1991

ParPlum : a system for evaluating parallel program optimization methods

Jingsong Fu
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the [Electrical and Computer Engineering Commons](#)

Let us know how access to this document benefits you.

Recommended Citation

Fu, Jingsong, "ParPlum : a system for evaluating parallel program optimization methods" (1991).
Dissertations and Theses. Paper 4177.
<https://doi.org/10.15760/etd.6061>

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

AN ABSTRACT OF THE THESIS OF Jingsong Fu for the Master of Science
in Electrical and Computer Engineering presented on August 9, 1991.

Title: ParPlum: A System for Evaluating Parallel Program Optimization Methods

APPROVED BY THE MEMBERS OF THE THESIS COMMITTEE:

Michael A. Driscoll, Chair

W. Robert Daasch

Warren Harrison

The diversity of application programs and parallel architectures makes the mapping problem complicated and hard to evaluate. The quality of mapping is machine and application dependent and varies due to inaccurate values of application and architecture characteristics.

A system for developing, applying and evaluating mappings must have four characteristics: (1) Simplicity: A mapping procedure can be evaluated by separately evaluating its submapping, so the complicated problem can be simplified. (2) Generality: A wide range of application programs and architectures can be easily represented and all mapping algorithms can be easily implemented. (3) Multifunctionality: all the mapping steps, application programs, target architectures, and related cost functions can vary and

are easy to evaluate. (4) Ability for the sensitivity analysis: The sensitivity of mapping quality to the inaccuracy of cost functions and characteristics of applications and architectures can be easily tested.

ParPlum, which is presented in this thesis, is aimed at creating and evaluating mappings on different parallel architectures with different application programs. Sensitivity analysis is another major focus. The design philosophy of ParPlum is to narrow down the multidimensional optimization problem into sub-problems with one or fewer dimensions. Mapping, for example, can be divided into three submappings, partitioning, allocating, and scheduling. This leads to the implementation of the ParPlum system, the use of data flow style, the distribution of ParPlum libraries, and the development of the ParPlum pipeline.

The experiments conducted in testing ParPlum are typical of experiments performed by parallel processing researchers. More specifically, the testing of ParPlum has been done on a UNIX LAN with three different types of graphs representing the calculations of FFT and vector inner products with several partition and allocation algorithms. In testing, about 500 test trials were conducted, varying parameters of the graphs and architectures, such as the number of Sun workstations, the number of actors in data flow graphs, and the grain size of graphs. Total execution time and speedup were used in evaluating the performance of different mapping algorithms. The speedup curves showed dramatically key aspects of speedup such as linearity and saturation. In the test of calculating 32000 element vector inner products on 2, 4, 8, and 16 Sun 3/50 workstations, speedups of 1.85, 3.81, 6.96, and 12.47 were observed. The experiments and the results showed that the ParPlum system is a general, easily controllable, multifunctional tool for mapping, applying, and evaluating mapping methods.

PARPLUM: A SYSTEM FOR EVALUATING PARALLEL PROGRAM
OPTIMIZATION METHODS

by
JINGSONG FU

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING

Portland State University
1991

TO THE OFFICE OF GRADUATE STUDIES:

The members of the Committee approve the thesis of Jingsong Fu presented
August 9, 1991.

Michael A. Driscoll, Chair

W. Robert Daasch

Warren Harrison

APPROVED:

Rolf Schaumann, Chairman, Department of Electrical and Computer Engineering

C. William Savery, Interim Vice Provost for Graduate Studies and Research

ACKNOWLEDGEMENTS

I am grateful to Dr. Michael A. Driscoll, my academic advisor, for giving me extensive advice not only in the design and implementation of the ParPlum system, but also in writing this document. The unforgettable experience of studying and working with him is a great pleasure and treasure in my entire academic life. Thanks to Dr. W. Robert Daasch and Dr. Warren Harrison for providing valuable comments on my thesis.

Thanks to all the other members of the ParPlum project: De-Zheng Tang, Chin-tamani M. Patwardhan, Kiswanto Thayib, Satish Maruti Pai, and Liono Setiowijoso, for their co-operation and the pleasure we shared in PCAT (Portland Center of Advanced Technology).

Thanks to Ms. Shirley Clark for her kindness and help in my entire study at PSU. Thanks to all who have helped and given me encouragement in my study and research.

I would like to acknowledge the Department of Electrical and Computer Engineering at Portland State University and the National Science Foundation for the financial support. Finally, thanks to my grandfather Mr. Ge, S-Liang and my family friend Mr. Wang, Fuyuan for their initial financial support for my attendance to PSU. Without their generosity it could be very hard or even impossible for me to present this thesis.

Jingsong Fu

TABLE OF CONTENTS

	PAGE
ACKNOWLEDGEMENTS	iii
LIST OF TABLES	vi
LIST OF FIGURES	vii
CHAPTER	
I INTRODUCTION	1
1.1 Problem Statement	1
1.2 ParPlum Research Goal and Thesis Objective	2
1.3 Thesis Overview	3
II EVALUATION OF MAPPING METHODS	4
2.1 Overview	4
2.2 Mapping Methods and Mapping Systems	5
2.3 Mapping Criteria and Evaluation Methods	8
2.4 Our Strategy.	16
III THE IMPLEMENTATION OF PARPLUM	20
3.1 Overview of the ParPlum System	20
3.2 User Interface.	21
3.2.1 Program Format	
3.2.2 Machine Format	
3.3 Core	25
3.4 Models	28
3.5 ParPlum Mapping Pipeline	28
3.5.1 Mapping Methods	

3.5.2	Utilities	
3.5.3	Multi-interpretor	
3.6	Performance Estimation Environment.	35
3.6.1	Interpreters	
3.6.2	Time Measurement	
3.7	Summary	41
IV	GENERAL APPROACH TO SYSTEM TESTING	43
4.1	Overview of System Testing.	43
4.2	Graphs, Algorithms and Architectures	43
4.3	Evaluation Models	49
V	EXPERIMENTAL RESULTS AND DATA ANALYSIS	54
5.1	Tree Graphs	54
5.2	Loop Graphs	58
5.3	Other Experiments	62
5.3.1	FFT Graphs	
5.3.2	Tree-loop Graphs with LAST Allocation Algorithm	
5.3.3	Tree-loop Graphs with Random Allocation Algorithm	
5.4	Modeling Execution Time of Tree-Loop Graphs	64
5.4.1	Model of Btree Partition and Modulo Allocation	
VI	SUMMARY AND CONCLUSIONS	70
	REFERENCES	73

LIST OF TABLES

TABLE		PAGE
I	Build Graph and Real Execution Time for Tree Graphs.	51
II	Setup Connection Times.	53
III	Execution Time for Tree Graphs (raw data).	55
IV	Execution Time and Speed Up for Tree Graphs	56
V	Total Execution Time and Speedup for Tree-loop Graphs.	59
VI	Selected Setup Connection Times	66
VII	Execution Time for Basic Actors and Graphs	66
VIII	Modeled and Simulated Execution Time for Tree Graphs.	68

LIST OF FIGURES

FIGURE		PAGE
1	Mapping Evaluation.	9
2	ParPlum Pipeline	18
3	ParPlum System.	21
4	Data Flow Graph	23
5	Input Format.	24
6	Machine Format	25
7	The List Structure in CORE	26
8	Core Organization	27
9	Relationship of Models	29
10	An Example for Send Actor	33
11	Multi-interpreter on A LAN with 4 Child Processes	34
12	Uni-interpreter State Diagram.	36
13	Parent Process State Diagram	38
14	Child Process State Diagram	39
15	15 Node Binary Tree	44
16	Basic Loop Graph (8loop)	45
17	4 Loop Tree-like Graph.	46
18	15 Node Tree with 3 Partitions	47
19	15 Node Tree with 4 Partitions	47
20	Time Chart for Four Even Partitions.	50
21	Execution Time of Tree Graphs.	57

22	Speedup of Tree Graphs.	58
23	Execution Time of Tree-loop Graphs	60
24	Speedup of Tree-loop Graphs	61
25	Speedup of 16x2000 Tree-loop Graph with LAST Allocation.	63
26	Speedup of 16x500 Tree-loop Graph with Random Allocation	64
27	4 Partitions of Tree-loop Graph.	67
28	Modeled Execution Time for Tree-loop Graphs	69

CHAPTER I

INTRODUCTION

1.1 PROBLEM STATEMENT

The growth of parallel computers in recent years has increased the importance of parallel processing. A fundamental task in parallel processing is mapping an input program to a parallel architecture, so that the input program can be divided into several pieces which may be executed simultaneously on different processors. There are many associated subproblems in a mapping procedure. Each mapping procedure can be divided into three submapping steps [1]: partitioning, allocating, and scheduling. Program partitioning makes use of fine-grained program structures and divides a program into a set of partitions which reflect program characteristics. Program allocation assigns one or more program modules or partitions to a specific processor. The allocation is primarily concerned with the topology of a target architecture. Job scheduling arranges the execution order of operations within each processor, making use of any fine-grained structure of individual processors and partitions and the dependences among all processors.

Each mapping procedure can be viewed as solving a multidimensional optimization problem in which some aspect of performance is optimized. Typical optimization criteria include minimizing total execution time, minimizing interprocessor communication, and balancing load. Good mappings enhance program parallelism and promote fast execution. Since many of the mapping problems and subproblems are NP-hard and because of the diversity of programs and architectures, the evaluation of mapping

methods is limited to small classes of programs and target architectures. Because of the complexity of programs and architectures, mapping methods must base their decisions on potentially inaccurate models of system performance. These inaccuracies may greatly affect the quality of mapping processes and the evaluation of mapping methods.

1.2 PARPLUM RESEARCH GOAL AND THESIS OBJECTIVE

The overall goal of this project is to develop and evaluate ParPlum, a general system for developing, applying, and evaluating automatic mapping procedures. Evaluating a mapping procedure using the ParPlum mapping pipeline has a number of advantages, including:

1. Evaluating mappings for a wide range of input programs and target architectures.
2. Evaluating any meaningful combination of submapping steps and procedures.
3. Examining the sensitivities of mapping qualities to the inaccuracies of cost functions, and the sensitivities of cost values to the inaccuracies of characteristics of applications and architectures.

Because of the great complexity of mapping methods and evaluations, the research to be presented in this thesis will focus on the following research objectives:

1. The design and implementation of the ParPlum system, including its internal data structures, mapping libraries, utilities, and the ParPlum pipeline.
2. The testing and evaluation of the ParPlum system on a UNIX LAN.

1.3 THESIS OVERVIEW

Chapter II begins with an examination of some mapping procedures and their evaluation methods, some mapping and evaluation systems, and their advantages and shortcomings. Then the relationship between performance models and mapping methods is reviewed, which leads to the design concept and implementation of the ParPlum system.

Chapter III presents the overall structure and implementation of the ParPlum system, the relationship of all the components and libraries in the ParPlum system, and the construction of the ParPlum pipeline and the execution environment. Chapter IV describes the general approach and experiments for testing and evaluating the ParPlum system. Briefly, ParPlum is used to perform several experiments typical of those performed by parallel processing researchers. These experiments show that ParPlum meets all the requirements for developing, applying, and evaluating automatic mapping problems. There are detailed discussions about the time measurement, input graphs, mapping algorithms and performance criteria used in the system testing. Chapter V gives experimental results and data analysis as the examples of mapping and mapping evaluation using the ParPlum system. Chapter VI presents conclusions about the design and implementation of the ParPlum system, describes the current status of the system and gives areas for further development in the future.

CHAPTER II

EVALUATION OF MAPPING METHODS

2.1 OVERVIEW

Since mapping problems are NP-hard, heuristic methods have been used to find near optimal solutions. A heuristic mapping procedure can be defined as one that attempts to minimize some cost functions (functions of application and system characteristics) without guaranteeing optimality.

Application characteristics are features of a program, such as program module dependences, program grain size, and types of calculations. System characteristics are features of a target architecture, such as system topologies, CPU throughput, etc. These characteristics form the basic environment for a mapping procedure. The evaluation of heuristic mapping methods is based on mapping criteria, such as total execution time and efficiency in resource utilization.

In Section 2 of this chapter, some mapping algorithms and mapping systems are examined in the light of application and system characteristics. Section 3 discusses mapping procedure evaluation methods and details cost functions, mapping criteria, and application and system characteristics. Section 4 presents the key concepts of the design strategy and implementation of the ParPlum system.

2.2 MAPPING METHODS AND MAPPING SYSTEMS

The parallel mapping problem for specific architecture and program characteristics has received generous attention. The architecture-oriented methods dominate the literature. Some of them have focused mainly on the development of specific mapping strategies for particular multiprocessor architectures, which are applicable to some limited class of multiprocessor architectures [2]. Representative of this category is the work of Bokhari [3]. He considered mapping algorithm communication graphs into a finite element machine architecture. In 1988, he presented a sum-bottleneck path algorithm to solve a small class of programs on a single-host, multiple-satellite system [4]. On the other hand, some other researchers have focused mainly on the development of specific mapping methods for particular application programs. For example, Iyer and Sholl [5] proposed a methodology for partitioning feed-forward, pipelined program structures in real-time distributed system. For chain graphs, which represent a large number of real applications, Girkar [6] presented an optimal algorithm which merges nodes into one partition based on node computation time and communication time. Two other similar approaches were also independently proposed by Polychronopoulos [7] and Bokhari [4].

As a number of multiprocessor architectures have been developed in recent years, such as CHiP [8], Clip [9], PASM [10], MPP [11], the Cosmic Cube [12], the Butterfly [13], Ultracomputer [14], the Connection Machine [15], RP3 [16] and pyramid [17], there is a general demand for mapping methods which are applicable to various multiprocessor architectures regardless of application and underlying architecture characteristics. There is some work on mapping methods when both input programs and architectures are allowed to vary. Preparata [18] discussed the allocation of several types of programs into "hypercube equivalent" networks such as the shuffle-exchange, cube-connected cycles and butterfly. Another approach is taken by Fishburn and Finkel [19] who allocate a small class of commonly used communication graphs into smaller-sized architectures of

the same graph type. Kim and Browne [2] mapped regular and irregular computation graphs to homogeneous and heterogeneous MIMD architectures, using conceptually simple and computationally tractable heuristics based on linear clustering.

Although these methods can map general programs to general multiprocessor architectures, they do not deal with the entire mapping problem. In other words, they concentrate on some subset of mapping problems. Some of them may focus on partitioning, while others may neglect that submapping and concentrate on allocating. For example, Preparata's algorithm only deals with allocation. Kim and Browne did not take scheduling into account.

Paralex is a general automatic mapping system developed by Babaglu [20] which automatically maps programs to multiprocessor architectures. A prototype of Paralex has been developed on a network of m680x0, SPARC, PRISM, MIPS and Vax-architecture workstations running UNIX. When executing a Paralex program, a clustering algorithm [21] is used to partition the program into subgraphs of chains. Then the subgraphs are allocated to a collection of workstations on a network with another heuristic algorithm using SPEC mark [22] as the cost function. Finally a loader launches the program execution in the multiprocessors.

The Paralex system provides evidence that a distributed system can be viewed and programmed as if it were a uniform multiprocessor parallel computer. Paralex adopts a very pragmatic and realistic approach — use only information that is structurally available and prefer simple, cheap heuristics to complex, expensive optimization computations. How well these choices work in practice has to be verified, because different kinds of heuristics only apply to different applications.

Pre-P [23] is another developing mapping system which focuses on automatically determining general mappings of programs into architectures. In the Pre-P project, although the original target parallel architecture is a CHiP machine, the design strategy

and protocols were developed with the general mapping problem in mind.

In Pre-P, an instance of a parallel program is represented as a communication graph, G_i , whose nodes represent processes and whose edges represent communication links between processes. The parallel program is then a family of communication graphs $\{G_i\}$, one for each problem instance. This representation presupposes that the program has already been decomposed into a set of processes which run concurrently. To represent the target architecture, an undirected computation graph, H , is used in which the nodes are processors and the edges are data paths.

Using these abstractions, the mapping process can then be viewed as an embedding problem from $\{G_i\}$ into H . One fruitful approach for performing the embedding was to divide the embedding process into the tasks of partitioning, placing these groups at processors (we call it allocation), and multiplexing the processes within each group to execute the original parallel program (we call it scheduling). A diverse group of benchmark programs and architecture interconnection structures were considered to evaluate the efficiency of mappings based on this approach. The complexity of a mapping was measured in terms of the amount of sequential simulation during multiplexing and the amount of edge expansion resulting from a given set of contraction, placement and routing transformations.

Because the initial design strategy of Pre-P focused on the general mapping problem, the authors claim that it is easy to port Pre-P to other message-passing architecture, such as the hypercube. It will be possible to extend Pre-P to an evaluation system for comparing mapping strategies with one another and with the "optimal" strategy and for the analysis of performance. However, there have not been any results found in the literature about these extensions.

2.3 MAPPING CRITERIA AND EVALUATION METHODS

The quality of mappings is controlled by mapping criteria, cost functions, and application and architecture characteristics. Figure 1 shows their relationships, which can be characterized by two formulas. First, the cost function, whose values are affected by application and architecture characteristics:

$$C = C(\theta(A), \eta(P));$$

where

A — a target architecture;

C — a cost function;

P — an application program (graph);

$\eta(p)$ — a vector of program characteristics, e.g., dependence and grain size;
and

$\theta(A)$ — a vector of architecture characteristics, e.g. number of processors,
communication time between two processors.

Second, the overall qualities of a mapping, which are estimated by a certain mapping criterion:

$$Q = F(\theta(A), \eta(P), C(\theta(A), \eta(P)), M(A, P)),$$

where

F — a criterion function;

Q — an overall quality; and

M — a mapping method.

Viewing the mapping procedure as an optimization problem, the mapping criterion is the definitive measure of performance. It is usually a measured quantity (such as execution

time) or a quantity derived from measurement (such as speedup). A cost function takes application and architecture characteristics as its inputs, and the cost values used in a mapping process should reflect the mapping criterion of the mapping method. For example, in the LAST algorithm [24], the overall quality is estimated by the mapping criteria of minimizing total graph execution time. Two cost functions were used in decision making, d_node_j and $strength_{j,i}$, which are functions of node computation time, which is restricted by CPU speed.

Figure 1. Mapping Evaluation.

Commonly used mapping criteria functions include:

1. *Total execution time* [25]: This is a function of mapping method and other factors including the number of processors and interprocessor communications cost.
2. *Speedup* [26]: This is the ratio of the total execution time on a uniprocessor to the total execution time on the parallel processors.
3. *Efficiency in resource utilization* [27]: This is a function of task-direct execution cost of subgraphs and overhead cost associated with subgraphs including the task scheduling and communication cost.
4. *Task reliability and delay* [28]: The task reliability measures the probability of executing successfully a task that is composed of a set of functions running on remote processing elements, while task delay describes the average delay incurred during the processing of a task.

Although the quality of mapping is controlled by some specific mapping criteria, it is affected by other factors which represent the architecture and program characteristics, mapping features and lower level performances of the mapping method. Almost every heuristic mapping method makes decisions based on some cost values. Most cost functions in the literature are performance-oriented, so cost function is a major factor which affects mapping performances. Two commonly used cost functions are: (1) *Total interprocessor communication cost* [21]: Interprocessor communication cost occurs when processes residing in different processors must communicate. Interprocessor communication cost is a function of the amount of data transferred and of network properties such as topology and link capacity. (2) *Total execution and communication cost* [29]: This is the sum of the total computation cost for each process and the total interprocessor communication cost.

There are many other cost functions which are based on execution cost and communication cost. *Completion time* [30] is a simple example. This cost function is the total execution time cost and interprocessor communication cost incurred by that processor whose cost is greater than all other processors. Below are some more detailed examples:

1. A cost function used in the simulated annealing heuristic [31] is given as $C = C_b + \omega C_c$. C_b is the total cost due to unbalance loading of a multiprocessor system. C_c is the total communication cost from task graph edges having nodes on different processors. ω is the weight given to the contribution of the communications cost relative to the computational load imbalance across the system.
2. In the LAST algorithm [24], two costs were used in decision making, d_node_j and $strength_{j,i}$. The d_node value computes the percentage of defined edges that a node possesses and the percentage of communication costs that belong to defined edges. The strength measure relates to how much a given node is attached to a group of nodes.
3. One of the two cost functions used for the Dominant Request Tree algorithm [2], in finding the largest weighted sum of computation and communication times among the graph nodes, which was defined for node n as:

$$C = \omega T_{comp} + (1-\omega)T_{comm}.$$

where T_{comp} is the computation time of node n , T_{comm} is the total communication time of node n with its adjacent node(s), and ω is a normalization factor.

4. In Chu's allocation strategy [32], a matrix presentation was used to calculate total cost, which was defined as the sum of processing cost and IPC cost. The allocation of modules to processors was defined in matrix X . The processing cost is given by the Q matrix. The measure of the communication cost is found in a distance matrix D .

The major boundary between mapping criteria and cost functions is that mapping criteria focus on evaluating the overall mapping performance, while cost functions help in decision making during the mapping process. From the detailed examples above, a cost function calculates its value based on a few other variables, such as a graph node computation time and the communication time between two nodes, which are the characteristics of programs and architectures. We can summarize mapping criteria, cost functions, and characteristics of programs and architectures and their relationship as

1. Mapping criteria: This is the overall measurement of mapping quality. It is determined by cost functions and characteristics of applications and architectures.
2. Characteristics of application and architecture: These are machine specific and program dependent parameters. Some of them may be given by system specifications such as the number of processors and memory space available. Some of them are measured in different applications, such as a graph node execution time.
3. Cost function: This characterizes the solution space of a mapping search strategy. The cost could be an indicator or estimator of mapping quality. For example, lower interprocess communication time cost indicates higher speedup in overall quality. The cost functions could be divided into several levels. A higher level cost function depends on a lower level cost function. The lowest level cost function only uses the characteristics of application and architecture to calculate its cost values.

The diversity of applications and architectures causes the diversity of mapping methods, which in turn causes the diversity of evaluation procedures for mapping algorithms. Although the common evaluation criteria such as total execution time and speedup can be used in evaluations, there is no optimal heuristic algorithm, so evaluations of mapping methods are conducted in a comparative fashion. Sheild [31], for exam-

ple, analyzed the performance of simulated annealing in comparison with the performance of iterative improvement. Four VLSI circuits were chosen in evaluating the annealing heuristic. Iqbal, Saltz and Bokhari [33] analyzed four strategies for load balancing. The performance of each of these strategies is compared on a set of problems whose structure permits the use of all four strategies.

Task Grapher, developed by El-Rewini and Lewis [34], is a rare general evaluation system which does performance analysis of allocation methods. For the implementation details of Task Grapher, see the reference [35]. In Task Grapher, programs must be represented by an acyclic-directed task graph $TG(M, E)$ [36], where M is a set of numbered nodes representing tasks and E is a set of edges representing ordering restrictions among the tasks. Each node is given a value for execution time of the task. Each edge is given a value equal to the size of the message delivered from one node to another. After the task graph is defined, a target machine interconnection topology must be specified. There are six built-in interconnection topologies: hypercube, fully connected, star, mesh and balanced binary tree. Given a task graph and a topology of a target machine, Task Grapher uses one or more of its seven allocation heuristics to produce the following displays: (1) Gantt Chart Schedule, (2) Speedup Line Graph, (3) Critical Path in Task Graph, (4) Processor Utilization Chart, (5) Processor Efficiency Chart, and (6) Dynamic Activity Display.

With Task Grapher, a user can: (1) model a parallel program as a task graph consisting of M tasks (one task per node); (2) choose a method of optimization from several allocation heuristics which will automatically allocate tasks onto N processors; (3) choose the topology of the desired target architecture (or design an arbitrary topology for the parallel processor of interest; and (4) observe anticipated allocating and performance estimates obtained from the mapping of task graph onto target machine.

In reviewing previous work, we can summarize a parallel mapping process as a

multi-dimensional problem by extending the previous formula to

$$Q = F(\theta(A), \eta(P), C(\theta(A),\eta(P)), p(\theta(A),\eta(P)), a(\theta(A),\eta(P)), s(\theta(A), \eta(P))),$$

F — a criterion function;

Q — an over all quality;

A — a target architecture;

C — a cost function;

P — a application program(graph);

$\theta(A)$ — a vector of program(graph) characteristics, e.g., dependence, grain size;

$\eta(P)$ — a vector of architecture characteristics, e.g., number of processors, communication time between two processors;

p — a partitioning algorithm;

a — an allocating algorithm; and

s — a scheduling algorithm;

Based on the above equation, most previous research can be divided into three categories:

1. Given an architecture and application program, and one or two fixed submapping methods (e.g., for scheduling or allocating or both), find other submapping methods (e.g., partitioning), such that overall quality is improved by minimizing a cost function. Examples of this category are the references [2, 3, 6], which describe submapping algorithms for a certain class of applications or architectures.
2. Given an architecture and application program, find a mapping method M , such that the overall quality is improved by minimizing a cost function. Paralex [20] and Pre-P [23] are two projects which treat the overall mapping problem as a

whole process and originally focus on determining general automatic mappings of programs into architectures (more than one application and architecture).

3. Given an application program and several architectures and submapping algorithms (e.g. for allocation), select architecture and algorithm pairs, such that several criteria functions are guaranteed. Task Grapher [34] is an example.

The three categories discussed above illustrate several important requirements for evaluating mapping methods. First is the desirability of simplifying mapping procedures. A mapping procedure can be evaluated by separately evaluating its submapping steps. This approach could narrow down a complex multidimensional problem to several lower dimensional subproblems. It not only simplifies the whole process, but also eases the choice of the "best" submappings and, eventually, of a "better" mapping method.

Second is the generality of mapping procedures. An evaluation system requires a general description language for application programs of any grain size and any topology and a general description for architecture topologies; The evaluation should take into account the performance of each mapping step.

Third is a multifunctional evaluation system, in which applications and architectures are substitutable, different mapping methods can be chosen for different applications and architectures, and various criteria functions can be used to estimate mapping performances. It is desirable to have a powerful system which can vary all the variables in a controllable manner: cost functions, programs, architectures, and mapping and submapping algorithms. The Task Grapher [34] has all these features for evaluating allocation algorithms.

Fourth is the consideration of the relationship between mapping criteria and cost functions, and the relation of cost functions with the characteristics of applications and architectures. Most current research has concentrated on finding workable combinations of mapping methods. However, the sensitivity analysis of mapping quality to inaccuracy

of cost functions and inaccuracy of application and architecture characteristics has not drawn great attention in the literature. Few papers have mentioned this problem in their performance analysis. There are a few exceptions. Polychronopoulos and Kuck presented a parameterized form of the mapping method which can be tuned for different systems [37]. Sarkar [38] presented the sensitivity of mapping strategies to changes in execution profile information due to different program inputs.

The points discussed above highlight several insufficiencies in the current evaluation of parallel mapping methods and mapping systems. Investigation of these issues calls for a more general, automatic mapping and evaluation system. The design strategy presented in the next section corrects some of the shortcomings in current mapping evaluation methods and systems.

2.4 OUR STRATEGY

The original interest of the ParPlum project has focused on a general mapping evaluation system which has the characteristics needed for (1) evaluating different mapping algorithms or combinations of submapping algorithms for different classes of programs on a wide range of architectures; and (2) investigating the sensitivity of mapping quality to the cost functions and application and architecture characteristics. The approach to implementing the ParPlum system came from three ideas, including:

1. Good mapping methods examine programs and architectures at the level of fine-grained parallelism [39].
2. The data flow style renders explicit all "natural" interinstruction parallelism [40, 41].
3. Good mappings can be obtained quickly by dividing the overall mapping process into the three submappings steps [1].

A ParPlum pipeline is established to facilitate the mapping procedure and evaluation. The inputs to the ParPlum pipeline are the descriptions of the program and the target architecture. Each step inside the ParPlum pipeline is a partitioning, allocating, or scheduling procedure, a parallel interpreter or a utility program such as chop, split, merge and combine. The submapping procedures are chosen from the partitioning, allocating, or scheduling libraries. Procedures dealing with the same submapping step are interchangeable. Any partitioning procedure, for example, may be substituted for any other partitioning procedures. The model library contains different kinds of cost functions which include explicit information on communication and computation loads, bandwidth, memory constraints, etc.

Figure 2 shows the arrangement of the ParPlum pipeline, which allows models and mapping strategies to be changed independently. With this pipeline and library style, the user can perform the mapping and evaluation in a flexible and controllable manner.

Generally speaking, there are two types of flexibility. First, the mapping strategy can be evaluated for a wide range of programs and architectures. Second, the mapping strategy can be evaluated for different mapping criteria and cost functions. Based on these, several types of experiments can be suggested:

1. For a given class of application programs and a given architecture, select a group of mapping or submapping methods, and compare the performance of the resulting mapping for each method to choose a "best" mapping strategy. In this way, mapping strategy is varied to fit the fixed application and architecture.
2. For a given class of application programs and a specific mapping or submapping method, compare the mapping using 2 processors, 4 processors, etc. or the results for homogeneous architectures and heterogeneous architectures. Here the target architecture is varied.

Figure 2. ParPlum Pipeline.

3. We can also vary application programs, while keeping the mapping algorithms and architecture unchanged.
4. Select a class of programs and a class of architectures. For a given mapping method and for each combination of program and architecture, vary the accuracy of cost functions and measure the performance of the resulting mapping to characterize the sensitivity of the mapping quality to different cost functions with different accuracies.
5. Select a class of programs and a class of architectures. For a given mapping method and for each combination of program and architecture, vary the accuracy of one of the characteristics of application or architecture to measure the inaccuracy of the resulting cost to characterize the sensitivity of the cost function to inaccuracies of the characteristics.

CHAPTER III

THE IMPLEMENTATION OF PARPLUM

3.1 OVERVIEW OF THE PARPLUM SYSTEM

The ParPlum system is organized to facilitate the ParPlum mapping pipeline approach described in Section 2.4. The ParPlum system can be viewed as a core surrounded by four components: model libraries, mapping method libraries, utilities and execution tools. It is implemented as a collection of C programs and libraries that are organized in a fashion suitable to carry out mapping and evaluation experiments, i.e., sensitivity analysis and quality comparison. Figure 3 shows two levels: user interface and ParPlum system. It gives an overview of the high-level structure of the ParPlum system, showing all components and their relationships.

The core library provides routines to read program descriptions and architecture descriptions, create and arrange the internal data structures for program descriptions, build low level models, and write out the results. The models include different cost functions and characteristics of architectures and applications. The cost functions are abstracted from different evaluation and mapping methods. The characteristics are low level features generated by the core based on program descriptions and architecture descriptions. The mapping method libraries are separated into three submapping steps: partition, allocate and schedule. Each step uses the internal data structures in the core and a few parameters generated by models. To carry out an evaluation experiment, a ParPlum pipeline is formed with one or several submapping steps and with the utilities. The

Figure 3. ParPlum System.

interpreter is used to execute a mapped process and measure its performance with some mapping criteria.

Section 2 describes the user interface. Section 3 shows the major core functions and the internal data structures. This is followed by the model libraries. Then, the mapping and evaluation environment which includes the mapping libraries, utilities and execution is discussed. Finally, a summary is given for Chapter III.

3.2 USER INTERFACE

The user interface provides the descriptions of input programs and target architectures for general mapping problems. They are presented in two input formats: the program format and the machine format. The formats can be characterized by

1. Generality: They can represent most application programs and multiprocessor architectures. No undesirable restriction is imposed on the formats.

2. Simplicity: They are easy to understand and can be easily developed from a data flow graph (DFG) and an architecture topology;
3. Extensibility: It is easy to add more information, which could be needed by some specific mapping strategy.

3.2.1 Program Format

A data flow graph description format has been developed to represent the input program. A DFG is a directed graph in which the vertices (or nodes) denote entities called actors and the edges (or arcs) represent paths that carry data values between nodes. The presence or absence of a value on an arc is indicated by the presence or absence of a token (data) [42]. In data flow computation, an actor can execute or fire when there is a token on each of its input arcs. The result of an actor's firing is that the actor "consumes" its input tokens and produces a new token on each of its output arcs. Figure 4 is a DFG for the addition of 8 integers.

Figure 5 shows the program format developed for the DFG in Figure 4. Each line starting with *actor* contains the following information:

1. Actor ID, which numbers the actors in the graph;
2. Actor operation, which includes addition, division, subtraction, etc. In Figure 5, all actors have operation "add," which stands for addition;
3. Actor priority, which gives the order of executing the graph. In Figure 5, all actors have been assigned initial priority zero;
4. The input data type for each input arc. It is "int" in Figure 5, which means integer; and
5. Output links, which give the arc number of the destination actor and the destination actor ID for each output of the actor.

Figure 4. Data Flow Graph.

Each line starting with *data* includes the information:

1. Data value;
2. Data type;
3. Iteration number of the data; and
4. Destination actor ID and arc number of the actor to which the data goes.

Figure 5 has two parts. The top half is the initial format without partition and allocation information. After each step of the ParPlum pipeline in Figure 2.1, the partition and allocation information will be added to the initial format. For example, Figure 5 shows that a partition algorithm has divided the graph into three partitions: actors 1, 2, 5 assigned to partition 1; actor: 3, 4, 6 assigned to partition 2; and actor 7 assigned to partition 3. Figure 5 also shows the results of an allocation algorithm when two processors are

```

actor(1, add, 0, input(int, int), output( (5, 1) ) );
actor(2, add, 0, input(int, int), output( (5, 2) ) );
actor(3, add, 0, input(int, int), output( (6, 1) ) );
actor(4, add, 0, input(int, int), output( (6, 2) ) );
actor(5, add, 0, input(int, int), output( (7, 1) ) );
actor(6, add, 0, input(int, int), output( (7, 2) ) );
actor(7, add, 0, input(int, int), output() );

data(1, int, 0, into( (1,1) ) );
data(1, int, 0, into( (1,2) ) );
data(1, int, 0, into( (2,1) ) );
data(1, int, 0, into( (2,2) ) );
data(1, int, 0, into( (3,1) ) );
data(1, int, 0, into( (3,2) ) );
data(1, int, 0, into( (4,1) ) );
data(1, int, 0, into( (4,2) ) );

```

```

partition(1, alist(1, 2, 5) );
partition(2, alist(3, 4, 6) );
partition(3, alist( 7 ) );
allocation(1, plist( 1 ) );
allocation(2, plist(2, 3) );

```

Figure 5. Input Format.

available: allocation 1 contains partition 1; allocation 2 contains partitions 2 and 3. The field in the format related to scheduling is the actor priority. Scheduling will change the value in the internal data structure which is responsible for actor priority.

3.2.2 Machine Format

The multiprocessor architecture is defined in the machine format. The format can represent a fixed number of homogeneous or heterogeneous processors. A communication time function is provided for all pairs of processors. Figure 6 is an example of the machine format for the a UNIX LAN with 4 processors.

```

Machine : SUN, EECS;
Numprocs : 4;
proc_name(peterpan, lady, goofy, eeecs);
exec_time(ADD, INT, 1.2, 2, (0, 1) );
exec_time(ADD, FLOAT, 2, 2.2, (0) );
exec_time(ADD, FLOAT, 2, 2, (1) );
exec_time(MUL, FLOAT, 3.2, 2, (0, 1) );
comm_time(0, 1, 3, 3 );
comm_time(1, 0, 2.3, 2.43);
comm_time(1, 1, 2, 4 );

```

Figure 6. Machine Format.

The first line gives a configuration name for the file. Line 2 sets the total number of processors. Then all the machine names are given by the line starting with *proc_name*. *exec_time* is a function to calculate the execution time for some basic DFG operations such as addition, multiplication and division operating on different data types like integer type and floating type. *comm_time* gives the formula to calculate the time to pass a certain amount of data between two processors.

3.3 CORE

As mentioned in Section 2.4, a mapping procedure is divided into three submappings (partition, allocation, and schedule) to ease the complexity of mapping problems. Because each submapping needs part of the graph information to carry out the process, an internal graph data structure which depicts the input program format is required. For establishing the ParPlum mapping pipeline, the other three components: models, utilities and execution also need the internal descriptions of the program format. Instead of creating the internal data structure for each of the components and submappings, all the code which deals with input format descriptions is constructed as one single *core* library.

There are three sets of internal data structures, as shown in Figure 7. The first is

an overall graph structure. The graph is represented in an actor list. The actor list contains all the actors, which are the basic elements in a DFG. The precedence relation between actors are guaranteed by pointers. All the operations or actions that an actor could perform are implemented in separate files in a core library *action*. As an actor will be fired when its input tokens are available, a ready queue is used during execution to contain all the ready actors with the order of priority, which is set by scheduling algorithms.

Figure 7. The List Structure in CORE.

The second and third data structures are for partitioning and allocating. A parti-

tion list contains actor lists as the basic elements. In turn, an allocation list uses partition lists as its basic elements.

The core structure implements the concept of dividing mappings into three sub-mappings so that each submapping only involves a few internal data structures. A partition algorithm uses the fine-grain level structures of the actor list. An allocation algorithm uses the coarse-grained structures at the level of partitions. A schedule algorithm uses only the graph dependence information in the actor list. This approach greatly simplifies the overall mapping process.

The core libraries can be divided into five parts according to their functions. Figure 8 shows the core library organization. At the center of the core are the operations used to create and handle the internal data structures for a DFG. Around the center of the core are the characteristics operations, input operations, partition operations and allocation operations. The characteristic operations contain the functions to generate the values of characteristics for different DFGs and architectures. The input operations read the program format and the architecture format using code generated by LEX and YACC. The partition operations create the basic data structures for partitioning, while the allocation operations do the same for allocating. These operations are used by mapping algorithms.

Figure 8. Core Organization.

3.4 MODELS

As described in Section 2.2 and Section 2.3, each heuristic mapping method needs a cost function in searching the solution space and a mapping criterion in estimating the mapping quality. This restricts a mapping method to a specific criterion and a particular cost function. The model libraries are implemented to break these restrictions.

We define models in two libraries: characteristics and cost functions. The characteristics of applications and architectures are generated by the core. Some of them may be measured parameters which are directly included in the machine format. The cost functions have been abstracted from various mapping algorithms. The value of a cost function is calculated based on the characteristics. The mapping criteria could be affected by both the cost functions and characteristics. It is most often measured with the interpreters.

Figure 9 shows the relationships of the models and the relationships of the models with the core, the mapping libraries and the interpreters. In general, a mapping algorithm could use any cost function to search the solution space, and the performance of a mapping method could be evaluated or measured using any criterion. A cost function and characteristic could have several versions or values for different accuracies, so the sensitivities can be easily measured. For example, an "add" actor's execution time could have three different versions in the model library, the model of the measured value, the model of the measured value with 20% noise, and the model of the measured value with 30% noise. Then the sensitivity of mapping quality to noise can be evaluated by using these three versions.

3.5 PARPLUM MAPPING PIPELINE

Three components are used to build the ParPlum mapping pipeline, three submap-

Figure 9. Relationship of Models.

pings, utilities, and executions. The core and models are hidden from the user inside these three components.

3.5.1 Mapping Methods

The purpose of our strategy is focused on the general mapping problem. In other words, for a given input program and a given architecture, we hope to compare mapping algorithms with one another and with the "best." This requires all the submapping algorithms to be implemented as executable files, so that submapping procedures in the ParPlum mapping pipeline can easily substitute for others in the same step. The three sub-mappings can be viewed as:

Partition: Given a program format with a large number of fine grain actors, find a partition which divides the program into m processes, such that a given cost function is minimized.

Allocation: Given a program format with a partition of m processes and a target machine, find an assignment of each process to one or more processors, such that a given cost function is minimized.

Schedule: Given a program format with allocation and machine information, find the order of operations within each processor, such that a given cost function is minimized.

Five partition algorithms, five allocation algorithms and seven schedule algorithms have been implemented in the mapping libraries, and some others are in progress.

The partition algorithms:

1. *Fine grain*: Each partition contains a single actor.
2. *Random*: Each partition contains a random collection of actors.
3. *Modulo*: The partition i contains actors whose ID gets the same remainder when divided by N , where N is the number of partitions.
4. *Chain* [20]: The actors among which the heaviest communication occurs are in same partition.
5. *Btree*: The left tree of a binary tree is put into one partition, while the right tree and root node are put in another partition.

The allocation algorithms:

1. *Random*: It assigns partitions randomly to processors.
2. *Modulo*: The allocation i contains partitions whose ID gets the same remainder when divided by N , where N is the number of allocations.
3. *LAST (Localized Allocation of Static Tasks)* [24]: It assigns partitions based on their connection with previously allocated partitions and on their speed of execution.
4. *Module clustering (MCA)* [43]: It assigns partitions to processors based on the connection among partitions. It first assigns the two partitions which have the heaviest connection to the same processor, then finds the next most heavily connected pair and assigns them to the same processor, and so on, until all partitions

have been assigned to a processor.

5. *Load balancing* [43]: It uses MCA and checks the result to see if the load among processors is approximately balanced. If yes, then it terminates, otherwise it identifies the overloaded and underloaded processors and transfers some partitions from overloaded processors to underloaded processors.

The schedule algorithms:

1. *Top-bottom* [44]: It first assigns priorities to the input nodes of a graph, then to the rest of the nodes on the basis of their levels.
2. *Bottom-top* [44]: It uses the same search strategy as that of top-bottom, but it starts at the bottom of a graph.
3. *Critical path top-bottom* [45]: It assigns priorities to the nodes while adopting the top-bottom search strategy. In the case when two nodes could have the same priority, it assigns a higher priority to the node which has a shorter path to the exit node.
4. *Critical path top-bottom* [45]: It uses the same decision making strategy as allocation algorithm 3, but the search is from bottom to top.
5. *CPCN (critical path conditional node)* [45]: This algorithm first assigns weights, based on execution time and weights of successors, to all actors of a DFG. On the basis of these weights and the critical path, it assigns priorities to actors of the graph. This algorithm can schedule graphs which have conditional actors.
6. *CPMISF (critical path most immediate successor first)* [46]: This algorithm is based on a strategy called Critical Path most immediate successor first. It involves finding the levels of nodes in a DFG and finding their execution times.
7. *CGS (cyclic graph schedule)* [47]: It first finds cycles or loops in a given

graph. Then the execution times of the actors in these loops are adjusted by multiplying them by the number of iterations of the loop. Once this is done other remaining actors in the graph are assigned by weights. Finally, the critical path technique is utilized to assign priorities to the actors of a graph.

To form a ParPlum mapping pipeline, any algorithm or combination of algorithms which are already implemented in mapping libraries can be selected. Some algorithms require specific models, but any reasonable models could be chosen in designing an evaluation experiment. Users can carry out the experiments described in Chapter 2 simply by varying the mapping steps and models.

3.5.2 Utilities

To actually construct and arrange a ParPlum mapping pipeline for a multiprocessor architecture, some utility programs are needed to connect the mapping steps. As different operating system could have some specific requirements for the pipeline arrangement, utility programs could be different from one system to another. The utilities created for UNIX systems are merge, combine, chop, and split.

At the beginning of a pipeline, merge is used to concatenate the two input format files into one file, so the parser embedded in the partition step can parse the formats. Merge is just the UNIX command CAT.

After the allocation steps, the output includes the partition and allocation information. In each allocation, there could be more than one partition. For the convenience of the schedule and execution steps, combine is used to put all the partitions within each allocation into one partition. After the combine step, each allocation only has one partition.

Because of the dependences among different processors, one processor may need to send data to some other processors. This function is performed by the send actor,

Figure 10. An Example for Send Actor.

which transfers data via UNIX sockets. As the send actor is not in the original input graph, it has to be added after the last submapping step. The chop utility adds send actors at the places which need to transfer data to another processor. An example of adding send actors is given in Figure 10. The top graph contains three partitions without adding send actors. The bottom is the output graph of chop, in which four send actors are

included.

Split is a utility which follows chop. It writes the format of the actors in different allocations into different files. If there are n allocations, n format files will be created by split. It also creates an information file for the multi-interpretor, which includes number of processors, processors' names, processors' input format file names, etc.

3.5.3 Multi-interpretor

The multi-interpretor executes the resulting mappings on a parallel architecture of interest. The target parallel architecture used in our initial implementation is a UNIX LAN. The idea of viewing a collection of workstations on a network as a parallel architecture appears to be a popular one. There are a number of projects that have been experimenting with different abstractions on top of such a system [20, 48].

The LAN multi-interpretor is implemented as a centrally controlled distributed system. Figure 11 shows the topology of this architecture. The parent processor is the console of whole system. It reads the information file created by split, then creates n

Figure 11. Multi-interpretor on A LAN with 4 Child Processes.

child processes according to the requirements described in the information file. After the communication links among child processes are established, the parent sends part of the

program to each child process. Then all children start to execute the program, calculate results, and record performance values, such as the total execution time. In the next section, we will present more details about the multi-interpreter.

3.6 PERFORMANCE ESTIMATION ENVIRONMENT

To estimate the performance of mapping or submapping methods and conduct performance analysis and sensitivity analysis experiments, two interpreters, uni-interpreter and multi-interpreter, are available in our system. Both interpreters are built with the same functions in the core library. Generally speaking, the multi-interpreter child program is the uni-interpreter program plus some interprocess communication functions. The uni-interpreter could be used to run original input programs which are not split, while the multi-interpreter is used to run programs after splitting. Comparisons can be made on the basis of their results. Several time points are recorded in the interpreters to help in evaluation.

3.6.1 Interpreters

The uni-interpreter executes input formats in sequential order. Figure 12 is the state diagram of the uni-interpreter. There are five main states. First is the "initialization" state, which creates all the internal data structures including an empty actor list and a ready queue. The actor list will be used to store input data flow graph information. The ready queue will contain pointers to all the ready actors. Second is the "parser" state. The parser in the uni-interpreter reads the DFG format and puts actors into the actor list and initial data into actors' inputs, checking to see if the actor is ready. If the actor is ready, an actor pointer to the ready actor will be added to the ready queue. The third state is "build link" which creates an arc between any two actors if they have a dependence relationship. Fourth is the "execution" state, in which actors are fired sequentially.

Finally, the "end" state terminates the process.

Figure 12. Uni-interpreter State Diagram.

The multi-interpreter contains a limited number of processors (SUN workstations) on a LAN. Each processor in the multi-interpreter executes a graph in the same sequence as that of the uni-interpreter, except for message passing among processors. Figure 13 depicts the state diagram of the parent process and Figure 14 shows the state diagram of a child process. At the left bottom of Figure 13, all information types received by the parent from child processes are listed under "info;" all command types sent by the parent to children are listed under "command." There are a total of 11 main states in the parent process. State 1 is the "initialization" state which gets the child machine information including machine names, input file, etc. State 2 is the "create children" state. State

3, "setup communication," creates communication channels among all processes. State 4 is the "receive child processes information." When it receives a PIECE information from any one child process, it enters state 6, "send graph," and sends a piece of graph to that child. When it receives READY information N times (N is the number of children), it enters state 5, "send EXEC," and sends a command EXEC to all child processes. When it receives DONE information, it enters state 7, "send TIME," and sends TIME commands to all child processes. State 8 is "receiving child time records." After the parent receives the time records from all children, state 9, "send DIE" is entered. It sends DIE commands to all child processes. Then state 10 prints out all the time records and state 11 terminates the parent.

There are 9 basic states in the child process state diagram shown in Figure 14. First, the "initialization" state creates the internal data structures and communication sockets. Second, in "send addresses to parent," a child process transfers its address to the parent. Third, in "receive all addresses," the child receives communication addresses of all other child processes. Fourth, in the "parser" state, the child parses the input programs from the socket and builds internal actor data structures into the actor list. The fifth state is "build link" which creates an arc between any two actors if they have a dependence relationship. The child can start to fire actors synchronously, i.e., all child processes start at approximately same time; or asynchronously, i.e., children start to fire actors right after "build link." The synchronization is controlled by the user. If synchronization is requested, child processes will enter state 6, and after receiving EXEC from the parent process, enter state 7, "execution." If synchronization is not requested, "execution" is entered right after "build link." The eighth state is wait to exit, and is invoked after the last actor in the graph is fired. The final state is "end," in which child processes send time records to the parent and exit.

Figure 13. Parent Process State Diagram.

Figure 14. Child Process State Diagram.

3.6.2 Time Measurement

The fundamental measurement in the testing procedure is the time record, which includes wall clock time, user cpu time, system cpu time and total cpu time (user cpu time plus system cpu time). For the convenience of collecting data, several time record points are inserted in the interpreters.

In Figures 12, 13, and 14, all the time record points are marked. When analyzing experimental data, several time intervals will be counted based on time record points. The name of the time intervals for the uni-interpreter are listed below (refer to Figure 12):

init time: the time difference between time 2 and time 1, the time to initialize the process.

parser time: the time difference between time 3 and time 2, the time to parse the DFG format and build the actor list.

build link time: the time difference between time 4 and time 3, the time to build dependence relations among the actors in the actor list.

build graph time: the time difference between time 4 and time 2, which is the sum of *parser time* and *build link time*.

real exec time: the time difference between time 5 and time 4, which is the time for firing actors.

total exec time: the time difference between time 5 and time 1, which is the total time for the process.

The name of the time intervals for the multi-interpreter can be summarized as (refer to Figure 13 and Figure 14):

init time: the time difference between time 2 and time 1, which is the time to initialize the parent process and create the child processes, see Figure 13.

setup connection time: the time difference between time 3 and time 2, which is the time for the parent to set up communication links among the parent and all child processes, see Figure 13.

total exec time: the time difference between time 5 and time 1, which is the overall time for the parent process, Figure 13.

create graph time: the time difference between time 4 and time 2, which is the time for a child to parse DFG formats, build the actor list and build dependence relations among the actors in the actor list, Figure 14.

wait exec time: the time difference between time 5 and time 4, which is the time spent waiting to fire actors after a child creates all internal data structures (Figure 14).

real exec time: the time difference between time 7 and time 4, if synchronous start is used; the time difference between time 7 and 5, if asynchronous. This is the time for firing actors, Figure 14.

idle time: the time accumulated waiting for input data from other child processes, which is included in *real exec time*, Figure 14.

wait exit time: the time difference between time 8 and time 7, which is the time spent waiting to terminate after a child fires the last actor, Figure 14.

3.7 SUMMARY

This chapter has described the overall structure and implementation of the ParPlum system, the relationship between the ParPlum system and the ParPlum mapping pipeline, and the relationship of all the components in the ParPlum system. The ParPlum mapping pipeline is established by the basic components in the ParPlum system. Although the organization of the ParPlum mapping pipeline is fixed, the submapping

steps and the embedded models can be controlled by the user. The system design encourages users to extend the submapping libraries by adding new mapping methods, to add new models, and to design evaluation experiments by selecting the mapping methods and models from the libraries.

CHAPTER IV

GENERAL APPROACH TO SYSTEM TESTING

4.1 OVERVIEW OF SYSTEM TESTING

Since the design of the ParPlum system focuses on the evaluation of general mappings of parallel programs on multiprocessor architectures, the testing approach concentrates on three major features of the system's behavior. The first is the ability of mapping different applications to different target architectures with different mapping methods. The second is the ease of changing mapping or submapping methods, application programs, and architectures. Finally, the ease with which different mapping methods can be compared with certain evaluation criteria must be tested. A number of different DFGs have been developed for use in testing. Each of the graphs are mapped to a LAN with differing numbers of workstations. The mappings are created by the mapping pipeline with several submapping algorithms from the mapping libraries. The evaluations of different mapping methods is conducted via the interpreters. The results of the multi-interpreter and uni-interpreter are used in performance analysis which is based on several evaluation criteria. The experiments which are conducted in the evaluation of the system are typical of those a researcher in the field would like to perform.

4.2 GRAPHS, ALGORITHMS AND ARCHITECTURES

For the input graphs, varying the type of DFG and the number of nodes in each type of graph tests the system's ability to handle graphs with different types or different

grain sizes. Three different types of graph topologies are used in the testing procedure. First are the binary tree graphs, which compute vector inner products. Graphs with 63, 127, 255, 511, 1023, 2047, and 4095 nodes are used. Figure 15 illustrates a binary tree graph with 15 nodes, which calculates the inner product of 8 element vectors. The 63 nodes binary tree calculates the inner products of 32 element vectors; The 127 tree node does the inner products of 64 element vectors; The 255 does 128, and so on. The tree graph is a fine grain representation of the inner product calculation.

Figure 15. 15 Node Binary Tree.

Second are the graphs with loop topology, which are often used in numerical calculations, such as integration, and also can be used to compute inner products. Figure 16 illustrates an 8loop graph in which the gdata actor has constant data=8 and loop counter=7. This is another way to calculate the inner product of 8 element vectors. It is equivalent to the binary tree in Figure 15. In this way, 32loop is equivalent to a 63 node tree; 64loop is equivalent to 127 node tree, etc. These two graph types make it possible to test the system with different grain sizes. Fine grain parallelism can be explored with the

tree graph, while the coarse grain can be represented in the loop graph structure. The tree-loop graph used in testing is a tree-like graph with each node at the first level of the tree replaced by the basic loop nodes. Figure 17 illustrates a tree-loop graph with 4 loop nodes. If each loop node is a 100loop, then it is equivalent to the calculation of a 400 element vector inner products, or 799 node binary tree. The 31 node tree-loop graph with 100loop, 200loop, 500loop, 1000loop, and 2000loop as the loop nodes has been used in the testing. If the loop nodes are extended to binary tree graph, the tree nodes number are 3199, 6399, 15999, 31999, and 63999. The tree-loop graph is a coarse grain representation of the inner product calculation.

Figure 16. Basic Loop Graph (8loop).

The tree graph and tree-loop graph are two different graph representations of inner product calculations. The tree graph is a fine grain representation, while the tree-

Figure 17. 4 Loop Tree-like Graph.

loop graph is a coarse grain representation. The third type of graph has FFT topology, which often occurs in digital signal processing operations. Graphs with 32, 64, 128, 256, and 512 input nodes are used.

A few partition and allocation algorithms have been used in the testing procedure. This approach shows the simplicity of varying mapping algorithms in the ParPlum pipeline. The two partition algorithms used in the testing are "btree" and "modulo." The btree algorithm is used in partitioning binary tree graphs and tree-loop graphs. When splitting a binary tree into two parts, it starts at a root node (the bottom node), puts all nodes of the left subtree to one partition, all nodes of the right subtree and the root node to another partition. If more than 2 partitions are required, it will partition the left subtree and right subtree recursively. Figure 18 and Figure 19 illustrates the "btree" result of splitting a 15 node tree into 3 and 4 partitions. The modulo algorithm is used in partitioning the FFT graph. For a given number of available machines and a graph, it puts the actor with ID=1 into the first partition, the actor with ID=2 into the second partition, ..., then the actor with ID=N into the Nth partition (N is number of machines). After that, it puts the actor with ID=N+1 into the first partition, ..., ID=2N into the Nth partition, ..., until all nodes are put into a partition.

Figure 18. 15 Node Tree With 3 Partitions.

Figure 19. 15 Node Tree With 4 Partitions.

Three allocation algorithms have been used. First is "modulo." For a given number of machines, N , the modulo algorithm puts on the same machine the partitions with partition IDs that have the same remainder when divided by N . For instance, if there are 4 machines available and 15 partitions, then partitions 1, 5, 9, and 13 will be assigned to machine 1; partitions 2, 6, 10, and 14 will be assigned to machine 2; partitions 3, 7, 11 and 15 will be assigned to machine 3; and finally, partitions 4, 8 and 12 will be assigned to machine 4. The second allocation algorithm is called "random." If N is the number of machines available, the random algorithm assigns partitions to machines in a random fashion. If there are 4 partitions and two machines, the result may be all the partitions in one allocation or two partitions in one allocation and the other two partitions into another allocation. The third algorithm used is the "LAST" algorithm [24].

No scheduling algorithms have been used in the testing. Scheduling's goal is to find the order of firing actors within each processor, such that the data waiting time (waiting for data from other processors) is minimized. As the tree, FFT, and tree-loop graphs' execution orders are determined by graph precedence relations, the priorities of the actors in these graphs are irrelevant to the execution order. Adding any scheduling step makes no difference.

For the multiprocessor execution, the number of machines (workstations) used in experiments has been varied from 2 to 8 for tree, from 2 to 6 for FFT graphs, and 2, 4, 8, and 16 for tree-loop graphs. These test the ability of the system to handle the different numbers of processors and the different types of machines required by different algorithms. This is a common approach used in evaluating mapping procedures, by varying the target machine in some sense, when different multiprocessor systems unavailable [25, 49, 50].

4.3 EVALUATION MODELS

Two commonly used evaluation criteria in performance analysis are total execution time and speedup. In our test experiments, total execution time is a directly measured time period in the interpreters, while speedup is a calculated value based on the total execution time.

The general definition of speedup for a parallel architecture with p processors (written $S(p)$) is the ratio of the total execution time on a uniprocessor to the total execution time on a parallel processor. The terms superunitary, unitary, and subunitary speedup have been used to describe $S(p) > p$, $S(p) = p$, and $S(p) < p$, respectively for this general definition [51]. There exists a general debate on the possibility of superunitary speedup [52, 53, 54], and a few modified models of speedup have been developed on the basis that superunitary speedup is possible only when the total amount of work performed by the p processors is *strictly less* than the total work performed by a single processor [26]. In our test experiments, the models used to calculate speedup still observe Amdahl's law, in which the speedup achievable by a parallel computer with p processors is less than p .

Following the general definition, the speedup could be expressed in the ParPlum system as:

$$speedup = \frac{\text{total execution time of uni-interpreter}}{\text{total execution time of multi-interpreter}}.$$

If this approach is used, superunitary speedup will occur when executing a graph with a large number of actor nodes. For example, consider a graph with n actor nodes, evenly partitioned into p pieces (each having $\frac{n}{p}$ nodes), and then allocated to p workstations. The total execution time of the uni-interpreter could be approximated by the summation of two significant time intervals, *build graph time* and *real exec time*. (as *init time* is less than one second, it will be neglected in this discussion). Figure 20 gives a

Figure 20. Time Chart For Four Even Partitions.

time chart of the parent and $p=4$ child processes, We assume that all workstations have the same speed and all child processes start at the same time. The total execution time of the multi-interpretter could be expressed as the addition of *set up connection time*, *build graph time* and *real execution time*, omitting the *wait exec time*, *child init time* and *wait exit time*. Now the speedup in our example can be formed as:

$$speedup = \frac{T_g(n) + T_e(n)}{T_{set}(p) + T_g(n/p) + T_e(n/p)}.$$

where

$T_g(n)$: — build graph time for a graph with n nodes;

$T_e(n)$: — real execution time for a graph with n nodes;

$T_{set}(p)$: — set up connection time for p workstations.

$T_e(n)$ is a linear function of n , but $T_g(n)$ is a function of n^2 , because of the internal implementation of the actor list structure (a linked list). When the graph size is increased, the *build graph time* dominates the total execution time, and the speedup can be simplified as $T_g(n)/T_g(n/p)$. The complexity of $T_g(n)$ is $O(n^2)$, so the speed up could reach p^2 . In our example p is 4, and the speedup could be 16 (in the actual experiments, we did get the speedup around 8 when using 4 workstations). This is an artifact of the experimental system, and not useful in comparing mapping methods. When testing a small size of graph with $T_g \ll T_e$, then T_g is negligible and this approach of speedup calculation can be used with minor error.

Table I shows *build graph time* and *real execution time* on the uni-interpreter for 63, 127, 255, 511, 1023, 2047 and 4095 node binary tree graphs. The *build graph time* grows much faster than that of *real exec. time*, as a function of number of actors. The superunitary speedup can also be seen here. If a graph of 4096 tree nodes is split into 2 even pieces with each containing 2048 nodes, the uni-interpreter needs 7634 seconds to build the graph. But in the multi-interpreter, each child takes 1607 seconds to build graph. Here, although the graph size in the parent is double that in children, the *build graph time* is more than four times as much.

TABLE I

BUILD GRAPH AND REAL EXECUTION TIME FOR TREE GRAPHS

Number of Actors	63	127	255	511	1023	2047	4095
Build Graph Time(sec.)	4	11	37	137	505	1604	7634
Real Exec Time(sec.)	1	3	5	10	22	45	98

By the general definition of speedup, another possible approach could be used in calculating speedup when parser time dominates the total execution time:

$$speedup = \frac{\text{total exec time of multi-interpreter on a single machine}}{\text{total exec time of multi-interpreter on several machines}}.$$

In this equation, the total execution time of the uni-interpreter is replaced by the total execution time of the multi-interpreter on a single machine. Running the multi-interpreter on a single machine means the parent process creates all its child processes on one processor. In this way, the *build graph time* of the multi-interpreter on a single machine is the summation of the *build graph time* spent on all the child processes. So, the above equation could be extended if we assume all the p child processes execute equal size subgraphs, $\frac{p}{n}$, and neglect insignificant time intervals:

$$speedup = \frac{T_{set_single}(p) + pT_g(n/p) + pT_e(n/p)}{T_{set_multi}(p) + T_g(n/p) + T_e(n/p)}$$

where:

$T_{set_single}(p)$: — set up connection time on a single machine;

$T_{set_multi}(p)$: — set up connection time on many machines.

In this form, when $T_{set_single}(p)$ and T_{set_multi} are relatively small, the speedup is less than or equal to p .

There is a problem with this definition. The set up connection time on a single machine is greater than that for many machines on the LAN we used in the testing. Table II contains the data for the set up connection time (some data are missing from the table due to the difficulty in executing more than 6 child processes on one machine). The difference of the set up connection times can be so great as to overwhelm any other time differences. Including the set up connection time on a single machine is not part of the general definition of speedup, as this is the communication overhead for the multemachine case. This leads to a modification of the second approach, which does not include the set up connection time on a single machine:

$$speedup = \frac{(\text{total execution time on a single machine}) - (\text{setup connection time})}{\text{total execution time of multi-interpreter on several machines}}.$$

This form includes the set up connection time in the total execution time of multi-interpreter on several machines, but does not include it in the total execution time of multi-interpreter a single machine. The speedup calculated by this form may underestimate actual speedup, since some setup time will be needed in the single machine case, no matter what form the interpreter takes.

TABLE II
SETUP CONNECTION TIMES

Number of children	2	3	4	5	6	8	16
Single Machine (sec.)	33	49	62	78	94	-	-
Multi Machine(sec.)	28	32	39	43	53	64	130

Both of the approaches are used in test experiments. The first approach is used for tree-loop graphs, as the *build graph time* is insignificant. The modified second approach is used for binary tree and FFT graphs to get rid of the superunitary problem.

CHAPTER V

EXPERIMENTAL RESULTS AND DATA ANALYSIS

5.1 TREE GRAPHS

The test run using binary tree graphs includes 77 trials in which 42 trials were conducted for 63, 127, 255, 511, 1023, 2047, and 4095 tree nodes on the multi-interpretor with 2, 3, 4, 5, 6, and 8 Sun 3/50 workstations running child processes and 35 trials for the same number of tree nodes on the multi-interpretor with one Sun 3/50 workstation running 2, 3, 4, 5, and 6 children processes. The partition algorithm used is "btree" and the allocation algorithm is "modulo." Table III gives the raw data (total execution time on wall clock) of one test run. The numbers in the first row are the number of tree nodes. The data in the rows labeled Uni Time are the total execution times of the multi-interpretor on a single machine. The data in the rows labeled Mul Time are the total execution times of the multi-interpretor on several machines. The absence of data in the row of 8 machines Uni Time is because of problem that we mentioned in last chapter.

Table IV gives the revised total execution time of the multi-interpretor on a single machine (deducting the set up connection time), and the speedups. The values of the speedups are calculated by the modified second approach from section 4.3. The data in rows labeled Rev Time are the total execution times of the multi-interpretor on a single machine after deducting the setup connection time. Figure 21 shows the total execution time curves. The curves fall into three sets on the basis of the number of processors: (2,3), (4,5,6), (8). The reason is that if the number of machines is not a power of 2, the

TABLE III
EXECUTION TIME FOR TREE GRAPHS (RAW DATA)

Number of Nodes	63	127	255	511	1023	2047	4095
2 Machines Uni Time(sec)	41	46	67	130	355	1199	4446
2 Machines Mul Time(sec)	29	34	47	75	196	618	2281
3 Machines Uni Time(sec)	53	67	86	133	318	1980	3582
3 Machines Mul Time(sec)	34	38	51	83	200	623	2322
4 Machines Uni Time(sec)	68	77	90	124	259	714	2400
4 Machines Mul Time(sec)	38	42	60	77	106	282	728
5 Machines Uni Time(sec)	85	94	104	145	266	675	2181
5 Machines Mul Time(sec)	46	43	61	74	103	261	684
6 Machines Uni Time(sec)	102	109	124	168	276	637	1974
6 Machines Mul Time(sec)	52	55	66	76	102	264	731
8 Machines Mul Time(sec)	64	69	70	71	82	123	262

number of actors in each partition is unbalanced. For example, in Figure 18, the tree is split into 3 partitions so that partition 1 has 3 actors, partition 2 contains 4 actors, but partition 3 has 8 actors, double the size of partition 1 and partition 2. After allocation, machine 3 gets half of the total tree nodes, while machine 1 and 2 only get 1/4 of the total tree nodes each. When machine 1 and 2 finish execution, machine 3 is still running. The total execution time will only depend on machine 1, which takes the longest time to finish its job. If only two machines are used, the loads are balanced. The two machines will finish execution at almost the same time (if the machines have the same speed). We can predict here that if the same size of tree graphs are mapped by "btree" and "modulo" algorithms to 9, 10, 11, 12, 13, 14, 15 machines, the execution time curves will be close to the curve for 8 machines. The imbalance of load also affects the speedup. Some important features can be clearly seen in Figure 22. (1) There is no speedup when increasing the number of processors until the graph size reaches 511 nodes. Before this

size, the set up connection time dominates the total execution time. (2) The speedup for 2 and 3 machines are at the same level, while speedups for 4, 5, and 6 machines are at another level. The reason is the use of the btree partition algorithm, as we have discussed in the analysis of execution time for tree graphs. (3) The curve slope drops between 2 and 3 processors and between 4 and 6 processors. There are two reasons: The more machines are used, the longer the set up connection time. On the other hand, since the btree partition algorithm causes the imbalanced load, there is no improvement in performance for adding machines from two to three or from four to six.

TABLE IV
EXECUTION TIME AND SPEED UP FOR TREE GRAPHS

Number of Nodes	63	127	255	511	1023	2047	4095
2 Machines Rev Time(sec)	7	13	33	97	323	1165	4411
2 Machines Mul Time(sec)	29	34	47	75	196	618	2281
Speed Up	0.24	0.38	0.7	1.29	1.65	1.89	1.91
3 Machines Rev Time(sec)	7	13	36	89	270	929	3537
3 Machines Mul Time(sec)	34	38	51	83	200	623	2322
Speed Up	0.20	0.34	0.71	1.1	1.35	1.49	1.52
4 Machines Rev Time(sec)	7	14	28	70	193	650	2339
4 Machines Mul Time(sec)	38	42	60	77	106	282	728
Speed Up	0.18	0.35	0.47	0.86	1.82	2.30	3.2
5 Machines Rev Time(sec)	9	15	27	67	187	597	2103
5 Machines Mul Time(sec)	46	43	61	74	103	261	684
Speed Up	0.18	0.35	0.47	0.9	1.82	2.29	3.08
6 Machines Rev Time(sec)	9	16	31	69	181	543	1882
6 Machines Mul Time(sec)	52	55	66	76	102	264	731
Speed Up	0.17	0.29	0.47	0.91	1.77	2.05	2.58
8 Machines Mul Time(sec)	64	69	70	71	82	123	262

Figure 21. Execution Time Of Tree Graphs.

Figure 22. Speedup of Tree Graphs.

5.2 LOOP GRAPHS

The test run for the 16 tree-loop graphs includes 25 trials, of which 20 are for 16x100, 16x200, 16x500, 16x1000, and 16x2000 tree-loop graphs on the multi-interpreter with 2, 4, 8 and 16 Sun 3/50 workstations running child processes and 5 for the same tree-loop graphs on the uni-interpreter. The "btree" and "modulo" algorithms are still used in these test runs.

Table V gives the total execution times (wall clock time) of one test run, and speedups which are calculated by the first approach described in section 4.3. Figure 23 shows the total execution times versus the number of machines. Figure 24 shows the speedups versus the number of machines.

TABLE V
TOTAL EXECUTION TIME AND SPEEDUP FOR TREE-LOOP GRAPHS

Type of Tree-loop	100	200	500	1000	2000
1 Machine Time(minutes)	3.60	8.10	27.35	47.85	132.2
Speed Up	1.00	1.00	1.00	1.00	1.00
2 Machine Time(minutes)	2.20	4.33	15.1	26.29	71.62
Speed Up	1.64	1.87	1.81	1.82	1.85
4 Machine Time(minutes)	1.53	2.70	7.76	13.3	34.79
Speed Up	2.35	3.00	3.52	3.60	3.81
8 Machine Time(minutes)	1.52	2.15	4.77	7.82	19.03
Speed Up	2.34	3.77	5.73	6.11	6.96
16 Machine Time(minutes)	2.60	3.20	4.78	6.17	10.6
Speed Up	1.38	2.53	5.72	7.76	12.47

Figure 24 shows some other features of speedup. First the 2000 loop curve gets very good linear speedup for all cases, as the real execution time overwhelms the set up connection time and interprocessor communication time and dominates the total execution time. Second, the curves of 100, 200, and 500 loops are going down after the break points of 4 machines or 8 machines. It is because, after the break point, the setup connection time and interprocessor communication time will dominate the total execution time. For a certain job size, the best number of machines is the number at the break point. When the loop counter is increased (grain size is increased), the speedup break point goes up as shown in Figure 24.

Figure 23. Exection Time of Tree-loop Graphs.

Figure 24. Speedup Of Tree-loop Graphs.

5.3 OTHER EXPERIMENTS

5.3.1 FFT Graphs

The results of three test runs in which 32, 64, 128, 256, and 512 input FFT graphs have been used with the uni-interpreter and the multi-interpreter on 1, 2, 3, 4, 5, and 6 Sun 3/50 workstations shows that the maximum speedup is around 2. The "modulo" algorithm is used both for partition and allocation in the experiments. The FFT topology has a large number of sequential connections among actors, which lead to a lot inter-processor communications. We have not found a good partition algorithm for this topology, so the low speedup is as expected.

5.3.2 Tree-loop Graphs With LAST Allocation Algorithm

The "LAST" allocation algorithm has been used in the ParPlum pipeline. After "btree" partitions a 16x2000 tree-loop graph into 16 partitions, if there are 2 machines available, "LAST" allocates 8 partitions into allocation 0, and another 8 partitions into allocation 1; if 4 machines are available, allocation 0 and 3 both contain seven partitions and allocation 1 and 2 contain one partition each; if 8 machines are available, allocation 0 and 7 have 5 partitions each, while allocation 1, 2, 3, 4, 5, and 6 all have only 1 partition; if 16 machines available, then all the allocations contain 1 partition. Because of the unbalance in the 4 machines and 8 machines situations, the total execution times of these two cases are much greater than those obtained when using the "modulo" allocation algorithm. Figure 25 shows the speedup curves of one test trial using "LAST" and "modulo" algorithms. The speedup of "LAST" is down at the points of 4 machines and 8 machines.

The poor performance of the LAST algorithm is due, in part, to the inaccurate values used for the cost of execution and interprocessor communication times. They are all assumed to be equal in our current implementation. The results may indicate the

Figure 25. Speedup of 16x2000 Tree-loop Graph with LAST Allocation.

LAST algorithm's sensitivity to the inaccurate values of cost functions, but further experiments are needed to be certain.

5.3.3 Tree-loop Graphs With Random Allocation Algorithm

As the "random" algorithm allocates partitions in a random fashion, the resulting allocations are usually unbalanced and the amount of interprocessor communication is also increased. Figure 26 gives the speedup curve of one test trial using the "btree" partition algorithm and the "random" allocation algorithm. The graph used in this trial is 16x500 tree-loop.

Figure 26. Speedup of 16x500 Tree-loop Graph with Random Allocation.

5.4 MODELING EXECUTION TIME OF TREE-LOOP GRAPHS

The previous sections of this chapter show that the execution results differ for different partitioning and allocating algorithms and application programs. An execution time model is often used in analyzing performances of mapping methods [25]. Good

models will help to save a large amount of computation time in performance evaluations, including sensitivity analysis, in selecting a best mapping for a specific application, etc.

Generally speaking, as mentioned in Chapter II, a model for execution time is a function of several variables, and could be expressed in the extended form of that in Section 2.3:

$$T = f(\theta(A), \eta(P), p, a, s),$$

where

T — total execution time;

A — a target architecture;

P — an application program (graph);

$\theta(A)$ — a vector of program (graph) characteristics, e.g. dependence, grain size;

$\eta(P)$ — a vector of architecture characteristics, e.g. number of processors, communication time between two processors; and

p, a, s — partition, allocation and scheduling algorithms which may be affected by the values of $\theta(A)$ and $\eta(P)$.

5.4.1 Model Of Btree Partition And Modulo Allocation

The execution time of the 16 tree-loop graphs used in experiments with the "btree" partition and "modulo" allocation on 2, 4, 8, and 16 Sun 3/50 workstations can be modeled based on some measured lower level parameters shown in Table VI and Table VII.

Table VI shows the set up connection time among the parent process and child processes on different numbers of machines. Each value in the table is a mean value of the results of 8 test trials.

TABLE VI
SELECTED SETUP CONNECTION TIMES

Number of machines	2	4	8	16
Setup Time(seconds)	28	39	64	130

TABLE VII
EXECUTION TIME FOR BASIC ACTORS AND GRAPHS

Actor or Graph	add	send	100loop	200loop	500loop	1000loop	2000loop
Exec. Time(sec)	0.001	0.1	12.5	28	100	207	415

Table VII shows the execution time for two basic actors, "add" and "send." "add" is an actor performing additions. "send" is an actor that transfers data tokens between two child processes. The other values in the table are the execution times for the basic loop graph with different iteration counters. Each entry in the table is an average of several test trials' results.

The model of execution time for the 16 tree-loop graphs using "btree" and "modulo" algorithms can be written as:

$$T = \text{set up connection time} + \text{execution time of critical path to finish execution.}$$

The total set up connection time is a function of the total number of child processes. The execution time of the critical path is determined by the "btree" and "modulo" algorithms. Figure 27 shows a 16 tree-loop graph with 4 partitions and allocations. The critical path starts from the leftmost partition with 4 loop nodes, followed by actor 1, actor 2, actor 9, actor 13, actor 14, actor 17 and actor 18. The execution time calculation of the 16 tree-loop graph with 4 partitions and allocations can be represented by following formula:

Figure 27. 4 Partitions of Tree-loop Graph.

$$T = t_{setup}(4) + \sum_{i=1}^4 (\text{basic loop exec time}) + \sum_{i=1}^5 (\text{add time}) + \sum_{i=1}^2 (\text{send time}).$$

If the basic loop iteration counter is 500, the total execution time can be calculated based using the values in Table VI and Table VII.

$$T = 39 + 4 \times 85 + 5 \times 0.001 + 2 \times 0.1 = 380(\text{seconds}) = 6.33(\text{minutes})$$

Similarly, the execution time of a 16 tree-loop graph of 4 partitions with iteration counters of 100, 200, 1000, and 2000 can be obtained by substituting for the execution time of the basic loop. A similar approach can be used in calculating the execution time of a loop graph with a different number of partitions or allocations. The calculation results are listed in Table VIII. Table VIII presents a comparison between the results given by the model and the results obtained from the execution. Percentage errors for execution time are also shown in the table.

TABLE VIII
MODELED AND SIMULATED EXECUTION TIME FOR TREE GRAPH

Type of Tree-loop	16x100	16x200	16x500	16x1000	16x2000
1 Machine Simul(min)	3.60	8.10	27.35	47.85	132.2
1 Machine Model(min)	3.33	7.84	26.45	46.32	120.96
Error (%)	7.5	3.2	3.3	3.2	8.5
2 Machine Simul(min)	2.20	4.33	15.1	26.29	71.62
2 Machine Model(min)	2.13	4.2	13.8	24.21	55.8
Error (%)	3.2	3.0	10.9	7.91	22
4 Machine Simul(min)	1.53	2.70	7.76	13.3	34.79
4 Machine Model(min)	1.48	2.52	7.31	12.60	28.31
Error (%)	3.3	6.7	8.6	5.26	18
8 Machine Simul(min)	1.52	2.15	4.77	7.82	19.03
8 Machine Model(min)	1.48	2.00	4.4	7.00	14.9
Error (%)	3.2	6.9	5.8	10.48	21
16 Simul(min)	2.60	3.20	4.78	6.17	10.6
16 Model(min)	2.38	2.38	3.83	5.15	9.08
Error (%)	8.5	25	7.8	16.53	14.3

Figure 28 depicts the simulated and modeled execution time curves. The shapes of the two curves are well matched. The values predicted by the model underestimated the observed values. The main reason is the diversity of the Sun workstations' load. This will cause more overhead in idle time and waiting time (waiting to execute and waiting to terminate), and increase total execution time.

Figure 28. Modeled Execution Time for Tree-loop Graphs.

CHAPTER VI

SUMMARY AND CONCLUSIONS

This thesis presented the design and evaluation of the ParPlum system. ParPlum is a general mapping and evaluation system developed to deal with the complexity, diversity and inaccuracy of mapping procedures. The basic approach used in the design and implementation of the ParPlum system is to split complex problems into subsets, then solve the subproblems in as isolated a manner as possible. This general approach led to: (1) the use of data flow style; (2) the division of mapping into three submappings and the construction of mapping libraries; (3) the construction of performance model libraries and the sensitivity analysis of models at different levels. (4) the construction of the ParPlum pipeline.

The system testing experiments have been conducted with three different types of DFGs executed on different numbers of Sun 3/50 workstations (at most 16). The experiments indicate that common mathematics applications like FFT and matrix calculations (inner product) can be implemented with the data flow format with no restriction on the program's grain size. In other existing mapping evaluation systems, the graph representations are restricted to coarse grain, such as in Pre-P, where each graph node is a process [23]. In our test experiments, different algorithms have been chosen to give different mappings and performances. This step shows another advantage over the other systems in which only a limited number of algorithms are used to find a "good" mapping, such as Paralex [20]. In Task Grapher [34], although several algorithms could be evaluated, results are based on calculation instead of simulation. The modeling of the total execu-

tion time for tree-loop graphs gives another way to estimate the performance of mappings based on a small number of previous simulation results. It can be used in adjusting simulation results and predicting performances.

The experiments used to test the system are typical of experiments performed by parallel processing researchers. These experiments with ParPlum showed several key characteristics of the system: (1) Simplicity: The whole mapping procedure can be contained in the ParPlum pipeline, which is just one UNIX command line with several options. Changes in mapping algorithms can be done with a change of options in the pipeline. (2) Extensibility: Any new mapping algorithm can be added to the mapping libraries, as all algorithms are implemented in separate files. (3) Versatility: Generally speaking, any reasonable mapping and submapping algorithm can be used with any application. There is no restriction on the types of applications and on algorithms. (4) Controllability: In perform evaluations, the user can control the formation of the mapping pipeline by choosing different mapping steps or can even split the mapping pipeline into several pieces when it is necessary.

Section 2.3 developed four requirements for evaluating mapping methods. In summary, they are: (1) The desirability of simplifying mapping procedures; (2) The generality of mapping procedures; (3) A multifunctional evaluation system; and (4) The consideration of the relationship between mapping criteria and cost functions, and the relation of cost functions with the characteristics of applications and architectures. The experiments using ParPlum show that it meets these requirements and is thus a useful tool for developing, applying, and evaluating mapping procedures. Specifically, with characteristics (1) and (4), the first requirement in Section 2.3 is matched. With the characteristics (2), (3), and (4), the second and third requirements are met. Characteristics (2) and (4) will fit the fourth requirement in Section 2.3.

The research presented in this thesis is the first step of the ParPlum Project and

includes the basic system concept, the design of internal data structures, the implementation of the ParPlum pipeline, and the system testing. Most of the software modules and libraries in Figure 3.1, the ParPlum System, have been completed and tested with the help of all project group members. Some other group members are conducting some experiments with the mapping pipeline to examine the performance of mapping algorithms.

As mentioned in Chapter I, the research goal of ParPlum is to evaluate mappings for a wide range of input programs and target architectures and to examine the sensitivities of mapping to inaccuracies in the models of programs and architectures. For pursuing these goals and to improve user friendliness, several tasks remain: (1) Building the model library. At this writing, some architecture characteristics have been implemented, such as the execution time of actors, communication time between two Sun workstations, etc. A well-defined model protocol and interface between different levels of models needs to be developed. (2) Constructing a simple data flow language and compiler. The data flow format used now is an intermediate representation of applications and is tedious to generate by hand. A data flow language suitable to most application programs is highly needed to solve this problem. One of the group members has started working on this task. (3) Porting the simulator to other architectures. The first and the easiest step is to do simulations with different types of machines, such as SPARC, SUN 3/50, and Sequent, on a LAN.

REFERENCES

- [1] L. Jamieson, "The Characteristics of Parallel Algorithms," *The MIT Press*, pp. 306-334, 1987.
- [2] S. J. Kim and J. C. Browne, "A General Approach to Mapping of Parallel Computations upon Multiprocessor Architectures," *Proceeding 1988 International Conference on Parallel Processing*, pp. 1-8, August 15-19, 1988.
- [3] S. H. Bokhari, "On the Mapping Problem," *IEEE Transaction on Computers*, vol. C-30, no. 3, pp. 207-214, March 1981.
- [4] S. H. Bokhari, "Partitioning Problems in Parallel, Pipelined, and Distributed Computing," *IEEE Transaction on Computers*, vol. C-37, no. 1, pp. 48-57, January 1988.
- [5] V. R. Iyer and H. A. Sholl, "Software Partitioning for Distributed, Sequential, Pipelined Applications," *IEEE Transaction on Software Engineering*, vol. SE-15, no. 10, pp. 1270-1279, October 1989.
- [6] M. Girkar, "Partitioning Programs for Parallel Execution," *Proceeding ACM International Conference on Supercomputer*, pp. 216-229, 1988.
- [7] C. D. Polychronopoulos, "On Program Restructuring, Scheduling and Communication for Parallel Processor Systems," *Rep. no. 595, Center for Supercomputing Research and Development*, August, 1986.
- [8] L. Snyder, "Introduction to the Configurable, Highly Parallel Computer," *Computer*, vol. 15, pp. 116-123, January 1982.
- [9] M. Duff, "Parallel Algorithms and Their Influence on the Specification of Application Problems," *Multicomputers and Image Processing*, Academic Press, New York, pp. 261-274, 1982.
- [10] H. J. Siegel, "PASM: A Partitionable SIMD/MIMD System for Image Processing And Pattern Recognition," *IEEE Transaction on Computers*, vol. C-14, pp. 25-33, February 1981.
- [11] K. Batcher, "Design of a Massively Parallel Processor," *IEEE Transaction on Computers*, vol. C-29, pp. 836-840, September 1980.
- [12] C. Seitz, "The Cosmic Cube," *Communications of the ACM*, vol. 28, pp. 22-33, January 1985.

- [13] W. Crother , "Performance Measurements on a 128-Node Butterfly Parallel Processor," *1985 International Conference on Parallel Processing* , pp. 531-540, August 1985.
- [14] A. Gottlieb, "The NYU Ultracomputer -- Designing an MIMD Shared Memory Parallel Computer," *IEEE Transaction on Computers* , vol. C-32, no. 2, pp. 175-189, Feb. 1983.
- [15] W. D. Hillis, "The Connection Machine," *MIT Press, Cambridge, MA*, 1985.
- [16] G. F. Pfister , "The IBM Research Parallel Processor Prototype(RP3): Introduction and Architecture," *1985 International Conference on Parallel Processing*, pp. 764-771, August 1985.
- [17] L. Uhr, "Pyramid Multi-computer Structures, and Augmented Pyramids," *Computing Structures for Image Processing*, M. J. B. Duff, editor, *Academic Press, London*, pp. 95-112, 1983 .
- [18] F. Preparata, "VLSI Algorithms and Architectures," *Proceeding of the Conference on Mathematical Foundations of Computer Science*, 1984.
- [19] J. Fishburn and R. Finkel, "Quotient Network," *IEEE Transaction on Computers*, April 1982.
- [20] O. Babaglu, L. Alvisi, A. Amoroso, and R. Davoli, "Paralex: An Environment for Parallel Programming in Distributed Systems," *Technical Report UB-LCS-91-01, University of Bologna*, February 6, 1991.
- [21] C. E. Houstis, "Module Allocation of Real-Time Applications to Distributed Systems," *IEEE Transaction on Software Engineering*, vol. SE-16, no. 7, pp. 699-709, July 1990.
- [22] R. E. Novak. , "SPEC, the Industry Standard for System Performance.," *Technical Report, MIPS Computer Systems*, 1990.
- [23] F. Berman and L. Snyder, "On Mapping Parallel Algorithms into Parallel Architectures," *Journal of Parallel and Distributed Computing*, vol. 4, pp. 439-458, 1987.
- [24] J. Baxter and J. H. Patel, "The LAST Algorithm: A Heuristic-Based Static Task Allocation Algorithm," *Proceeding 1989 International Conference on Parallel Processing*, pp. II217-II222, 1989.
- [25] M. A. Driscoll and P. D. Fisher, "Modeling Execution Times for Parallel Algorithms," *Proceedings of the Third Annual Parallel Processing Symposium*, pp. 413-426, March 29-31, 1989.
- [26] D. P. Helmbold and C. E. McDowell, "Modeling Speedup (n) Greater than n," *IEEE Transaction on Parallel and Distributed Systems*, vol. 1, no. 2, pp. 250-256, April 1990.
- [27] J. P. Huang, "Modeling of Software Partition for Distributed Real-Time Applications," *IEEE Transaction on Software Engineering*, vol. SE-11, no. 10, pp. 1113-1126, October 1985.

- [28] S. Hariri and C. S. Raghavendra, "Distributed Functions Allocation for Reliability and Delay Optimization," *Proceedings of the 1986 Fall Joint Computer Conference*, pp. 344-352, 1986.
- [29] H. S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Transaction on Software Engineering*, vol. SE-3, no. 1, pp. 85-93, Jan. 1977.
- [30] C. Shen and W. Tsai, "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion," *IEEE Transaction on Computers*, vol. C-34, no. 3, pp. 197-203, March 1985.
- [31] J. Sheild, "Partitioning Concurrent VLSI Simulation Programs onto A Multiprocessor by Simulated Annealing," *IEE Proceedings*, vol. 134, no. 1, pp. 24-30, January 1987.
- [32] W. W. Chu, L. J. Holloway, M. Lan, and K. Efe, "Task Allocation in Distributed Data Processing," *Computer*, vol. 13, pp. 57-69, November 1980.
- [33] M. A. Iqbal and J. H. Saltz, "A Comparative Analysis of Static and Dynamic Load Balancing Strategies," *1986 International Conference on Parallel Processing*, 1986.
- [34] H. El-Rewini and T. G. Lewis, "Scheduling Parallel Program Tasks onto Arbitrary Target Machines," *Journal of Parallel and Distributed Computing*, vol. 9, pp. 138-158, 1990.
- [35] W. Su, "Task Grapher: Arbitrary Machine Topology and Performance Tools Environment," *TR 90-60-6 Department of Computer Science, Oregon State University*, 1990.
- [36] B. Kruatrachue, "Static Task Scheduling and Grain Packing in Parallel Processing Systems," *Ph.D Thesis, Department of Computer Science, Oregon State University*, 1987.
- [37] C. D. Polychronopoulos and D. J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Transaction on Computers*, vol. C-36, no. 12, pp. 1425-1439, December 1987.
- [38] V. Sarkar, "Partitioning and Scheduling Parallel Programs for Multiprocessors," *The MIT Press*, 1989.
- [39] M. A. Driscoll, J. Fu, S. M. Pai, C. Patwardhan, L. Setiowijoso, D. Tang, and K. Thayib, "Sensitivity Analysis and Mapping Programs to Parallel Architectures," *Proceeding of 1991 International Conference on Parallel Processing*, pp. II272-II273, August 1991.
- [40] J. A. Sharp, "Data Flow Computing," *John Wiley & Sons*, 1985.
- [41] W. B. Ackerman, "Data Flow Languages," *IEEE Computer*, vol. 15, no. 2, pp. 15-25, February 1982.
- [42] S. Dasgupta, "Computer Architecture," *John Wiley & Sons*, pp. 322-347, 1987.
- [43] K. Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *Computer*, vol. 15, pp. 50-56, June 1982.

- [44] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A Comparison of List Schedules for Parallel Processing Systems," *Communications ACM*, vol. 17, no. 12, pp. 685-690, December 12, 1974.
- [45] M. Granski, "The Effect of Operation Scheduling on the Performance of a Data Flow Computer," *IEEE Transaction on Computers*, vol. C-36 no. 9, September 1987.
- [46] C. Ramamoorthy, K. M. Chandy, and M. J. Gonzales, "optimal scheduling Strategies in a multiprocessor system," *IEEE Transaction Computers*, vol. C-21, pp. 37-146, Feb. 1972.
- [47] D. Martin and G. Estrin, "Models of Computation and System Cyclic to Acyclic Graph Transformations," *IEEE Transaction Elec. Comput.*, vol. C-16, pp. 70-79, Feb. 1967.
- [48] L. R. Schaefer, "Analysis of a Coordination Framework for Mapping Coarse-Grain Applications to Distributed Systems," *Master Thesis Portland State University*, May 31, 1991.
- [49] C. U. Martel, "A Parallel Algorithm for Preemptive Scheduling of Uniform Machines," *J. of Parallel and Distributed Computing*, vol. 5, 1988.
- [50] *Butterfly (TM) Parallel Processor Overview*, Beranek and Newman Inc., Cambridge, MA, June, 1985.
- [51] E. Miya, "Suggestion on Superlinear Speed Up Terminology," *Network News Posting*, December 1988.
- [52] D. L. Eager, J. Zahorjan, and E. D. Lazowska, "Speedup versus Efficiency in Parallel Systems," *IEEE Transaction Computers*, vol. C-38, pp. 408-432, March, 1989.
- [53] V. Faber, P. M. Lubeck, and A. B. White, "Superlinear Speedup of an Efficient Sequential Algorithm is not Possible," *Parallel Comput.*, vol. 3, pp. 259-260, 1986.
- [54] D. Parkinson, "Parallel Efficiency can be Greater than Unity," *Parallel Comput.*, vol. 3, pp. 261-262, 1986.