][p1];
        }
    }
    TriangleSolver(LU, inSubMat, outSubMat, subdist, 2);
    //update
    for (p1 = i; p1 < i+M; p1++)
    {
        for (p2 = k1; p2 <= k2; p2++)
        {
            outMat[p2][p1] = outSubMat[p2-k1][p1-i];
        }
    }
}

_cleanup_space(_distributor, "123");
_cleanup_space(_constructor, "123");

/* <token action="SET" idxset="(k1)"/> */
sprintf(_tp_name, "token#%s", "123");
sprintf(_tp_token, "(k1:%d~%d,%d:#%d)", i+M, N, M, M);
_tp_size = sizeof(_tp_token);
_tokens =
_set_token(_distributor, _tp_name, (char *)_tp_token, _tp_size);
if (_tokens < 0) exit(-1);

/* <send var="i" type="int"/> */
sprintf(_tp_name, "int:i#%s", "123");
_tp_size = sizeof(int);
_tp_i_123 = &i;
_status =
_send_data(_distributor, _tp_name, (char *)_tp_i_123, _tp_size);
if (_status < 0) exit(-1);

/* <send var="outMat" type="float[N(i~N) ][N(i~N) ]"/> */
sprintf(_tp_name,
        "float(%d) (%d):outMat#%s[%d~%d,%d][%d~%d,%d]@%d",
        (N), (N), "123", (i), (N), 1, (i), (N), 1,
        sizeof(float));
_tp_size =
(((N) - (i) - 1) / 1 + 1) * (((N) - (i) - 1) / 1 + 1) *
sizeof(float);
_tp_outMat_123 = (float *)malloc(_tp_size);
for (_x0_123 = (i), _y0_123 = 0; _x0_123 < (N);
    _x0_123 += 1, _y0_123 += 1) {
    for (_x1_123 = (i), _y1_123 = 0; _x1_123 < (N);
        _x1_123 += 1, _y1_123 += 1) {
        _tp_outMat_123[_y0_123 *
            ((N) - (i) - 1) / 1 + 1) +

```

```

        _y1_123] =
            outMat[_x0_123][_x1_123];
    }
}
_status =
_send_data(_distributor, _tp_name, (char *)_tp_outMat_123,
           _tp_size);
if (_status < 0) exit(-1);
free(_tp_outMat_123);

/* <read var="outMat" type="float[N(i+M~N) ][N(i+M~N)]"/> */
sprintf(_tp_name,
        "float(%d) (%d):outMat#%s[%d~%d,%d][%d~%d,%d]@%d",
        (N), (N), "123", (i+M), (N), 1, (i+M), (N), 1,
        sizeof(float));
_tp_size =
(((N) - (i+M) - 1) / 1 + 1) * (((N) - (i+M) - 1) / 1 + 1) *
sizeof(float);
_tp_outMat_123 = (float *)malloc(_tp_size);
_tp_size =
_read_data(_constructor, _tp_name, (char *)_tp_outMat_123,
           _tp_size);
if (_tp_size < 0) exit(-1);
for (_x0_123 = (i+M), _y0_123 = 0; _x0_123 < (N);
    _x0_123 += 1, _y0_123++) {
    for (_x1_123 = (i+M), _y1_123 = 0; _x1_123 < (N);
        _x1_123 += 1, _y1_123++) {
        outMat[_x0_123][_x1_123] =
            _tp_outMat_123[_y0_123 *
                (((N) - (i+M) - 1) / 1 + 1) +
                _y1_123];
    }
}
free(_tp_outMat_123);
}

_close_space(_constructor, "123", 1);
_close_space(_distributor, "123", 1);
/* </master> */

/*
for (i = 0; i < N; i++)
{
    for (j = 0; j < N; j++)
    {
        printf("%6.3f ", outMat[i][j]);
    }
    printf("\n");
}
*/
return 0;
}
/* </parallel> */

```

- The Worker Program

```
/* =====
 * Compiler-Generated Worker Program, "blu_123.c".
 */
#include "parallel.h"
/* <reference> */
#include "blu.h"
/* </reference> */

/* <reference> */
int i, j, dist, k1, k2, p1, p2, q1, q2, q3;
int subdist, rowdist, coldist;
float inSubMat[M][M], outSubMat[M][M], LU[M][M], L[M][M], U[M][M];
/* </reference> */

/* <parallel appname="blu"> */
int * _tp_i_123;
float * _tp_outMat_123;
int _x0_123;
int _y0_123;
int _x1_123;
int _y1_123;
int _k1_start = 0;
int _k1_stop = 0;
int _k1_step = 0;

main(int argc, char **argv[])
{
    /* <worker id="123"> */
    _distributor = _open_space("distributor", 0, "123");
    _constructor = _open_space("constructor", 0, "123");

    while (1)
    {
        /* <token action="GET" idxset="(k1)"/> */
        sprintf(_tp_name, "token#%s", "123");
        _tp_size = 0;
        _tp_size =
            _get_token(_distributor, _tp_name, (char *)_tp_token, _tp_size);
        if (_tp_size < 0) exit(-1);
        if (_tp_token[0] == '!') break;
        sscanf(_tp_token, "%d@(k1:%d~%d,%d)",
            &_tokens, &_k1_start, &_k1_stop, &_k1_step);

        /* <read var="i" type="int"/> */
        sprintf(_tp_name, "int:i#%s", "123");
        _tp_size = sizeof(int);
        _tp_i_123 = &i;
        _tp_size =
            _read_data(_distributor, _tp_name, (char *)_tp_i_123, _tp_size);
        if (_tp_size < 0) exit(-1);

        /* <read var="outMat" type="float[N(i~i+M)][N(i~N) ]"/> */
        sprintf(_tp_name,
```

```

        "float(%d) (%d):outMat#%s[%d~%d,%d][%d~%d,%d]@%d",
        (N), (N), "123", (i), (i+M), 1, (i), (N), 1,
        sizeof(float));
    _tp_size =
    (((i+M) - (i) - 1) / 1 + 1) * (((N) - (i) - 1) / 1 + 1) *
    sizeof(float);
    _tp_outMat_123 = (float *)malloc(_tp_size);
    _tp_size =
    _read_data(_distributor, _tp_name, (char *)_tp_outMat_123,
    _tp_size);
    if (_tp_size < 0) exit(-1);
    for (_x0_123 = (i), _y0_123 = 0; _x0_123 < (i+M);
        _x0_123 += 1, _y0_123 += 1) {
        for (_x1_123 = (i), _y1_123 = 0; _x1_123 < (N);
            _x1_123 += 1, _y1_123 += 1) {
            outMat[_x0_123][_x1_123] =
            _tp_outMat_123[_y0_123] *
            (((N) - (i) - 1) / 1 + 1) +
            _y1_123];
        }
    }
    free(_tp_outMat_123);

    /* <read var="outMat" type="float[N(k1) ][N(i~N) ]"/> */
    sprintf(_tp_name,
        "float(%d) (%d):outMat#%s[%d~%d,%d][%d~%d,%d]@%d",
        (N), (N), "123", _k1_start, _k1_stop, 1, (i), (N), 1,
        sizeof(float));
    _tp_size =
    (((_k1_stop - _k1_start - 1) / 1 + 1) *
    (((N) - (i) - 1) / 1 + 1)) *
    sizeof(float);
    _tp_outMat_123 = (float *)malloc(_tp_size);
    _tp_size =
    _read_data(_distributor, _tp_name, (char *)_tp_outMat_123,
    _tp_size);
    if (_tp_size < 0) exit(-1);
    for (_x0_123 = _k1_start, _y0_123 = 0; _x0_123 < _k1_stop;
        _x0_123 += 1, _y0_123 += 1) {
        for (_x1_123 = (i), _y1_123 = 0; _x1_123 < (N);
            _x1_123 += 1, _y1_123 += 1) {
            outMat[_x0_123][_x1_123] =
            _tp_outMat_123[_y0_123] *
            (((N) - (i) - 1) / 1 + 1) +
            _y1_123];
        }
    }
    free(_tp_outMat_123);

    //A = A - WZ

    /*<target index="k1" order="1" limits="(i+M,N,M)" chunk="M">*/
    for (k1 = _k1_start; k1 < _k1_stop; k1 += _k1_step)
    /*</target>*/
    {

```

```

k2 = k1 + M - 1;
rowdist = M;
if (k2 > N-1)
{
    k2 = N - 1;
    rowdist = N - k1;
}

for (p1 = i + M; p1 < N; p1 = p1 + M)
{
    p2 = p1 + M - 1;
    coldist = M;
    if (p2 > N-1)
    {
        p2 = N - 1;
        coldist = N - p1;
    }

    /*
    * matrix multiplication
    * outMat[k1:k2][p1:p2] = outMat[k1:k2][p1:p2] -
    *   outMat[k1:k2][i:i+M-1]* outMat[i:i+M-1][k1:k2];
    */
    for (q1 = k1; q1 <= k2; q1++)
    {
        for (q2 = p1; q2 <= p2; q2++)
        {
            for (q3 = 0; q3 < M; q3++)
            {
                outMat[q1][q2] = outMat[q1][q2] -
                    outMat[q1][i+q3]*outMat[i+q3][q2];
            }
        }
    }
}

/* <send var="outMat" type="float[N(k1)      ][N(i+M~N)]"/> */
sprintf(_tp_name,
        "float(%d) (%d):outMat#%s[%d~%d,%d][%d~%d,%d]@%d",
        (N), (N), "123", _k1_start, _k1_stop, 1, (i+M), (N), 1,
        sizeof(float));
_tp_size =
    (((_k1_stop - _k1_start - 1) / 1 + 1) *
    (((N) - (i+M) - 1) / 1 + 1)) *
    sizeof(float);
_tp_outMat_123 = (float *)malloc(_tp_size);
for (_x0_123 = _k1_start, _y0_123 = 0; _x0_123 < _k1_stop;
    _x0_123 +=1, _y0_123 ++) {
    for (_x1_123 = (i+M), _y1_123 = 0; _x1_123 < (N);
        _x1_123 +=1, _y1_123 ++) {
        _tp_outMat_123[_y0_123 *
            ((N) - (i+M) - 1) / 1 + 1) +
            _y1_123] =
            outMat[_x0_123][_x1_123];
    }
}

```

```
        }
    }
    _status =
    _send_data(_constructor, _tp_name, (char *)_tp_outMat_123,
              _tp_size);
    if (_status < 0) exit(-1);
    free(_tp_outMat_123);
}

_close_space(_constructor, "123", 0);
_close_space(_distributor, "123", 0);
/* </worker> */

exit(0);
}
/* </parallel> */
```

REFERENCES

- [1]. ALLEN, R. and KENNEDY, K. "Automatic translation of Fortran programs to vector form". *ACM Trans. on Programming Languages and Systems*, 9(4):491-542, Oct. 1987.
- [2]. ALMASI, G. S. and GOTTLIEB, A. "Highly Parallel Computing, 2nd edition". The Benjamin Cummings Publishing Company, Inc., 390 Bridge Parkway, Redwood City, CA, 1994.
- [3]. ARVIND, "Decomposing a Program for Multiple Processor System", *Proceedings of the 1980 International Conference on Parallel Processing*, pp. 7-14, August, 1980.
- [4]. Banerjee, P., Chandy, J., Gupta, M., Hodges, E. IV, Holm, J., Lain, A., Palermo, D., Ramaswamy, S., and Su, E., "The PARADIGM Compiler for Distributed-Memory Multicomputers", *IEEE Computer*, Vol. 28, No. 10, pages 37-47, October 1995.
- [5]. BEGUELIN, A., DONGARRA, J., GEIST, A. and SUNDERAM, V. "Visualization and debugging in a heterogeneous environment". *IEEE Computer*, Vol. 26, No. 6, pp. 88-95, June 1993.
- [6]. BLATHRAS, K. "A Systematic Dataflow Relaxation Approach Asynchronous Parallel Iterative Algorithm". Ph.D. Thesis, Temple University, Philadelphia, August 1996.
- [7]. CARRIERO, N. and GELERNTER, D. "The S/Net's Linda Kernel". *ACM Transactions on Computer Systems*, Vol. 4, No. 2, pp. 110-129, May 1986.
- [8]. CARRIERO, N. and GELERNTER, D. "Linda in Context". *Communications of the ACM*, 32(4):444-458, April 1989.
- [9]. CARRIERO, N. and GELERNTER, D. "How to Write Parallel Programs - A First Course". The MIT Press, Cambridge, MA, 1990
- [10]. DEMMEL J., HIGHAM, N. and SCHREIBER, R., "Block LU Factorization", February 16, 1992.
- [11]. DENNIS, J. B., "Data Flow Supercomputers". *IEEE Computer* 13(1): 48-56, 1980.
- [12]. DONGARRA, J. and WALKER, D., "SOFTWARE LIBRARIES FOR LINEAR ALGEBRA COMPUTATIONS ON HIGH PERFORMANCE COMPUTERS", The Netlib Web Site, Feb 9, 1997.

- [13]. FOX, G. C., HIRANANDANI, S., KENNEDY, K., KOELBEL, C., KREMER, U., TSENG, C.-W. and WU, M.-Y. "Fortran D language specification". Technical Report SCCS-42c, Syracuse University, Syracuse, NY, April 1991. Rice Center for Research in Parallel Computation; CRPC-TR90079.
- [14]. GARSHOL, L. M., "An Introduction To XML", <http://www.garshol.priv.no/>, (October 1999)
- [15]. Gupta, M. and Banerjee P., "PARADIGM: A Compiler for Automatic Data Distribution on Multicomputers", In *Proceedings of the 7th ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.
- [16]. GURD, J. R., KIRKHAM, C. C. and WATSON, I. "The manchester prototype dataflow computer". *Communications of the ACM*, 28(1):34-52, January 1985.
- [17]. HIGH PERFORMANCE FORTRAN FORUM. "High Performance Fortran Language Specification". Version 1.1. Center for Research on Parallel Computation, Rice University, Houston, TX, Nov. 1994.
- [18]. HODGES, E. W., "High Performance Fortran Support for the PARADIGM Compiler". *Master's thesis, Department of Electrical Engineering, University of Illinois, Urbana, IL*, October 1995. Center for Reliable and High-Performance Computing.
- [19]. HUMMEL, S. F., SCHONBERG, E. and FLYNN, L. E., "Factoring -- A Method for Scheduling Parallel Loops" *CACM*, Vol., 35, No.8 (August 1992).
- [20]. HWANG, K. "Advanced Computer Architecture: Parallelism, Scalability, Programmability". McGraw-Hill, Inc., New York, 1993.
- [21]. KAPPS, C. A., and MATHYS, W. L., "An Analysis of AC versus DC", In *AREA IONIZATION*, STATIC INC., June 25, 1984
- [22]. KNOBE, K., LUKAS, J. D. and STEELE, G. L. "Compiling Fortran 8x Array Features for the Connection Machine Computer Systemization on SIMD machines". *the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, 1988.
- [23]. KRISHNAMURTHY, E. V. "Parallel Processing: Principles and Practice". Addison-Wesley, New York, 1989.
- [24]. KUCK, D. J., KUHN, R. H., LEASURE, B. and WOLFE, M. J. "The structure of an advanced vectorizer for pipelined processors". In *Proceedings of the 4th International Computer Software and Applications Conference. IEEE Computer*

Society, Washington, D.C., 709-715, 1980.

- [25]. Lamport, L., "The parallel execution of do loops", *Communications of the ACM*, 17(2):83-93.
- [26]. LAWRENCE LIVERMORE NATIONAL LABORATORY, "OpenMP Tutorial", <http://www.llnl.gov/computing/tutorials/openMP/>, (January, 2004)
- [27]. MESSAGE-PASSING INTERFACE FORUM, "MPI: A message-passing interface standard". *Technical Report CS-94-230*, University of Tennessee, Knoxville, TN, 1994.
- [28]. PADUA, D. A. and WOLFE, M. J. "Advanced compiler optimizations for supercomputers". *Commun. ACM* 29, 12 (Dec.), 1184-1201, 1986.
- [29]. PFISTER, G. F. "In Search of CLUSTERS - The Coming Battle in Lowly Parallel Computing". Prentice Hall PTR, Upper Saddle River, NJ, 1995.
- [30]. POLYCHRONOPOULOS, C. D. "Parallel programming and compilers". Kluwer Academic Publishers, Norwell, MA, 1988.
- [31]. POLYCHRONOPOULOS, C. D and KUCK, D., "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Computers" *IEEE Transactions on Computers*, C-36, 12 (December 1987).
- [32]. SABOT, G. "A Compiler for a Massively Parallel Distributed Memory MIMD Computer". *The Fourth Symposium on the Frontiers of Massively Parallel Computation*, 1992.
- [33]. SHI, Y., "A Distributed Programming Model and Its Applications to Computation Intense Problems for Heterogeneous Environments", *AIP Conference Proceedings* 283, *Earth and Space Science Information Systems*, 827-848, 1994.
- [34]. SHI, Y., "Multicomputer System and Method", *U. S. PATENT OFFICE*, PATENT #5,517,656, May 1996.
- [35]. SHI, Y. "Timing Models: Towards the Scalability Analysis of Parallel Programs". *Technical Report*, CIS, Temple University, Philadelphia, 1994.
- [36]. SHI, Y., BLATHRAS, K. and DOUGHERTY, J. "The Synergy System: Tools for Computing the Future". In *Supercomputing '92, Minneapolis*, 1992.
- [37]. THE OPENMP WEB SITE, <http://www.openmp.org/>.

- [38]. THE THINKING MACHINE CORPORATION. "CM Fortran User's Guide version 0.7-f". July 1990.
- [39]. The W3C XML WEB SITE, <http://www.w3.org/XML/>
- [40]. TRELEAVEN, P. C., BROWNBIDGE, D. R. and HOPKINS, R. P. "Data-driven and demand-driven computer architecture". *Computing Surveys*, 14(1):93-143, March 1982.