Portland State University

PDXScholar

Dissertations and Theses

Dissertations and Theses

8-11-1995

Data Allocation for Distributed Programs

Liono Setiowijoso

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

Setiowijoso, Liono, "Data Allocation for Distributed Programs" (1995). *Dissertations and Theses*. Paper 5102.

<https://doi.org/10.15760/etd.6978>

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.

THESIS APPROVAL

The abstract and thesis of Liono Setiowijoso for the Master of Science in Electrical and Computer Engineering were presented August 11, 1995, and accepted by the thesis committee and department.

COMMITTEE APPROVALS:

Michael A. Driscoll, Chair

W. Robert Daasch

Jingke Li
Representative of the Office of Graduate Studies

DEPARTMENT APPROVAL:

Rolf Schaumann, Chair
Department of Electrical and Computer Engineering

ACCEPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

by

on *15 February 1996*

ABSTRACT

An abstract of the thesis of Liono Setiowijoso for Master of Science in Electrical and Computer Engineering presented August 11, 1995

Title: Data Allocation For Distributed Programs

This thesis shows that both data and code must be efficiently distributed to achieve good performance in a distributed system. Most previous research has either tried to distribute code structures to improve parallelism or to distribute data to reduce communication costs. Code distribution (exploiting functional parallelism) is an effort to distribute or to duplicate function codes to optimize parallel performance. On the other hand, data distribution tries to place data structures as close as possible to the function codes that use it, so that communication cost can be reduced.

In particular, dataflow researchers have primarily focused on code partitioning and assignment. We have adapted existing data allocation algorithms for use with an existing dataflow-based system, ParPlum. ParPlum allows the execution of dataflow graphs on networks of workstations. To evaluate the impact of data allocation, we extended ParPlum to more effectively handle data structures. We then implemented tools to extract from dataflow graphs information that is relevant to the mapping algorithms and fed this information to our version of a data distribution algorithm. To

see the relation between code and data parallelism we added optimization to optimize the distribution of the loop function components and the data structure access components. All of these are done automatically without programmer or user involvement.

We ran a number of experiments using matrix multiplication as our workload. We used different numbers of processors and different existing partitioning and allocation algorithm. Our results show that automatic data distribution greatly improves the performance of distributed dataflow applications. For example, with 15×15 matrices, applying data distribution speeds up execution about 80% on 7 machines. Using data distribution and our code-optimizations on 7 machines speeds up execution over the base case by 800%.

Our work shows that it is possible to make efficient use of distributed networks with compiler support and shows that both code mapping and data mapping must be considered to achieve optimal performance.

DATA ALLOCATION FOR DISTRIBUTED PROGRAMS

by

Liono Setiowijoso

A thesis submitted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

in

ELECTRICAL AND COMPUTER ENGINEERING

Portland State University

1996

ACKNOWLEDGMENTS

First, I want to thank God for the completion of this thesis. I wish to extend my gratitude to my advisor Dr. Michael A. Driscoll, who spent hours reviewing the many drafts of this thesis. I would like to thank my committee members: Dr. W. Robert Daasch and Dr. Jingke Li for their unlimited support during the defense. Special thanks go to Shirley Clark for her kindness and help during my study at Portland State University and I am grateful to my friends for their friendship, support, love, and feedback.

Finally, my love and appreciation go to my mother, my brothers, my sister, and my uncles for their love, endless support and warm wishes.

TABLE OF CONTENTS

	PAGE
LIST OF TABLES.....	iv
LIST OF FIGURES.....	v
CHAPTER	
I INTRODUCTION.....	1
Problem Statement.....	1
Research Goal and Thesis Objective.....	4
Contributions.....	5
Thesis Overview.....	6
II PARPLUM SYSTEM.....	7
Overview.....	7
Current System.....	8
Basic Concept.....	8
Parplum System.....	9
Submappings.....	12
Utilities.....	16
Experiments on Parplum System.....	19
Limitations of The Parplum System.....	20

Timing For ParPlum Simulator.....	21
Adding Data Structures to Purplum.....	22
Introduction.....	22
Tree Form Data Structures.....	22
N-level Data Structures.....	24
Programming with Data Structures.....	27
Summary.....	33
III DATA PARTITIONING METHODS.....	35
Overview.....	35
Related Work.....	35
Array Alignment.....	40
Element Distribution.....	42
Cross References.....	44
Summary.....	45
IV DESIGN CONCEPT.....	46
Overview.....	46
Data Distribution Graph.....	48
Data Alignment.....	54
The First Part (Alignment).....	55
The Second Part (Partitioning).....	59
The Third Part (Optimization).....	60

	Example on Matrix Multiplication.....	62
	Summary.....	68
V	EXPERIMENTAL RESULTS AND DATA ANALYSIS.....	69
	Overview.....	69
	The Experimental Result.....	70
	Data Analysis.....	78
	Summary.....	80
VI	CONCLUSIONS AND FUTURE DEVELOPMENT.....	81
	REFERENCES.....	84
APPENDICES		
A	Dataflow Graph Program In ParPlum Format.....	87
B	Data Distribution Graph In ParPlum Format.....	94
C	Output Graph Of Data Alignment In ParPlum Format.....	96
D	Output Graph Of Allocation Sub Mapping	103
E	Experiment Result In Graph Format.....	110

LIST OF TABLES

TABLE	PAGE
5.1 Total execution time in seconds for matrix 3x3 with Allocation3.....	71
5.2 Total execution time in seconds for matrix 3x3 with Allocation1.....	71
5.3 Total execution time in seconds for matrix 7x7 with Allocation3.....	73
5.4 Total execution time in seconds for matrix 7x7 with Allocation1.....	74
5.5 Total execution time in seconds for matrix 15x15 with Allocation3..	75
5.6 Total execution time in seconds for matrix 15x15 with Allocation1.....	76

LIST OF FIGURES

FIGURES	PAGE
2.1 Dataflow graph	9
2.2 Dataflow graph input format	9
2.3 Loop structure in Dataflow graph.....	11
2.4a Partitioned graph.....	13
2.4b Partitioned graph in the ParPlum language.....	14
2.5a Allocated graph.....	15
2.5b Allocated graph in the ParPlum language.....	15
2.6a Output graph of the combine utility.....	16
2.6b Output graph of the combine utility in the ParPlum language.....	17
2.7 Dataflow graph format after ‘Split’ utility.....	17
2.8a Input file for processor #1.....	18
2.8b Input file for processor #2.....	19
2.9 A 2 x 3 matrix structure in tree form.....	23
2.10 Modification of an element in matrix structure.....	24
2.11 1D matrix structure.....	24
2.12 2D matrix structure.....	25

2.13	3D matrix structure.....	26
2.14	Creation of 2D matrix data structure.....	29
2.15a	After actor 1 fired.....	29
2.15b	After actor 2 fired.....	29
2.15c	After actor 3 fired.....	29
2.15d	After actor 4 fired.....	30
2.15e	After actor 5 fired.....	30
2.16	Matrix access and initialization.....	30
2.17	After storing data to array A.....	31
2.18	Allocation for fig. 2.14.....	31
3.1	Anti-diagonal Partition.....	38
3.2	Row-Column alignment in example 2.....	41
3.3	Line distribution in 3D system.....	43
3.4	Plane distribution in 3D system.....	44
3.5	Cross reference between arrays A and B.....	45
4.1	Alignment Process.....	47
4.2	Pre-alignment Category.....	48
4.3	Dataflow Graph of Two Dimension Matrix.....	49
4.4	Algorithm to create a simplified graph.....	50
4.5	Simplified Graph.....	51
4.6	Data Distribution Graph.....	53

4.7	Alignment Category.....	54
4.8	Data Structure of Matrix A and B.....	55
4.9	Matrix Multiplication in Dataflow Graph.....	56
4.10	Aligned Matrix Data Structure.....	57
4.11	Aligned Matrix Multiplication in Dataflow Graph.....	57
4.12	Replacing with constant data tokens.....	58
4.13	Creation and Initialization an Element of Matrix A.....	60
4.14	Creation and Initialization an Element of Matrix A After Optimization.....	61
4.15a	Initialization of matrix A and matrix B.....	62
4.15b	Structure of matrix C and multiplication of matrix A and B.....	63
4.16	The output of the first submapping.....	65
4.17	The output of the second submapping.....	66
4.18	The aligned graph of matrix A.....	67
5.1	Total execution time graph in seconds of table 5.1.....	72
5.2	Total execution time graph in seconds of table 5.2.....	73
5.3	Total execution time graph in seconds of table 5.3.....	74
5.4	Total execution time graph in seconds of table 5.4.....	75
5.5	Total execution time graph in seconds of table 5.5.....	76
5.6	Total execution time graph in seconds of table 5.6.....	77

CHAPTER I

INTRODUCTION

1.1 PROBLEM STATEMENT

Due to the demand for faster computation, data distributed computing systems have become more important. These systems provide a cost-effective way to exploit the power of parallel processing. Writing good data-distributed programs is not easy, however, because it is much faster and less expensive to access local data than to access them from another processor. To achieve high performance, it is essential for a computation to use local data as much as possible. There are two common approaches in distributing code or data among the processors. In the first, the programmer distributes code or data in addition to explicitly managing communication among tasks. In the second, the compiler takes a program written in a particular shared memory parallel language and then automatically generates the target parallel program or distributed data for the multicomputer. [1]

Dataflow graphs are designed to be the machine languages for parallel machines. A dataflow graph has many advantages, exposing parallelism and access patterns, but dataflow languages have proven difficult to use compared to other languages such as FORTRAN, PASCAL, etc., especially in dealing with data structures, because dataflow has no concept of memory and since data

tokens passed directly from actor to another actor, there are no variables. This makes it very difficult to implement data structure efficiently.

In mapping a dataflow program to a distributed system, it is possible to allocate the instruction codes (functional parallelism) as well as the data structures (data parallelism) to each processor to achieve good parallelism. Functional parallelism distributes different functions or operations as allowed by data and control dependencies. [15] At this level functions typically handle only scalar data tokens. With the introduction of data structures, a scalar data token can be a pointer to the location of a data element. In this case, even though we distribute the functions, communication is still needed to locate the real data. Thus to achieve data parallelism, alignment of data elements is also very crucial. In other words, to reduce communication in dataflow requires both function and data allocation, but most work done in dataflow system has focused on functional parallelism.

A dataflow graph simulator (ParPlum) was developed to explore functional parallelism in a UNIX local area network (LAN) [7]. This LAN system is a representative distributed memory system having expensive data communication when compared to systems such as Monsoon [17], Epsilon2 [18], or P-RISC[19] that were designed specifically for Dataflow computing. However, the LAN system is less expensive and more commonly available.

Most application programs are focused on data-driven models, especially those which process matrices, the heart of most scientific computing, as Pande wrote in his paper [15]:

Most partitioning and scheduling methods use data-driven code partitioning or code-based data allocation. Some of these approaches

automatically partition a program's data or code, but not both. Data-driven code partitioning in particular relies heavily on user judgment.

To achieve the maximum performance we need to partition and allocate both code and data [20].

The main idea of data partitioning is to divide the matrices among the processors, so that each processor works with local data as much as possible. For instance if an operation in a particular processor needs the first column of matrix A and the third row of matrix B, it is better to allocate those elements of matrices A and B to that particular processor so that no inter-processor communication is needed to access them.

Ramanujam and Sadayappan introduced a matrix notation that leads to a communication-free partitioning method for array accesses in fully parallel loops if certain conditions are met [2]. Li and Chen partition the program data in such a way that communication overhead is reduced and workload is balanced and then align the relative location of data structures so as to minimize inter-processor communication [3] [4]. Gupta and Banerjee analyze each loop, and based on performance considerations, identify some constraints on the distribution of various data structures being referenced in that loop to minimize access to non-local data [1].

Every approach leads to the same optimization criterion, reducing inter-processor communication in the data distributed applications to minimize the total execution time. Since finding optimal data storage patterns for communication free parallel processing in some problems is NP-complete, even for one or two dimensional matrices, the evaluation and testing of most methods are limited to particular programs and target architectures.

1.2 RESEARCH GOAL AND THESIS OBJECTIVE

The primary goal of this project is to reduce communication time in the ParPlum Dataflow system by providing automated partitioning of both data structures and functions without user interaction or programmer involvement. This thesis designs and implements a data structure for handling multi-dimensional matrices in dataflow graphs using ParPlum, and tools to automatically partition the data structures and optimize function tokens as well. The system tests and evaluates the automated data partitioning method in a distributed memory system. This research will show that data alignment and data partitioning is as important as functional parallelism and advantageous not only for computers designed especially for parallel computation, but also for relatively inexpensive local area networks (LANs). In the evaluation of the system, comparison of the same program executed under different data partitioning methods and without data partitioning will show the advantage of data partitioning.

Due to the complexity of data alignment and data partitioning methods and some limitations of the ParPlum system and the matrix data structure, the research and the experiments presented in this thesis will focus on the following objectives:

1. Design and implement data structure for multi-dimensional matrix handling in dataflow graphs using the ParPlum system.

2. Design and implement the alignment and partitioning method for two dimensional matrices. This step forms an intermediate graph that can be used by existing alignment methods.
3. Test the system run-time behavior under different numbers of processors, different sizes of matrices, and with or without alignment or optimization.
4. Compare the old ParPlum method to the new method using data partitioning to show the difference in the total execution time.

1.3. CONTRIBUTIONS

- New data structure to the ParPlum system which is like I-structure [12] but with a faster way to locate the data element.
- Auto loop-unrolling for matrix initialization for further optimization.
- Data alignment method to minimize communication among the data structure tokens.
- Loop optimization to optimize communication among the scalar data tokens.
- Using both functional parallelism and data parallelism at the same time to achieve better performance.
- Additional optimization on data structure access.
- Using Local Area Network (LAN).

1.4 THESIS OVERVIEW

Chapter 2 explains the dataflow concept, the data structure, and the ParPlum system. This includes a discussion of limitations and a brief explanation of programming with data structures and also a detailed explanation of the advantages of both function and data allocation. Chapter 3 examines some existing data partitioning methods in other systems and their evaluation methods and introduces the array alignment method we will use in our experiments. Chapter 4 describes the design concept and the implementation of the components of data alignment and data partitioning. Chapter 5 introduces the experimental environment, shows the experimental results of the different methods, and analyzes the results. Chapter 6 presents the conclusions about the design and implementation of the data partitioning and discusses future developments.

CHAPTER II

PARPLUM SYSTEM

2.1. OVERVIEW

The ParPlum system can be viewed as a core surrounded by four components: model libraries, mapping method libraries, utilities, and simulation tools. It is implemented as a collection of C programs and libraries that are organized to carry out mapping and evaluation experiments. [7] Input to the ParPlum system is machine information and a program written in dataflow graph format. The mapping process consists of several submappings such as partitioning and allocating. The flexibility of this system is that the procedures dealing with the same submapping are interchangeable, and a new procedure can be put between two submappings while the effect of every submapping can be monitored.

Section 2.2 discusses the ParPlum system before the addition of data structures and the timing of the system is discussed in section 2.3. Section 2.4 explains the method we chose for handling data structures in ParPlum. Section 2.5 introduces the new actors to handle operations on data structures. Finally, a summary of the improvements is given in section 2.6.

2.2. CURRENT SYSTEM

2.2.1. Basic Concept

The dataflow concept has been introduced in an effort to easily sequence operations in fully distributed systems and to provide easy programmability for multiprocessors. In a dataflow environment, each data token takes control of the processes, replacing the central controller (a program counter in a von Neumann machine).

A dataflow graph is a directed graph where each node/actor represents a machine instruction. Each actor has one or more input arcs and output arcs. The actor may execute or fire whenever there is a token available on every one of its input arcs and all carry the same iteration tags. In practice the tag is composed of two fields, a loop iteration number and a unique number to identify multiple activations of the same loop body. When an actor fires, it consumes the input tokens, computes the result value, and produces a result token on every output arc. [5] Figure 2.1 shows an example dataflow graph to evaluate the expression: $(7 + 6) * (5 + 4) - 3$. We can see that actor 1 (calculating $7 + 6$) and actor 2 (calculating $5 + 4$) can be executed simultaneously because all of the input arcs of these actors have at least one ready token, in this case a constant data value. The multiplication actor (actor 3) has to wait until a result value is produced by the addition actors for each of its input arcs, while the subtraction actor (actor 4) is the last actor to fire.

Figure 2.1. Dataflow Graph

2.2.2. ParPlum System

The graph in Figure 2.1 can be written in an input format for the ParPlum system as shown in Figure 2.2.

```

ACTOR(1,ADD,0,INPUT(INT,INT),OUTPUT((3,1)));
ACTOR(2,ADD,0,INPUT(INT,INT),OUTPUT((3,2)));
ACTOR(3,MUL,0,INPUT(INT,INT),OUTPUT((4,1)));
ACTOR(4,SUB,0,INPUT(INT,INT),OUTPUT());
DATA(7,INT,0,0,INTO((1,1)));
DATA(6,INT,0,0,INTO((1,2)));
DATA(5,INT,0,0,INTO((2,1)));
DATA(4,INT,0,0,INTO((2,2)));
DATA(3,INT,0,0,INTO((4,2)));

```

Figure 2.2. Dataflow Graph input format

The format for an actor contains the following information [7]:

1. Actor id, which is the unique node number in the graph;
2. Operation code, e.g., ADD, MUL, SUB, etc.;
3. Priority number, which gives the order of firing among the ready actors;
4. Input data type for each input arc, in this example INT for integer;

5. Output links, which give the input arc number and actor id for each actor that receives a copy of this actor's result.

The constant data format contains the following information:

1. Data value;
2. Data type;
3. Initial iteration number of the current data;
4. Initial relabeling number;
5. Actor id and input arc number to which the data goes.

The iteration number acts as a loop counter. It is set to zero (using a special actor) when entering the loop. The relabeling number is the activation id of a loop, which is set to a unique id number every time the loop is entered and is reset to the original id number when leaving the loop. This id number is very important in multiple activations of the same loop. It is possible that the firing of actors from different activations is interleaved, so this number differentiates tokens with the same iteration number but coming from different loop activations.

In the ParPlum system an actor is considered ready to fire if there is at least one token in each input arc with the same iteration number and the same label number.

Figure 2.3. Loop structure in Dataflow graph

Figure 2.3 is an example of a loop written in the dataflow graph format. The related C code is shown for comparison. A token entering the loop is labeled with a unique number by the RELABEL actor, and the original label is returned when leaving the loop by the UN-LABEL actor. Structure tokens for

arrays A and B are labeled as constant tokens, because these tokens are expected to be there through out the loop. The DCRE actor is used to reset the iteration number to zero as the initial value of the loop, and the ICT actor is used to get the iteration number of the input token and to transform it into a token on the output arc. This transformation makes the iteration number available as a token so that its value can be compared or analyzed. The LESS actor compares the value of two input tokens and returns a Boolean token that is used by the SEL actor to choose which output arc will receive the output token. This actor acts as a decision maker in the IF..THEN statement of high level languages. In this example the iteration number generated by the ICT actor will be passed to the COPY actor as long as the Boolean output of the LESS actor is true, indicating the iteration number is still less than n. The COPY actor copies the token to the body of the loop, in this example to the READ and STORE actors, as an index of the array, and also copies it to the INC actor to increase the iteration number. The output token is sent back to the ICT actor to start another loop. These steps are done repeatedly until the iteration number is equal to n as the limit of the loop.

2.2.3. Submappings

There are two basic submappings in the ParPlum system: partitioning and allocating. The partitioning submapping partitions the actors in such a way that a cost function is minimized. Several partitioning methods have been developed, such as [7]:

1. *Fine grain*: Every partition contains a single actor/instruction node.

2. *Modulo*: The partition i contains actors whose ID number divided by N , (N is the number of partitions) yields the remainder i .
3. *Chain*: The actors with the heaviest communication among them are put in the same partition. [16]

The result of partitioning the sample graph in Figure 2.1 is shown in Figure 2.4a, where actor 1 and actor 2 are put in different partitions but actor 3 and actor 4 are put in the same partition.

Figure. 2.4a. Partitioned Graph

```

ACTOR(1,ADD,0,INPUT(INT,INT),OUTPUT((3,1)));
ACTOR(2,ADD,0,INPUT(INT,INT),OUTPUT((3,2)));
ACTOR(3,MUL,0,INPUT(INT,INT),OUTPUT((4,1)));
ACTOR(4,SUB,0,INPUT(INT,INT),OUTPUT());
DATA(7,INT,0,0,INTO((1,1)));
DATA(6,INT,0,0,INTO((1,2)));
DATA(5,INT,0,0,INTO((2,1)));
DATA(4,INT,0,0,INTO((2,2)));
DATA(3,INT,0,0,INTO((4,2)));
PARTITION(1,ALIST(1));
PARTITION(2,ALIST(2));
PARTITION(3,ALIST(3,4));

```

Figure 2.4b. Partitioned graph in the ParPlum language

The partitioned graph is the input for the allocating step, which allocates the partitions to each processor to minimize some cost function. Some allocation methods that have been developed are:

1. *Random*: It assigns a partition randomly to a processor.
2. *LAST (Localized Allocation of Static Tasks)* [9]: It assigns partitions based on their connectivity with previously allocated partitions to minimize the data transfer among the processors.
3. *Module clustering (MCA)* [10]: This method assigns partition to processors based on the connection among the partitions.

The mapping of the partitioned graph (Figure 2.4a) into a two-processor system is shown in Figure 2.5a and 2.5b, where partitions 1 and 3 are put in the same allocation and partition 2 is put in a different allocation.

Figure 2.5a. Allocated graph

```

ACTOR(1,ADD,0,INPUT(INT,INT),OUTPUT((3,1)));
ACTOR(2,ADD,0,INPUT(INT,INT),OUTPUT((3,2)));
ACTOR(3,MUL,0,INPUT(INT,INT),OUTPUT((4,1)));
ACTOR(4,SUB,0,INPUT(INT,INT),OUTPUT());
DATA(7,INT,0,0,INTO((1,1)));
DATA(6,INT,0,0,INTO((1,2)));
DATA(5,INT,0,0,INTO((2,1)));
DATA(4,INT,0,0,INTO((2,2)));
DATA(3,INT,0,0,INTO((4,2)));
PARTITION(1,ALIST(1));
PARTITION(2,ALIST(2));
PARTITION(3,ALIST(3,4));

ALLOCATION(1,PLIST(1,3));
ALLOCATION(2,PLIST(2));

```

Figure 2.5b. Allocated graph in The ParPlum language

2.2.4. Utilities

Besides these two basic submappings, there are some utility procedures that restructure the graph before feeding it to the ParPlum simulator. The first utility combines all the partitions allocated to the same processor. Figure 2.6a and 2.6b show the Dataflow graph notation of the output graph from the combine utility for the allocated graph in Figure 2.5a and 2.5b.

Figure 2.6a. Output graph of the combine utility

```

ACTOR(1,ADD,0,INPUT(INT,INT),OUTPUT((3,1)));
ACTOR(2,ADD,0,INPUT(INT,INT),OUTPUT((3,2)));
ACTOR(3,MUL,0,INPUT(INT,INT),OUTPUT((4,1)));
ACTOR(4,SUB,0,INPUT(INT,INT),OUTPUT());
DATA(7,INT,0,0,INTO((1,1)));
DATA(6,INT,0,0,INTO((1,2)));
DATA(5,INT,0,0,INTO((2,1)));
DATA(4,INT,0,0,INTO((2,2)));
DATA(3,INT,0,0,INTO((4,2)));

PARTITION(1,ALIST(1,3,4));
PARTITION(2,ALIST(2));

ALLOCATION(1,PLIST(1));
ALLOCATION(2,PLIST(2));

```

Figure 2.6b. Output of the combine utility in the ParPlum language

Figure 2.7. Dataflow graph format after 'Split' utility

The next utility chops and splits the allocated graph into different files as the input files for each processor. A SEND actor is generated to connect an actor to another actor in a different allocation. For example, from the dataflow graph in Figure 2.7, actor 2 needs to send a result token to actor 3, which is on a different processor. The split utility will create a SEND actor that tells processor 2 where to send the result. Figure 2.8a and 2.8b are the final graphs which are ready to be sent to each processor. Each of these graphs is put in a separate data file for the children in the ParPlum simulator. The SEND actor has four input arcs, three constant data values and a data token. The first constant data value is the processor id where a data token will be sent to, the second one is the actor id where the token will go, and the last value is the queue id number of the input arc of the destination actor.

The final piece of the simulator input is machine information as the input data for the parent process in the ParPlum simulator. The parent controls and manages all the child processes. This parent process will distribute the input files to each child and tell the child processes to stop when the program is finished.

```

ACTOR(1,ADD,0,INPUT(INT,INT),OUTPUT((3,1)));
ACTOR(3,MUL,0,INPUT(INT,INT),OUTPUT((4,1)));
ACTOR(4,SUB,0,INPUT(INT,INT),OUTPUT());

```

```

DATA(7,INT,0,0,INTO((1,1)));
DATA(6,INT,0,0,INTO((1,2)));
DATA(3,INT,0,0,INTO((4,2)));

```

Figure 2.8a. Input file for processor #1

```

ACTOR(2,ADD,0,INPUT(INT,INT),OUTPUT((3,4)));
ACTOR(3,SEND,0,INPUT(INT,INT,INT,INT),OUTPUT());

DATA(5,INT,0,0,INTO((2,1)));
DATA(4,INT,0,0,INTO((2,2)));
DATA(1,INT,0,0,INTO((3,1)));
DATA(3,INT,0,0,INTO((3,2)));
DATA(2,INT,0,0,INTO((3,3)));

```

Figure 2.8b. Input file for processor #2

The ParPlum simulator is started with a parent process which creates child processes according to the machine information. After the control links from the parent to all of the children and the communication links among the children are established, the parent sends the program (from the input files) to each child process and tells them to execute.

The child processes execute the program, calculate, and record information about the result value and the process performance in output files. When the program is finished, the last actor of the program (END actor) tells the parent to stop the child processes. As soon as the parent process receives this information, it will stop and kill all of the child processes.

2.2.5. Experiments on ParPlum System

Some experiments that have been done using the ParPlum system include:

1. Tree Graphs [7]: experiments were conducted for various sizes of tree graphs (63 to 4095 tree nodes) on the ParPlum simulator with 2, 3, 4, 5, 6, and 8 Sun 3/50 workstations.

2. Loop Graphs [7]: experiments were conducted for various loop sizes of 16 tree-loop graphs (200 to 2000 loops for each tree-loop graph) on the ParPlum simulator with 2, 4, 8, and 16 Sun 3/50 workstations.
3. FFT Graphs [7]: experiments were conducted for various input FFT graphs (32 to 512 inputs) on the ParPlum simulator with 1, 2, 3, 4, 5 and 6 Sun 3/50 workstations.
4. Experiments we conducted to identify the various inaccuracies and sensitivity during the mapping procedure in matrix multiplication with automatic data generation, Gaussian elimination, etc. [8]

2.2.6. Limitations of The ParPlum System

There are some limitations of the ParPlum system and the nature of the dataflow graph environment. Therefore we need to make some assumptions and rules. In terms of the nature of dataflow behavior, the limitation is:

- A matrix element is only initialized once and is not updated (write once read many.) This is also known as the 'no side effect rule,' in functional languages. Since there are no memory-based variables in dataflow, conceptually data in the form of tokens are passed directly from actor to actor without residing in memory. Special structure handling is needed if we allow a modification to the data element of the matrix. Jean-Luc Gaudiot [6] describes this approach in his research.

In terms of the ParPlum data structure, the limitations are as follow:

- There is exactly one copy of each element of the matrix data structure and only the processor that holds the matrix element is allowed to update or initialize the element.

- The location of the data elements is fixed, i.e., no relocation is allowed during execution. This data allocation is performed only during compile time and all the decisions are made based on the reference patterns in the source code.

2.3 TIMING FOR PARPLUM SIMULATOR

According to the testings that were conducted on the ParPlum system, there are some overheads to the total execution time of the simulator. There are two type of overheads, the first overhead is the time to create and the time to manage the child processors. This overhead is about 50% to 90% of the overall execution time. This overhead varied depending on the load of the system, number of child processors created, and the size of the code is fed to each child processors.

The second overhead is the time to simulate a function code of the ParPlum system. For example to simulate an addition of two floating-point numbers, it takes about 2.23 millisecond compared to 400 nanosecond of the C language code addition.

In terms of communication time, it takes 160 millisecond in the ParPlum system compared to the 'real world' machine to machine communication that takes about 1 millisecond. So the ratio between communication and computation for the ParPlum system is about 70 and 2500 for the 'real world.' We can see here the communication time is more expensive in the 'real world' which makes data distribution and code optimization even more important. This

suggests that repeating our experiments outside of ParPlum would show even larger improvements from data distribution.

2.4 ADDING DATA STRUCTURES TO PARPLUM

2.4.1. Introduction

There are several ways to organize arrays of data in a dataflow system. The most popular data structure used is a tree. To use a tree data structure in ParPlum we need to modify it according to the nature of the ParPlum system. This section explains the tree data structure generally and the modified tree form that we call an n-level data structure.

2.4.2. Tree Form Data Structures

There are three categories of data structures that are generally used in dataflow systems: [6]

- **Scalar Value:** This is known as a constant data value and is constructed as a token.
- **Sequential Access Structures:** This is known as a ‘stream,’ where the data consumption is done in a sequential pattern. This is constructed as a sequence of tokens.
- **Random Access Data Structures** are the most generally used but the hardest to implement since there is no information about the access pattern available during compile time.

Figure 2.9. A 2 x 3 matrix structure in tree form

Dennis [11] has introduced a directed cyclic graph (also referred to as a heap) to represent the random access data structure. Each data element, consisting of the selector (index) and the data value, is represented as a branch of a tree (Figure 2.9). Since dataflow semantics prevent modification of the original data value, the set of selectors that form the first array must be duplicated to form the new set of selectors that index the new data value (Figure 2.10). This could be expensive if the structure contains a large number of selectors. This thesis is limited to no modification of the original data value.

A heap access cannot be performed until the construction of the whole structure is completed. Arvind and Thomas [12] introduced I-structures to allow the read access before the completion of the array structure. This implementation uses a 'presence' bit to mark the availability of the element of the array. If a read access attempts to read an empty element, it will be deferred until the 'presence' bit is set. [6]

Figure 2.10. modification of an element in matrix structure

2.4.3. N-level Data Structures

By simplifying the Tree Form Data Structure using linked lists, a matrix is organized in an n-level structure of rows or columns. Each level is a linked list that keeps pointers to other levels of a linked list or to a data element. The basic data structure, shown in Figure 2.11, is a 1-level data structure (one dimensional matrix) which, in this example, consists of 2 data elements. This data structure is composed of a linked list to keep both data values instead of two separate locations in the Tree data structure. The advantage of this approach is there is only one allocator for each level or dimension. Once we know the location of a certain level, we can get any indexed element easily, but in the Tree data structure we have to go back to the parent node in order to find the location of another index element even for the same dimension.

Figure 2.11. 1D Matrix Structure

An $n \times n$ matrix structure is composed of $n+1$ linked lists that are organized in a 2-level data structure, one linked list in the level 1 as the entry point to the data structure, and two linked lists in the level 2 to store the data elements. Figure 2.12 shows how the data elements of a 2×2 matrix are stored in the 2-level data structure.

Figure 2.12. 2D Matrix Structure

In this method we can expand the data structure to unlimited dimensions by adding one level of linked list per dimension. Figure 2.13 demonstrates the 3-level data structure (3D matrix) with 2 data elements in each dimension ($2 \times 2 \times 2$). For simplicity, the experiments in this thesis will be conducted on 2D matrices only.

When using this structure in a parallel computing environment, it is possible that the matrix access (read) operation occurs before the matrix element is updated or initialized. To handle this problem, a waiting list is stored in the matrix structure. The list keeps information about the index of each element to be read and the destination node for the element's value. As soon as the particular element is updated (a value is stored) the value of that element is also distributed to all of the destination nodes kept in the waiting list. Tokens

referring to a data structure store a locator of the processor that owns the structure. This allows other processors to access a data structure from its owner. Recall that only the owner of the data structure can update its data element.

Figure 2.13. 3D Matrix Structure

The disadvantage of this structure is the whole linked list has to be stored in the same processor because we can not separate an element of a linked list to a different processor. In other words, we can not distribute each element of the same row or column to different processor.

Similar to the Tree data structure, the matrix has to be accessed sequentially. For instance to access element [1,1] of a matrix, we have to read the first element of the entry point linked list (level 1) to get the pointer of the first column of linked list (level 2) and then read the first element to get the data. Thus, it is impossible to distribute the elements of the matrix so there is no communication. An access optimization that can be done to reduce the communication cost will be described in chapter 4.

The advantages of this data structure compared to the tree structure are it is faster to access data elements from the same row or column, and it faster to allocate the data elements because we allocate the data element by row or column, not per element as in the tree structure. Since the data elements are stored in a linked list with variable numbers of element, we have the flexibility to efficiently represent banded and sparse matrices. The disadvantage of this data structure is we cannot align or allocate the data elements individually.

2.5. PROGRAMMING WITH DATA STRUCTURES

Actors introduced to access this new data structure in the ParPlum system are:

STRC: to create a matrix structure. The STRC actor receives the data type of structure elements as an input. This actor returns a structure pointer to the data structure.

READ: to read an element of a matrix structure. This actor has two input arcs, the matrix data structure and the index number of the data to be accessed. This actor returns the accessed data.

RREAD: This is a remote read actor to read an element of a matrix structure from the structure's owner in a different processor and then send the data value back to the processor that requested it. This actor has five input arcs: the matrix data structure, the index number of the data value to be read, the processor id where the data token will be sent, the actor id of the recipient, and the queue number of the input arc. This actor

will send the data token internally (instead of using the SEND actor) based on the information from the input arcs and nothing is generated in the output arc.

STORE: stores a data value in a matrix structure. Three input arcs are needed in this actor. The first arc is the matrix data structure. The second one is the index number where the data will be stored. The last arc is the data value. STORE returns nothing.

Because input-output in the ParPlum system is very limited, the dataflow program has to initialize each data element of the matrix data structure explicitly using the actors mentioned above. Initialization is the construction of a matrix data structure with pre-calculated values. We do not need to initialize an empty data structure, because the data element is created when we store a value in a data structure and only one STORE actor can be performed for each data element. This means that the matrix data structure creator (STRC) only creates a pointer to the data structure and the data element is created by the STORE actor. This also implies a flexibility in the data structure size and that the data element does not exist until we store a data value in it.

Figure 2.14 shows an example of the dataflow graph and ParPlum notation to initialize a two-dimensional array, A. The data structure created and updated is described step by step in Figure 2.15, assuming that actor 4 fires first and then actor 5 later. It is possible that actor 5 fires first so that index 2 of array A is created first followed by index 1 when actor 4 is fired.

```

ACTOR(1,STRC,0,INPUT(INT),OUTPUT((4,1),(5,1)));
ACTOR(2,STRC,0,INPUT(INT),OUTPUT((4,3)));
ACTOR(3,STRC,0,INPUT(INT),OUTPUT((5,3)));
ACTOR(4,STORE,0,INPUT(STRUCT,INT,STRUCT),OUTPUT());
ACTOR(5,STORE,0,INPUT(STRUCT,INT,STRUCT),OUTPUT());
DATA(0,STRUCT,0,0,INTO((1,1)));
DATA(0,INT,0,0,INTO((2,1)));
DATA(0,INT,0,0,INTO((3,1)));
DATA(1,INT,0,0,INTO((4,2)));
DATA(2,INT,0,0,INTO((5,2)));

```

Figure 2.14. creation of 2D matrix data structure

Figure 2.15a. After actor 1 fired

Figure 2.15b. After actor 2 fired

Figure 2.15c. After actor 3 fired

Figure 2.15d. After actor 4 fired

Figure 2.15e. After actor 5 fired

Figure 2.16 shows the dataflow graph and the ParPlum notation to access the array created in Figure 2.14. When actor 6 fires, the READ actor will read the structure pointer in index number 2 from the array A (A^2) and then actor 7 will store a value of 9 in the structure element A^2 at index number 2. Figure 2.17 shows the result of this modification.

Figure 2.16. matrix access and initialization

Figure 2.17. After storing data to array A

Since only the owner of the data structure can write or read a data element from the data structure, we prepared actors to do so in case there is READ or STORE actor that is located in another processor. We call these actors remote read actor (RREAD) and store actor (STORE) but they have no input arcs. These actors will be accessed by READ or STORE actors from other processors that need to access the data structure in this processor. For the READ or the STORE actor to know which actor to send the request to, we give a fixed number for RREAD and STORE actors in every processor.

Figure 2.18. Allocation for fig. 2.14.

If the example in Figure 2.14 is allocated as in Figure 2.18 and every allocation is located in a different processor, actor 4 (STORE) needs to store the result of actor 2, but the owner/creator of the data element (actor 1) is in allocation 2. In this case, the STORE actor 4 will send a request to store the data to STORE actor 99997 in allocation 2. The processor location of the owner of the data structure can be recognized by looking at the header of the data structure locator. This header was set when the data structure was created by STRC actor. In this example is actor 1.

Related to the example above, for instance the accessing of matrix A (Figure 2.16) , is put in allocation 3. The READ actor 6 in allocation 3 needs to read data structure locator with index 2. This locator is created by actor 3 (STRC) in allocation 2, so the owner of this locator is the processor where allocation 2 is located. The READ actor 6 in allocation 3 cannot find the data structure locally to get the locator from, so it needs to send a request to RREAD actor 99998 in allocation 2. Besides sending the request for a particular element, the READ actor 6 also send the information about the destination of its output. In this example it is STORE actor 7. Then after the RREAD actor in allocation 2 receives all of the information needed, it will get the data element of matrix A with index 2 and send the result internally via SEND actor back to actor 7 (STORE) in allocation 3. In this illustration we can see that even though we partition and allocate the instruction codes so as to reduce the communication among the allocated actors, the internal communication caused by this data structure is a significant overhead. Ideally if we combine all of these 3 allocations (allocation 1,2 and 3,) in the same processor there is no internal

communication occur because the READ actor 6 and the STORE actor 4 will access data element locally.

Usually we store or read data elements using a loop as an index counter and in the dataflow system the loop counter update can be executed in parallel with the loop body. Chapter 4 will explain the loop structure with more detail. This means that the second index counter can be generated without waiting for the first operation to complete. Suppose that we have done the optimization of the data distribution as described above but the loop structure is scattered among the processors so that the index generation took longer than the read or store operation. Then there is no benefit in optimizing the data distribution. The processes still have to wait until the index counter generated to continue because index counter is one of their input arcs. It is clear here that both optimization of data distribution and code distribution are crucial. We cannot optimize data distribution only without optimizing the code distribution as well.

2.6. SUMMARY

This chapter has described the ParPlum system and the addition of data structures to the old system including new actors to deal with this data structure.

Optimization can be achieved not by data alignment or code allocation only. We need both.

Although there are some limitations of this data structure, now we can allocate not only the program or functions but the data structure as well. By

having an ability to allocate both program and data, we can exploit both data and control parallelism to reduce communication.

CHAPTER III

DATA PARTITIONING METHODS

3.1 OVERVIEW

This chapter discusses related work in this field and tries to adopt a method that we can use in the ParPlum system. Section 3.2 reviews related work. Section 3.3 explains the method of data alignment that is used in our research and the summary is given in section 3.4.

3.2 RELATED WORK

Some researchers have explored methods to partition and distribute data among the processors to eliminate, or at least reduce, interprocessor communication. Most of these methods are related to matrix applications.

Ramanujam and Sadayappan [2] developed a matrix notation technique to describe array accesses in fully parallel loops, and then derived sufficient conditions for communication-free partitioning of arrays. Consider a nested loop accessing arrays A and B:

```
for  $i = 1$  to  $N$ 
  for  $j = 3$  to  $N$ 
     $A[i,j] \leftarrow B[i',j']$ 
```


where i' and j' are linear functions of i and j and can be written as:

$$i' = f(i, j) = a_{11}i + a_{12}j + a_{10} \quad (a)$$

$$j' = g(i, j) = a_{21}i + a_{22}j + a_{20} \quad (b)$$

Every element of array A has its image(s) or map(s) in array B. If we can partition these elements into a family of parallel hyperplanes, there will be no communication caused by cross references. A hyperplane in two dimension is a line, so we need to partition array A and B into parallel lines. The family of parallel lines in array A is defined by:

$$\alpha i + \beta j = c, \quad (c)$$

and a corresponding family of lines in array B is defined by:

$$\alpha' i' + \beta' j' = c' \quad (d)$$

Now we have to find a solution of the above equations with conditions: at most one of α and β can be zero; similarly at most one of α' and β' can be zero. Otherwise the equations do not define parallel lines and that means the communication among processors cannot be eliminated. A solution that satisfies these conditions is known as nontrivial solution and the corresponding partition is also called nontrivial partition.

The matrix notation derived from the equations (a), (b), (c), and (d) as a solution to the above system is

$$\begin{pmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & 0 \\ -a_{10} & -a_{20} & 1 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ c' \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \\ c \end{pmatrix}$$

For an example using the above sufficient condition, consider this loop:

```

for  $i = 2$  to  $N$ 
  for  $j = 2$  to  $N$ 
     $A[i,j] \leftarrow B[i-1,j] + B[i,j-1]$ 

```

For each element $A[i,j]$, we need two elements of B . The matrix notation of the element $B[i-1,j]$ is:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ c' \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \\ c \end{pmatrix}$$

Similarly the matrix notation for the element $B[i,j-1]$ is:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ c' \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \\ c \end{pmatrix}$$

Both systems above must have a common solution and this reduces to the following system:

$$\begin{aligned} \alpha &= \alpha' \\ \beta &= \beta' \\ c &= c' + \alpha' \\ c &= c' + \beta' \end{aligned}$$

Those equations above can be reduced to $\alpha = \alpha' = \beta = \beta'$ which has a solution say $\alpha = 1$. This implies that both arrays A and B are partitioned in anti-diagonal lines. Figure 3.1 demonstrates such a partition with zero communication.

Figure 3.1. Anti-diagonal Partition

Li and Chen [4] represented the reference patterns of the source program as an undirected, weighted graph called a component affinity graph. (CAG) Given a cross-reference pattern between matrix A and matrix B:

$$\lceil a(i_1, \dots, i_p, \dots, i_n) \leftarrow b(\tau_1, \dots, \tau_q, \dots, \tau_m) \rceil$$

Domain components, domain p of array a $[\text{dom}(a, p)]$ and domain q of array b $[\text{dom}(b, q)]$ are said to have affinity if

$$\lceil i_p \cong \tau_q + c \rceil \quad \text{where } c \text{ is a small constant.}$$

From the following cross-reference pattern:

$$\lceil a(i, j) \leftarrow b(j, i) \rceil$$

two affinity relations can be derived, the first between $\text{dom}(a, 1)$ and $\text{dom}(b, 2)$ and the second between $\text{dom}(a, 2)$ and $\text{dom}(b, 1)$. If we align the two domains according to these relation, for every i and j , $a(i, j)$ and $b(j, i)$ will be mapped to the same processor and no communication will be needed.

Li and Chen [4] also introduced a modeled component alignment called a *component affinity graph* (CAG) which is constructed from the source program based on the cross-reference pattern described above. Each node in the graph represents the component of index domain to be aligned. Each node, that represents component from the same index domain, is grouped in the same

column and using the affinity concept, edges are generated between nodes that have affinity in their domain components. Each edge is given weight according to the strength of the preference. Then, the Gaussian elimination is used to optimize the component affinity graph and the component alignment problem can be defined. The component alignment problem, unfortunately, is expensive to solve, Li and Chen [4] observed in their research:

“ for a group of six 3-dimensional index domains, an exhaustive search algorithm may take two or more hours to find the optimal alignment on a Sun 3/50.”

Li and Chen [3] also addressed the problem by analyzing the source program references and matching the syntactic reference patterns with the appropriate aggregate communication routines. The explicit communication metric was used to guide the optimizations. By first distributing the data structures in a virtual network and determining the relative location based on the communication overhead, the interprocessor communication is then minimized. The compiler estimates the corresponding communication cost for each of the candidate partitions, and then selects the one with the minimum cost.

Gupta and Banerjee [1] introduced the notion of constraints on data distribution which was used by the compiler to obtain a complete and consistent picture of the data distribution scheme that offers good performance in the overall execution time. The compiler analyzes each loop to identify some constraints on the distribution of various data structure and then tries to distribute them so that the overall interprocessor communication time is minimized.

All of this work was done for parallel computer architectures such as Warp systolic machine, Intel iPCS/2 or NCUBE that have relatively faster communication time compared to a distributed system. Thus reducing communication is even more important for distributed machines.

3.3 ARRAY ALIGNMENT

The location of data structures is very crucial in reducing cross-references between distributed arrays. On one hand, we need to utilize a parallel computing environment by distributing the instructions (functional parallelism) and the data structures among the processors (data parallelism,) but on the other hand we need to eliminate the cross-references that occur because of this data distribution. Alignment can be done by generating alignment information based on the array reference patterns from the source program. The problem becomes more subtle in aligning multiple data structures. In this case, a communication-cost function can be used to determine the suitable location of the data structure to reduce the communication cost.

This research will focus on addressing a problem of ‘array alignment’ in the matrix data structure that was design for the dataflow graph simulator (PARPLUM.) Because of the limitation of the matrix data structure in ParPlum system (chapter 2), this research is limited to the row or column alignment of two dimensional matrix data structure only.

Example 1 shows an array reference pattern of arrays A and B. Although both arrays look the same, it is not necessary to allocate each array element in the same way.

Example 1:

$$a(j,i) \leftarrow b(i,j)$$

In this example, if we either align row j of array A and column j of array B or column i of array A and row i of array B in the same processor, there are no cross references among these alignments. Example 2 shows another array reference pattern of arrays A and B that if we align row i of array A and column i of array B, no cross references occur; but not vice versa. Such alignment is shown in Figure 3.2.

Example 2:

$$a(i,j) \leftarrow b(k,i)$$

Figure 3.2. Row-Column alignment in example 2.

In some cases, communication cannot be reduced if there are conflicting reference patterns. In example 3, either we align $a(i,j)$ with $b(j,i)$ or $b(i,j-2)$, and the communication due to one of these cross-references can not be eliminated.

Example 3:

$$a(i,j) \leftarrow b(i,j-2)$$

$$b(i,j) \leftarrow a(j,i)$$

Element Distribution

This research simplifies the method introduced by Ramanujam and Sadayappan [2] for row and column alignment only, but instead of using matrix notation to determine the rule of the data element distribution, we use a dataflow graph as the source of information. We do not use the matrix notation, first because we always distribute in row or column, so from the position of matrix index itself we can identify whether row or column distribution (line distribution) should be used. It is called 'line distribution' because the shape of the alignment is a line. The second reason is that the graph notation also tells us the relation among the data element implicitly by using the arc.

Figure 3.3. Line distribution in 3D system

How can the matrix index location tell the type of data distribution? In the example 2., both matrices have index i , but the position of index i in matrix A is in the first dimension and in the second dimension for matrix B, so we have to use row alignment in matrix A (the first dimension) and column alignment in matrix B (the second dimension.) The following 3-dimensional arrays problem gives us more example of aligning by looking at the position of the matrix index.

Example 4.

$$a(i,j,k) \leftarrow b(k-1,j-2,m)$$

In example 4. there are two matching indices (j and k), meaning only one unknown index, so we can align them in a 'line distribution.' If we choose a constant value for each j and k , we can align matrix A along the first dimension and matrix B along the third dimension as shown in Figure 3.3. In the case of two indexes unknown the alignment will be a 'plane distribution.' Example 5 and Figure 3.4 show the matrix relation and its 'plane distribution.'

Figure 3.4. Plane distribution in 3D system

Example 5.

$$a(i,j,k) \leftarrow b(k-l,m,n)$$

Cross References

In finding the cross references among the matrices we do not use a reference graph or a component graph that is constructed from the source program, because we do not use high level language in the ParPlum system.

Cross references among the data elements are described by input and output arcs of the graph because the input program in the ParPlum system is in a graph format already. Figure 3.5 shows the dataflow graph for example 2. Cross reference between array B and array A are shown by the output arc of the READ actor and the input arc of the STORE actor. Cross references between the index i of arrays A and the index i of matrix B can be determined by checking the source actor of these arcs. If both arcs come from the same actor or from the

same loop body, this means there is a link between the index i of array A and array B.

Figure 3.5. Cross reference between arrays A and B

3.4. SUMMARY

We have explored some methods of data alignment and have adopted a method that can be used in the ParPlum system. Because of the limitation of the system, we tried to modify the method so as to fit in the data structure that had been developed for the ParPlum system. The row-column alignment is the alignment method we will use in this experiment with the dataflow graph as the input program that carries all of the information about cross references and distribution of matrices' data elements.

CHAPTER IV

DESIGN CONCEPT

4.1 OVERVIEW

Basically the whole idea of the data distribution procedure is to place the matrix element creator and the matrix accesses to that particular element on the same processor to reduce accesses to different processors. If a communicationless partition is impossible, an optimization is used to reduce the cross references.

Our approach uses four submapping procedures to replace the partitioning step in ParPlum. We group these submapping procedures into two categories, pre-alignment and alignment (figure 4.1). The pre-alignment category consists of three submappings procedures. The first procedure extracts nodes dealing with matrix data structures from the graph. This results in a simplified graph that still contains all the information needed for data distribution. The second procedure organizes the simplified graph into a data distribution graph which contains more organized information about cross references among the matrix data structures. We also call this procedure a pre-alignment process. The third procedure combines the data distribution graph with the original graph.

The alignment category consists of the fourth procedure that aligns the graph to reduce cross references, partitions the whole graph, and adds some optimization to further increase the performance. This forth procedure or submapping is interchangeable, meaning we can develop a different method of the data distribution without loosing the integrity of the whole system.

Figure 4.1. Alignment Process

After this process is completed, the usual ParPlum steps of allocation, combine, and split can be performed. The detail of the four procedures are given in the following sections.

4.2. DATA DISTRIBUTION GRAPH

The three submapping procedures in the pre-alignment category describes in figure 4.2.

Figure 4.2. Pre-alignment Category

The first submapping step creates a simplified graph which is composed of only actor nodes related to the matrix data structures such as read or write functions. The idea of this approach is to speed up the organization process in the next step, because we work on a relatively small graph, compared to the original dataflow graph. For matrix multiplication of two 10 by 10 matrices this procedure reduces the graph size by 50 % and for two 20 by 20 matrices this procedure reduces graph size by 33 %. This approach also allows changes in the

alignment method without starting all over again from the original dataflow graph.

Figure 4.3. Dataflow Graph of Two Dimension Matrix

A submapping procedure called AG reads a dataflow graph as its input graph and searches all the actor nodes that deal with matrix data structures, i.e. STRC, STORE, and READ actors. The partition format is used to describe the selected graph. Since data structures always begin with the creation actor

(STRC) of their pointer to the first linked-list which will be used by all actors accessing the data structure, we can search the whole graph related to the matrix structure started from these actors.

Figure 4.3 shows part of a dataflow graph that initializes and accesses a two dimensional matrix structure. AG creates the simplified graph by identifying all subgraphs that start with a STRC actor. The subgraph includes all the branches of actor nodes below this actor. If an arithmetic actor is found, it terminates a branch. This arithmetic actor indicates the processing of the data structure that leads to a cross-reference to another data structure. Figure 4.4 describes the algorithm for AG.

```

Find STRC actor;
While found STRC actor
{
    Create a temporary actor list;
    While not an arithmetic actor or still more actor found
    {
        Put actor to temporary actor list;
        Get the next actor;
    }
    Find STRC actor;
}

```

Figure 4.4. Algorithm to create a simplified graph

Since a two dimensional matrix consists of one access actor (actor 1) and two branch actors (actor 2 and 3) in Figure 4.3, this procedure will find three STRC actors and group them in three partitions. This procedure also makes an adjustment to the STORE actors that immediately follow the STRC actor (actor 4 and 5 in Figure 4.3). If this STORE actor has inputs from two different STRC

actors, it will be grouped with the STRC actor as the branch graph, in this example the STORE actor 4 will be grouped with the STRC actor 2 and the STORE actor 5 with the STRC actor 3. The result of AG procedure on the graph in Figure 4.3 is demonstrated in Figure 4.5.

Figure 4.5. Simplified Graph

The pre-alignment procedure performed in the second submapping collects the matrix initialization actors (subgraphs ending with a STORE actor) and then combines all of them into the same partition with their matrix element

creator (STRC actor). Each partition is given a label, so if later we need to switch row initialization to column initialization, this can be done correctly (e.g. reversing the subscripts in a matrix). With this label we can also recognize the array access actor quickly. Because we use the partition format to carry all of this information and there is no special field added to this partition structure, the partition number is used to carry the label. For example, partition x00 is an entry point actor to array X, partition x000 is an access actor to array X, and partition x00n is an initialization of element n of array X, so the partition 100 carries the entry point actor of array 1, the partition 1000 carries the access actor to array 1, and the partition 1002 carries the initialization of the second element of array 1.

Applying this pre-alignment submapping to the graph in Figure 4.5 will re-label the partitions, split the second graph into two partitions, one with the array access actor and another with the rest of the graph, and combine all of the first matrix element initializations to partition 1001 and the second matrix initialization to partition 1002. This submapping creates four partitions as shown in Figure 4.6. It is possible that the matrix initialization is performed in a loop. In such cases, the loop will be unrolled and the counter will be replaced with a constant data token related to the index number of the matrix element creation. In the case of multiple loops for multidimensional matrix initialization, the number of loops can be reduced by duplicating the inner loop into each partition that has the related matrix structure creation.

The third submapping combines the data distribution graph (pre-aligned graph) with the original dataflow graph and keeps the partition information for use in the next submapping. Before we can align the data distribution graph, the

graph needs to be combined with the whole dataflow graph. This is because alignment requires further information, especially about the matrix index numbers which are not constant. Besides matching matrix elements with the same index number, alignment also matches matrix elements with the same source actor at their index number, such as a counter from a loop.

Figure 4.6. Data Distribution Graph

4.3. DATA ALIGNMENT

The second category is the fourth submapping procedure that can be divided into 3 major parts, data alignment, function partitioning, and optimizations. The optimizations are optional. Figure 4.7 shows the second category.

Figure 4.7. Alignment Category

The fourth submapping is the major part of the whole data partitioning procedures. There are two major parts involved in this submapping. The first part is data alignment in which the dataflow graph from the previous submapping is aligned to minimize the data cross-references, and the second part is a function partitioning that puts the rest of the dataflow graph into partitions that will be ready for allocation. After the second part, optimizations can be added to reduce the overall communication time.

Figure 4.8. Data Structure of Matrix A and B

4.3.1. THE FIRST PART (ALIGNMENT)

It was mentioned in chapter 3 that the matrix data structure can be aligned by row or column. In this step the matrix data structures are aligned based on the information from the data distribution graph. For instance, the original data structure of matrix A and matrix B (in Figure 4.8) are initialized by column, and the matrix access and the matrix process described in Figure 4.9.

This example is a part of the matrix multiplication process. We can see that the index numbers of the first READ actor of matrix B and the second READ actor of matrix A are from the same actor k . If we align matrix A by row and keep matrix B by column, there will be no cross references between these two matrices. The data alignment procedure will reverse the order of the initialization of matrix A by row as shown in Figure 4.10, and also the data access to matrix A will be reversed.

Figure 4.9. Matrix Multiplication in Dataflow Graph

Figure 4.11 shows the graph of Figure 4.9 after the alignment procedure. We can see that the read order of matrix A is reversed so that both arcs from actor k are subscripts of the first dimension of matrices A and B. Now we can remove actor k and replace it with a constant token as shown in Figure 4.12. At the same time, this graph is duplicated and distributed into different partitions with different constant index numbers according to the size of the matrix data structure and the number of partitions.

Figure 4.10. Aligned Matrix Data Structure

Figure 4.11. Aligned Matrix Multiplication in Dataflow Graph

As the matrix data structure is changed, the matrix initialization and all of the other matrix accesses to this data structure have to be changed to maintain consistency. This can be done easily because all actors that access the same data structure are stored in the same partition. We know that partition x000 keeps all the READ accesses and the arithmetic actor of the x^{th} matrix data structure which is kept in partition x00. In Figure 4.6, partition 1000 keeps all the READ accesses and the arithmetic actor of the matrix structure is kept in partition 100. The cross-reference to other data structures can be shown in this partition too (partition 1000) by checking other partitions that send results to the same arithmetic actor.

Figure 4.12. Replacing with constant data tokens

4.3.2 THE SECOND PART (PARTITIONING)

Following the alignment procedure, the partitioning procedure groups the aligned graphs with the same constant token number as created during the alignment process. Matrix creation, matrix initialization, and matrix access with the same index number are grouped in the same partition. Processing actors that can not be duplicated are put into different partitions or into the partition with a closely related graph. In the above example, where the arithmetic actors access two data structures, either we include this actor in the partition of matrix A or matrix B, and the cross reference cannot be eliminated. The partitioning procedure also searches all of the matrix access actors that relate to a particular matrix element creator and try to put them into the same partition. Remember that the matrix creator holds the data structure, so every data access to this data structure will be performed in the processor that has the matrix creator.

After all of the dataflow graphs that relate to a matrix data structure are put in the partition, it is time to partition the rest of the graph. While doing the partitioning, an optimization can be performed to reduce the communication time. The partitioning methods developed for the ParPlum system can be used internally in this procedure. We cannot produce an intermediate graph to be used as input to the standard ParPlum partitioning systems as the procedure cannot accept a partially partitioned graph as input.

4.3.3. THE THIRD PART (OPTIMIZATION)

Better parallelism can be achieved by optimizing the partitioned graph. There are 2 important optimizations developed in this research, the loop optimization and the matrix entry point optimization. A loop consists of many actors linked to each other to be repeated. If these actors are distributed to different processors, they will cause major data traffic. The first optimization puts all of the loop control actors into the same partition.

Figure 4.13. Creation and Initialization an Element of Matrix A

The second optimization reduces the steps in accessing a matrix data structure. We know that there is only one entry point to access a matrix data structure. If we already know the index number of the matrix access, which we have discovered in the alignment process, we can replace the actor directly with

an arc from that particular element of the matrix. Figure 4.13 shows the matrix creation and matrix initialization of a data structure. If the constant token values in Figure 4.13 are the same, we can reduce the communication arcs from the entry point of this data structure (the access actor.) The READ actors here (actor 11 and actor 13) are used to read the pointer to the element of the data structure stored by actor 10 which is structure created by actor 9. Because we know that all constant numbers are the same, we can remove the READ actors and connect the STORE actors directly to the STRC actor. In this way we reduce the communication to this partition from three to one. Figure 4.14 shows the optimized result of the graph in Figure 4.13.

Figure 4.14. Creation and Initialization an Element of Matrix A
After Optimization

4.4. EXAMPLE ON MATRIX MULTIPLICATION

The following is an experiment in applying the data allocation method to a matrix multiplication program. Figure 4.15 shows the original dataflow program of matrix multiplication:

$$\begin{pmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & 1 & 2 \end{pmatrix} * \begin{pmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & 1 & 2 \end{pmatrix} = C$$

A * B = C

Figure 4.15a. Initialization of matrix A and matrix B

Figure 4.15b. Structure of matrix C and multiplication of matrix A and B

For a reference, the complete dataflow program is described in appendix A.

Figure 4.15a is the creator of matrices A and B, and to simplify the program, matrix A and matrix B are initialized with the column index number. Actor 7 and 27 get the index number and the value from the same loop counter. Figure 4.15b shows the data structure creator of matrix C, which holds the result value of the matrix multiplication, and also the graph to multiply matrix A and matrix B. Actor 320201 multiplies every element of matrices A and B as read by actors 3002 and 3102. Actor 3201 (EQU) compares index k to zero and returns a Boolean value to determine which input will be put to the output arc of actor 3202 (MER), either the constant token (0) or the token that is sent by actor 3206 (SEL). This process initializes the value of the addition actor (320202) to zero if index k is equal to zero. If not, the last value that is send back from actor 3206 will be added to the new value. Remember that in the loop process, the loop index is also the iteration number. Actor 3203 (INC) increases the iteration number by 1 and actor 3204 (ICT) converts this iteration number to a token value so that can be compared to 2 by actor 3205 (LGR). If it is larger than 2, actor 3206 will pass the summation value from actor 3203 to actor 3302, which stores the value to matrix C. If the comparison is false, this value will be sent back to actor 3202 for another iteration.

The first submapping of this data allocation is to collect all of the matrix related graph nodes and the result of this procedure can be seen in a dataflow graph notation in Figure 4.16. Partitions 1000, 2000, and 3000 are the main root (entry point) of the graph for matrices A, B and C. The other partitions contain the matrix elements for each branch of the graph. The partition number 100x contains an element of matrix graph in partition 1000, partition 200x

contains an element of matrix graph in partition 2000, and so on. In this example a cross reference at actor 320201 between matrix A and matrix B can be seen in partition 1000 and partition 2000. Based on this information the next submapping will try to align the index of matrix A and B such a way to reduce or eliminate the cross reference.

```
partition(1000,alist(6,7,99,300,390,3001,3002,320201));
partition(1001,alist(301,401));
partition(1002,alist(302,402));
partition(1003,alist(303,403));
partition(2000,alist(26,27,299,600,690,3101,3102,320201));
partition(2001,alist(601,701));
partition(2002,alist(602,702));
partition(2003,alist(603,703));
partition(3000,alist(800,890,2299,3301,3302));
partition(3001,alist(801,901));
partition(3002,alist(802,902));
partition(3003,alist(803,903));
```

Figure 4.16. The output of the first submapping

Figure 4.17 shows part of the result of the second submapping. The complete graph is shown in appendix B. In this step the entry points of matrices A, B and C are separated into partitions 100, 200 and 300, so that only the matrix access graph is left in the partitions 1000, 2000, and 3000. This method gives us the ability to duplicate this graph or unroll a loop and allocate it to a different processor to exploit data parallelism. The matrix A is aligned by reversing row and column. The input of matrix initialization (actor 6 and 7) from actors 7778 and 7779 is aligned, so is the matrix read (actor 3001 and 3002.) This aligned graph is shown in Figure 4.18.

....

```
actor (7779, COPY, 0,input(INT),output((6, 2),(7, 3)));
actor (7778, COPY, 0,input(INT),output((7, 2)));
actor (6, READ, 0,input(STRUCT,INT),output((7, 1)));
actor (7, STORE, 0,input(STRUCT,INT,INT),output());
actor (99, UNLBL, 0,input(STRUCT,INT,INT),output((3002, 1),(6, 1)));
actor (3001, READ, 0,input(STRUCT,INT),output((320201, 1)));
actor (3002, READ, 0,input(STRUCT,INT),output((3001, 1)));
```

.....

```
partition(300,alist(800,890));
partition(200,alist(600,690));
partition(100,alist(300,390));
partition(1,alist(7778,7779));
partition(1000,alist(6,7,99,3001,3002,320201));
partition(1001,alist(301,401));
partition(1002,alist(302,402));
partition(1003,alist(303,403));
partition(2000,alist(26,27,299,3101,3102,320201));
partition(2001,alist(601,701));
partition(2002,alist(602,702));
partition(2003,alist(603,703));
partition(3000,alist(2299,3301,3302));
partition(3001,alist(801,901));
partition(3002,alist(802,902));
partition(3003,alist(803,903));
```

Figure 4.17. The output of the second submapping

Figure 4.18. The aligned graph of matrix A

The next step is to combine this subgraph with the rest of the graph and to partition the whole graph and apply our optimizations. In this step the matrix

access graph is duplicated according to the size of the matrix and the number of processor will be used to execute the program. Appendix C shows the dataflow graph notation after the partition procedure and appendix D is the final graph notation after applying allocation submapping and ready to be splited into the input files for child processors.

4.5. SUMMARY

There are two categories of submapping procedures in this data alignment process. The first category consists of three submapping procedures that creates a data distribution graph, and the second category consists of the major submapping procedure that does the data alignment and function partitioning. During the second category we can add an optimization to optimize the whole graph to gain a good parallelism of both data and codes. There are two optimizations we introduced in this research. The loop optimization that reduces the cross references among the loop body and the matrix access optimization that reduces the matrix data access for a certain condition.

The second category as an alignment/partition method submapping and a codes optimizer are changeable, that means we can design another method of data alignment or codes optimizer without effecting the other processes.

CHAPTER V

EXPERIMENTAL RESULTS AND DATA ANALYSIS

5.1 OVERVIEW

The experiments we run compare the total execution time among many combinations of data alignment, optimization, partitioning and allocation. We choose matrix multiplication in this experiment because matrix multiplication is the kernel of most of matrix applications especially in the numeric analysis method. For this purpose we chose 2 partition methods (part1 and part3) and 2 allocation methods (alloc1 and alloc3) that have been developed for the ParPlum system. Part1 is fine grain partitioning that puts each node into a partition and part3 is pipeline partitioning that is a little bit smarter than the first method we chose. On the other hand alloc1 is a random allocation method and alloc3 is using the LAST algorithm [9]. From all of these methods we get 8 combinations:

- Using data alignment with loop optimization + allocation1
- Using data alignment with loop optimization + allocation3
- Using data alignment without loop optimization + allocation1
- Using data alignment without loop optimization + allocation3
- Using partition1 + allocation1
- Using partition1 + allocation3

- Using partition3 + allocation1
- Using partition3 + allocation3

The combination between partition3 and allocation1 is not used because allocation1 does not produce enough allocated nodes to be fed to the number of processors used in our experiments.

5.2 THE EXPERIMENTAL RESULT

The matrix sizes we used in this experiment are 3 by 3, 5 by 5, 7 by 7, 10 by 10, and 15 by 15 run on 3, 5, and 7 processors. Ideally these experiments have to be done on machines under controlled conditions, but it is hard to achieve this condition on a LAN that can be used anytime for other purposes. Another factor that influences the experiment is the variation of the speed of system. To simulate an inter-processor communication among the workstation in UNIX LAN, ParPlum uses unreliable datagram sockets. In this experiment we experienced some lose of data because of the different speed of each system and also because of the heavy traffic of other processes. To solve this problem we implemented a delay for repeating communications to the same processor in a certain time interval. We ran an experiment to determine this time delay, and after we found the smallest value for which the program functioned reliably, we used this value for all other experiments. This adjustment causes an increase in communication time.

The experiments to get the total execution time for all of these partition and allocation combinations were conducted 6 times each for experiments running less than an hour and 4 times each for longer experiments.

Table 5.1 shows the total execution time (in seconds) of various combinations of partitioning and alignment with allocation 3 on matrix 3 by 3.

	Number of Processors		
	3	5	7
Alloc3 with loop optimization	50.2	17.3	21.0
Alloc3 without loop optimization	97.4	105.0	108.4
Partition1 with allocation3	204.2	111.8	104.8
Partition3 with allocation3	135.3	73.9	63.0

Table 5.1 Total execution time in seconds for matrix 3x3 with Allocation3.

Table 5.2 shows the total execution time (in seconds) of the same combination of partitioning and alignment with allocation 1 of a 3 by 3 matrix.

	Number of Processors		
	3	5	7
Alloc1 with loop optimization	20.2	17.3	15.0
Alloc1 without loop optimization	53.5	33.8	27.3
Part1 with alloc1	91.1	62.0	57.8

Table 5.2. Total execution time in seconds for matrix 3x3 with Allocation1

Figure 5.1. Total execution time graph in seconds of table 5.1

Figure 5.2. Total execution time graph in seconds of table 5.2

Table 5.3 and table 5.4 show the same combination of partitioning and alignment with allocation 3 and allocation1 respectively of matrix 7 by 7.

	Number of Processors		
	3	5	7
Alloc3 with loop optimization	399.9	208.3	141.1
Alloc3 without loop optimization	240.4	928.6	555.0
Part1 with alloc3	1682.4	759.6	874.8
Part3 with alloc3	1863.8	862.5	731.6

Table 5.3. Total execution time in seconds for matrix 7x7 with Allocation3

	Number of Processors		
	3	5	7
Alloc1 with loop optimization	164.9	90.4	98.9
Alloc1 without loop optimization	487.1	207.4	258.8
Part1 with alloc1	1188.2	641.1	461.0

Table 5.4. Total execution time in seconds for matrix 7x7 with Allocation1

Figure 5.3. Total execution time graph in seconds of table 5.3

Figure 5.4. Total execution time graph in seconds of table 5.4

Table 5.5 and 5.6 show the result of combination of partitioning and alignment with allocation 3 and allocation1 respectively of matrix 15 by 15.

	Number of Processors		
	3	5	7
Alloc3 with loop optimization	4021.6	1835.9	1015.3
Alloc3 without loop optimization	12865.6	7235.1	5010.0
Part1 with alloc3	22005.3	12529.6	9397.0
Part3 with alloc3	26631.1	14257.5	9732.5

Table 5.5. Total execution time in seconds for matrix 15x15 with Allocation3

	Number of Processors		
	3	5	7
Alloc1 with loop optimization	4912.1	2651.0	1126.3
Alloc1 without loop optimization	5184.0	2351.7	1730.6
Part1 with alloc1	18883.2	10514.1	7523.8

Table 5.6. Total execution time in seconds for matrix 15x15 with Allocation1

Figure 5.5. Total execution time graph in seconds of table 5.5

Figure 5.6. Total execution time graph in seconds of table 5.6

5.3. DATA ANALYSIS

From table 5.1 and table 5.2, we can see that the optimum total execution times for matrix 3 by 3 are obtained by combining data alignment and loop optimization. Data alignment itself, without loop optimization, took longer (about 45% to 70%) for number of processors more than 3 in table 5.1, primarily because of the inefficiency of using 5 or more processors for small matrix sizes. But with a different allocation method (allocation1 in table 5.2,) we can improve the performance even for more than 3 processors and with data alignment but without the optimization. Here we can see that the combination of data alignment and code alignment is very important and can support each other to improve performance.

Table 5.3 and table 5.4 in general show the same thing for matrix sizes 7 by 7, but there is an unpredictable result on combination of data alignment with allocation3 in table 5.3 that for 3 processors, the optimum result is obtained from a combination of data alignment without loop optimization. It is about 40% faster than the combination of data alignment with loop optimization. This may be because the experimental environment we used was different for each experiment. For small matrices that take a short time to execute we finished all the experiments in one time and the load of each machine can be very different every day. We done these in one time because to get the final input files for the simulator we have to do at least 5 steps of procedure and we have to do them again every time we change the experiment method.

We compared table 5.1 to table 5.2 and also table 5.3 to table 5.4; we obviously see that allocation1 (random) is a lot better than allocation3 (LAST

algorithm). All experiments using allocation1 produced a good prove that data alignment with loop optimization or without loop optimization has a better result than combination using allocation3. This shows that good combination of data alignment and code allocation is also very important. Allocation3 (LAST algorithm) tries to combine nodes that are closely related, but it seems not to balance the code distribution among the processors. This also shows that it is better using a simple allocation method after the data alignment, code partitioning and optimization. Otherwise we have to develop an allocation method to utilize the optimized code and improve the performance. For bigger matrix sizes (15 by 15) we can see improvement in total execution time for with either allocation1 or allocation3.

The improvement for 3 by 3 matrices with optimization is about 250% - 280% and without optimization the improvement is about 85% - 110%. With bigger matrix sizes (7 by 7 and 15 by 15), the improvements are about 280% - 625% with optimization and about 85% - 350% without optimization. This shows that optimization also has a big role beside the data alignment itself. The complete experiment results (tables and graphs) can be found in appendix E.

Table 5.5 and table 5.6 show that these alignments are scaleable. Performance improves for the bigger matrix sizes and over a certain number of processors. In these experiments we cannot show for more than 7 processors because of the limited number of workstations available. For 10 by 10 and 15 by 15 matrix sizes, the improvement of the total execution time is about 30% - 50% by adding 2 more processors, in this experiment from 3 to 5 and from 5 to 7 processors. For 5 by 5 and 7 by 7 matrix sizes performance seems optimal

around 5 and 7 processors and will increase the total execution time as we increase the number of processors.

SUMMARY

This experiment was conducted on several combinations of existing parallel and allocation methods with data alignment and optimization developed in this thesis. This experiment used 3 by 3, 5 by 5, 7 by 7, 10 by 10, and 15 by 15 matrix sizes in 7 processors. The overall improvements are between 85% to 625% over the old ParPlum partition and allocation method. Bigger matrix sizes have better speedup.

A load balancing in random allocation method can improve the total execution time in certain way compare to a smarter allocation algorithm such as LAST algorithm.

CHAPTER VI

CONCLUSIONS AND FUTURE DEVELOPMENT

This thesis presented the design and automated evaluation of data alignment and function partitioning in the ParPlum system to improve the performance without user or programmer involvement. The addition of new data structure to the system allows us to run more variety applications on the ParPlum system, such as matrices calculations. An optimization of the instruction codes was also introduced to show that both data and code parallelism are very important in dataflow graphs.

With different allocation methods, the system performance of small matrix sizes with data alignment and function partitioning is almost the same, specifically if the number of processors is more than the size of the matrix, we obtained unexpected results. But better performance can be achieved with the addition of optimization. On the bigger matrix sizes, better performance can be achieved using data alignment and function partitioning alone.

We see that the data alignment and function partitioning alone give about 85% - 350% improvement over the old function partitioning method, but with the addition of code optimization, additional improvement can be achieved. Also we saw from the experimental results that a matched function partitioning should be used after the data alignment. Even a smart function partitioning algorithm is

not a guarantee of better results. This experiment shows that the random partitioning algorithm does better over the LAST algorithm because the data alignment has already optimized some of the codes. In this case random algorithm balance the system load equally but not the LAST algorithm.

As the main goal of this research is to see the effect of using data and function partitioning at the same time that no other research has done before, we obtained a good result on the overall performance by choosing a match data and function partitioning even on the unreliable environment and tool such as ParPlum. We believe that using other people experiments on data distribution and function partitioning we can combine and create a good method to improve the result of this experiment.

In this thesis we used a simple linked list data structure which limits our method to align the matrix data elements, though our experiments show a big improvement in the total execution time. We believe that using a better data structure that allows us to use better alignment method will further improve performance. With a more reliable data communication method than the datagram so that we do not need to implement the delay will speedup the process, and also with a high level language to describe the dataflow graph will help speedup the process to find data relations and distributions.

More research and experiments are also needed to see the impact of these data alignment and function partitioning methods with or without code optimizations for bigger matrix sizes on more reliable dataflow simulator, that has lower overhead time to start the simulator, we can get the result as close as to the real world.

In this experiment we use the Dataflow language that has a good way to express parallel processing code, but these results can be used to see the same effect for other language as long as we can find a way to get the data relations or data connections among the distributed data. We can obtain a reference from them to optimize the distribution of the data and the function code as well to improve the performance.

Finally, we conclude that both matched data alignment and function partitioning can be used in a system to improve the overall performance. With more research and development, the dataflow system in general and the ParPlum in specific can be very effective to compute matrix calculations in less expensive system such as LAN.

REFERENCES

- [1] Manish Gupta and P. Banerjee, "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers," *IEEE Transactions on Parallel and Distributed System*, Vol. 3., No. 2, p179-193, March 1992.
- [2] J. Ramanujam and P. Sadayappan, "Compile-Time Techniques for Data Distribution in Distributed Memory Machines," *IEEE Transactions on Parallel and Distributed System*, Vol. 2., No. 4, p472-482, October 1991.
- [3] Jingke Li and Marina Chen, "Generating Explicit Communication from Shared-Memory Program References," *Proceedings "Supercomputing '90,"* p865-876, November 12-16, 1990.
- [4] Jingke Li and Marina Chen, "Index Domain Alignment: Minimizing Cost of Cross-Referencing Between Distributed Arrays," Technical Report YALEU/DCS/TR-725, Yale University, October 1989.
- [5] Arvind and D. E. Culler, "Dataflow Architectures," *Annual Reviews Computer Science*, Massachusetts Institute of Technology, Cambridge, p225-253, 1986
- [6] Jean-Luc Gaudiot, "Structure Handling in Data-Flow System," *IEEE Transactions on Computer*, Vol. c-35, No. 6, p489-502, June 1986.
- [7] Jingsong Fu, "ParPlum: A System for Evaluating Parallel Program Optimization Methods," MS thesis, Portland State University, August 1991.
- [8] Dezheng Tang and Michael A. Driscoll, "Mapping Programs to Parallel Architectures in The Real World," Technical Report PSU-EE-92-010, Portland State University, March 18, 1992.

- [9] Jeff Baxter and Janak H. Patel, "The LAST Algorithm: A Heuristic-Based Static Task Allocation Algorithm," *1989 International Conference on Parallel Processing*, Computer System Group, Coordinated Science Lab., University of Illinois at Urbana-Champaign.
- [10] Kemal Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *Computer*, p50-56, June 1982.
- [11] J. B. Dennis, "First Version of a Data Flow Procedure Language," *Programming Symp.: Porc. Colloque sur la Programmation*, Paris, France. Apr. 1974.
- [12] Arvind and R. E. Thomas, "I-Structure: An Efficient Data Type for Functional Languages," Technical Report LCS/TM-178, Lab. Computer Science, Massachusetts Institute of Technology, Cambridge, June 1980.
- [13] Bilha Mendelson and Gabriel M. Silberman, "Mapping Data Flow Programs on a VLSI Array of Processors," *1987 International Conference on Parallel Processing*, p871-873
- [14] Michael A. Driscoll, Jingsong Fu, Satish M. Pai, Chintamani Patwardhan, Liono Setiowijoso, De-Zheng Tang, Kiswanto Thayib, "Sensitivity Analysis and Mapping Program to Parallel Architectures," *Proceeding of 1991 International Conference on Parallel Processing*, p272-273, August 1991.
- [15] Santosh S. Pande, Dharma P. Agrawal, and Jon Mauney, "Compiling Functional Parallelism on Distributed-Memory System," *IEEE: Parallel and Distributed Technology*, Vol. 2., No. 1, p64-76, Spring 1994.
- [16] O. Babaglu, L. Alvisi, A. Amoroso, and R. Davoli, "Paralex: An Environment for Parallel Programming in Distributed System," *Technical Report UB-LCS-91-01, University of Bologna*, Feb 6, 1991.
- [17] G.M. Papadopoulos and K.R. Traub, "Multithreading: A Revisionist View of Dataflow Architectures," *Proc. 18th Annual Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1991, pp. 342-351.
- [18] V.G. Grafe and J.E. Hoch, "The Epsilon-2 Multiprocessor System," *J. Parallel & Distributed Computing*, Vol. 10, 1990, pp. 309-318.

- [19] R.S. Nikhil and Arvind, "Can Dataflow Subsume von Neumann Computing?" *Proc. 16th Annual Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1989, pp. 262-272.
- [20] Ben Lee and A. R. Hurson, "Dataflow Architectures and Multithreading," *Computer*, pp. 27-39, August 1994.

APPENDIX A: Dataflow Graph Program for Matrix Multiplication

```
# Machine information for allocation submapping
Machine : SUN;
Numprocs : 3;
proc_name(GEPPETTO,GEPPETTO,GEPPETTO);

# Structure of matrix A (3 x 3)
actor(300,STRC,0,input(int),output((390,1),(99,1),(2990,1)));
actor(301,STRC,0,input(int),output((401,3)));
actor(302,STRC,0,input(int),output((402,3)));
actor(303,STRC,0,input(int),output((403,3)));

actor(390,COPY,0,input(struct),output((401,1),(402,1),(403,1)));

actor(401,STORE,0,input(struct,int,struct),output());
actor(402,STORE,0,input(struct,int,struct),output());
actor(403,STORE,0,input(struct,int,struct),output());

data(0,int,0,0,into((300,1)));
data(0,int,0,0,into((301,1),(302,1),(303,1)));
data(0,int,0,0,into((401,2)));
data(1,int,0,0,into((402,2)));
data(2,int,0,0,into((403,2)));

# initialization of matrix A
actor(7001,COPY,0,input(int),output((7002,1),(7003,1),(7004,1),(7005,1)));
actor(7002,LBL,0,input(int),output((7003,2),(7004,2),(7005,2),(100,1)));
actor(7003,LBL1,0,input(int,int),output((7006,1)));
actor(7004,LBL2,0,input(int,int),output((7012,2)));
actor(7005,LBL3,0,input(int,int),output((7012,3)));
actor(7006,ICT,0,input(int),output((7007,1),(7008,2)));
actor(7007,LGR,0,input(int,int),output((7008,1)));
actor(7008,SEL,0,input(bool,int),output((7011,1),(7009,1)));
actor(7009,COPY,0,input(int),output((7777,1),(7010,1)));
actor(7010,INC,0,input(int),output((7006,1)));
actor(7011,DCRE,0,input(int),output((7012,1)));
actor(7012,UNLBL,0,input(int,int,int),output((100,1)));

actor(7777,COPY,0,input(int),output((8001,1),(8501,1)));

actor(8001,COPY,0,input(int),output((8002,1),(8003,1),(8004,1),(8005,1)));
actor(8002,LBL,0,input(int),output((8003,2),(8004,2),(8005,2),(8502,1)));
actor(8003,LBL1,0,input(int,int),output((8006,1)));
actor(8004,LBL2,0,input(int,int),output((8012,2)));
actor(8005,LBL3,0,input(int,int),output((8012,3)));
actor(8006,ICT,0,input(int),output((8007,1),(8008,2)));
actor(8007,LGR,0,input(int,int),output((8008,1)));
actor(8008,SEL,0,input(bool,int),output((8011,1),(8009,1)));
```

```

actor(8009,COPY,0,input(int),output((7779,1),(8010,1)));
actor(8010,INC,0,input(int),output((8006,1)));
actor(8011,DCRE,0,input(int),output((8012,1)));
actor(8012,UNLBL,0,input(int,int,int),output((100,1)));
actor(8501,COPY,0,input(INT),output((8503,1),(8504,1),(8505,1)));
actor(8502,COPY,0,input(int),output((8503,2),(8504,2),(8505,2)));
actor(8503,LBL1,0,input(INT,int),output((8506,1),(8508,2)));
actor(8504,LBL2,0,input(INT,int),output((8512,2)));
actor(8505,LBL3,0,input(INT,int),output((8512,3)));
actor(8506,ICT,0,input(INT),output((8507,1)));
actor(8507,LGR,0,input(int,int),output((8508,1)));
actor(8508,SEL,0,input(bool,INT),output((8511,1),(8509,1)));
actor(8509,COPY,0,input(INT),output((7778,1),(8510,1)));
actor(8510,INC,0,input(INT),output((8506,1),(8508,2)));
actor(8511,DCRE,0,input(INT),output((8512,1)));
actor(8512,UNLBL,0,input(INT,int,int),output((100,1)));

```

```

actor(7778,COPY,0,input(int),output((6,2)));
actor(7779,COPY,0,input(int),output((7,2),(7,3)));

```

```

actor(6,READ,0,input(struct,int),output((7,1)));
actor(7,STORE,0,input(struct,int,int),output());

```

```

# Number of elements of matrix A: 0 - n
data(2,int,-1,-1,into((7007,2),(8007,2),(8507,2)));

```

```

# Structure of matrix B (3 x 3)
actor(600,STRC,0,input(int),output((690,1),(299,1),(298,1)));
actor(601,STRC,0,input(int),output((701,3)));
actor(602,STRC,0,input(int),output((702,3)));
actor(603,STRC,0,input(int),output((703,3)));

```

```

actor(690,COPY,0,input(struct),output((701,1),(702,1),(703,1)));

```

```

actor(701,STORE,0,input(struct,int,struct),output());
actor(702,STORE,0,input(struct,int,struct),output());
actor(703,STORE,0,input(struct,int,struct),output());

```

```

actor(299,UNLBL,0,input(struct,int,int),output((3101,1),(26,1)));
actor(298,ICT,0,input(struct),output((9001,1)));

```

```

data(0,int,0,0,into((600,1)));
data(0,int,0,0,into((601,1),(602,1),(603,1)));
data(0,int,0,0,into((701,2)));
data(1,int,0,0,into((702,2)));
data(2,int,0,0,into((703,2)));

```

```

# Initialization of matrix B
actor(9001,COPY,0,input(int),output((9002,1),(9003,1),(9004,1),(9005,1)));
actor(9002,LBL,0,input(int),output((9003,2),(9004,2),(9005,2),(100,1)));
actor(9003,LBL1,0,input(int,int),output((9006,1)));
actor(9004,LBL2,0,input(int,int),output((9012,2)));
actor(9005,LBL3,0,input(int,int),output((9012,3)));
actor(9006,ICT,0,input(int),output((9007,1),(9008,2)));

actor(9007,LGR,0,input(int,int),output((9008,1)));
actor(9008,SEL,0,input(bool,int),output((9011,1),(9009,1)));
actor(9009,COPY,0,input(int),output((8887,1),(9010,1)));
actor(9010,INC,0,input(int),output((9006,1)));
actor(9011,DCRE,0,input(int),output((9012,1)));
actor(9012,UNLBL,0,input(int,int,int),output((100,1)));

actor(8887,COPY,0,input(int),output((18001,1),(18501,1)));

actor(18001,COPY,0,input(int),output((18002,1),(18003,1),(18004,1),(18005,1)));
actor(18002,LBL,0,input(int),output((18003,2),(18004,2),(18005,2),(18502,1)));
actor(18003,LBL1,0,input(int,int),output((18006,1)));
actor(18004,LBL2,0,input(int,int),output((18012,2)));
actor(18005,LBL3,0,input(int,int),output((18012,3)));
actor(18006,ICT,0,input(int),output((18007,1),(18008,2)));
actor(18007,LGR,0,input(int,int),output((18008,1)));
actor(18008,SEL,0,input(bool,int),output((18011,1),(18009,1)));
actor(18009,COPY,0,input(int),output((8889,1),(18010,1)));
actor(18010,INC,0,input(int),output((18006,1)));
actor(18011,DCRE,0,input(int),output((18012,1)));
actor(18012,UNLBL,0,input(int,int,int),output((100,1)));
actor(18501,COPY,0,input(INT),output((18503,1),(18504,1),(18505,1)));
actor(18502,COPY,0,input(int),output((18503,2),(18504,2),(18505,2)));
actor(18503,LBL1,0,input(INT,int),output((18506,1),(18508,2)));
actor(18504,LBL2,0,input(INT,int),output((18512,2)));
actor(18505,LBL3,0,input(INT,int),output((18512,3)));
actor(18506,ICT,0,input(INT),output((18507,1)));
actor(18507,LGR,0,input(int,int),output((18508,1)));
actor(18508,SEL,0,input(bool,INT),output((18511,1),(18509,1)));
actor(18509,COPY,0,input(INT),output((8888,1),(18510,1)));
actor(18510,INC,0,input(INT),output((18506,1),(18508,2)));
actor(18511,DCRE,0,input(INT),output((18512,1)));
actor(18512,UNLBL,0,input(INT,int,int),output((100,1)));

# Number of elements of matrix B: 0 - n
data(2,int,-1,-1,into((9007,2),(18007,2),(18507,2)));

actor(8888,COPY,0,input(int),output((26,2)));
actor(8889,COPY,0,input(int),output((27,2),(27,3)));

actor(26,READ,0,input(struct,int),output((27,1)));
actor(27,STORE,0,input(struct,int,int),output());

```

```

# Structure of matrix C (3 x 3)
actor(800,STRC,0,input(int),output((890,1),(2299,1)));
actor(801,STRC,0,input(int),output((901,3)));
actor(802,STRC,0,input(int),output((902,3)));
actor(803,STRC,0,input(int),output((903,3)));

actor(890,COPY,0,input(struct),output((901,1),(902,1),(903,1)));

actor(901,STORE,0,input(struct,int,struct),output());
actor(902,STORE,0,input(struct,int,struct),output());
actor(903,STORE,0,input(struct,int,struct),output());

actor(2299,UNLBL,0,input(struct,int,int),output((3301,1)));
actor(2990,ICT,0,input(struct),output((1001,1),(7001,1)));

data(0,int,0,0,into((800,1)));
data(0,int,0,0,into((801,1),(802,1),(803,1)));
data(0,int,0,0,into((901,2)));
data(1,int,0,0,into((902,2)));
data(2,int,0,0,into((903,2)));

# Loop i=0 to 2
#   Loop j=0 to 2
#     Loop k=0 to 2
actor(1001,COPY,1,input(int),output((1002,1),(1003,1),(1004,1),(1005,1)));
actor(1002,LBL,0,input(int),output((1003,2),(1004,2),(1005,2),(100,1)));
actor(1003,LBL1,0,input(int,int),output((1006,1)));
actor(1004,LBL2,0,input(int,int),output((1012,2)));
actor(1005,LBL3,0,input(int,int),output((1012,3)));
actor(1006,ICT,0,input(int),output((1007,1),(1008,2)));
actor(1007,LGR,0,input(int,int),output((1008,1)));
actor(1008,SEL,0,input(bool,int),output((1011,1),(1009,1)));
actor(1009,COPY,0,input(int),output((111,1),(1010,1)));
actor(1010,INC,0,input(int),output((1006,1)));
actor(1011,DCRE,0,input(int),output((1012,1)));
actor(1012,UNLBL,0,input(int,int,int),output((100,1)));

actor(111,COPY,0,input(int),output((2001,1),(2501,1)));

actor(2001,COPY,0,input(int),output((2002,1),(2003,1),(2004,1),(2005,1)));
actor(2002,LBL,0,input(int),output((2003,2),(2004,2),(2005,2),(2502,1)));
actor(2003,LBL1,0,input(int,int),output((2006,1)));
actor(2004,LBL2,0,input(int,int),output((2012,2)));
actor(2005,LBL3,0,input(int,int),output((2012,3)));
actor(2006,ICT,0,input(int),output((2007,1),(2008,2)));
actor(2007,LGR,0,input(int,int),output((2008,1)));
actor(2008,SEL,0,input(bool,int),output((2011,1),(2009,1)));
actor(2009,COPY,0,input(int),output((112,1),(2010,1)));
actor(2010,INC,0,input(int),output((2006,1)));
actor(2011,DCRE,0,input(int),output((2012,1)));

```

```

actor(2012,UNLBL,0,input(int,int,int),output((100,1)));
actor(2501,COPY,0,input(INT),output((2503,1),(2504,1),(2505,1)));
actor(2502,COPY,0,input(int),output((2503,2),(2504,2),(2505,2)));
actor(2503,LBL1,0,input(INT,int),output((2506,1),(2508,2)));
actor(2504,LBL2,0,input(INT,int),output((2512,2)));
actor(2505,LBL3,0,input(INT,int),output((2512,3)));
actor(2506,ICT,0,input(INT),output((2507,1)));
actor(2507,LGR,0,input(int,int),output((2508,1)));
actor(2508,SEL,0,input(bool,INT),output((2511,1),(2509,1)));
actor(2509,COPY,0,input(INT),output((5501,1),(2510,1)));
actor(2510,INC,0,input(INT),output((2506,1),(2508,2)));
actor(2511,DCRE,0,input(INT),output((2512,1)));
actor(2512,UNLBL,0,input(INT,int,int),output((100,1)));
actor(112,COPY,0,input(int),output((4001,1),(4501,1)));
actor(4001,COPY,0,input(int),output((4002,1),(4003,1),(4004,1),(4005,1)));
actor(4002,LBL,0,input(int),output((4003,2),(4004,2),(4005,2),(5502,1),(4502,1)));
actor(4003,LBL1,0,input(int,int),output((4006,1)));
actor(4004,LBL2,0,input(int,int),output((4012,2)));
actor(4005,LBL3,0,input(int,int),output((4012,3)));
actor(4006,ICT,0,input(int),output((4007,1),(4008,2)));
actor(4007,LGR,0,input(int,int),output((4008,1)));
actor(4008,SEL,0,input(bool,int),output((4011,1),(4009,1)));
actor(4009,COPY,0,input(int),output((116,1),(4010,1)));
actor(4010,INC,0,input(int),output((4006,1)));
actor(4011,DCRE,0,input(int),output((4012,1)));
actor(4012,UNLBL,0,input(int,int,int),output((100,1)));
actor(113,COPY,0,input(int),output((5502,1),(4502,1)));
actor(4501,COPY,0,input(INT),output((4503,1),(4504,1),(4505,1)));
actor(4502,COPY,0,input(int),output((4503,2),(4504,2),(4505,2)));
actor(4503,LBL1,0,input(INT,int),output((4506,1),(4508,2)));
actor(4504,LBL2,0,input(INT,int),output((4512,2)));
actor(4505,LBL3,0,input(INT,int),output((4512,3)));
actor(4506,ICT,0,input(INT),output((4507,1)));
actor(4507,LGR,0,input(int,int),output((4508,1)));
actor(4508,SEL,0,input(bool,INT),output((4511,1),(4509,1)));
actor(4509,COPY,0,input(INT),output((115,1),(4510,1)));
actor(4510,INC,0,input(INT),output((4506,1),(4508,2)));
actor(4511,DCRE,0,input(INT),output((4512,1)));
actor(4512,UNLBL,0,input(INT,int,int),output((100,1)));
actor(5501,COPY,0,input(INT),output((5503,1),(5504,1),(5505,1)));
actor(5502,COPY,0,input(int),output((5503,2),(5504,2),(5505,2)));
actor(5503,LBL1,0,input(INT,int),output((5506,1),(5508,2)));
actor(5504,LBL2,0,input(INT,int),output((5512,2)));
actor(5505,LBL3,0,input(INT,int),output((5512,3)));
actor(5506,ICT,0,input(INT),output((5507,1)));
actor(5507,LGR,0,input(int,int),output((5508,1)));
actor(5508,SEL,0,input(bool,INT),output((5511,1),(5509,1)));
actor(5509,COPY,0,input(INT),output((114,1),(5510,1)));
actor(5510,INC,0,input(INT),output((5506,1),(5508,2)));
actor(5511,DCRE,0,input(INT),output((5512,1)));
actor(5512,UNLBL,0,input(INT,int,int),output((100,1)));

```



```

# counter i
actor(114,COPY,0,input(int),output((3001,2),(3301,2)));
# counter j
actor(115,COPY,0,input(int),output((3102,2),(3302,2)));
# counter k
actor(116,COPY,0,input(int),output((3002,2),(3101,2),(3201,1)));

actor(99,UNLBL,0,input(struct,int,int),output((3001,1),(6,1)));

# Read element of matrix A
actor(3001,READ,0,input(struct,int),output((3002,1)));
actor(3002,READ,0,input(struct,int),output((320201,1)));

# Read element of matrix B
actor(3101,READ,0,input(struct,int),output((3102,1)));
actor(3102,READ,0,input(struct,int),output((320201,2)));

# Multiply the elements of matrix A and matrix B
actor(3201,EQL,0,input(int,int),output((3202,1)));
actor(3202,MER,0,input(bool,int,int),output((320202,2)));
actor(320201,MUL,0,input(int,int),output((320202,1)));
actor(320202,ADD,0,input(int,int),output((3203,1)));
actor(3203,INC,0,input(int),output((3204,1),(3206,2)));
actor(3204,ICT,0,input(int),output((3205,1)));
actor(3205,LGR,0,input(int,int),output((3206,1)));
actor(3206,SEL,0,input(bool,int),output((3210,1),(3207,1)));
actor(3207,COPY,0,input(int),output((3202,3),(3202,2)));
actor(3209,UNLBL,0,input(int,int,int),output((3202,3),(3202,2)));
actor(3210,DCRE,0,input(int),output((911,1)));
actor(3211,ICT,0,input(int),output((3209,1)));

data(0,int,-1,-1,into((3201,2)));
data(0,int,-1,-1,into((3209,2)));
data(-1,int,-1,-1,into((3209,3)));
data(0,int,0,-1,into((3202,2)));
data(0,int,0,-1,into((3202,3)));

# Store the result to matrix C
actor(911,COPY,0,input(int),output((3302,3),(3211,1)));
actor(3301,READ,0,input(struct,int),output((3302,1)));
actor(3302,STORE,0,input(struct,int,int),output((6001,2)));

actor(100,COPY,0,input(int),output());

# Counter to decide when to stop
actor(6001,CNTR,0,input(int,int),output((6002,1)));
actor(6002,END,0,input(int),output());

```

```
# total result n x n
data(9,int,-1,-1,into((6001,1)));

# loop index for marix multiplying 0 - n
data(2,int,-1,-1,into((1007,2),(2007,2),(2507,2)));
data(2,int,-1,-1,into((4007,2),(4507,2),(5507,2)));
data(2,int,-1,-1,into((3205,2)));

# Data for constant token (iter # = -1)
data(-1,int,0,0,into((99,2),(99,3)));
data(-1,int,0,0,into((299,2),(299,3)));
data(-1,int,0,0,into((2299,2),(2299,3)));
```

APPENDIX B: Data Distribution Graph In ParPlum Format

```

Machine : SUN;
Numpprocs : 3;
proc_name(GEPPETTO, GEPPETTO, GEPPETTO);
actor (7779, COPY, 0,input(INT),output((6, 2),(7, 3)));
actor (7778, COPY, 0,input(INT),output((7, 2)));
actor (6, READ, 0,input(STRUCT,INT),output((7, 1)));
actor (7, STORE, 0,input(STRUCT,INT,INT),output());
actor (99, UNLBL, 0,input(STRUCT,INT,INT),output((3002, 1),(6, 1)));
actor (300, STRC, 0,input(INT),output((390, 1),(99, 1),(2990, 1)));
actor (390, COPY, 0,input(STRUCT),output((401, 1),(402, 1),(403, 1)));
actor (3001, READ, 0,input(STRUCT,INT),output((320201, 1)));
actor (3002, READ, 0,input(STRUCT,INT),output((3001, 1)));
actor (320201, MUL, 0,input(INT,INT),output((320202, 1)));
actor (301, STRC, 0,input(INT),output((401, 3)));
actor (401, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (302, STRC, 0,input(INT),output((402, 3)));
actor (402, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (303, STRC, 0,input(INT),output((403, 3)));
actor (403, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (26, READ, 0,input(STRUCT,INT),output((27, 1)));
actor (27, STORE, 0,input(STRUCT,INT,INT),output());
actor (299, UNLBL, 0,input(STRUCT,INT,INT),output((3101, 1),(26, 1)));
actor (600, STRC, 0,input(INT),output((690, 1),(299, 1),(298, 1)));
actor (690, COPY, 0,input(STRUCT),output((701, 1),(702, 1),(703, 1)));
actor (3101, READ, 0,input(STRUCT,INT),output((3102, 1)));
actor (3102, READ, 0,input(STRUCT,INT),output((320201, 2)));
actor (320201, MUL, 0,input(INT,INT),output((320202, 1)));
actor (601, STRC, 0,input(INT),output((701, 3)));
actor (701, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (602, STRC, 0,input(INT),output((702, 3)));
actor (702, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (603, STRC, 0,input(INT),output((703, 3)));
actor (703, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (800, STRC, 0,input(INT),output((890, 1),(2299, 1)));
actor (890, COPY, 0,input(STRUCT),output((901, 1),(902, 1),(903, 1)));
actor (2299, UNLBL, 0,input(STRUCT,INT,INT),output((3301, 1)));
actor (3301, READ, 0,input(STRUCT,INT),output((3302, 1)));
actor (3302, STORE, 0,input(STRUCT,INT,INT),output((6001, 2)));
actor (801, STRC, 0,input(INT),output((901, 3)));
actor (901, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (802, STRC, 0,input(INT),output((902, 3)));
actor (902, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (803, STRC, 0,input(INT),output((903, 3)));
actor (903, STORE, 0,input(STRUCT,INT,STRUCT),output());
data (-1, INT, 0, 0, into((99, 3)));
data (-1, INT, 0, 0, into((99, 2)));
data (0, INT, 0, 0, into((300, 1)));

```

```

data (0, INT, 0, 0, into((301, 1)));
data (0, INT, 0, 0, into((401, 2)));
data (0, INT, 0, 0, into((302, 1)));
data (1, INT, 0, 0, into((402, 2)));
data (0, INT, 0, 0, into((303, 1)));
data (2, INT, 0, 0, into((403, 2)));
data (-1, INT, 0, 0, into((299, 3)));
data (-1, INT, 0, 0, into((299, 2)));
data (0, INT, 0, 0, into((600, 1)));
data (0, INT, 0, 0, into((601, 1)));
data (0, INT, 0, 0, into((701, 2)));
data (0, INT, 0, 0, into((602, 1)));
data (1, INT, 0, 0, into((702, 2)));
data (0, INT, 0, 0, into((603, 1)));
data (2, INT, 0, 0, into((703, 2)));
data (0, INT, 0, 0, into((800, 1)));
data (-1, INT, 0, 0, into((2299, 3)));
data (-1, INT, 0, 0, into((2299, 2)));
data (0, INT, 0, 0, into((801, 1)));
data (0, INT, 0, 0, into((901, 2)));
data (0, INT, 0, 0, into((802, 1)));
data (1, INT, 0, 0, into((902, 2)));
data (0, INT, 0, 0, into((803, 1)));
data (2, INT, 0, 0, into((903, 2)));

```

Aligned Graph.

```

partition(300,alist(800,890));
partition(200,alist(600,690));
partition(100,alist(300,390));
partition(1,alist(7778,7779));
partition(1000,alist(6,7,99,3001,3002,320201));
partition(1001,alist(301,401));
partition(1002,alist(302,402));
partition(1003,alist(303,403));
partition(2000,alist(26,27,299,3101,3102,320201));
partition(2001,alist(601,701));
partition(2002,alist(602,702));
partition(2003,alist(603,703));
partition(3000,alist(2299,3301,3302));
partition(3001,alist(801,901));
partition(3002,alist(802,902));
partition(3003,alist(803,903));

```

APPENDIX C: Output Graph Of Data Alignment In ParPlum Format

```

Machine : SUN;
Numprocs : 3;
proc_name(GEPETTO, GEPETTO, GEPETTO);

actor (6, READ, 0,input(STRUCT,INT),output((7, 1)));
actor (7, STORE, 0,input(STRUCT,INT,INT),output());
actor (26, READ, 0,input(STRUCT,INT),output((27, 1)));
actor (27, STORE, 0,input(STRUCT,INT,INT),output());
actor (99, UNLBL, 0,input(STRUCT,INT,INT),output((3002, 1),(6, 1),(320210, 1),(320205, 1),(320221, 1),(320216, 1)));
actor (100, COPY, 0,input(INT),output());
actor (111, COPY, 0,input(INT),output((2001, 1),(2501, 1)));
actor (112, COPY, 0,input(INT),output((4001, 1),(4501, 1),(115, 1)));
actor (113, COPY, 0,input(INT),output((5502, 1),(4502, 1)));
actor (114, COPY, 0,input(INT),output((3001, 2),(3301, 2),(320209, 2),(320220, 2)));
actor (115, COPY, 0,input(INT),output((3102, 2),(3302, 2),(320212, 2),(320223, 2)));
actor (116, COPY, 0,input(INT),output((3201, 1)));
actor (298, ICT, 0,input(STRUCT),output((9001, 1)));
actor (299, UNLBL, 0,input(STRUCT,INT,INT),output((3101, 1),(26, 1),(320211, 1),(320207, 1),(320222, 1),(320218, 1)));
actor (300, STRC, 0,input(INT),output((390, 1),(99, 1),(2990, 1)));
actor (301, STRC, 0,input(INT),output((401, 3)));
actor (302, STRC, 0,input(INT),output((402, 3)));
actor (303, STRC, 0,input(INT),output((403, 3)));
actor (390, COPY, 0,input(STRUCT),output((401, 1),(402, 1),(403, 1)));
actor (401, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (402, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (403, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (600, STRC, 0,input(INT),output((690, 1),(299, 1),(298, 1)));
actor (601, STRC, 0,input(INT),output((701, 3)));
actor (602, STRC, 0,input(INT),output((702, 3)));
actor (603, STRC, 0,input(INT),output((703, 3)));
actor (690, COPY, 0,input(STRUCT),output((701, 1),(702, 1),(703, 1)));
actor (701, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (702, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (703, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (800, STRC, 0,input(INT),output((890, 1),(2299, 1)));
actor (801, STRC, 0,input(INT),output((901, 3)));
actor (802, STRC, 0,input(INT),output((902, 3)));
actor (803, STRC, 0,input(INT),output((903, 3)));
actor (890, COPY, 0,input(STRUCT),output((901, 1),(902, 1),(903, 1)));
actor (901, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (902, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (903, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (911, COPY, 0,input(INT),output());
actor (1001, COPY, 1,input(INT),output((1002, 1),(1003, 1),(1004, 1),(1005, 1)));
actor (1002, LBL, 0,input(INT),output((1003, 2),(1004, 2),(1005, 2),(100, 1)));

```

```

actor (1003, LBL1, 0,input(INT,INT),output((1006, 1)));
actor (1004, LBL2, 0,input(INT,INT),output((1012, 2)));
actor (1005, LBL3, 0,input(INT,INT),output((1012, 3)));
actor (1006, ICT, 0,input(INT),output((1007, 1),(1008, 2)));
actor (1007, LGR, 0,input(INT,INT),output((1008, 1)));
actor (1008, SEL, 0,input(BOOL,INT),output((1011, 1),(1009, 1)));
actor (1009, COPY, 0,input(INT),output((111, 1),(1010, 1)));
actor (1010, INC, 0,input(INT),output((1006, 1)));
actor (1011, DCRE, 0,input(INT),output((1012, 1)));
actor (1012, UNLBL, 0,input(INT,INT,INT),output((100, 1)));
actor (2001, COPY, 0,input(INT),output((2002, 1),(2003, 1),(2004, 1),(2005, 1)));
actor (2002, LBL, 0,input(INT),output((2003, 2),(2004, 2),(2005, 2),(2502, 1)));
actor (2003, LBL1, 0,input(INT,INT),output((2006, 1)));
actor (2004, LBL2, 0,input(INT,INT),output((2012, 2)));
actor (2005, LBL3, 0,input(INT,INT),output((2012, 3)));
actor (2006, ICT, 0,input(INT),output((2007, 1),(2008, 2)));
actor (2007, LGR, 0,input(INT,INT),output((2008, 1)));
actor (2008, SEL, 0,input(BOOL,INT),output((2011, 1),(2009, 1)));
actor (2009, COPY, 0,input(INT),output((112, 1),(2010, 1)));
actor (2010, INC, 0,input(INT),output((2006, 1)));
actor (2011, DCRE, 0,input(INT),output((2012, 1)));
actor (2012, UNLBL, 0,input(INT,INT,INT),output((100, 1)));
actor (2299, UNLBL, 0,input(STRUCT,INT,INT),output((3301, 1)));
actor (2501, COPY, 0,input(INT),output((2503, 1),(2504, 1),(2505, 1)));
actor (2502, COPY, 0,input(INT),output((2503, 2),(2504, 2),(2505, 2)));
actor (2503, LBL1, 0,input(INT,INT),output((2506, 1),(2508, 2)));
actor (2504, LBL2, 0,input(INT,INT),output((2512, 2)));
actor (2505, LBL3, 0,input(INT,INT),output((2512, 3)));
actor (2506, ICT, 0,input(INT),output((2507, 1)));
actor (2507, LGR, 0,input(INT,INT),output((2508, 1)));
actor (2508, SEL, 0,input(BOOL,INT),output((2511, 1),(2509, 1)));
actor (2509, COPY, 0,input(INT),output((5501, 1),(2510, 1),(114, 1)));
actor (2510, INC, 0,input(INT),output((2506, 1),(2508, 2)));
actor (2511, DCRE, 0,input(INT),output((2512, 1)));
actor (2512, UNLBL, 0,input(INT,INT,INT),output((100, 1)));
actor (2990, ICT, 0,input(STRUCT),output((1001, 1),(7001, 1)));
actor (3001, READ, 0,input(STRUCT,INT),output((320201, 1)));
actor (3002, READ, 0,input(STRUCT,INT),output((3001, 1)));
actor (3101, READ, 0,input(STRUCT,INT),output((3102, 1)));
actor (3102, READ, 0,input(STRUCT,INT),output((320201, 2)));
actor (3201, EQL, 0,input(INT,INT),output((3202, 1)));
actor (3202, MER, 0,input(BOOL,INT,INT),output());
actor (3203, INC, 0,input(INT),output((3204, 1),(3206, 2)));
actor (3204, ICT, 0,input(INT),output((3205, 1)));
actor (3205, LGR, 0,input(INT,INT),output((3206, 1)));
actor (3206, SEL, 0,input(BOOL,INT),output());
actor (3207, COPY, 0,input(INT),output((3202, 3),(3202, 2)));
actor (3209, UNLBL, 0,input(INT,INT,INT),output((3202, 3),(3202, 2)));
actor (3210, DCRE, 0,input(INT),output());
actor (3211, ICT, 0,input(INT),output((3209, 1)));
actor (3212, DCRE, 0,input(INT),output((911, 1)));

```

```

actor (3301, READ, 0,input(STRUCT,INT),output((3302, 1)));
actor (3302, STORE, 0,input(STRUCT,INT,INT),output((6001, 2)));
actor (4001, COPY, 0,input(INT),output((4002, 1),(4003, 1),(4004, 1),(4005, 1)));
actor (4002, LBL, 0,input(INT),output((4003, 2),(4004, 2),(4005, 2),(5502, 1),(4502, 1)));
actor (4003, LBL1, 0,input(INT,INT),output((4006, 1)));
actor (4004, LBL2, 0,input(INT,INT),output((4012, 2)));
actor (4005, LBL3, 0,input(INT,INT),output((4012, 3)));
actor (4006, ICT, 0,input(INT),output((4007, 1),(4008, 2)));
actor (4007, LGR, 0,input(INT,INT),output((4008, 1)));
actor (4008, SEL, 0,input(BOOL,INT),output((4011, 1),(4009, 1)));
actor (4009, COPY, 0,input(INT),output((116, 1),(4010, 1)));
actor (4010, INC, 0,input(INT),output((4006, 1)));
actor (4011, DCRE, 0,input(INT),output((4012, 1)));
actor (4012, UNLBL, 0,input(INT,INT,INT),output((100, 1)));
actor (4501, COPY, 0,input(INT),output((4503, 1),(4504, 1),(4505, 1)));
actor (4502, COPY, 0,input(INT),output((4503, 2),(4504, 2),(4505, 2)));
actor (4503, LBL1, 0,input(INT,INT),output((4506, 1),(4508, 2)));
actor (4504, LBL2, 0,input(INT,INT),output((4512, 2)));
actor (4505, LBL3, 0,input(INT,INT),output((4512, 3)));
actor (4506, ICT, 0,input(INT),output((4507, 1)));
actor (4507, LGR, 0,input(INT,INT),output((4508, 1)));
actor (4508, SEL, 0,input(BOOL,INT),output((4511, 1),(4509, 1)));
actor (4509, COPY, 0,input(INT),output((4510, 1)));
actor (4510, INC, 0,input(INT),output((4506, 1),(4508, 2)));
actor (4511, DCRE, 0,input(INT),output((4512, 1)));
actor (4512, UNLBL, 0,input(INT,INT,INT),output((100, 1)));
actor (5501, COPY, 0,input(INT),output((5503, 1),(5504, 1),(5505, 1)));
actor (5502, COPY, 0,input(INT),output((5503, 2),(5504, 2),(5505, 2)));
actor (5503, LBL1, 0,input(INT,INT),output((5506, 1),(5508, 2)));
actor (5504, LBL2, 0,input(INT,INT),output((5512, 2)));
actor (5505, LBL3, 0,input(INT,INT),output((5512, 3)));
actor (5506, ICT, 0,input(INT),output((5507, 1)));
actor (5507, LGR, 0,input(INT,INT),output((5508, 1)));
actor (5508, SEL, 0,input(BOOL,INT),output((5511, 1),(5509, 1)));
actor (5509, COPY, 0,input(INT),output((5510, 1)));
actor (5510, INC, 0,input(INT),output((5506, 1),(5508, 2)));
actor (5511, DCRE, 0,input(INT),output((5512, 1)));
actor (5512, UNLBL, 0,input(INT,INT,INT),output((100, 1)));
actor (6001, CNTR, 0,input(INT,INT),output((6002, 1)));
actor (6002, END, 0,input(INT),output());
actor (7001, COPY, 0,input(INT),output((7002, 1),(7003, 1),(7004, 1),(7005, 1)));
actor (7002, LBL, 0,input(INT),output((7003, 2),(7004, 2),(7005, 2),(100, 1)));
actor (7003, LBL1, 0,input(INT,INT),output((7006, 1)));
actor (7004, LBL2, 0,input(INT,INT),output((7012, 2)));
actor (7005, LBL3, 0,input(INT,INT),output((7012, 3)));
actor (7006, ICT, 0,input(INT),output((7007, 1),(7008, 2)));
actor (7007, LGR, 0,input(INT,INT),output((7008, 1)));
actor (7008, SEL, 0,input(BOOL,INT),output((7011, 1),(7009, 1)));
actor (7009, COPY, 0,input(INT),output((7777, 1),(7010, 1)));
actor (7010, INC, 0,input(INT),output((7006, 1)));
actor (7011, DCRE, 0,input(INT),output((7012, 1)));

```

```

actor (7012, UNLBL, 0,input(INT,INT,INT),output((100, 1)));
actor (7777, COPY, 0,input(INT),output((7, 2),(320206, 2),(320217, 2)));
actor (7778, COPY, 0,input(INT),output());
actor (7779, COPY, 0,input(INT),output((6, 2),(7, 3)));
actor (8001, COPY, 0,input(INT),output((8002, 1),(8003, 1),(8004, 1),(8005, 1)));
actor (8002, LBL, 0,input(INT),output((8003, 2),(8004, 2),(8005, 2),(8502, 1)));
actor (8003, LBL1, 0,input(INT,INT),output((8006, 1)));
actor (8004, LBL2, 0,input(INT,INT),output((8012, 2)));
actor (8005, LBL3, 0,input(INT,INT),output((8012, 3)));
actor (8006, ICT, 0,input(INT),output((8007, 1),(8008, 2)));
actor (8007, LGR, 0,input(INT,INT),output((8008, 1)));
actor (8008, SEL, 0,input(BOOL,INT),output((8011, 1),(8009, 1)));
actor (8009, COPY, 0,input(INT),output((7779, 1),(8010, 1),(320204, 1),(320215, 1)));
actor (8010, INC, 0,input(INT),output((8006, 1)));
actor (8011, DCRE, 0,input(INT),output((8012, 1)));
actor (8012, UNLBL, 0,input(INT,INT,INT),output((100, 1)));
actor (8501, COPY, 0,input(INT),output((8503, 1),(8504, 1),(8505, 1)));
actor (8502, COPY, 0,input(INT),output((8503, 2),(8504, 2),(8505, 2)));
actor (8503, LBL1, 0,input(INT,INT),output((8506, 1),(8508, 2)));
actor (8504, LBL2, 0,input(INT,INT),output((8512, 2)));
actor (8505, LBL3, 0,input(INT,INT),output((8512, 3)));
actor (8506, ICT, 0,input(INT),output((8507, 1)));
actor (8507, LGR, 0,input(INT,INT),output((8508, 1)));
actor (8508, SEL, 0,input(BOOL,INT),output((8511, 1),(8509, 1)));
actor (8509, COPY, 0,input(INT),output((7778, 1),(8510, 1)));
actor (8510, INC, 0,input(INT),output((8506, 1),(8508, 2)));
actor (8511, DCRE, 0,input(INT),output((8512, 1)));
actor (8512, UNLBL, 0,input(INT,INT,INT),output((100, 1)));
actor (8887, COPY, 0,input(INT),output((27, 2),(27, 3),(320208, 2),(320208, 3),(320219, 2),(320219, 3)));
actor (8888, COPY, 0,input(INT),output((26, 2)));
actor (8889, COPY, 0,input(INT),output());
actor (9001, COPY, 0,input(INT),output((9002, 1),(9003, 1),(9004, 1),(9005, 1)));
actor (9002, LBL, 0,input(INT),output((9003, 2),(9004, 2),(9005, 2),(100, 1)));
actor (9003, LBL1, 0,input(INT,INT),output((9006, 1)));
actor (9004, LBL2, 0,input(INT,INT),output((9012, 2)));
actor (9005, LBL3, 0,input(INT,INT),output((9012, 3)));
actor (9006, ICT, 0,input(INT),output((9007, 1),(9008, 2)));
actor (9007, LGR, 0,input(INT,INT),output((9008, 1)));
actor (9008, SEL, 0,input(BOOL,INT),output((9011, 1),(9009, 1)));
actor (9009, COPY, 0,input(INT),output((8887, 1),(9010, 1)));
actor (9010, INC, 0,input(INT),output((9006, 1)));
actor (9011, DCRE, 0,input(INT),output((9012, 1)));
actor (9012, UNLBL, 0,input(INT,INT,INT),output((100, 1)));
actor (18001, COPY, 0,input(INT),output((18002, 1),(18003, 1),(18004, 1),(18005, 1)));
actor (18002, LBL, 0,input(INT),output((18003, 2),(18004, 2),(18005, 2),(18502, 1)));
actor (18003, LBL1, 0,input(INT,INT),output((18006, 1)));
actor (18004, LBL2, 0,input(INT,INT),output((18012, 2)));
actor (18005, LBL3, 0,input(INT,INT),output((18012, 3)));
actor (18006, ICT, 0,input(INT),output((18007, 1),(18008, 2)));
actor (18007, LGR, 0,input(INT,INT),output((18008, 1)));

```



```

actor (18008, SEL, 0,input(BOOL,INT),output((18011, 1),(18009, 1)));
actor (18009, COPY, 0,input(INT),output((8889, 1),(18010, 1)));
actor (18010, INC, 0,input(INT),output((18006, 1)));
actor (18011, DCRE, 0,input(INT),output((18012, 1)));
actor (18012, UNLBL, 0,input(INT,INT,INT),output((100, 1)));
actor (18501, COPY, 0,input(INT),output((18503, 1),(18504, 1),(18505, 1)));
actor (18502, COPY, 0,input(INT),output((18503, 2),(18504, 2),(18505, 2)));
actor (18503, LBL1, 0,input(INT,INT),output((18506, 1),(18508, 2)));
actor (18504, LBL2, 0,input(INT,INT),output((18512, 2)));
actor (18505, LBL3, 0,input(INT,INT),output((18512, 3)));
actor (18506, ICT, 0,input(INT),output((18507, 1)));
actor (18507, LGR, 0,input(INT,INT),output((18508, 1)));
actor (18508, SEL, 0,input(BOOL,INT),output((18511, 1),(18509, 1)));
actor (18509, COPY, 0,input(INT),output((8888, 1),(18510, 1),(320203, 1),(320214, 1)));
actor (18510, INC, 0,input(INT),output((18506, 1),(18508, 2)));
actor (18511, DCRE, 0,input(INT),output((18512, 1)));
actor (18512, UNLBL, 0,input(INT,INT,INT),output((100, 1)));
actor (320201, MUL, 0,input(INT,INT),output((320202, 1)));
actor (320202, ADD, 0,input(INT,INT,INT),output((3302, 3)));
actor (320203, COPY, 0,input(INT),output((320207, 2)));
actor (320204, COPY, 0,input(INT),output((320205, 2),(320206, 3)));
actor (320205, READ, 0,input(STRUCT,INT),output((320206, 1)));
actor (320206, STORE, 0,input(STRUCT,INT,INT),output());
actor (320207, READ, 0,input(STRUCT,INT),output((320208, 1)));
actor (320208, STORE, 0,input(STRUCT,INT,INT),output());
actor (320209, READ, 0,input(STRUCT,INT),output((320213, 1)));
actor (320210, READ, 0,input(STRUCT,INT),output((320209, 1)));
actor (320211, READ, 0,input(STRUCT,INT),output((320212, 1)));
actor (320212, READ, 0,input(STRUCT,INT),output((320213, 2)));
actor (320213, MUL, 0,input(INT,INT),output((320202, 2)));
actor (320214, COPY, 0,input(INT),output((320218, 2)));
actor (320215, COPY, 0,input(INT),output((320216, 2),(320217, 3)));
actor (320216, READ, 0,input(STRUCT,INT),output((320217, 1)));
actor (320217, STORE, 0,input(STRUCT,INT,INT),output());
actor (320218, READ, 0,input(STRUCT,INT),output((320219, 1)));
actor (320219, STORE, 0,input(STRUCT,INT,INT),output());
actor (320220, READ, 0,input(STRUCT,INT),output((320224, 1)));
actor (320221, READ, 0,input(STRUCT,INT),output((320220, 1)));
actor (320222, READ, 0,input(STRUCT,INT),output((320223, 1)));
actor (320223, READ, 0,input(STRUCT,INT),output((320224, 2)));
actor (320224, MUL, 0,input(INT,INT),output((320202, 3)));

data (-1, INT, 0, 0, into((99, 3)));
data (-1, INT, 0, 0, into((99, 2)));
data (-1, INT, 0, 0, into((299, 3)));
data (-1, INT, 0, 0, into((299, 2)));
data (0, INT, 0, 0, into((300, 1)));
data (0, INT, 0, 0, into((301, 1)));
data (0, INT, 0, 0, into((302, 1)));
data (0, INT, 0, 0, into((303, 1)));
data (0, INT, 0, 0, into((401, 2)));

```

```

data (1, INT, 0, 0, into((402, 2)));
data (2, INT, 0, 0, into((403, 2)));
data (0, INT, 0, 0, into((600, 1)));
data (0, INT, 0, 0, into((601, 1)));
data (0, INT, 0, 0, into((602, 1)));
data (0, INT, 0, 0, into((603, 1)));
data (0, INT, 0, 0, into((701, 2)));
data (1, INT, 0, 0, into((702, 2)));
data (2, INT, 0, 0, into((703, 2)));
data (0, INT, 0, 0, into((800, 1)));
data (0, INT, 0, 0, into((801, 1)));
data (0, INT, 0, 0, into((802, 1)));
data (0, INT, 0, 0, into((803, 1)));
data (0, INT, 0, 0, into((901, 2)));
data (1, INT, 0, 0, into((902, 2)));
data (2, INT, 0, 0, into((903, 2)));
data (2, INT, -1, -1, into((1007, 2)));
data (2, INT, -1, -1, into((2007, 2)));
data (-1, INT, 0, 0, into((2299, 3)));
data (-1, INT, 0, 0, into((2299, 2)));
data (2, INT, -1, -1, into((2507, 2)));
data (0, INT, -1, -1, into((3002, 2)));
data (0, INT, -1, -1, into((3101, 2)));
data (0, INT, -1, -1, into((3201, 2)));
data (0, INT, 0, -1, into((3202, 3)));
data (0, INT, 0, -1, into((3202, 2)));
data (2, INT, -1, -1, into((3205, 2)));
data (-1, INT, -1, -1, into((3209, 3)));
data (0, INT, -1, -1, into((3209, 2)));
data (2, INT, -1, -1, into((4007, 2)));
data (2, INT, -1, -1, into((4507, 2)));
data (2, INT, -1, -1, into((5507, 2)));
data (9, INT, -1, -1, into((6001, 1)));
data (2, INT, -1, -1, into((7007, 2)));
data (0, INT, -1, -1, into((7779, 1)));
data (2, INT, -1, -1, into((8007, 2)));
data (2, INT, -1, -1, into((8507, 2)));
data (0, INT, -1, -1, into((8888, 1)));
data (2, INT, -1, -1, into((9007, 2)));
data (2, INT, -1, -1, into((18007, 2)));
data (2, INT, -1, -1, into((18507, 2)));
data (1, INT, -1, -1, into((320203, 1)));
data (1, INT, -1, -1, into((320204, 1)));
data (1, INT, -1, -1, into((320210, 2)));
data (1, INT, -1, -1, into((320211, 2)));
data (2, INT, -1, -1, into((320214, 1)));
data (2, INT, -1, -1, into((320215, 1)));
data (2, INT, -1, -1, into((320221, 2)));
data (2, INT, -1, -1, into((320222, 2)));

```

```

partition(0,alist(800,890));
partition(1,alist(600,690));
partition(2,alist(300,390));
partition(3,alist(7778,7779));
partition(4,alist(6,7,99,3001,3002,320201,26,27,299,3101,3102,301,401,601,701));
partition(5,alist(320203,320204,320205,320206,320207,320208,320209,320210,320211,320212,320213,302,402,602,702));
partition(6,alist(320214,320215,320216,320217,320218,320219,320220,320221,320222,320223,320224,303,403,603,703));
partition(7,alist(2299,3301,3302,802,902));
partition(8,alist(801,901));
partition(9,alist(803,903));
partition(10,alist(100,1002,1012,2010,2508,3205,4004,4502,4512,5510,7006,8003,8501,8511,9006,18004,18502,18512));
partition(11,alist(111,1003,2001,2011,2509,3206,4005,4503,5501,5511,7007,8004,8502,8512,9007,18005,18503,320202));
partition(12,alist(112,1004,2002,2012,2510,3207,4006,4504,5502,5512,7008,8005,8503,8887,9008,18006,18504));
partition(13,alist(113,1005,2003,2501,2511,3209,4007,4505,5503,6001,7009,8006,8504,8888,9009,18007,18505));
partition(14,alist(114,1006,2004,2502,2512,3210,4008,4506,5504,6002,7010,8007,8505,8889,9010,18008,18506));
partition(15,alist(115,1007,2005,2503,2990,3211,4009,4507,5505,7001,7011,8008,8506,9001,9011,18009,18507));
partition(16,alist(116,1008,2006,2504,3201,3212,4010,4508,5506,7002,7012,8009,8507,9002,9012,18010,18508));
partition(17,alist(298,1009,2007,2505,3202,4001,4011,4509,5507,7003,7777,8010,8508,9003,18001,18011,18509));
partition(18,alist(911,1010,2008,2506,3203,4002,4012,4510,5508,7004,8001,8011,8509,9004,18002,18012,18510));
partition(19,alist(1001,1011,2009,2507,3204,4003,4501,4511,5509,7005,8002,8012,8510,9005,18003,18501,18511));

```

APPENDIX D: Output Graph Of Allocation Sub Mapping

Machine : SUN;

Numprocs : 3;

proc_name(GEPETTO, GEPETTO, GEPETTO);

```

actor (6, READ, 0,input(STRUCT,INT),output((7, 1)));
actor (7, STORE, 0,input(STRUCT,INT,INT),output());
actor (26, READ, 0,input(STRUCT,INT),output((27, 1)));
actor (27, STORE, 0,input(STRUCT,INT,INT),output());
actor (99, UNLBL, 0,input(STRUCT,INT,INT),output((3002, 1),(6, 1),(320210, 1),(320205,
1),(320221, 1),(320216, 1)));
actor (100, COPY, 0,input(INT),output());
actor (111, COPY, 0,input(INT),output((2001, 1),(2501, 1)));
actor (112, COPY, 0,input(INT),output((4001, 1),(4501, 1),(115, 1)));
actor (113, COPY, 0,input(INT),output((5502, 1),(4502, 1)));
actor (114, COPY, 0,input(INT),output((3001, 2),(3301, 2),(320209, 2),(320220, 2)));
actor (115, COPY, 0,input(INT),output((3102, 2),(3302, 2),(320212, 2),(320223, 2)));
actor (116, COPY, 0,input(INT),output((3201, 1)));
actor (298, ICT, 0,input(STRUCT),output((9001, 1)));
actor (299, UNLBL, 0,input(STRUCT,INT,INT),output((3101, 1),(26, 1),(320211, 1),(320207,
1),(320222, 1),(320218, 1)));
actor (300, STRC, 0,input(INT),output((390, 1),(99, 1),(2990, 1)));
actor (301, STRC, 0,input(INT),output((401, 3)));
actor (302, STRC, 0,input(INT),output((402, 3)));
actor (303, STRC, 0,input(INT),output((403, 3)));
actor (390, COPY, 0,input(STRUCT),output((401, 1),(402, 1),(403, 1)));
actor (401, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (402, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (403, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (600, STRC, 0,input(INT),output((690, 1),(299, 1),(298, 1)));
actor (601, STRC, 0,input(INT),output((701, 3)));
actor (602, STRC, 0,input(INT),output((702, 3)));
actor (603, STRC, 0,input(INT),output((703, 3)));
actor (690, COPY, 0,input(STRUCT),output((701, 1),(702, 1),(703, 1)));
actor (701, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (702, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (703, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (800, STRC, 0,input(INT),output((890, 1),(2299, 1)));
actor (801, STRC, 0,input(INT),output((901, 3)));
actor (802, STRC, 0,input(INT),output((902, 3)));
actor (803, STRC, 0,input(INT),output((903, 3)));
actor (890, COPY, 0,input(STRUCT),output((901, 1),(902, 1),(903, 1)));
actor (901, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (902, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (903, STORE, 0,input(STRUCT,INT,STRUCT),output());
actor (911, COPY, 0,input(INT),output());
actor (1001, COPY, 1,input(INT),output((1002, 1),(1003, 1),(1004, 1),(1005, 1)));
actor (1002, LBL, 0,input(INT),output((1003, 2),(1004, 2),(1005, 2),(100, 1)));

```

```

actor (1003, LBL1, 0,input(INT,INT),output((1006, 1)));
actor (1004, LBL2, 0,input(INT,INT),output((1012, 2)));
actor (1005, LBL3, 0,input(INT,INT),output((1012, 3)));
actor (1006, ICT, 0,input(INT),output((1007, 1),(1008, 2)));
actor (1007, LGR, 0,input(INT,INT),output((1008, 1)));
actor (1008, SEL, 0,input(BOOL,INT),output((1011, 1),(1009, 1)));
actor (1009, COPY, 0,input(INT),output((111, 1),(1010, 1)));
actor (1010, INC, 0,input(INT),output((1006, 1)));
actor (1011, DCRE, 0,input(INT),output((1012, 1)));
actor (1012, UNLBL, 0,input(INT,INT,INT),output((100, 1)));
actor (2001, COPY, 0,input(INT),output((2002, 1),(2003, 1),(2004, 1),(2005, 1)));
actor (2002, LBL, 0,input(INT),output((2003, 2),(2004, 2),(2005, 2),(2502, 1)));
actor (2003, LBL1, 0,input(INT,INT),output((2006, 1)));
actor (2004, LBL2, 0,input(INT,INT),output((2012, 2)));
actor (2005, LBL3, 0,input(INT,INT),output((2012, 3)));
actor (2006, ICT, 0,input(INT),output((2007, 1),(2008, 2)));
actor (2007, LGR, 0,input(INT,INT),output((2008, 1)));
actor (2008, SEL, 0,input(BOOL,INT),output((2011, 1),(2009, 1)));
actor (2009, COPY, 0,input(INT),output((112, 1),(2010, 1)));
actor (2010, INC, 0,input(INT),output((2006, 1)));
actor (2011, DCRE, 0,input(INT),output((2012, 1)));
actor (2012, UNLBL, 0,input(INT,INT,INT),output((100, 1)));
actor (2299, UNLBL, 0,input(STRUCT,INT,INT),output((3301, 1)));
actor (2501, COPY, 0,input(INT),output((2503, 1),(2504, 1),(2505, 1)));
actor (2502, COPY, 0,input(INT),output((2503, 2),(2504, 2),(2505, 2)));
actor (2503, LBL1, 0,input(INT,INT),output((2506, 1),(2508, 2)));
actor (2504, LBL2, 0,input(INT,INT),output((2512, 2)));
actor (2505, LBL3, 0,input(INT,INT),output((2512, 3)));
actor (2506, ICT, 0,input(INT),output((2507, 1)));
actor (2507, LGR, 0,input(INT,INT),output((2508, 1)));
actor (2508, SEL, 0,input(BOOL,INT),output((2511, 1),(2509, 1)));
actor (2509, COPY, 0,input(INT),output((5501, 1),(2510, 1),(114, 1)));
actor (2510, INC, 0,input(INT),output((2506, 1),(2508, 2)));
actor (2511, DCRE, 0,input(INT),output((2512, 1)));
actor (2512, UNLBL, 0,input(INT,INT,INT),output((100, 1)));
actor (2990, ICT, 0,input(STRUCT),output((1001, 1),(7001, 1)));
actor (3001, READ, 0,input(STRUCT,INT),output((320201, 1)));
actor (3002, READ, 0,input(STRUCT,INT),output((3001, 1)));
actor (3101, READ, 0,input(STRUCT,INT),output((3102, 1)));
actor (3102, READ, 0,input(STRUCT,INT),output((320201, 2)));
actor (3201, EQL, 0,input(INT,INT),output((3202, 1)));
actor (3202, MER, 0,input(BOOL,INT,INT),output());
actor (3203, INC, 0,input(INT),output((3204, 1),(3206, 2)));
actor (3204, ICT, 0,input(INT),output((3205, 1)));
actor (3205, LGR, 0,input(INT,INT),output((3206, 1)));
actor (3206, SEL, 0,input(BOOL,INT),output());
actor (3207, COPY, 0,input(INT),output((3202, 3),(3202, 2)));
actor (3209, UNLBL, 0,input(INT,INT,INT),output((3202, 3),(3202, 2)));
actor (3210, DCRE, 0,input(INT),output());
actor (3211, ICT, 0,input(INT),output((3209, 1)));
actor (3301, READ, 0,input(STRUCT,INT),output((3302, 1)));

```

```

actor (3302, STORE, 0,input(STRUCT,INT,INT),output((6001, 2)));
actor (4001, COPY, 0,input(INT),output((4002, 1),(4003, 1),(4004, 1),(4005, 1)));
actor (4002, LBL, 0,input(INT),output((4003, 2),(4004, 2),(4005, 2),(5502, 1),(4502, 1)));
actor (4003, LBL1, 0,input(INT,INT),output((4006, 1)));
actor (4004, LBL2, 0,input(INT,INT),output((4012, 2)));
actor (4005, LBL3, 0,input(INT,INT),output((4012, 3)));
actor (4006, ICT, 0,input(INT),output((4007, 1),(4008, 2)));
actor (4007, LGR, 0,input(INT,INT),output((4008, 1)));
actor (4008, SEL, 0,input(BOOL,INT),output((4011, 1),(4009, 1)));
actor (4009, COPY, 0,input(INT),output((116, 1),(4010, 1)));
actor (4010, INC, 0,input(INT),output((4006, 1)));
actor (4011, DCRE, 0,input(INT),output((4012, 1)));
actor (4012, UNLBL, 0,input(INT,INT,INT),output((100, 1)));
actor (4501, COPY, 0,input(INT),output((4503, 1),(4504, 1),(4505, 1)));
actor (4502, COPY, 0,input(INT),output((4503, 2),(4504, 2),(4505, 2)));
actor (4503, LBL1, 0,input(INT,INT),output((4506, 1),(4508, 2)));
actor (4504, LBL2, 0,input(INT,INT),output((4512, 2)));
actor (4505, LBL3, 0,input(INT,INT),output((4512, 3)));
actor (4506, ICT, 0,input(INT),output((4507, 1)));
actor (4507, LGR, 0,input(INT,INT),output((4508, 1)));
actor (4508, SEL, 0,input(BOOL,INT),output((4511, 1),(4509, 1)));
actor (4509, COPY, 0,input(INT),output((4510, 1)));
actor (4510, INC, 0,input(INT),output((4506, 1),(4508, 2)));
actor (4511, DCRE, 0,input(INT),output((4512, 1)));
actor (4512, UNLBL, 0,input(INT,INT,INT),output((100, 1)));
actor (5501, COPY, 0,input(INT),output((5503, 1),(5504, 1),(5505, 1)));
actor (5502, COPY, 0,input(INT),output((5503, 2),(5504, 2),(5505, 2)));
actor (5503, LBL1, 0,input(INT,INT),output((5506, 1),(5508, 2)));
actor (5504, LBL2, 0,input(INT,INT),output((5512, 2)));
actor (5505, LBL3, 0,input(INT,INT),output((5512, 3)));
actor (5506, ICT, 0,input(INT),output((5507, 1)));
actor (5507, LGR, 0,input(INT,INT),output((5508, 1)));
actor (5508, SEL, 0,input(BOOL,INT),output((5511, 1),(5509, 1)));
actor (5509, COPY, 0,input(INT),output((5510, 1)));
actor (5510, INC, 0,input(INT),output((5506, 1),(5508, 2)));
actor (5511, DCRE, 0,input(INT),output((5512, 1)));
actor (5512, UNLBL, 0,input(INT,INT,INT),output((100, 1)));
actor (6001, CNTR, 0,input(INT,INT),output((6002, 1)));
actor (6002, END, 0,input(INT),output());
actor (7001, COPY, 0,input(INT),output((7002, 1),(7003, 1),(7004, 1),(7005, 1)));
actor (7002, LBL, 0,input(INT),output((7003, 2),(7004, 2),(7005, 2),(100, 1)));
actor (7003, LBL1, 0,input(INT,INT),output((7006, 1)));
actor (7004, LBL2, 0,input(INT,INT),output((7012, 2)));
actor (7005, LBL3, 0,input(INT,INT),output((7012, 3)));
actor (7006, ICT, 0,input(INT),output((7007, 1),(7008, 2)));
actor (7007, LGR, 0,input(INT,INT),output((7008, 1)));
actor (7008, SEL, 0,input(BOOL,INT),output((7011, 1),(7009, 1)));
actor (7009, COPY, 0,input(INT),output((7777, 1),(7010, 1)));
actor (7010, INC, 0,input(INT),output((7006, 1)));
actor (7011, DCRE, 0,input(INT),output((7012, 1)));
actor (7012, UNLBL, 0,input(INT,INT,INT),output((100, 1)));

```

```

actor (7777, COPY, 0,input(INT),output((7, 2),(320206, 2),(320217, 2)));
actor (7778, COPY, 0,input(INT),output());
actor (7779, COPY, 0,input(INT),output((6, 2),(7, 3)));
actor (8001, COPY, 0,input(INT),output((8002, 1),(8003, 1),(8004, 1),(8005, 1)));
actor (8002, LBL, 0,input(INT),output((8003, 2),(8004, 2),(8005, 2),(8502, 1)));
actor (8003, LBL1, 0,input(INT,INT),output((8006, 1)));
actor (8004, LBL2, 0,input(INT,INT),output((8012, 2)));
actor (8005, LBL3, 0,input(INT,INT),output((8012, 3)));
actor (8006, ICT, 0,input(INT),output((8007, 1),(8008, 2)));
actor (8007, LGR, 0,input(INT,INT),output((8008, 1)));
actor (8008, SEL, 0,input(BOOL,INT),output((8011, 1),(8009, 1)));
actor (8009, COPY, 0,input(INT),output((7779, 1),(8010, 1),(320204, 1),(320215, 1)));
actor (8010, INC, 0,input(INT),output((8006, 1)));
actor (8011, DCRE, 0,input(INT),output((8012, 1)));
actor (8012, UNLBL, 0,input(INT,INT,INT),output((100, 1)));
actor (8501, COPY, 0,input(INT),output((8503, 1),(8504, 1),(8505, 1)));
actor (8502, COPY, 0,input(INT),output((8503, 2),(8504, 2),(8505, 2)));
actor (8503, LBL1, 0,input(INT,INT),output((8506, 1),(8508, 2)));
actor (8504, LBL2, 0,input(INT,INT),output((8512, 2)));
actor (8505, LBL3, 0,input(INT,INT),output((8512, 3)));
actor (8506, ICT, 0,input(INT),output((8507, 1)));
actor (8507, LGR, 0,input(INT,INT),output((8508, 1)));
actor (8508, SEL, 0,input(BOOL,INT),output((8511, 1),(8509, 1)));
actor (8509, COPY, 0,input(INT),output((7778, 1),(8510, 1)));
actor (8510, INC, 0,input(INT),output((8506, 1),(8508, 2)));
actor (8511, DCRE, 0,input(INT),output((8512, 1)));
actor (8512, UNLBL, 0,input(INT,INT,INT),output((100, 1)));
actor (8887, COPY, 0,input(INT),output((27, 2),(27, 3),(320208, 2),(320208, 3),(320219, 2),(320219, 3)));
actor (8888, COPY, 0,input(INT),output((26, 2)));
actor (8889, COPY, 0,input(INT),output());
actor (9001, COPY, 0,input(INT),output((9002, 1),(9003, 1),(9004, 1),(9005, 1)));
actor (9002, LBL, 0,input(INT),output((9003, 2),(9004, 2),(9005, 2),(100, 1)));
actor (9003, LBL1, 0,input(INT,INT),output((9006, 1)));
actor (9004, LBL2, 0,input(INT,INT),output((9012, 2)));
actor (9005, LBL3, 0,input(INT,INT),output((9012, 3)));
actor (9006, ICT, 0,input(INT),output((9007, 1),(9008, 2)));
actor (9007, LGR, 0,input(INT,INT),output((9008, 1)));
actor (9008, SEL, 0,input(BOOL,INT),output((9011, 1),(9009, 1)));
actor (9009, COPY, 0,input(INT),output((8887, 1),(9010, 1)));
actor (9010, INC, 0,input(INT),output((9006, 1)));
actor (9011, DCRE, 0,input(INT),output((9012, 1)));
actor (9012, UNLBL, 0,input(INT,INT,INT),output((100, 1)));
actor (18001, COPY, 0,input(INT),output((18002, 1),(18003, 1),(18004, 1),(18005, 1)));
actor (18002, LBL, 0,input(INT),output((18003, 2),(18004, 2),(18005, 2),(18502, 1)));
actor (18003, LBL1, 0,input(INT,INT),output((18006, 1)));
actor (18004, LBL2, 0,input(INT,INT),output((18012, 2)));
actor (18005, LBL3, 0,input(INT,INT),output((18012, 3)));
actor (18006, ICT, 0,input(INT),output((18007, 1),(18008, 2)));
actor (18007, LGR, 0,input(INT,INT),output((18008, 1)));
actor (18008, SEL, 0,input(BOOL,INT),output((18011, 1),(18009, 1)));

```

```

actor (18009, COPY, 0,input(INT),output((8889, 1),(18010, 1)));
actor (18010, INC, 0,input(INT),output((18006, 1)));
actor (18011, DCRE, 0,input(INT),output((18012, 1)));
actor (18012, UNLBL, 0,input(INT,INT,INT),output((100, 1)));
actor (18501, COPY, 0,input(INT),output((18503, 1),(18504, 1),(18505, 1)));
actor (18502, COPY, 0,input(INT),output((18503, 2),(18504, 2),(18505, 2)));
actor (18503, LBL1, 0,input(INT,INT),output((18506, 1),(18508, 2)));
actor (18504, LBL2, 0,input(INT,INT),output((18512, 2)));
actor (18505, LBL3, 0,input(INT,INT),output((18512, 3)));
actor (18506, ICT, 0,input(INT),output((18507, 1)));
actor (18507, LGR, 0,input(INT,INT),output((18508, 1)));
actor (18508, SEL, 0,input(BOOL,INT),output((18511, 1),(18509, 1)));
actor (18509, COPY, 0,input(INT),output((8888, 1),(18510, 1),(320203, 1),(320214, 1)));
actor (18510, INC, 0,input(INT),output((18506, 1),(18508, 2)));
actor (18511, DCRE, 0,input(INT),output((18512, 1)));
actor (18512, UNLBL, 0,input(INT,INT,INT),output((100, 1)));
actor (320201, MUL, 0,input(INT,INT),output((320202, 1)));
actor (320202, ADD, 0,input(INT,INT,INT),output((3302, 3)));
actor (320203, COPY, 0,input(INT),output((320207, 2)));
actor (320204, COPY, 0,input(INT),output((320205, 2),(320206, 3)));
actor (320205, READ, 0,input(STRUCT,INT),output((320206, 1)));
actor (320206, STORE, 0,input(STRUCT,INT,INT),output());
actor (320207, READ, 0,input(STRUCT,INT),output((320208, 1)));
actor (320208, STORE, 0,input(STRUCT,INT,INT),output());
actor (320209, READ, 0,input(STRUCT,INT),output((320213, 1)));
actor (320210, READ, 0,input(STRUCT,INT),output((320209, 1)));
actor (320211, READ, 0,input(STRUCT,INT),output((320212, 1)));
actor (320212, READ, 0,input(STRUCT,INT),output((320213, 2)));
actor (320213, MUL, 0,input(INT,INT),output((320202, 2)));
actor (320214, COPY, 0,input(INT),output((320218, 2)));
actor (320215, COPY, 0,input(INT),output((320216, 2),(320217, 3)));
actor (320216, READ, 0,input(STRUCT,INT),output((320217, 1)));
actor (320217, STORE, 0,input(STRUCT,INT,INT),output());
actor (320218, READ, 0,input(STRUCT,INT),output((320219, 1)));
actor (320219, STORE, 0,input(STRUCT,INT,INT),output());
actor (320220, READ, 0,input(STRUCT,INT),output((320224, 1)));
actor (320221, READ, 0,input(STRUCT,INT),output((320220, 1)));
actor (320222, READ, 0,input(STRUCT,INT),output((320223, 1)));
actor (320223, READ, 0,input(STRUCT,INT),output((320224, 2)));
actor (320224, MUL, 0,input(INT,INT),output((320202, 3)));

data (-1, INT, 0, 0, into((99, 3)));
data (-1, INT, 0, 0, into((99, 2)));
data (-1, INT, 0, 0, into((299, 3)));
data (-1, INT, 0, 0, into((299, 2)));
data (0, INT, 0, 0, into((300, 1)));
data (0, INT, 0, 0, into((301, 1)));
data (0, INT, 0, 0, into((302, 1)));
data (0, INT, 0, 0, into((303, 1)));
data (0, INT, 0, 0, into((401, 2)));
data (1, INT, 0, 0, into((402, 2)));

```



```

data (2, INT, 0, 0, into((403, 2)));
data (0, INT, 0, 0, into((600, 1)));
data (0, INT, 0, 0, into((601, 1)));
data (0, INT, 0, 0, into((602, 1)));
data (0, INT, 0, 0, into((603, 1)));
data (0, INT, 0, 0, into((701, 2)));
data (1, INT, 0, 0, into((702, 2)));
data (2, INT, 0, 0, into((703, 2)));
data (0, INT, 0, 0, into((800, 1)));
data (0, INT, 0, 0, into((801, 1)));
data (0, INT, 0, 0, into((802, 1)));
data (0, INT, 0, 0, into((803, 1)));
data (0, INT, 0, 0, into((901, 2)));
data (1, INT, 0, 0, into((902, 2)));
data (2, INT, 0, 0, into((903, 2)));
data (2, INT, -1, -1, into((1007, 2)));
data (2, INT, -1, -1, into((2007, 2)));
data (-1, INT, 0, 0, into((2299, 3)));
data (-1, INT, 0, 0, into((2299, 2)));
data (2, INT, -1, -1, into((2507, 2)));
data (0, INT, -1, -1, into((3002, 2)));
data (0, INT, -1, -1, into((3101, 2)));
data (0, INT, -1, -1, into((3201, 2)));
data (0, INT, 0, -1, into((3202, 3)));
data (0, INT, 0, -1, into((3202, 2)));
data (2, INT, -1, -1, into((3205, 2)));
data (-1, INT, -1, -1, into((3209, 3)));
data (0, INT, -1, -1, into((3209, 2)));
data (2, INT, -1, -1, into((4007, 2)));
data (2, INT, -1, -1, into((4507, 2)));
data (2, INT, -1, -1, into((5507, 2)));
data (9, INT, -1, -1, into((6001, 1)));
data (2, INT, -1, -1, into((7007, 2)));
data (0, INT, -1, -1, into((7779, 1)));
data (2, INT, -1, -1, into((8007, 2)));
data (2, INT, -1, -1, into((8507, 2)));
data (0, INT, -1, -1, into((8888, 1)));
data (2, INT, -1, -1, into((9007, 2)));
data (2, INT, -1, -1, into((18007, 2)));
data (2, INT, -1, -1, into((18507, 2)));
data (1, INT, -1, -1, into((320203, 1)));
data (1, INT, -1, -1, into((320204, 1)));
data (1, INT, -1, -1, into((320210, 2)));
data (1, INT, -1, -1, into((320211, 2)));
data (2, INT, -1, -1, into((320214, 1)));
data (2, INT, -1, -1, into((320215, 1)));
data (2, INT, -1, -1, into((320221, 2)));
data (2, INT, -1, -1, into((320222, 2)));

```

```

partition(0,alist(800,890,7778,7779,6,7,26,27,99,299,301,401,601,701,3001,3002,3101,3102,320201));
partition(0,alist(303,403,603,703,320214,320215,320216,320217,320218,320219,320220,320221,3202
22,320223,320224,803,903,100,1002,1012));
partition(0,alist(2010,2508,3205,4004,4502,4512,5510,7006,8003,8501,8511,9006,18004,18502,18512
,114,1006,2004,2502,2512));
partition(0,alist(3210,4008,4506,5504,6002,7010,8007,8505,8889,9010,18008,18506,911,1010,2008,2
506,3203,4002,4012,4510));
partition(0,alist(5508,7004,8001,8011,8509,9004,18002,18012,18510));
partition(1,alist(600,690,300,390,302,402,602,702,320203,320204,320205,320206,320207,320208,320
209,320210,320211,320212,320213));
partition(1,alist(111,1003,2001,2011,2509,3206,4005,4503,5501,5511,7007,8004,8502,8512,9007,180
05,18503,320202,112,1004));
partition(1,alist(2002,2012,2510,3207,4006,4504,5502,5512,7008,8005,8503,8887,9008,18006,18504,
113,1005,2003,2501,2511));
partition(1,alist(3209,4007,4505,5503,6001,7009,8006,8504,8888,9009,18007,18505,115,1007,2005,2
503,2990,3211,4009,4507));
partition(1,alist(5505,7001,7011,8008,8506,9001,9011,18009,18507,116,1008,2006,2504,3201,4010,4
508,5506,7002,7012));
partition(1,alist(8009,8507,9002,9012,18010,18508));
partition(7,alist(802,902,2299,3301,3302,801,901,298,1009,2007,2505,3202,4001,4011,4509,5507,700
3,7777,8010));
partition(7,alist(8508,9003,18001,18011,18509,1001,1011,2009,2507,3204,4003,4501,4511,5509,7005
,8002,8012,8510,9005,18003));
partition(7,alist(18501,18511));

allocation( 0, partlist(7) );
allocation( 1, partlist(1) );
allocation( 2, partlist(0) );

```

APPENDIX E: Experiment Result In Graph Format

	3 x 3	5 x 5	7 x 7	10 x 10	15 x 15
3	20.2	71.9	164.9	1544.2	4912.1
5	17.3	64.2	90.4	784.8	2651
7	15	167.7	98.9	299.8	1126.3

	3 x 3	5 x 5	7 x 7	10 x 10	15 x 15
3	50.2	160.1	399.9	1153.8	4021.6
5	17.3	78.2	208.3	564.6	1835.9
7	21	80.8	141.1	229.4	1015.3

	3 x 3	5 x 5	7 x 7	10 x 10	15 x 15
3	53.5	155.1	487.1	1297.1	5184.0
5	33.8	86.7	207.4	639.8	2351.7
7	27.3	102.2	258.8	429.3	1730.6

	3 x 3	5 x 5	7 x 7	10 x 10	15 x 15
3	97.4	273.6	240.4	4773.1	12865.6
5	105.0	404.9	928.6	2038.4	7235.1
7	108.4	149.7	555.0	1631.0	5010.0

	3 x 3	5 x 5	7 x 7	10 x 10	15 x 15
3	91.1	328.8	1188.2	5847.0	18883.2
5	62.0	258.7	641.1	2904.2	10514.0
7	57.8	174.6	461.0	2202.2	7523.8

	3 x 3	5 x 5	7 x 7	10 x 10	15 x 15
3	204.2	653.4	1682.4	6087.6	22005.3
5	111.8	366.0	759.6	3295.1	12529.6
7	104.8	258.6	874.8	2807.3	9397.0

	3 x 3	5 x 5	7 x 7	10 x 10	15 x 15
3	135.3	476.6	1863.8	8080.2	26631.1
5	73.9	240.6	862.5	3868.7	14257.5
7	63.0	202.8	731.6	2719.9	9732.5