

# UC Irvine

## ICS Technical Reports

### Title

Exploiting iteration-level parallelism in declarative programs

### Permalink

<https://escholarship.org/uc/item/0j11x8nt>

### Author

Roy, John M.A.

### Publication Date

1991

Peer reviewed

Department of Information and Computer Science

University of California at Irvine

Irvine, CA 92717

Z  
699  
C3  
no. 91-24

## Exploiting Iteration-Level Parallelism in Declarative Programs

John M.A. Roy

March 1991

Notice: This Material  
may be protected  
by Copyright Law  
(Title 17 U.S.C.)

Technical Report #91-24

### *Abstract*

In order to achieve viable parallel processing three basic criteria must be met: (1) the system must provide a programming environment which hides the details of parallel processing from the programmer; (2) the system must execute efficiently on the given hardware; and (3) the system must be economically attractive.

The first criterion can be met by providing the programmer with an *implicit* rather than *explicit* programming paradigm. In this way all of the synchronization and distribution are handled automatically. To meet the second criterion, the system must perform synchronization and distribution in such a way that the available computing resources are used to their utmost. And to meet the third criterion, the system must *not* require esoteric or expensive hardware to achieve efficient utilization.

This dissertation reports on the Process-Oriented Dataflow System (PODS), which meets all of the above criteria. PODS uses a hybrid von Neumann-Dataflow model of computation supported by an automatic partitioning and distribution scheme. The new partitioning and distribution algorithm is presented along with the underlying principles. Four new mechanisms for distribution are presented: (1) a distributed array allocation operator for data distribution; (2) a distributed L operator for code distribution; (3) a range filter for restriction index ranges for different PEs; and (4) a specialized apply operator for functional parallelism.

Simulations show that PODS balances communication overhead with distributed processing to achieve efficient parallel execution on distributed memory multiprocessors. This is partially due to a new software array caching scheme, called *remote caching*, which greatly reduces the amount of remote memory reads. PODS is designed to use off-the-shelf components, with no specialized hardware. In this way a real PODS machine can be built quickly and cost effectively. The system is currently being retargeted to the Intel iPSC/2 so that it can be run on commercially available equipment.

**Keywords:** single assignment, dataflow, multiprocessor, declarative programming, matrix multiply, SIMPLE

UNIVERSITY OF CALIFORNIA

IRVINE

Exploiting Iteration-Level Parallelism

in Declarative Programs

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

John Marc Andre Roy

Dissertation Committee:

Professor Lubomir Bic, Chair

Professor Nikil Dutt

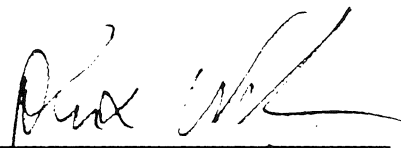

Professor Alexandru Nicolau

1991

© by John Marc Andre Roy

All rights reserved.

The dissertation of John Marc Andre Roy is approved,  
and is acceptable in quality and form  
for publication on microfilm:

  
\_\_\_\_\_  
Neil Dutt  
\_\_\_\_\_  
  
\_\_\_\_\_  
Committee Chair

University of California, Irvine

1991

iii

## DEDICATION

To  
my wonderful wife  
for all of her love, support, and understanding.

I love you Charlene.

## TABLE OF CONTENTS

LIST OF FIGURES .....	viii
LIST OF TABLES .....	xi
ACKNOWLEDGEMENTS .....	xii
CURRICULUM VITAE .....	xiii
PUBLICATIONS .....	xiii
ABSTRACT OF THE DISSERTATION.....	xiv
Background.....	1
1.1. Basic Issues in Parallel Processing.....	5
1.1.1. Parallel Programming.....	5
1.1.2. Distributed Memory MIMD .....	6
1.2. Previous Research.....	6
1.2.1. Single Assignment Principle .....	6
1.2.2. ID Nouveau Dataflow Language .....	8
Single Assignment Approach .....	9
Iteration.....	10
I-Structures.....	11
Discussion.....	13
1.2.3. Hybrid Dataflow.....	13
1.3. Overview of PODS Execution Model.....	14
1.3.1. Subcompact Processes (SP) .....	15
1.3.2. State Transitions.....	17
1.3.3. Distributed Memory Approach.....	19
1.3.4. Discussion.....	20
1.4. Contributions of this Research.....	21
1.4.1. Execution Model Extensions.....	21
1.4.2. Partitioning and Distribution Model.....	21
1.4.3. Remote Array Caching.....	22
1.4.4. Logical Architecture.....	22
1.4.5. Simulations.....	23
PODS Partitioning and Distribution Model.....	24
2.1. Overview .....	25
2.2. Underlying Principles.....	27
2.2.1. Basic Principles.....	28
2.2.2. PODS Specific Principles .....	30
Grouping Principle.....	30
Virtual Sources Principle .....	31
Collector Writes Principle .....	32
2.3. PODS Instructions and Processes .....	33
2.3.1. Activity Names .....	34
2.3.2. PODS Instruction Format .....	36

2.3.3. PODS Dataflow Operator Implementation.....	38
Arithmetic and Logical Operators.....	40
switch and forkjump .....	41
d and d_inverse.....	43
l and l_inverse .....	45
a and a_inverse .....	47
2.4. Array Partitioning and Distribution.....	48
2.5. Distributing Processes .....	56
2.5.1. Data-Distributed Execution Principle.....	57
2.5.2. Range Filters.....	61
Objective and Usage .....	61
Boundary Table.....	64
Master Array .....	65
Algorithm.....	65
2.5.3. LCD Effects.....	67
2.5.4. Remote Array Accesses.....	71
Remote Reads.....	71
Remote Writes .....	73
2.5.5. For-Loop Distribution Algorithm.....	74
2.5.6. Examples .....	76
LCD Examples.....	76
Matrix Multiply .....	84
2.6. Functional Distribution.....	89
2.7. Deadlock Handling.....	90
PODS Logical Implementation.....	95
3.1. System Overview .....	95
3.2. Logical PE Architecture .....	97
3.2.1. Execution Unit.....	99
3.2.2. Routing Unit .....	100
• 3.2.3. Array Manager.....	102
3.2.4. Memory Manager.....	104
3.2.5. Matching Unit.....	104
3.3. Remote Array Caching.....	104
3.4. Software Support.....	108
3.4.1. ID World and GITA Compiler.....	109
3.4.2. Translator.....	109
3.4.3. Partitioner .....	111
3.4.4. Simulator.....	113
PODS Simulations .....	114
4.1. Overview .....	114
4.1.1. Simulator Approach.....	114
4.1.2. Timing Assumptions .....	116
Execution Unit.....	116
Array Manager.....	117
Routing Unit .....	118
Memory Manager.....	119
Matching Store.....	119
Network.....	119
4.2. Measures of Effectiveness (MOEs).....	119



4.3. Example Programs .....	121
4.3.1. Matrix Multiply .....	121
Discussion.....	121
Results.....	122
4.3.2. SIMPLE .....	128
Discussion.....	129
Results.....	134
4.4. Summary .....	142
Conclusions.....	145
5.1. Related Work.....	145
5.1.1. Iannucci's Hybrid Architecture .....	145
5.1.2. Gao's Hybrid Machine .....	146
5.1.3. Alfalfa.....	147
5.1.4. Decoupled Multilevel Dataflow Model.....	147
5.1.5. Dynamic Structured Dataflow.....	148
5.1.6. Pingali and Rogers' Compiler .....	148
5.2. Advantages and Disadvantages of Single Assignment.....	149
5.3. Summary .....	150
5.4. Future Research.....	153
5.4.1. HyperPODS.....	153
5.4.2. PODS Compiler .....	154
References.....	156
Appendix A: Range Filter Algorithms .....	165

## LIST OF FIGURES

Figure 1.1.	Lines of Research.....	2
Figure 1.2.	ID Nouveau Quicksort Code.....	9
Figure 1.3.	ID Nouveau Iteration Example.....	11
Figure 1.4.	ID Nouveau I-Structure Example.....	11
Figure 1.5.	Subcompact Process Example Code.....	15
Figure 1.6.	PODS Subcompact Processes Example.....	17
Figure 1.7.	Process State Transition Diagram.....	18
Figure 1.8.	PODS Memory Accessing Scheme.....	20
Figure 2.1.	Simple Array Assignment.....	28
Figure 2.2.	Equal Distribution Principle.....	29
Figure 2.3.	Grouping Principle.....	30
Figure 2.4.	Virtual Sources Principle.....	31
Figure 2.5.	Collector Writes Principle.....	32
Figure 2.6.	Basic Dataflow Operator.....	34
Figure 2.7.	Activity Name Components.....	35
Figure 2.8.	SP Components.....	38
Figure 2.9.	ID vs PODS Statement "Addressing".....	40
Figure 2.10.	PODS SWITCH and FORKJUMP Instruction Examples.....	42
Figure 2.11.	PODS Branch.....	43
Figure 2.12.	PODS Code Fragment for a Loop.....	45
Figure 2.13.	Example L Operators.....	46
Figure 2.14.	Example Apply and Inv_Apply Operators.....	48
Figure 2.15.	Matrix Multiply ID Nouveau Source Code.....	50
Figure 2.16.	PODS Partitioning of A 2-D Array.....	52
Figure 2.17.	2-D Array Read Pseudo-Code.....	55

Figure 2.18. Example 2-D Array Remote Read. ....	55
Figure 2.19. Example 2-D Array Local Read. ....	56
Figure 2.20. Partitioning a 2D Iteration Space.....	58
Figure 2.21. Partitioning a 3D Iteration Space.....	59
Figure 2.22. Simple 2-D Array Fill. ....	62
Figure 2.23. 2-D Array Fill with Range Filter.....	63
Figure 2.24. Algorithm for Second Level, Descending Range Filter for A[ci*i+ki,cj*j+kj]. ....	66
Figure 2.25. Non-rectangular Array Partitioning Example.....	67
Figure 2.26. Effects of Communication Speed on Overlapping Iterations. ....	69
Figure 2.27. Remote Read Code Example. ....	72
Figure 2.28. Remote Write Code Example.....	73
Figure 2.29. Impossible Collector Writes.....	74
Figure 2.30. Simple Array Filling Example Code.....	76
Figure 2.31. Simple Row-Major Array Partitioning. ....	77
Figure 2.32. LCD Execution Wavefronts. ....	84
Figure 2.33. Example Execution Trace for Matrix Multiply on 4 PEs.....	88
Figure 2.34. ID Nouveau Deadlock Code Example.....	92
Figure 3.1. Logical Units of a PODS PE. ....	98
Figure 3.2. Routing Table.....	100
Figure 3.3. Routing Unit Block Diagram.....	101
Figure 3.4. Effects of Cache Size on Percentage of Remote Reads. ....	107
Figure 3.5. Remote Reads for the Livermore Loops using Remote Caching.....	108
Figure 3.6. PODS Programming System.....	109
Figure 3.7. PODS Partitioner Block Diagram.....	111
Figure 4.1. 2-D Array Read Pseudo-Code. ....	117
Figure 4.2. Matrix Multiply ID Nouveau Source Code.....	122

Figure 4.3.	Utilization for Each Functional Unit (16 x 16 MM).....	123
Figure 4.4.	Average Execution Unit Utilization for Matrix Multiply.....	124
Figure 4.5.	Utilization for each Execution Unit (16 x 16 MM on 8 PEs).....	125
Figure 4.6.	Utilization for each Execution Unit (16 x 16 MM on 16 PEs). ....	126
Figure 4.7.	Speed-Up of Matrix Multiply. ....	128
Figure 4.8.	Sweep For-Loops in Conduction Code.....	130
Figure 4.9.	Original Conduction Code with Multiple LCDs.....	131
Figure 4.10.	Scalar Expanded Conduction Code Fragment.....	132
Figure 4.11.	Utilization for Each Functional Unit (16 x 16 SIMPLE).....	135
Figure 4.12.	Execution Unit Utilization for SIMPLE.....	136
Figure 4.13.	Execution Unit Utilization (16 x 16 SIMPLE on 32 PEs). ....	137
Figure 4.14.	Execution Unit Utilization (32 x 32 SIMPLE on 32 PEs). ....	138
Figure 4.15.	Execution Unit Utilization (64 x 64 SIMPLE on 32 PEs). ....	139
Figure 4.16.	Speed-Up of SIMPLE. ....	141
Figure A.1.	Base Range Filter Algorithm for Outermost Level Distribution. ....	165
Figure A.2.	Base Range Filter Algorithm for Second Outermost Level Distribution. ..	166
Figure A.3.	Base Range Filter Algorithm for Third Outermost Level Distribution.....	167
Figure A.4.	Range Filter Algorithm for Step size -1.....	168
Figure A.5.	Second Level Distribution Range Filter for $A[ci*i+ki,cj*j+kj]$ . ....	169
Figure A.6.	Range Filter for Third Level Distribution with Step size C.....	170

## LIST OF TABLES

Table 2.1.	PODS Array Header Information.....	53
Table 2.2.	2-D Array Example Header.....	54
Table 2.3.	Example Boundary Table for a Given PE. ....	64
Table 2.4.	Effects of Outer Loop Distribution with No LCDs.....	78
Table 2.5.	Effects of Inner Loop Distribution with No LCDs. ....	79
Table 2.6.	Effects of Inner Loop Distribution with LCDs.....	81
Table 2.7.	Effects of No Distribution due to LCDs. ....	83
Table 4.1.	Measured Times of Operations on iPSC/2.....	116
Table 4.2.	Percent Overhead Instructions for Matrix Multiply.....	126
Table 4.3.	SP Statistics for Conduction. ....	134
Table 4.4.	Percent Overhead Instructions for SIMPLE.....	139

## ACKNOWLEDGEMENTS

I would like to express my thanks to my committee chair, Professor Lubomir Bic, for his continuing insights through out the years, and for this diligence in helping me prepare this dissertation.

I would like to particularly thank my research associate, Mark Nagel, for his ideas and programming expertise. Without Mark this work would still be in the programming stages. Good luck Mark.

Special thanks to my parents for their support and encouragement through out my entire life.

Financial support has been provided by a number of sources through out the years: Hughes Aircraft Company, Fail-Safe Technology, JMAR Research Group, and from my wonderful wife, Charlene. I would also like to thank the National Science Foundation for their support of the PODS research through NSF grant #CCR-8709817.

## CURRICULUM VITAE

John M.A. Roy

- 1982            B.S. in Electrical Engineering, University of California, San Diego.
- 1982-1987      Systems Engineer, Hughes Aircraft Company, Fullerton, California.
- 1984            M.S. in Electrical Engineering, University of Southern California.
- 1987-1988      Senior Member of Technical Staff, Fail-Safe Technology, Los Angeles, California.
- 1988-1989      Systems Consultant, JMAR Research Group, Irvine, California.
- 1989            M.S. in Information and Computer Science, University of California, Irvine.
- 1989-1990      Vice-President, Engineering and Operations, Trintech USA, Irvine, California.
- 1991-Present   Vice-President, Engineering, National Paging, Santa Ana, California.
- 1991            Ph.D. in Information and Computer Science, University of California, Irvine.  
Dissertation: "Exploiting Iteration-Level Parallelism in Declarative Programs."  
Professor Lubomir Bic, Chair.

## PUBLICATIONS

- L. Bic, M. D. Nagel, J. M. A. Roy. Automatic Data/Program Partitioning Using the Single Assignment Principle. *Supercomputing '89* (1989), pp. 551-556.
- L. Bic, M. D. Nagel, J. M. A. Roy. Executing Matrix Multiply on a Process Oriented Dataflow Machine. *Technical Report 90-08* (April 1990), Department of ICS, University of California, Irvine.
- L. Bic, M. D. Nagel, J. M. A. Roy. On Array Partitioning in PODS. In *Advanced Topics in Data-Flow Computing*. J. L. Gaudiot, L. Bic, Eds. (Prentice Hall, Englewood Cliffs, New Jersey, 1990), pp. 305-325.
- J. M. A. Roy, M. D. Nagel, L. Bic. Partitioning Declarative Programs into Communicating Processes. *Supercomputing '90* (1990), pp. 846-855.

# ABSTRACT OF THE DISSERTATION

Exploiting Iteration-Level Parallelism

in Declarative Programs

by

John Marc Andre Roy

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1991

Professor Lubomir Bic, Chair

In order to achieve viable parallel processing three basic criteria must be met: (1) the system must provide a programming environment which hides the details of parallel processing from the programmer; (2) the system must execute efficiently on the given hardware; and (3) the system must be economically attractive.

The first criterion can be met by providing the programmer with an *implicit* rather than *explicit* programming paradigm. In this way all of the synchronization and distribution are handled automatically. To meet the second criterion, the system must perform synchronization and distribution in such a way that the available computing resources are used to their utmost. And to meet the third criterion, the system must *not* require esoteric or expensive hardware to achieve efficient utilization.

This dissertation reports on the Process-Oriented Dataflow System (PODS), which meets all of the above criteria. PODS uses a hybrid von Neumann-Dataflow model of computation supported by an automatic partitioning and distribution scheme. The new partitioning and distribution algorithm is presented along with the underlying principles. Four new mechanisms for distribution are presented: (1) a distributed array allocation operator for data distribution; (2) a distributed L operator for code distribution; (3) a range filter for restriction index ranges for different PEs; and (4) a specialized apply operator for functional parallelism.

Simulations show that PODS balances communication overhead with distributed processing to achieve efficient parallel execution on distributed memory multiprocessors. This is partially due to a new software array caching scheme, called *remote caching*, which greatly reduces the amount of remote memory reads. PODS is designed to use off-the-shelf components, with no specialized hardware. In this way a real PODS machine can be built quickly and cost effectively. The system is currently being retargeted to the Intel iPSC/2 so that it can be run on commercially available equipment.



## CHAPTER 1

### Background

Scientific programmers are the primary users of parallel systems today. The current parallel programming systems do not meet the needs of this important group. Recent user surveys show that only one user program in twenty executed on the Cornell supercomputer is parallel, [P&B90]. These surveys also indicate that many more scientists would program for parallel systems if they were not so difficult to program. Hand-coded parallelism is too difficult and time consuming, while parallelizing compilers do not achieve significant speed-up.

What is needed is a system which provides scientific programmers with a means to express their problem clearly *and* to have it execute efficiently in parallel automatically. Add to this the desire to run on standard MIMD architectures (e.g., iPSC/2) and the problem becomes very difficult. MIMD architectures require that programs be decomposed into independent processes, running asynchronously on the different processor nodes and communicating with one another through message passing or through shared memory. The current state of the art in programming such machines efficiently is to let the programmer explicitly partition the program into processes and insert the necessary synchronization and communication primitives. This is very time-consuming and error-prone. Automatic generation of parallel programs from conventional languages has not, as yet, achieved sufficient speed-up to warrant wide-spread usage.

To achieve these goals many declarative programming languages [A&E88] have been designed. Declarative programming languages are much better suited for program decomposition than procedural languages such as C or FORTRAN. Declarative languages allow the programmer to describe the problem using high-level constructs, yet their

semantics eliminate uncontrolled side-effects though functional expressions and single assignment restrictions.

Declarative languages have been developed primarily in the context of approach of radically different computer architectures, in particular, dataflow architectures, where parallelism is to be exploited at the *instruction level*. For conventional loosely-coupled MIMD systems, this level of parallelism is too low; the communications costs are too high. By moving to *iteration level* parallelism this problem can be overcome [Burns, 88]. Iteration level parallelism is achieved when different iterations (or groups of iterations) from the same loop are run on different PEs.

Process-Oriented Dataflow Systems (PODS) make use of iteration level parallelism and declarative programming on distributed memory MIMD machines. The PODS line of research is shown in Figure 1.1 as the bold arrow.

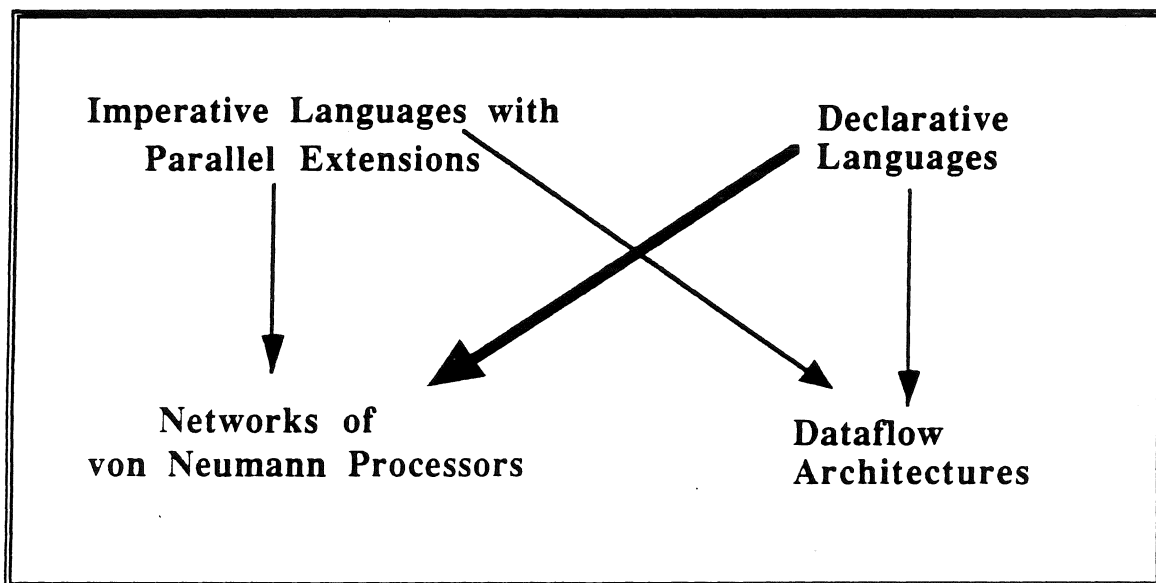


FIGURE 1.1. LINES OF RESEARCH.

Figure 1.1 shows the different lines of research in parallel processing. The first line involves running imperative languages with parallel extensions (e.g., FORTRAN\*

[K&B88]) on Networks of von Neumann Processors (e.g., iPSC/2 [Intel, 89]). This approach is the least revolutionary and has had some commercial success. The second line of research is to take imperative languages and execute them on dataflow architectures (e.g., Monsoon [Pap88]). This direction has not seen much research, only the ASTOR [U&Z89, Z&U87] project in Germany has looked into this. The next line of research is to take a declarative language (e.g., ID [ANP87b], SISAL[A&O85, MSS85]) and run them on von Neumann networks. This is where PODS is, and there are a number of others, notably Pingali and Rogers at Cornell [P&R90]. The final approach is the most revolutionary, running declarative languages on dataflow architectures involves both new hardware and software. P-RISC [N&A89] and the Monsoon project are both taking this approach.

In [Bic87], the basic principles of PODS were presented. The algorithms for subdividing dataflow graphs into communicating processes, however, were too simplistic, concentrating on only functional parallelism. In scientific code, most parallelism comes from loops iterating over large data structures (i.e., data parallelism). This issue has been addressed in subsequent studies [BNR89a, Bic90, BNR90a, BNR90b] which show that, for languages based on the single assignment principles (declarative languages), a simple automatic partitioning of arrays exposes significant parallelism that can be exploited at run-time.

In PODS, the programming language ID Nouveau [Nik88] is used because it is one of the most developed and supported dataflow languages to date and has single-assignment. Single-assignment is central to PODS. Given an ID Nouveau program, a compiler would produce a dataflow graph, where nodes represent individual instructions and arcs show all data dependencies. This graph is then used to generate light-weight processes, referred to as "subcompact processes" (SPs). This is accomplished by partitioning the dataflow graph

into subgraphs, each of which is executed as a sequential process on a given processing element (PE).

This dissertation describes the partitioning method used to form the SPs, the SP distribution criteria, the logical implementation of PODS, the remote caching scheme used, and the results of experiments with an event-driven, instruction-level, simulator. The dissertation is organized as follows:

- Chapter 1 Background — an overview of the pertinent basic concepts. This includes discussions on parallel programming, distributed memory MIMD architectures, the ID dataflow language, and the previous work on PODS. Knowledgeable readers may skip any or all of this chapter.
- Chapter 2 PODS Partitioning and Distribution Model — a detailed discussion of the inner workings of the partitioning of programs into SPs and their distribution.
- Chapter 3 PODS Logical Implementation — a discussion of the tasks necessary to make the PODS system work. The array caching scheme is presented along with a discussion of the special PODS instructions. This is followed up with a description of the PE architecture and the necessary support software.
- Chapter 4 Simulations — a presentation of the experiments using Matrix Multiply and SIMPLE. The simulation approach is discussed and the results are examined.
- Chapter 5 Conclusions — a discussion of the findings about PODS. Future research and related work are also discussed.

## 1.1. Basic Issues in Parallel Processing

### 1.1.1. Parallel Programming

Parallel processing has been touted as the wave of the future for a number of years, yet its use is not yet common. This is because parallel processing requires parallel programming. For the average, highly-intelligent, but inexperienced, scientific programmer, the task of programming a parallel system can be daunting.

In [K&B88], Karp and Babb discuss the complications which arise when trying to program in any one of twelve parallel FORTRAN dialects. They state that even trivial examples frequently become a challenge. Programming parallel systems present complications not found in sequential programming. Often parallel programming environments force the programmer to explicitly partition function and data according to the constraints of the architecture. Thus requiring the scientific programmer to become knowledgeable about the particular computer architecture being used.

In debugging parallel programs, synchronization and timing are often the problem [K&T88]. By requiring the programmer to explicitly state the communication and synchronization points in a program, the system is opening itself to subtle timing errors. The difficult thing about timing errors is their unpredictability. Often a timing error may disappear based upon some seemingly unrelated fact (e.g. the load on the I/O network to the host), and reappear at a later date.

In their 1989 report on supercomputers, the IEEE Scientific Supercomputer Subcommittee sited the lack of software as *the* major problem [IEEE89] in supercomputing today.

All of the above problems are addressed in PODS, the parallelization is implicit not explicit, the synchronization is handled automatically, and, due to the dataflow nature of PODS, the special timing problems of parallel programs are non-existent.

### 1.1.2. Distributed Memory MIMD

Distributed-memory MIMD computers can be made massively parallel by adding PE's in a modular fashion. This modularity allows dramatic increases the theoretical maximum speed. As an example, the latest supercomputer from Intel, called the Delta System, will incorporate 528 i860 microprocessors and have a theoretical peak processing rate of 32 billion floating-point operations per second [Ins91]. The problem is exploiting all of this parallelism.

## 1.2. Previous Research

### 1.2.1. Single Assignment Principle

The Single Assignment Principle simply states that *no variable will be assigned a value more than once*. This would seem like a very limiting restriction, i.e. one may not even write  $x = x + 1$ . However, researchers have found that a number of benefits can be derived from using single assignment in combination with a *functional* language. A functional language is one which is based on function application and is therefore free of side effects. Some of the programming benefits of single assignment functional languages [Veg88] are:

- Programs can be written at a higher level. Time can be spent concentrating on the algorithm rather than the program details.
- More algorithmic work can be expressed per line of code. This is important because evidence suggests that the number of lines

of *correct* code per day is roughly a constant for a given programmer, independent of the language used.

- Functional languages are free of side effects. This greatly reduces unexpected modification of variables in other routines.
- Programs are easier to verify because proofs can be based upon the concept of a function rather than some complex von Neumann model.
- Functional programs can contain a great deal of *implicit* rather than explicit parallelism. This is crucial to the PODS concept.

As is described in the next section, ID Nouveau is the single assignment functional language which PODS uses. Some of the basic ID Nouveau principles are discussed in the next section.

PODS specifically uses the following abilities of single assignment functional languages:

- **Implicit Parallelism** — the ability of a programmer to code a parallel program without explicitly specifying the parallelism.
- **Parallel Program Synchronization** — single assignment automatically synchronizes the data reads and writes of a program, thus preventing innocuous timing bugs.
- **Automatic Cache Coherency** — single assignment allows remote caching to avoid the cache coherency problem. Thus an efficient implementation can be designed, see Section 3.3, Remote Array Caching.

### 1.2.2. ID Nouveau Dataflow Language

ID (Irvine Dataflow) was born at the University of California, Irvine in a 1978 technical report [AGP78]. This report laid the foundations for all further versions of ID. ID has gone through many changes but still retains the basic dataflow ideas, the single assignment concept, and the compiler approach outlined by Arvind. The latest version is being worked at MIT and is called ID Nouveau. The ID Nouveau language environment, called ID-World, is a complete parallel language simulator. There are over twenty sites using ID-World and many more will be appearing as ID-World expands outside of the LISP machine world and onto UNIX workstations.

The syntax of ID Nouveau and its functional nature lead to clean algorithms, which in turn is easier to read and understand. Consider the quicksort code in Figure 1.2 below. Notice that ID Nouveau allows standard list operations which are easy to understand.



```

def Quicksort A =
{
  Split L =
  { startvalue = hd L;
    for v in L;
      if (v < startvalue) then cons Llist v;
      if (v == startvalue) then center = v;
      if (v > startvalue) then cons Rlist v;
    end for
  in
    Llist, center, Rlist
  } ; % Split

in
  % Quicksort routine body
  if (length A < 2)
  then
    A
  else
    {
      L, Middle, R = Split(Data)
    in
      cons Quicksort(L) Middle Quicksort(R);
    }
  }; % Quicksort

```

FIGURE 1.2. ID NOUVEAU QUICKSORT CODE.

The split function is repeatedly called until each sublist has only one element in it. Then the sublists are concatenated in order. This is a very clean and clear program for quicksort.

### Single Assignment Approach

The central issue for PODS in ID is its single assignment nature. All dataflow languages begin with single assignment, yet many diverge as further developments are made. ID has tried to stay true to its original single assignment concept:

...a dataflow operation is purely functional and produces no side-effects as a result of its execution.

This is the essence of single assignment; however, the issue of array handling is in conflict. To provide arrays this constraint has to be relaxed. ID Nouveau arrays (called *I-structures*) produce a side-effect, but are not allowed to be updated to ensure determinacy. Yet, with no update how useful is an array? The answer to this question is still being researched. Arvind, Nikhil, and Pingali feel that they are very useful and that this is the best approach [ANP87a]. They believe that an update operator is inadequate and *over-specifies* algorithms in such a way that unnecessary copying of intermediate data structures and substantial unnecessary sequentialization occur. They also feel that automatic detection is not tractable in general, contrary to other researchers' beliefs [A&K87, A&N87, P&W86].

### Iteration

Iteration is a major source of parallelism. How a language handles iteration is going to affect the ability of the programmer and compiler to exploit the parallelism in the loops. In ID Nouveau the evaluation of loops and conditionals is not eager. This is the same as VAL/SISAL for expressions [A&O85]. This forces the predicate to be fired before either of the two branches of a conditional are fired.

As an example of iteration, consider the program below, taken from [Tra86]. It fills each element of its argument array with a value and returns the sum of all the elements. The loop body contains ordinary bindings (like the variable *val*), I-structure stores (for  $A[i]$ ), and some *newified* variable bindings. These newified variable bindings describe how to compute the values the newified variables take on the next iteration of the loop, e.g. the variable *i* is incremented each time through the loop. These newified variables must have an initial binding outside the loop, otherwise it would have no value for the first iteration. Newified variables do not make sense outside of loops and are not allowed there.

```

{ def fill_it A =
  let
    i = lower_bound A;
    sum = 0;
  in
    while i ≤ upper_bound A;
      val = (upper_bound A - lower_bound A) ^ 2 - i*i;
      A[i] = val;
      new sum = sum + val;
      new i = i + 1;
    return sum
}

```

FIGURE 1.3. ID NOUVEAU ITERATION EXAMPLE.

I-Structures

The basic array structure mechanism in ID Nouveau is the I-structure [ANP87a]. An I-structure is an *incremental structure* which obeys the single assignment rule. An I-structure is available as soon as it is allocated and the array elements are individually accessible. Consider the wavefront example below:

```

{ A = matrix ((1, m) , (1, n)) ;
  {for i from 1 to m do
    A[i,1] = 1}
  {for j from 2 to n do
    A[1,j] = 1}
  {for i from 2 to m do
    {for j from 2 to n do
      A[i, j] = A[i-1, j] + A[i-1, j-1] + A[i, j-1] }}
  in
  A}

```

FIGURE 1.4. ID NOUVEAU I-STRUCTURE EXAMPLE.

Here a matrix has its upper and left borders filled with 1's, while its interior is filled with the sum of the upper, left, and diagonal elements. The matrix A will be returned as the value of the entire expression as soon as it is allocated. Meanwhile, all the loop bodies are initiated in parallel, but some will be delayed until the loop bodies to the left and top

(cartesian coordinate wise) complete. Thus a "wavefront" of processes fills the matrix in parallel.

To achieve this flexibility I-structures use a *presence bit*. Each cell of an I-structure has a logical bit attached to it to determine if the cell's value is present. If a read occurs before the cell is written, the read is enqueued by the I-structure. When a write occurs, all pending reads are dequeued and processed. If a write occurs to a cell which has already been written, then a run-time error occurs. This is an efficient way to enforce single assignment.

I-structures do have a referential transparency problem. Referential transparency demands that the values returned by two calls to the same constructor function with the same arguments must never be distinguishable. Thus, in a functional language, one can never alter a data structure once it has been created, and consequently one must specify the contents of all elements of the structure at creation time (as in VAL/SISAL [A&O85] and LUCID [W&A85]). Since ID Nouveau includes I-structures, and I-structures do not specify the contents of all elements at creation, ID Nouveau is not a completely functional language. Yet it is still single assignment and declarative.

Referential transparency can be given up but determinacy cannot. If a language possess the Church-Rosser property [Lan65], also called the confluence property, then overall program determinacy is guaranteed even if the machine exhibits non-determinacy in instruction scheduling. The Church-Rosser property requires that the answer computed by an expression be unaffected by the choice of which subexpressions are evaluated first. Since I-structures enqueue all early reads until the cell is written to, and each cell is single assignment, I-structures have the Church-Rosser property. No matter how one interleaves the execution of reads and writes, every fetch to a given I-structures element always returns the same value.

## Discussion

ID Nouveau is highly developed language system with many sites using its development environment (ID-World). The ID Nouveau language reference manual [Nik87a] describes a complete environment with a compiler, a context sensitive editor, and simulators with parallelism detectors.

In [A&E88] a convincing argument is made for single assignment programming of scientific programming. In this technical report the SIMPLE hydrodynamics and heat conduction problem is detailed, and an efficient ID Nouveau program is designed. This design is then contrasted with a parallel version of the program in annotated FORTRAN where each program does the same number of arithmetic, load, and store operations.

### **1.2.3. Hybrid Dataflow**

Since Dennis first described the first dataflow execution model [Den75], many architecture designers have attempted to apply the model to real systems. Dataflow is attractive because all parallelism in a program is exposed for potential concurrent execution. In spite of the elegance of the model, dataflow is not widely used after more than twenty years of research. The focus has instead turned to the evolution of modern systems by extending them with dataflow techniques. The results of research in this area include hybrid systems using large-grain or macro dataflow [Bab84, B&E87, DFL89, Ian88, Kap86, L&G86, S&H87].

Iannucci [Ian88] has reported on a hybrid dataflow / von Neumann architecture. This approach is similar to PODS in its use of ID Nouveau as the input language and split-phased structure access. However the Iannucci approach uses a finer grain scheduling approach, called *scheduling quanta* (SQ). An SQ of two to three instructions is desirable for Iannucci's approach, and each iteration of a loop is a new SQ. In PODS, however, the

natural decomposition of the program is used and SPs are allowed to run-in-place, thus reducing overhead. Another difference is in data structure distribution. There is no mechanism for spreading iterations of a single loop across processors in Iannucci's approach. Combining data structure distribution with loop distribution is a central goal in PODS. Finally Iannucci's model requires a special purpose architecture capable of fast context switching among very small SQs. PODS tries to generate SPs large enough to produce good computation-communication ratios on available distributed memory multiprocessors. Certainly PODS would benefit from a tailored architecture, but the model itself is not restricted to such.

In [G&H89], Goldberg and Hudak presented Alfalfa, a system similar at a high level to PODS. They have implemented the ALFL functional programming language and run-time system on an Intel iPSC hypercube using what they call *serial combinators*. Serial combinators are similar to PODS SPs in that they are sequential threads that execute on a von Neumann processor. The run-time system handles thread creation and distribution. The main focus of their work is the study of the effects of dynamic scheduling (diffusion scheduling) of parallel threads of execution. They show that diffusion scheduling works well in many cases, however, they have not addressed the problem of distributing large data structures such as arrays. This is illustrated through the relatively poor performance achieved with the Matrix Multiply algorithm.

### 1.3. Overview of PODS Execution Model

The primary objective of PODS is to achieve an efficient execution model for dataflow programs by reducing the overhead associated with scheduling each instruction individually [Bic90]. The greatest deficiency of the pure dataflow model is the excessive communication and token matching overhead associated with passing data from one

operation to another. These operations may lie on the same or different processors, thus potentially forcing token traffic over the processor interconnection network.

Originally it was thought the normal communication overhead could be reduced by grouping the instructions into threads. This was based on the observation that many threads of instructions in the dataflow graph must be executed sequentially due to inherent data dependencies. Grouping instructions in this manner is similar to Babb's Large Grain Data Flow (LGDF) [Bab84]. However, it was found this produced SPs which were too small for the communication to computation ratio of typical distributed-memory machines.

### 1.3.1. Subcompact Processes (SP)

In order to overcome the small SP problem, a different approach was tried and found to be sufficient. This approach uses the code-blocks inherent in the program. Each code-block is a different SP, which will then be distributed by the Partitioner as necessary. This is how PODS exploits the iteration level parallelism in a program.

The code fragment below in Figure 1.5 shows a simple nested loop. For this loop there are three different program scopes which turn into SPs. The first takes care of initial actions, mainly array allocation. The second handles the  $L$  level of the loop, and the third handles the  $K$  level and the actual computations.

```
(initial A := < >; Y := < >; ZX := < >
  for L from 1 to LOOP do
    new A := (initial X := < >
      for K from 1 to 1000 do
        new X[K] := Q + Y[K]*(R* ZX[K+10]+T*ZX[K+11])
      return X)
    return A[1])
```

FIGURE 1.5. SUBCOMPACT PROCESS EXAMPLE CODE.

Figure 1.6 shows the code fragment as a dataflow diagram. The SPs are outlined in bold lines. Notice that the SPs are grouped so that each one will be as independent from the others as possible. This is where the parallelism is. SP1 allocates the arrays and then passes that information on to SP2. There may be multiple versions of SP2 running (if it is distributed), each executing only part of the  $L$ -loop. Each SP2 will then spawn SP3, which will run in-place (SP3 would never be distributed if SP2 were). In Chapter 2 the algorithm for distributing SPs is discussed in detail.



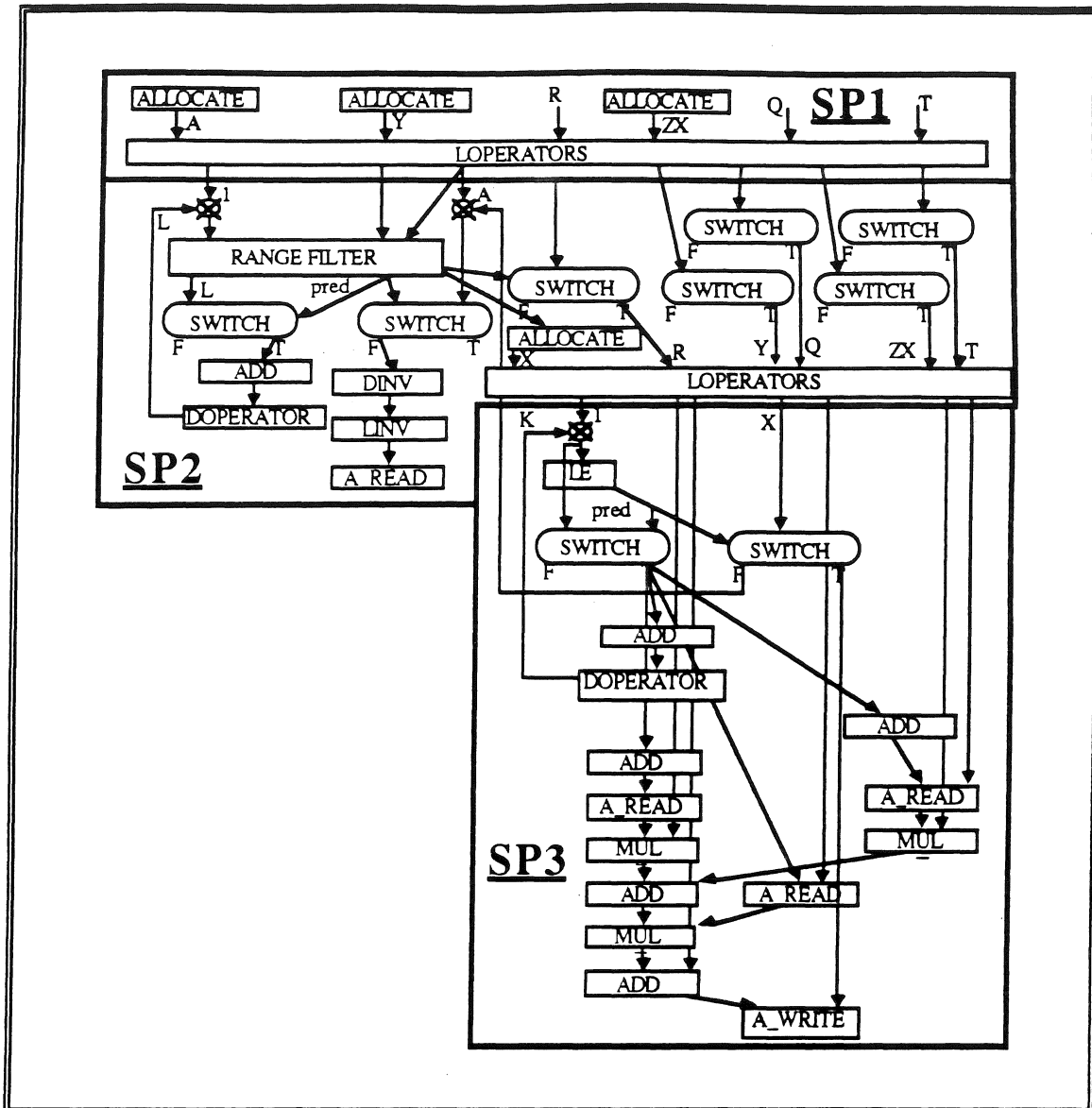


FIGURE 1.6. PODS SUBCOMPACT PROCESSES EXAMPLE.

### 1.3.2. State Transitions

Once the static SPs are formed they will need to be scheduled for execution. Instead of scheduling individual operators of a dataflow graph for execution, the level of granularity is changed to that of an SP. An SP is *passive* as long as its *first* operator is disabled (i.e., it is still missing some operands). A passive SP resides in program memory. When all

operands for the first operator have arrived, the SP becomes *active*. This is accomplished by loading the SP into execution memory and creating a simple process control block (PCB) for it. The PCB contains the following information:

- the starting address of the SP in execution memory
- a program counter pointing to the current instruction
- a status field indicating whether the process is running, ready, or blocked.

The three states are defined as follows. An SP is said to be *running* when a PE is currently fetching and executing instructions from that sequence. An SP is *ready* when its current instruction is enabled (has all its operands), but the PE is not available to execute that SP. Finally, an SP is *blocked* when its current instruction is not enabled.

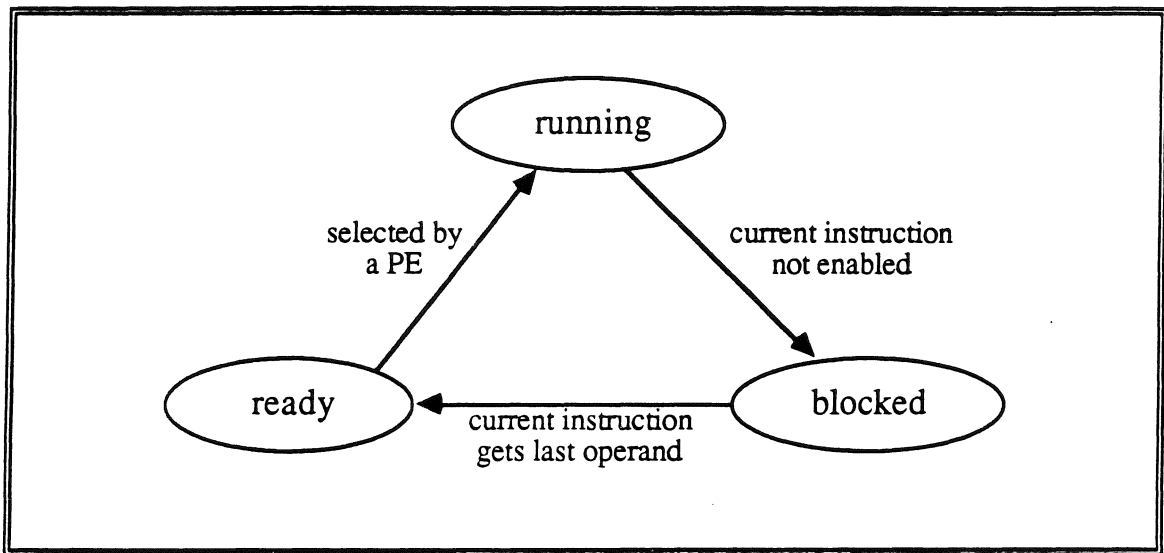


FIGURE 1.7. PROCESS STATE TRANSITION DIAGRAM.

The possible state transitions are illustrated in Figure 1.7. Initially, an SP is loaded into execution memory in the ready state. Whenever the PE becomes free, it begins executing one of the ready SPs in its execution memory; at that time, the status of the selected SP changes from ready to running. The PE continues executing the SP until it reaches the end

of the SP (at which time it is destroyed) or until it encounters an operator that does not yet have all its operands present. In the latter case, the SP is blocked and the PE switches to another ready SP. The blocked SP changes its status to ready as soon as the last operand for the current instruction arrives.

This process-oriented viewpoint permits us to execute a dataflow program as a collection of *communicating* SPs. A given dataflow program is transformed into one or more SPs, which are mapped onto the available PEs. Each SP continues executing as long as it has all the operands necessary to perform its current operation. When an operation produces a result token destined for a subsequent operation within the same SP, it is passed directly to the destination operand slot using a simple memory operation. Only when the token is destined for a different SP must it travel through the dataflow routing network (within the same PE or to another PE) and pass through the matching store. It is important to note that the amount of resources need for a particular SP is known at load time. With this information the amount of parallelism can be reduced if necessary.

### **1.3.3. Distributed Memory Approach**

In PODS, the memory is distributed as shown in Figure 1.8 below. The physical separation between the PEs is recognized and exploited. Remote memory requests are performed in a *split-phase* manner. This allows the CPU to continue processing during the long remote memory latency. Local memory requests are handled instantly and do not cause the CPU to context switch. This is one reason PODS is able to exploit the power of massively parallel distributed-memory machines.

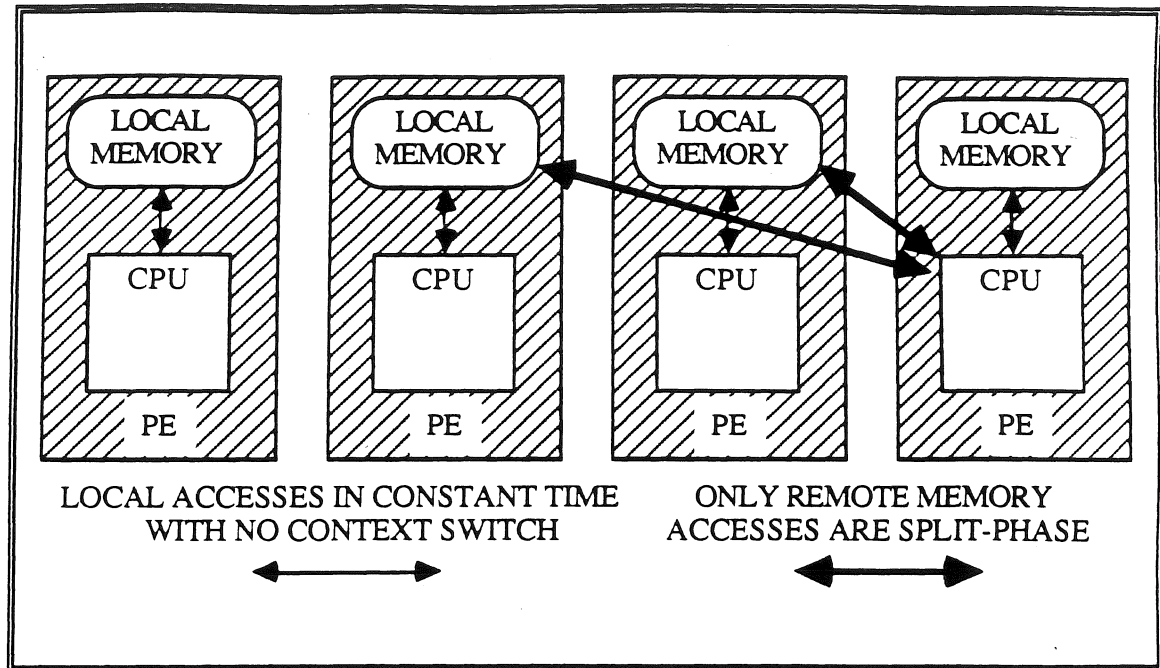


FIGURE 1.8. PODS MEMORY ACCESSING SCHEME.

#### 1.3.4. Discussion

This model of execution has a number of advantages. Since it uses a program counter, loops can be run in place efficiently. If necessary, due to dependencies, PODS can drop into completely sequential execution. When a process block occurs, the execution unit performs a simple context switch (no register storage is necessary) and takes the next ready SP off the ready list. And array accesses are split-phased to allow the long memory latency to be tolerated.

In summary, PODS uses a combination of dataflow and von Neumann models of computation. It uses single assignment to reduce side-effects which aides parallelism. The declarative nature of ID, and its implicit programming of parallelism, allows the programmer to ignore the architecture, which increases programmer productivity. For a more detailed description of the execution model, the reader is referred to [Bic87, Bic90].

## **1.4. Contributions of this Research**

This research has made contributions on many levels. It extends the existing models (the PODS Execution Model and ID Instruction Set). It presents new principles and algorithms (for partitioning and distribution). It exploits the abilities of old concepts in new ways (Remote Array Caching). It explains how all of these can work together in a logical manner (Logical Architecture). And it shows that this approach is efficient and scalable (the simulations).

### **1.4.1. Execution Model Extensions**

The PODS Execution Model was extended to allow iteration level parallelism. The previous model, based on the concept of sequential threads, produced SPs which were too small. The extension to iteration level parallelism allows larger SPs which are more easily distributed.

### **1.4.2. Partitioning and Distribution Model**

The new PODS Partitioning and Distribution Model is based upon two existing and three new principles of parallel execution. The existing principles (the Equal Distribution Principle and the Centralization Principle) are well known and are continually pushing in opposite directions. The new principles (the Grouping Principle, the Virtual Sources Principle, and the Collector Writes Principle) explain ways in which the two existing principles can be managed.

From these five principles, two partitioning and distribution algorithms were derived. The first shows how data should be partitioned and distributed to balance work load and speed up accesses. The second describes how code should be partitioned and distributed to balance parallel execution with communication costs.

Three primary and two secondary mechanisms were devised to make these algorithms work. The first primary mechanism is a distributed array allocate operator which distributes data. The second is a distributed L operator; it spawns processes across the PEs to distribute code. The third is an index range filter for restricting the indices for different PEs. These form the basis for PODS distributed processing. The secondary mechanisms are: an APPLY operator for functional distribution; and remote array caching for efficient array accesses. Together these provide an efficient means of applying the new partitioning and distribution algorithms.

#### **1.4.3. Remote Array Caching**

Remote Array Caching is a new approach similar to the concept of virtual memory and based upon the Virtual Sources Principle. This allows arrays to be accessed as if there were local to every PE. The locality-of-reference of computer programs is heavily exploited in Remote Array Caching.

#### **1.4.4. Logical Architecture**

A description of how all of these new concepts and approaches are implemented are contained in the Logical Architecture. The functional units in a PODS PE are: the Execution Unit, the Matching Store, the Routing Unit, the Array Manager, and the Memory Manager. Each of these is designed to run in parallel with the others.

Extensions to the ID instruction set were necessary to allow PODS to execute on a von Neumann CPU. Some of these extensions involve the addition of a program counter to each instruction's semantics. Others involve extensive modifications of existing instructions (e.g. the L operator), and finally others involved totally new instructions to support the PODS Range Filters (e.g. INTERVAL\_COUNT).

#### **1.4.5. Simulations**

The PODS Translator, Partitioner, and Simulator were designed and written to test PODS concepts. The simulations were necessary to test the logical architecture for correctness and efficiency. These simulations have shown PODS to be an efficient and viable approach.

## CHAPTER 2

### PODS Partitioning and Distribution Model

The performance of PODS comes from its ability to map the inherent granularity of a program onto a given architecture. The inherent granularity of a program comes from its block structure. The larger (smaller) the loops and procedures, the larger (smaller) the granularity. This granularity controls the size of the PODS SPs. The partitioning and distribution model allows the hybrid nature of PODS to be exploited: sequential code is run on an efficient von Neumann processor, and parallel code is distributed such that communication costs are not prohibitively high. This is not to say that *all* programs will run well on PODS, bad code can be written for any computer system. The aim of this model is to handle the large majority of code which will be executed on distributed memory MIMD machines and to flag code which is poorly written.

The key elements of PODS partitioning and distribution are:

1. array partitioning, which uses a simple page grouping scheme to allow equal load across the PEs;
2. array distribution, which follows the partitioning such that each PE produces only those elements for which it is responsible;
3. loop distribution, which considers data dependencies when distributing;
4. functional distribution, which attempts to off-load functions if the calling PE is overloaded.



Chapter 2 is organized as follows: (1) a quick overview of the model; (2) presentation and discussion of the underlying principles; (3) a detailed discussion of PODS instructions and processes; (4) a discussion of array partition and distribution; (5) an in-depth examination of process distribution; (6) a discussion of functional distribution; and finally (7) a discussion of deadlock handling.

### 2.1. Overview

In order to exploit a program's parallelism, the program must be partitioned, an activity that has been the subject of much research. Because optimal partitioning is NP-complete, these partitioning techniques strive for near-optimality, usually through the use of heuristics or programmer supplied directives. PODS performs partitioning automatically using the decomposition implied by the program structure. Programs are broken into code-blocks by the ID Nouveau compiler and replicated on each PE, making all processors homogeneous with respect to code. The key problem with partitioning and distribution in PODS is that of determining where to send tokens that activate SPs. Since the PEs are homogeneous, an instance of a specific SP can be executed anywhere simply by routing the initial activating tokens to a specific PE. Because each PE is aware only of its own state, this routing decision is binary: should an SP execute locally or remotely? PODS decides which SPs will be distributed and which will run locally at compile time. At run-time PODS decides where the distributed SPs will be executed. The exact methods for this distribution are explained in this chapter.

Simply put, the PODS partitioning and distribution uses data distribution to control execution distribution. There are two basic conceptual steps to achieve this.

1. Using a simple global algorithm, partition the data and allocate each partition to a PE.

2. Execute the program such that the owner of a particular array element will write that element.

By using a simple global algorithm for array partitioning, each PE can easily calculate where a particular array element is located during execution. This additional checking costs 29% more cycles for each array read or write, but allows arrays to be accessed in parallel with little or no communication and without context switching.

In order to realize the above, the following tasks are performed:

1. Arrays are cut-up into pages of fixed size  $X$ , where  $X$  is determined by the hardware architecture.
2. Arrays are grouped into superpages which are assigned to PEs sequentially.
3. Execution follows the array partitioning and distribution if it is executing loop code which has no Loop-Carried Dependencies (LCDs).
4. For code with LCDs, the execution will stay on the current PE unless a function call is made.
5. When a function call is made the execution may move to another PE depending upon the length to the current PE's task list.

There are three primary mechanisms for achieving data parallelism. These mechanisms are:

1. The ALLOCATE Operator: used to distribute data (data parallelism).

2. The DIST-L Operator: used to spawn processes on all PEs.
3. The RANGE-FILTER Operator: used to restrict loop indices ranges for different PEs.

The basic approach to distribute code for data parallelism is to:

1. distribute the arrays
2. decide which level of the nested loop to distribute
3. this level gets the RANGE\_FILTER while its parent gets the DIST-L operators.

The mechanism for functional parallelism:

1. The APPLY Operator: used to spawn function calls on a single remote PE (functional parallelism).

In this way the work load is partitioned at compile time and distributed using an efficient run-time algorithm without the programmer's explicit instructions.

## **2.2. Underlying Principles**

There are two basic principles which apply to any parallel system. They are:

1. The Equal Distribution Principle
2. The Centralization Principle

These two are supplemented by three PODS specific principles. These principles show ways in which the two basic principles can be reconciled somewhat. The PODS specific principles are:

1. The Grouping Principle
2. The Virtual Sources Principle
3. The Collector Writes Principle

By using each of these principles, PODS is able to provide efficient execution of scientific programs on MIMD machines. Each principle is explained below.

### 2.2.1. Basic Principles

For any assignment to be accomplished, the RHS calculations must be performed and the writing of the element must occur. Consider the simple assignment below:

```
A[i] = sqrt (B[i+1] + C[i]) * exp (D[m+i])
```

FIGURE 2.1. SIMPLE ARRAY ASSIGNMENT.

In this statement  $B[i+1]$ ,  $C[i]$ , and  $D[m+i]$  are data sources which need to be collected together so that the calculations can be performed. Once they are performed the assignment can occur. The diagram below illustrates these how these three agents interact. Note that each data source, the data collector, and the data storage could be on different PEs.

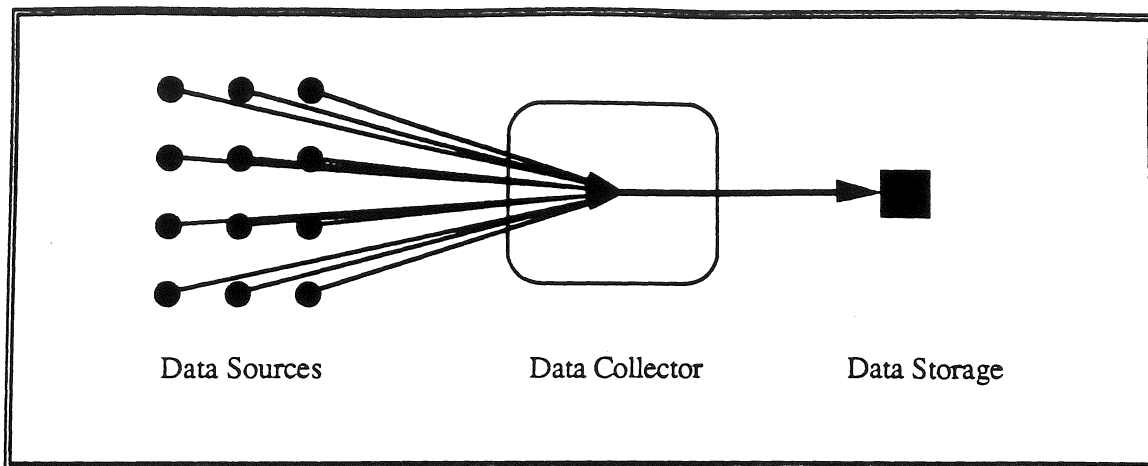


FIGURE 2.2. EQUAL DISTRIBUTION PRINCIPLE.

In order for the data sources to respond to multiple data collectors simultaneously they should be spread over all the available PEs. Since the access patterns are not known a priori, each PE should get an equal number of data sources. This is the Equal Distribution Principle. More concisely,

**Definition: Equal Distribution Principle**

In order to allow maximum parallel access, data sources, data collectors, and data storage should be distributed equally among the available PEs.

This principle is implemented in PODS by partitioning each array and distributing the pieces equally among the PEs.

The Centralization Principle concerns the cost of communication and the overloading of the interconnect network. Once the agents are widely distributed a problem occurs. The communication costs become extremely high. In order to reduce the effects of communication delays, all of the items (data sources, data collectors, and data storage) should be kept together (i.e. centralized). This is the Centralization Principle which states:

### Definition: Centralization Principle

In order to reduce communication costs and network overloading, data sources, data collectors, and data storage should be centralized on one PE.

These two principles are obviously in conflict. The PODS specific principles below show how the balance can be tilted in favor of distribution.

#### 2.2.2. PODS Specific Principles

##### Grouping Principle

In order to reduce the effects of communication delays without completely centralizing, the data sources should be grouped together until some size,  $x$ , is reached. The diagram below shows how the number of communication lines is reduced by grouping.

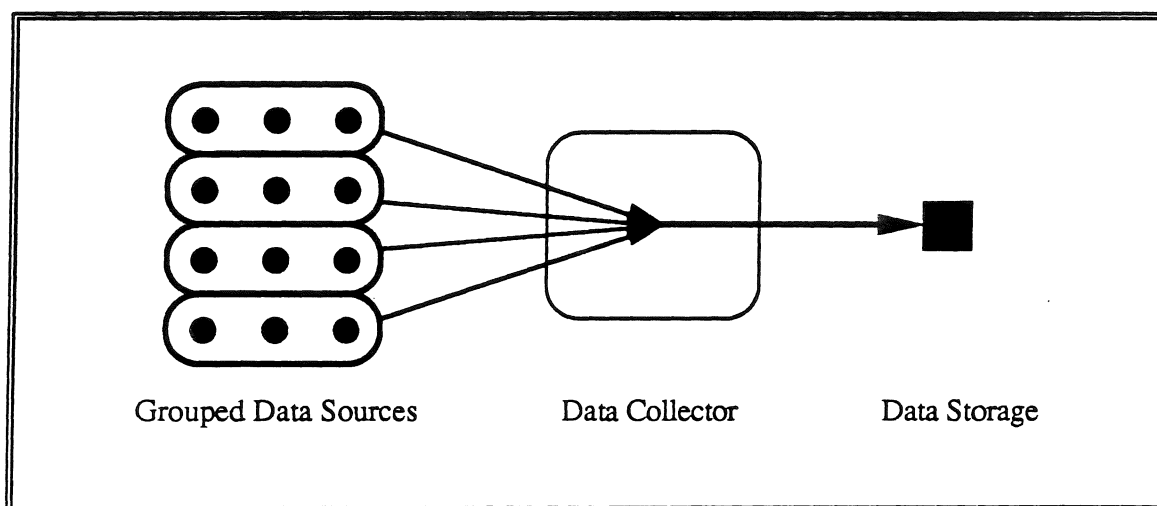


FIGURE 2.3. GROUPING PRINCIPLE.

This is the Grouping Principle which states the following.

### Definition: Grouping Principle

In order to reduce communication over the network, data sources should be grouped together until some reasonable size is reached.

This principle fights against the Equal Distribution Principle, a balance between them must be maintained. In PODS this is achieved by grouping the arrays into pages of a fixed size which is only dependent on the hardware architecture.

### Virtual Sources Principle

One aspect of single assignment is that data sources never need to be updated. This can be exploited by moving copies of the data sources into the collector for easy access. Locality of reference implies that the grouped data sources should be moved in toto when one of the data sources is needed. The diagram below shows how the amount of communication can be reduced by caching the data source in the collector without any cache coherency problems; the dashed lines are truly one way.

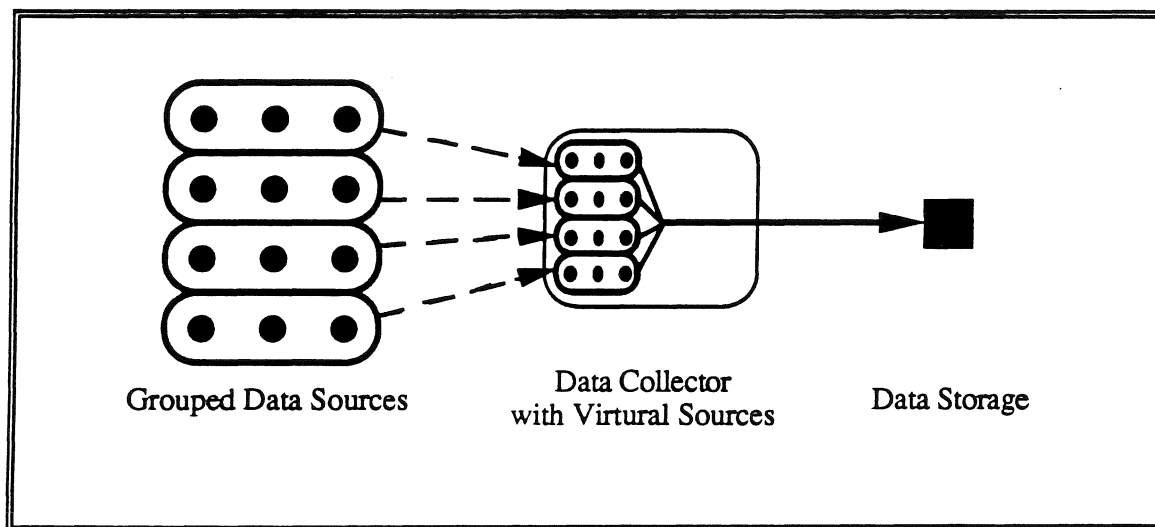


FIGURE 2.4. VIRTUAL SOURCES PRINCIPLE.

This is the Virtual Sources Principle which states the following.

### Definition: Virtual Sources Principle

Since each data source will never need to be updated, a copy should be moved into the data collector when any one of the grouped data sources is needed. The Virtual Sources Principle states that a single assignment system should cache data sources in its local memory to form a virtual source to reduce communication.

This principle allows remote reads to be reduced in PODS, and is implemented by remote access caching.

### Collector Writes Principle

In a single assignment system there will be only one write to a particular array element. The thick black arrow in the diagrams above represents this write. Since there is only one collector and one write, these two should be on the same PE. The diagram below shows this.

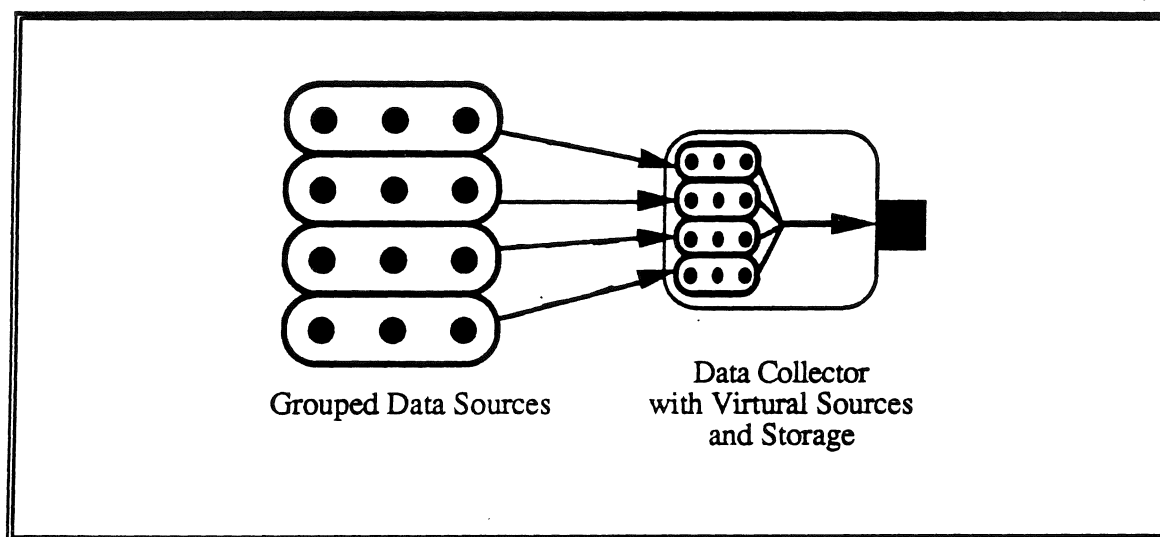


FIGURE 2.5. COLLECTOR WRITES PRINCIPLE.



The producer of an array element is the PE which collects the RHS calculations needed for the formation of a LHS value. This PE, the collector, becomes the writer by executing the WRITE\_ARRAY instruction which assigns that array element a value. Since the single-assignment principle is in force; there will be one writer. This is the Collector Writes Principle which states the following.

**Definition: Collector Writes Principle**

The Collector Writes Principle states that the system should map an array element such that the PE which holds that array element in its local memory (the owner) shall be the collector of the RHS data sources, and shall also be the writer of that array element.

This principle, in collaboration with the other principles, forces the execution to follow the data distribution. In PODS this is called Data Distributed Execution.

**2.3. PODS Instructions and Processes**

The basic concept of a dataflow operator has *not* changed, only the *implementation* of that concept. In PODS dataflow operators are implemented using PODS instructions. The basic dataflow concept (shown below) allows the dataflow graph to execute cleanly; without leaving tokens unconsumed.

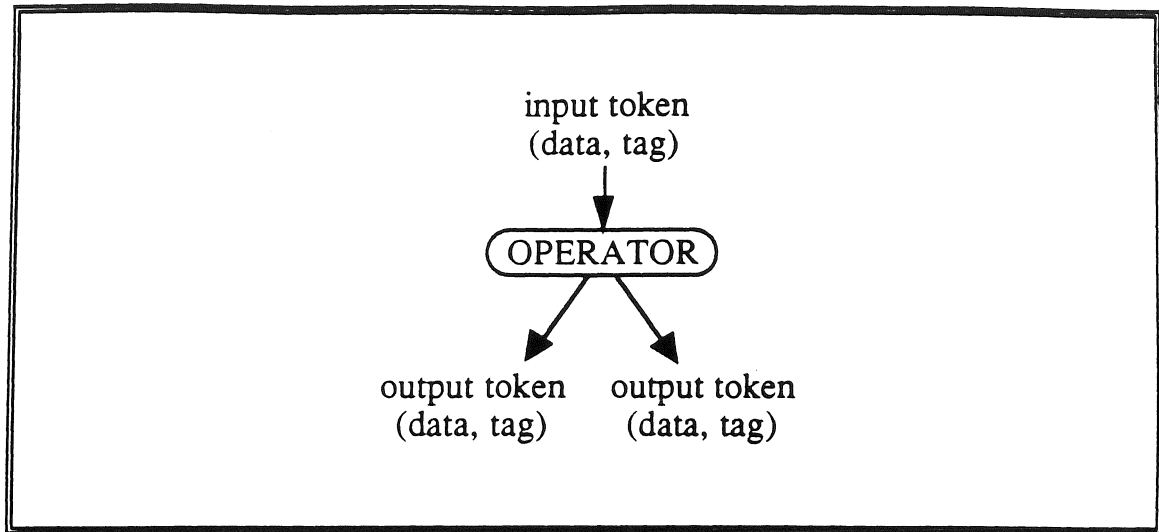


FIGURE 2.6. BASIC DATAFLOW OPERATOR.

The standard dataflow implementation of this concept performs the following steps when a token arrives:

1. consume input tokens
2. compute new data value
3. compute new tag
4. form new output tokens
5. send output tokens to destination operators

For PODS this implementation needs to be modified to contain the concept of an SP's state. An SP's state is basically a PODS *activity name*, which is discussed next in Section 2.3.1.

### 2.3.1. Activity Names

An activity name is the colored tag which identifies a token's complete context. What is presented below is a logical implementation, a physical implementation would use unique

frame IDs. Logically, activity names consist of two parts: (1) the static part which is known at compile time; and (2) the dynamic part which is built as the token moves from context to context. Figure 2.7 below shows the make-up of an activity name.

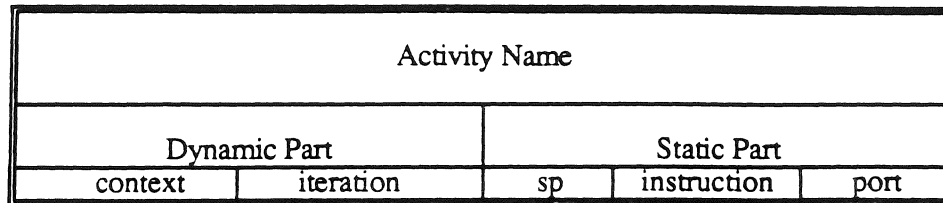


FIGURE 2.7. ACTIVITY NAME COMPONENTS.

The static part is known by the compiler from the dataflow graph once the SPs are built. The dynamic part is based upon the incoming token's activity name and is only affected by the context manipulating functions: D and D\_INVERSE, L and L\_INVERSE, A and A\_INVERSE. The activity name is also known as the *tag*. The individual subparts are listed below, along with their function.

- context: holds the pointer to past activity names, affected by L and L\_INVERSE, A and A\_INVERSE. The context holds a token's tag in a linked list. This list represents all of the execution scopes through which a given token has passed. This information is necessary for PODS to know how to move a token from one execution scope (i.e. SP) to another.
- iteration: holds the current iteration number, affected by D and D\_INVERSE.
- sp: holds the SP number, based on partitioned dataflow graph.
- instruction: holds the instruction number within this SP.

- port : holds the port number within this instruction, usually 0 or 1.

### 2.3.2. PODS Instruction Format

There are three types of PODS instructions. These types indicate how the instruction was derived from the output of the ID Nouveau compiler. The first type is formed from a simple mapping from TTDA instructions and PODS instructions. These are the basic instructions such as ADD, and ARRAY\_READ. The second type actually disappears when the output is translated. These are the IDENT instructions which are used for synchronization. These are not needed because the sequential nature of SPs synchronizes instructions automatically. The third type is composed of new instructions which are added or modified to accomplish the distribution. These are the SWITCH, FORKJUMP, D and D\_INVERSE, L and L\_INVERSE (in both dist and local forms), A and A\_INVERSE, and ALLOCATE. Each of these will be explained as they are encountered in this chapter.

PODS instructions have the following fields (see Figure 2.10 for an example):

1. Op Code — operation to be performed.
2. Number Arguments — the number of arguments this operation needs before it is ready to fire.
3. Operand List — slots for values of operands. Initially some of the operands are constants which are set at compile time. Each constant is represented by the pair (value, port). Other operand ports are flagged with a special "sticky bit" (STKY) which means that once a token is received on that port, it is then held there and does not need to be replenished for the instruction to fire.

4. Local Destination List — output value destinations which are within this SP. Each destination is represented by the pair [instruction number, port].
5. Route ID — ID of route to be used when output tokens are to be sent to other SPs. This is not a list because the routing information is stored in the Routing Unit and not in the Execution Unit. A route ID is simply a short-hand for: [SP ID, instruction number, port] [SP ID, instruction number, port] [SP ID, instruction number, port] ..., see Chapter 3 for complete details.
6. Comments — variable names from the source code, shown in brackets, "{}".

Values can be sent using any of the following paths:

1. Using the local destination list. This is the way almost all of the operators communicate. Only L and A operators can send tokens to other SPs.
2. Using the route list. This is performed in one of three ways depending on the type of L or A operator. Only L or A operators have routes.
  - (1) the DIST-L operator sends tokens to SPs on every PE.
  - (2) the LOCAL-L operator sends the token to a different SP on the same PE.

(3) the A operator sends the token to a different SP on some PE. Which PE is decided by a hash function.

### 2.3.3. PODS Dataflow Operator Implementation

In PODS, an SP contains code and a state. The code represents the operations to be performed and the state holds the status of these operations.

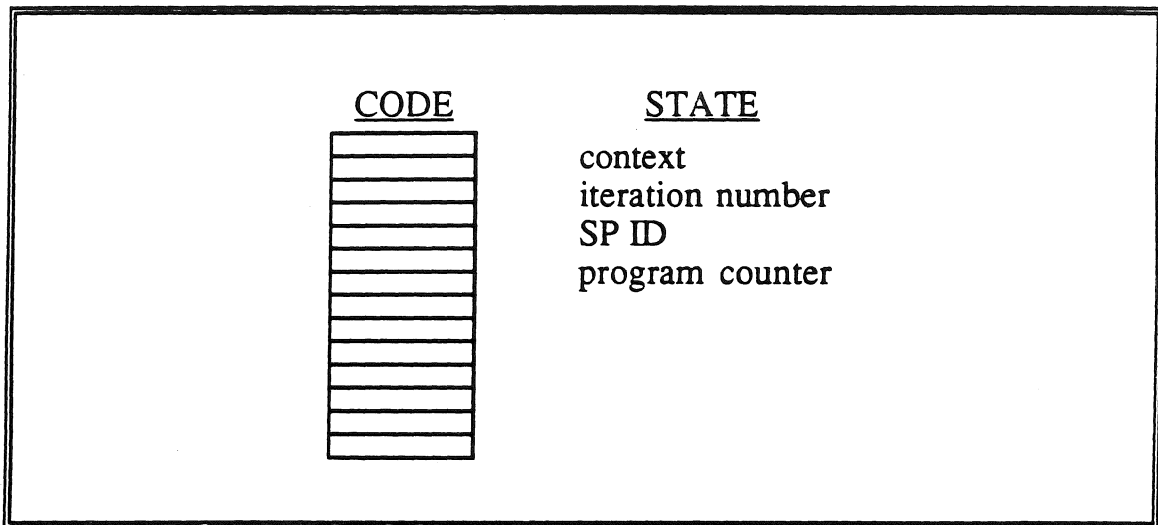


FIGURE 2.8. SP COMPONENTS.

When a token arrives at a PODS operator the state of the SP is used to decide the steps to execute this operator. All of the original ID operators which are not special operators are called basic PODS operators. All of the special operators are discussed individually after a discussion of the basic PODS operator implementation.

The basic PODS operator implementation performs the following steps when a token arrives:

1. Consume input tokens.
2. Compute new data value.

3. Compute new tag.
4. If the context and SP ID are the same, then no tokens are formed, only data is stored into destination instruction and port. If either of these has changed, then form new output tokens and route them using the routes specified for this operator.
5. Increment the program counter.

This implementation is the same as the basic dataflow version in Steps 1 - 3. Step 4 however now checks the SP state to see how to deal with the output data, whether to store it locally within this SP or to form a token and route it to another SP. Notice that Step 4 does not check the iteration number of the tag. This is because the iteration number can only be changed by a D operator, and D operators do not change SP. Step 5 has been added to increment the program counter. There are a couple of operators (the D and FORKJUMP operators) which set the program counter to a value rather than just incrementing it. All other operators follow these steps exactly. What follows is a description of the new PODS instructions, and why these implement the same semantics as the original ID operators.

In order to show that the semantics of the original ID operators have not changed each operation type will be addressed. It is quite simple to understand the way in which PODS implements the semantics of ID. The original ID had the following fields in its tag: context  $c$ , procedure  $p$ , statement number  $s$ , and iteration  $i$ . As explained above in the section on activity names, PODS uses a context  $c$ , a SP ID  $sp$ , an instruction number  $si$ , and an iteration  $i$ . PODS uses the context and iteration exactly the same, it is only the procedure and statement number which differ.

Basically the procedure cuts the dataflow program into subsets, and the statement number identifies the operator within the subset. PODS uses the same approach but just cuts the collection into smaller subsets. In Figure 2.9 below, the set of all operators is cut into procedures Proc1 - Proc4 (in bold lines), while the SPs are just subsets sp1 - sp8. In this way the combination of the two field holds exactly the same information, i.e. the "address" of a particular operator. Also note that since each procedure cut is also an SP cut, then when a procedure change is made an SP change is also make.

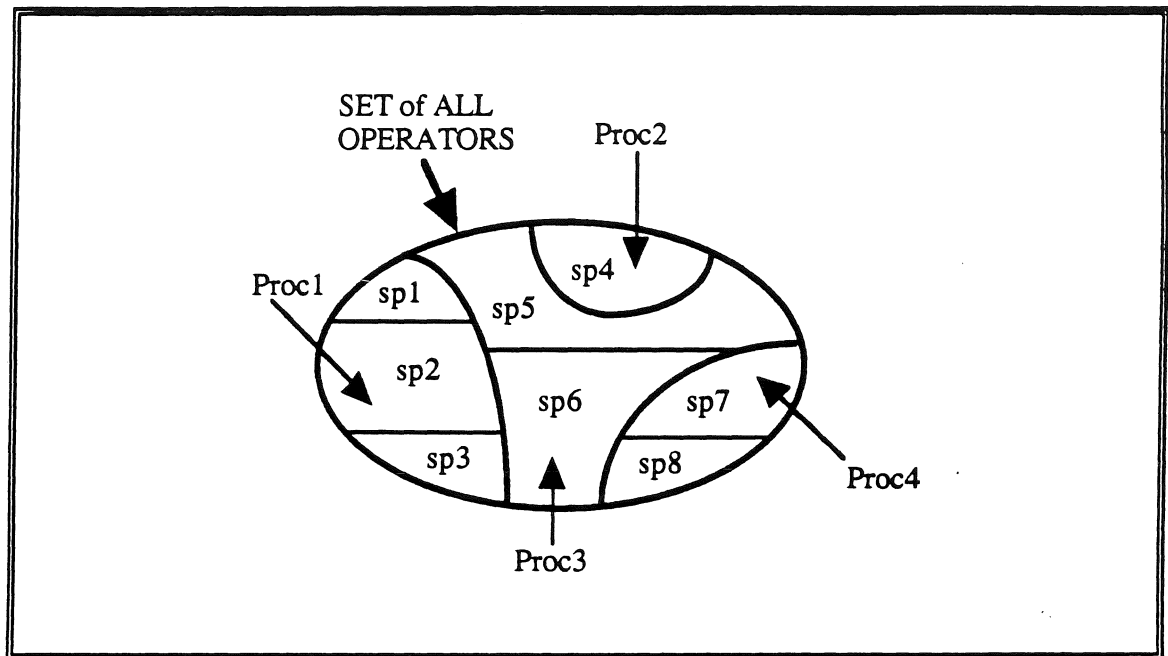


FIGURE 2.9. ID VS PODS STATEMENT "ADDRESSING".

### Arithmetic and Logical Operators

The vast majority of ID operators fit into this the class of arithmetic and logical operators. In the original ID these operators only changed the statement number and the value of the token. This can be expressed by:

ID Arithmetic & Logical

$c, p, s, i, v \rightarrow c, p, s', i, v'$



In PODS exactly the same value calculation is performed, and the instruction number is changed. Expressing this in a similar format to the above:

PODS Arithmetic & Logical  $c, sp, si, i, v \rightarrow c, sp, si', i, v'$

Notice that the "address" ( $sp, si'$ ) for the output token specifies the receiving operator just as is done in ID with ( $p, s'$ ).

The switch operator falls into this class and is discussed along with a new instruction (forkjump) below.

### SWITCH and FORKJUMP

The SWITCH and FORKJUMP work in conjunction to form a branch type of operation. The PODS SWITCH is much like the original ID SWITCH with the following exception: once tokens are passed along, the program counter is modified by a true or false relative offset. The original ID SWITCH performed the following:

ID SWITCH  $c, p, s, i, v \rightarrow c, p, s', i, v$

PODS performs the following which is exactly the same except the addressing differences, which are equivalent.

PODS SWITCH  $c, sp, si, i, v \rightarrow c, sp, si', i, v$

In order to execute a PODS SWITCH Steps 4 and 5 of the basic implementation need to be replaced. The new Steps 4 and 5 are:

4. If the predicate is true, then store output values into true destination instructions. If the predicate is false, then store the output values into the false destination instructions.

5. If the predicate is true, then increment the program counter by the true relative jump. If the predicate is false, then increment the program counter by the false relative jump.

Once the input tokens are present the SWITCH fires, sending tokens to either the true or false branch and jumping to the next instruction to execute. The PODS instructions below were taken from Matrix Multiply. As described previously, the fields have the following meanings: (1) instruction number; (2) op code; (3) number of arguments; (4) operand slots; (5) destinations; and (6) a comment. For SWITCH the number of arguments is always five, port 0 is the predicate, port 1 is the value, port 2 is the true relative offset, port 3 is the false relative offset, and port 4 is the number of true destinations. The destinations are ordered such that the false destinations are last. The FORKJUMP always takes two arguments: one is the value to be passed (port 0), the other is the relative offset (port 1).

10	SWITCH	5	(1.00,2)	(11.00,3)	(2.00,4)	->	[18,0]	[19,0]	[21,0]	{I}
18	FORKJUMP	2	(-17.00,0)			->	[1,0]	[2,1]		

FIGURE 2.10. PODS SWITCH AND FORKJUMP INSTRUCTION EXAMPLES.

To form a simple branch the SWITCH and FORKJUMP are used together as shown in Figure 2.11 below. The true relative jump of the SWITCH is set to 1, the false relative jump is set such that the program counter will jump to the first false instruction on a false predicate. The FORKJUMP is used to skip the false instructions, its relative jump is set to go to the beginning of the unbranched instructions.



For PODS the implementation performs something very similar. As for arithmetic and logical operators, the new "address" of the output token will be  $(sp, si')$  rather than the ID  $(p, s')$ . Otherwise PODS does exactly the same as ID.

PODS D	c, sp, si, i, v -> c, sp, si', i+1, v
PODS D_INVERSE	c, sp, si, i, v -> c, sp, si', 0, v

In order to execute a PODS D instruction Steps 4 and 5 of the basic implementation need to be replaced. The new Steps 4 and 5 are:

4. Increment the iteration number,  $i$ , and store output values into destination instruction and port.
5. Increment the program counter by the relative jump.

The D\_INVERSE operator implementation is very similar to the D operator's. It merely resets the iteration number to zero rather than incrementing it. Specifically, the Step 4 of the basic implementation should read:

4. Set the iteration number,  $i$ , to 0 and store output values into destination instruction and port.

In order to produce a loop, the SWITCH takes the iteration variable and passes it into the loop body on a true predicate. Inside the loop body the iteration variable is modified (usually just incremented by one), and the D operator is placed at the end, see the code fragment from Matrix Multiply below. The D operator feeds both the predicate and the switch so that the loop test can be performed. In the example below the relative offset of the D operator is -11, which will cause the program counter to be set to 9 ( $20-11=9$ ) after the D operator is executed. The loop body is from instruction 11 to instruction 19. The

D\_INVERSE will reset the iteration number once the loop has exited. The loop will be exited from the SWITCH on a false predicate. Note that the SWITCH at instruction number 10 has a false relative offset of 11 and the last destinations offset is to instruction 21 ( $21 = 10 + 11$ ).

9	LE	2 (STKY, 1)	-> [10,0]
10	SWITCH	5 (1.00,2) (11.00,3) (2.00,4)	-> [18,0] [19,0] [21,0]
11	DIST_LOPERATOR	1 (STKY,0)	-> (12)
12	DIST_LOPERATOR	1 (STKY,0)	-> (14)
13	DIST_LOPERATOR	1 (STKY,0)	-> (15)
14	DIST_LOPERATOR	1 (STKY,0)	-> (10)
15	DIST_LOPERATOR	1 (STKY,0)	-> (11)
16	DIST_LOPERATOR	1 (STKY,0)	-> (13)
17	DIST_LOPERATOR	1 (STKY,0)	-> (16)
18	DIST_LOPERATOR	1	-> (1)
19	PLUS	2 (1.00,1)	-> [20,0] {NEXT-I}
20	D	2 (-11.00,1)	-> [9,0] [10,1] {I}
21	DINV	1	->

FIGURE 2.12. PODS CODE FRAGMENT FOR A LOOP.

### L and L\_INVERSE

In order to perform code distribution the original ID L operators need to be changed from their original implementation. In PODS L and L\_INVERSE are used to route tokens between SPs. There are also two versions of each operator: a DISTRIBUTE version and a LOCAL version.

In the original ID L operators were for entering and exiting loops. This is still true; however, in PODS entering and exiting loops means entering and exiting an SP. In the original ID the procedure  $p$  of a tag does not change as the token passed though the L and L\_INVERSE, however a new and unique context  $c$  is created. The new context is the concatenation of the old context, statement number, and iteration. This is shown below:

ID L  $c, p, s, i, v \rightarrow (clsli), p, s', 0, v$

ID L\_INVERSE (clsli), p, s', i', v -> c, p, s, i, v

In PODS the implementation is as follows:

PODS L c, sp, si, i, v -> (clspli), sp', si', 0, v

PODS L\_INVERSE (clspli), sp', si', i, v -> c, sp, si, i, v

This implementation also generates a new, unique context *c*. This stored context is then used in the L\_INVERSE for returning to the previous context. The only real difference is that the change in SP must be recorded in the tag. Referring back to Figure 2.9, L operators move the scope from one SP to another *within* the same procedure (e.g. from sp1 to sp2). Since the output token no longer has the same context, it will be sent to the Routing Unit to be routed to the receiving SP.

L and L\_INVERSE operators perform routing by referencing a particular route list. The figure below shows two L type operators from Matrix Multiply. The LOCAL\_LOPERATOR is using route list 7 with the LOCAL\_LINV operator is using route list 9. A route list is a list of destination addresses, each consisting of an SP, an instruction, and a port. This information is static and known at compile time. By duplicating this route table in every PE, each Routing Unit can find a particular instance of an SP.

20	LOCAL_LOPERATOR	1	-> (7)
12	LOCAL_LINV	1	-> (9)

FIGURE 2.13. EXAMPLE L OPERATORS.

The LOCAL and DISTRIBUTE versions of each operator tell the Routing Unit to (1) send the token only to its own PE, or (2) to distribute copies of this token to all PEs. Tokens are distributed when the receiving SP is distributed. This way all of the PEs are given the



This is a simple and efficient method for calling procedures and is somewhat akin to the fastcall apply used by Iannucci, [Ian88]. The instructions below were taken from SIMPLE, and form a function call to and return from the procedure TLU. APPLY operators take a variable number of arguments. One for the return instruction (port 0), one for the number of parameters to pass (port 1), and then one for each parameter (ports 2 to n+1). The INV\_APPLY takes two arguments: one for the return value (port 0), and one for the instruction number to return to (port 1).

```

from CONDUCTION-3.pods
9  APPLY          6 (10.00,0) (4.00,1) (STKY,2) (3.00,5) -> (121) {TLU}
from TLU.pods
18 INV_APPLY      2 -> (121)

```

FIGURE 2.14. EXAMPLE APPLY AND INV\_APPLY OPERATORS.

#### 2.4. Array Partitioning and Distribution

In scientific code a number of large arrays are used. It is critical that access to these arrays be efficient. This is the idea vector processors are based upon [H&B84]. In PODS, modified I-structures form the basis for array operations. I-structures are data structures which can be resized as necessary and enforce the single assignment principle with presence bits [ANP87a, ANP89]. PODS also uses presence bits, but arrays are of a fixed size which is determined at allocation time.

The single assignment principle guarantees that only *one* instruction will ever write to an array element; it is the *producer* of that data. PODS exploits this fact by attempting to map each array element onto the same PE as its producer instruction, this is how PODS uses the Collector Writes Principle. However, it is not always possible, nor efficient for the collector to be the owner, as is explained below. By locality of reference, the statements which read an array element will be "close" to the writer. Thus having the writer and



owner the same will allow most array reads to be local rather than remote. Having local array reads is important, since once the array element is written there can be read many more times. Making these array reads efficient is central to PODS.

In order to make the array reads efficient, the array caching scheme detailed in Chapter 3 is used. This simple scheme produces excellent results [BNR89b] as long as the array is accessed in the same direction as it is partitioned. For two dimensional arrays this means that arrays accessed in a row-major manner should be partitioned row-major. Generalizing to multiple dimensions, this means that first-major (last-major) code should be used to access first-major (last-major) arrays.

One approach to ensure that the direction is correct is to analyze each array's accesses and estimate which direction would be more efficient. Analyzing the one filling algorithm (there usually will be only one due to the single-assignment principle) could be done, but the reads matter more because there are many more of them. Analyzing the reads would require that the entire execution trace of the program be known at compile time, which is not possible. To see some of the difficulties, consider a matrix-multiply function which takes arrays A and B as arguments. In ID Nouveau the code would be:

```

Def mm A B = {(l1,u1), (l2,u2) = 2D_bounds A;
C = i_matrix ((l1,u1), (l2,u2));
In
{ For i <- l1 To u1 Do
  { For j <- l2 to u2 Do
    s = 0;
    C[i,j] =
      { For k <- l1 To u1 Do
        Next s = s + A[i,k] * B[k,j];
      Finally s
    }
  };
  Finally C
}
};

```

FIGURE 2.15. MATRIX MULTIPLY ID NOUVEAU SOURCE CODE.

By examining this code it is easily seen that array A should be row-major and array B should be column-major based on the reads. However, an array is partitioned at allocation time and stays that way for its entire lifetime. So if the Matrix Multiply function was called with MM X Y, array X should be row-major and array Y should be column-major, and if called with MM Y X then the reverse is true. However, the binding between A (B) and X (Y) is dynamic and hence PODS cannot take advantage of it. This late binding also prevents the proper direction for each array to be used every time.

A better approach is to pick a direction and use it, letting the programmer know which direction is appropriate. This is the approach used by many popular languages today. For example, 'C' is row-major and FORTRAN is column-major. PODS uses row-major partitioning.

In order to better understand this partitioning, consider the following example. A two dimensional array which is 8 x 256 is to be partitioned and distributed over 20 PEs. For the iPSC/2 and the simulations herein, the best page size is 32 elements or approximately 2 kilobytes. Previous studies have shown that this is not a critical parameter [BNR89b].

Following the simple array partitioning algorithm, each array is divided into pages of 32 elements in row-major fashion.

Once the array is cut into pages (linearly, in row-major), the pages are grouped together sequentially to form superpages; one superpage per PE, see Figure 2.16 below. The algorithm for achieving this is as follows:

1. calculate the number of pages,

$$\#pgs = \text{floor}(\text{number of elements} / \text{page size})$$

2. calculate the number of pages per PE,

$$\#ppp = \text{floor}(\#pgs / \text{number of PEs})$$

3. each PE gets #ppp pages
4. the extra elements left over from step 1 are assigned to the last PE
5. the extra pages from step 2 are assigned, one to each PE, starting with the second to last PE and continuing to the first PE

Often a superpage will wrap around the logical array limits. This only needs to be handled properly when the array is accessed. It is also the case that sometimes a few PEs will end up with one more page in its superpage than the others. Both of these situations are handled by the boundary table. The handling of these cases will be explained in detail in Chapter 3, PODS Logical Implementation. For the example PE #0 through PE #15 have 3 pages, while PE #16 through PE #19 will have 4 pages.

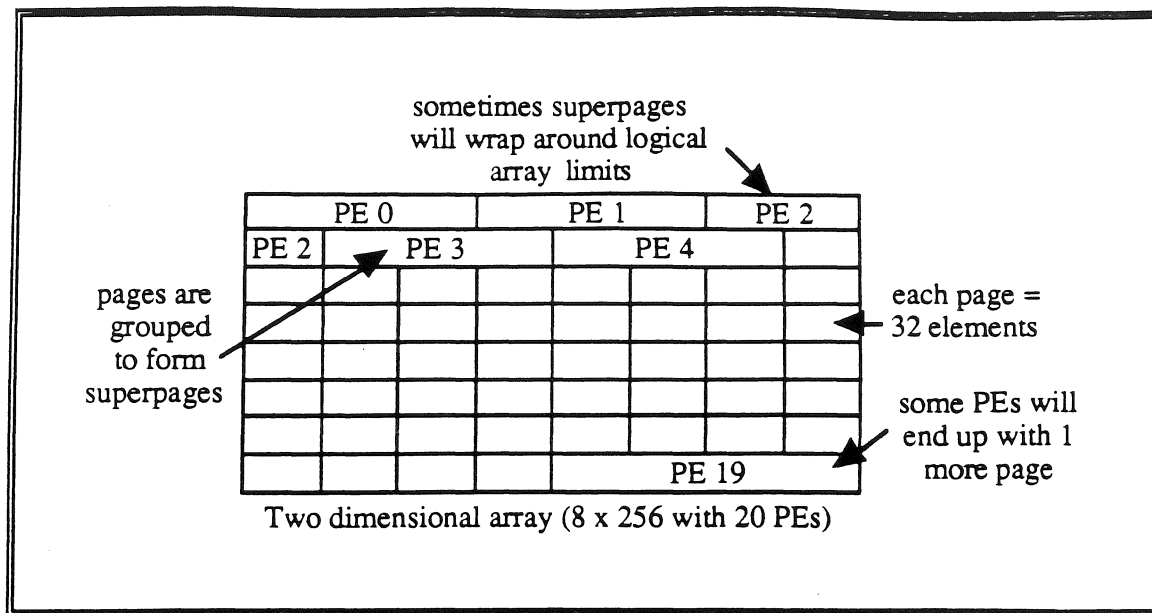


FIGURE 2.16. PODS PARTITIONING OF A 2-D ARRAY.

One key concept of this approach is that it is known globally and requires limited information to use. It is the ALLOCATE instruction which performs this data distribution. Each ALLOCATE works with a FORKJUMP and performs the following:

1. The ALLOCATE requests an array ID from the local Array Manager (see Chapter 3).
2. The SP continues executing until the ALLOCATE's companion FORKJUMP (placed directly after the ALLOCATE). The SP will either block, until the Array Manager responds with an array ID or will continue executing if the value has already returned.
3. When the Array Manager receives the allocate request, it will allocate the necessary space, build the array header, build the boundary table, send the array ID to the requesting SP, and then send a remote allocation request onto all of the other PEs with

the array ID attached. In this way all of the PEs have the same ID for the same array. The PE which executes the ALLOCATE is called the *host* PE, this PE number is also sent as part of the request.

4. The remote PEs will receive the remote allocate request and build the header and tables, and allocate the appropriate space.

For a two dimensional array PODS stores the following array header information in each PE:

Field Name	Description
<code>beginning_offset</code>	start of this PEs responsibility
<code>ending_offset</code>	end of responsibility
<code>number_of_dimensions</code>	2
<code>size_dim1</code>	size of first dimension
<code>size_dim2</code>	size of second dimension
<code>ELEMENT_SPACE</code>	space allocated for this array on this PE
<code>beginning_range1_dim1</code>	start of first range interval in dim 1
<code>ending_range1_dim1</code>	end of first range interval in dim 1
<code>beginning_range1_dim2</code>	start of first range interval in dim 2
<code>ending_range1_dim2</code>	end of first range interval in dim 2
<code>beginning_range2_dim1</code>	start of second range interval in dim 1
<code>ending_range2_dim1</code>	end of second range interval in dim 1
<code>beginning_range2_dim2</code>	start of second range interval in dim 2
<code>ending_range2_dim2</code>	end of second range interval in dim 2
...	
NULL	

TABLE 2.1. PODS ARRAY HEADER INFORMATION.

The *beginning\_offset* and *ending\_offset* are the starting and stopping points of this PEs area-of-responsibility expressed in the row-major linearized version of the array. The *number\_of\_dimensions*, *size\_dim1*, and *size\_dim2* fields hold the number of dimensions and sizes of each for this array. The `ELEMENT_SPACE` is where the actual data is stored,

excluding the cache. The *beginning\_rangeX\_dimY* and *ending\_rangeX\_dimY* fields hold the starting and stopping points for each range interval of this array. Superpages can wrap around an array dimension, like PE #2 in Figure 2.16 above, this causes multiple range intervals in the boundary table. The bolded fields make up the boundary table for this array on a given PE. Boundary Tables will be discussed in detail in the section on range filters.

The header is similar for other dimension arrays. For example, for a three dimensional array the *number\_of\_dimensions* would be 3, there would be an extra dimension size field, *size\_dim3*, and there would be an additional *beginning\_range* and *ending\_range* for each segment. Notice that the header size is fixed at allocation time and will not grow.

Continuing with the two dimensional array example in Figure 2.16, the header for PE #2 would be:

Field Name	Value
beginning_offset	192
ending_offset	287
number_of_dimensions	2
size_dim1	8
size_dim2	256
ELEMENT_SPACE	space allocated for this array on this PE
beginning_range1_dim1	0
ending_range1_dim1	0
beginning_range1_dim2	192
ending_range1_dim2	255
beginning_range1_dim1	0
ending_range1_dim1	0
beginning_range1_dim2	0
ending_range1_dim2	31
NULL	

TABLE 2.2. 2-D ARRAY EXAMPLE HEADER.

To perform a two dimensional read the offset into the array must be calculated first. Then the beginning and ending offsets must be checked. If the offset is not within the bounds then the read is remote and a message must be sent to the owning PE. If the read is local, the presence bit must be checked. If it is not present then the read must be enqueued, as in

I-structures. If the value is present then the memory location is read. The pseudo-code for performing the read is:

```

offset = size_dim2 * i + j
if (offset < beginning_offset) goto REMOTE_READ
if (offset ≥ ending_offset) goto REMOTE_READ
if (element not present) goto ENQUEUE_READ
value = array[offset]

```

FIGURE 2.17. 2-D ARRAY READ PSEUDO-CODE.

Continuing with the above example, assume the expression below is being executed on PE #2.

$$\text{result} = A[0,10] + A[1,10];$$

Assuming both elements have already been written, the first array read,  $A[0,1]$ , would perform the following read calculations.

```

offset = size_dim2 * i + j
        = 256 * 0 + 10
        = 10
if (offset < beginning_offset) goto REMOTE_READ
    10 < 192
    goto REMOTE_READ

```

FIGURE 2.18. EXAMPLE 2-D ARRAY REMOTE READ.

The REMOTE\_READ sends a message to the owning PE (PE #1), who will respond with  $A[0,10]$ . PE #2 will continue on and encounter the second array read,  $A[1,10]$ . Note that PE #2 did not block this SP when the read was determined to be remote. Only when the instruction which consumes the result is reached will the SP block. By that time  $A[0,10]$  may have been received. The second array read calculations would be:

```

offset = size_dim2 * i + j
        = 256 * 1 + 10
        = 266
if (offset < beginning_offset) goto REMOTE_READ
    266 < 192
if (offset ≥ ending_offset) goto REMOTE_READ
    266 ≥ 287
if (element not present) goto ENQUEUE_READ
    present
value = array[offset]
        A[266]

```

FIGURE 2.19. EXAMPLE 2-D ARRAY LOCAL READ.

The value of array A @ offset 266 would be stored in the consuming instruction. When the consuming instruction was reached the SP would block if A[0,10] had not yet arrived, and PE #2 would start executing the next SP from the task ready list.

Array caching complicates this somewhat, but, it is independent of the PODS partitioning and distribution. In Chapter 3 array caching is examined. On a typical RISC processor (MIPS R3000) the caching version would take 22 cycles while a regular two dimensional read would take 17 cycles. This 29% additional overhead is well worth it.

Note that it is not necessary to distribute all arrays. In the future more analysis may show that certain arrays should be kept local and other distributed, this is an area of current research.

## 2.5. Distributing Processes

Distributing code (i.e., processes) is the key issue in parallel processing. In PODS this is accomplished by following an execution distribution principle which tries to map the calculation of an array element to its owner as much as possible (i.e., Collector Writes Principle). The PODS implementation of the Collector Writes Principle is called Data Distributed Execution (DDE).



### 2.5.1. Data-Distributed Execution Principle

The central concept in PODS code distribution is to follow the data distribution as much as possible. Placing the execution of an operation on the same PE as the location of its data will reduce communication costs and context switches. A system performs DDE when it moves execution to the PE where the data resides.

Consider an  $n$ -dimensional index space, where the dimensions are ordered by the levels of nesting. Say this multiple nested loop has index levels  $i_1, i_2, \dots, i_n$ , and that there is an array write at the inner-most level ( $A[i_1, i_2, \dots, i_n] = x$ ). The goal is to distribute the computations evenly across the PEs using DDE. This is achieved by picking one of the levels of the nest, say  $i_a$ , and cutting up the index space along  $i_a$  into  $\text{number\_of\_PE}$  ranges. The levels previous to  $i_a$  are executed on one PE, while levels after  $i_a$  are executed on every PE. Since the array write needs the value of every index, all of the previous indices ( $i_1, i_2, \dots, i_{a-1}$ ) must be broadcast to every PE, and, every following index ( $i_{a+1}, i_{a+2}, \dots, i_n$ ) must be generated locally — it is the  $i_a$  level which is used to partition the iteration space. However, the data distribution is still followed.

To better visualize this consider a 2-dimensional iteration space with indices  $i$  and  $j$ . Figure 2.20 (a) shows the data partitioning of an array where the superpage assigned to each PE does not reach the end of the array dimension. Figure 2.20 (d) shows the data partitioning of a larger array, where the superpage is larger than the dimension. When the superpage just happens to match the array dimension size the partitioning acts just like it were smaller than the array dimension. Figures 2.20 (b), (c), (e) and (f) show the iteration space partitioning when  $i$  or  $j$  are used for  $i_a$ .

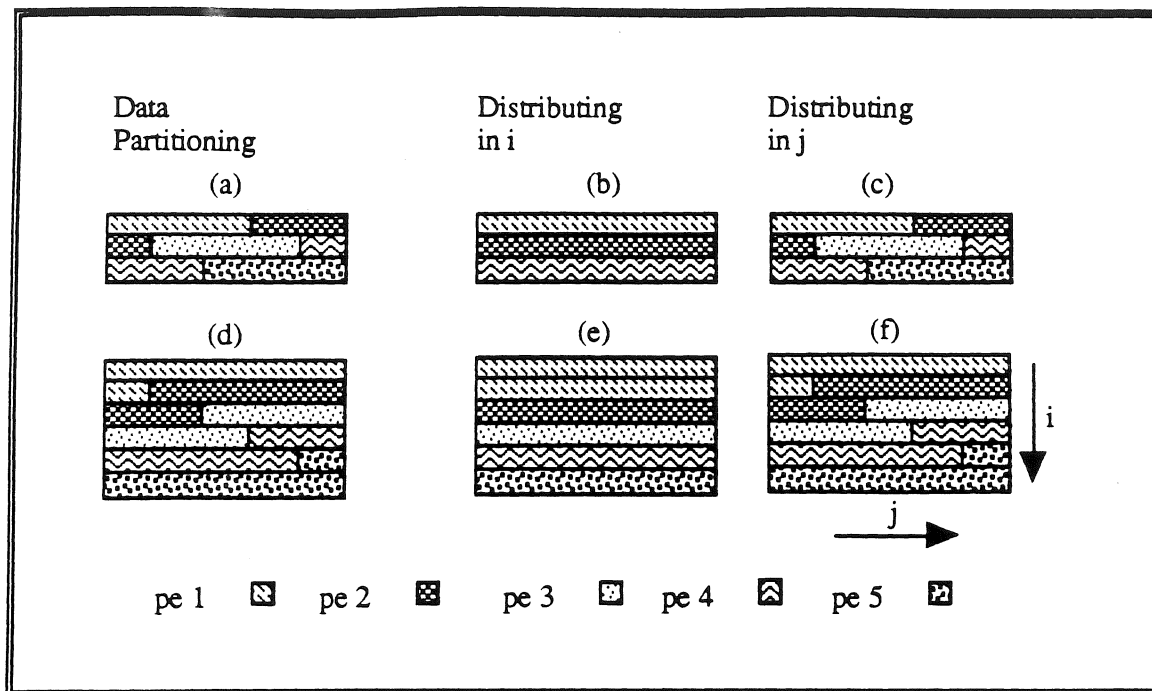


FIGURE 2.20. PARTITIONING A 2D ITERATION SPACE.

In order to ensure single assignment, the iteration space cannot exactly follow the data partitioning in every case. When any level other than the last level is used to partition on, the remaining levels cannot be partitioned and must be assigned based upon the upper levels. Figure 2.20 (b) and (e) show on which PE the calculations will be performed if the iterations were partitioned along  $i$ . This assignment is achieved by simply assigning iteration space areas based upon the first element in each row. This causes some interesting situations. In case (b) PE #3 has *no* iterations to run. While in case (e) PE #1 has two full rows to calculate. Notice that there can be some remote writes, e.g. PE #1 writes to some of PE #2's elements.

When that last level is used to partition on the mapping is exact. This is because all  $i_1, i_2, \dots, i_n$  are available and each PE can completely decide which iterations to perform.

To generalize this to multiple dimensions consider the figure below. In general, the data partitioning, case (a) below, will not exactly match any dimension size. When a level is picked to distribute, all levels below it will use this level's partitioning. Case (b) shows the planes of iteration space responsibility when the  $i$ -th level is used. Case (c) shows the iteration spaces if the  $j$ -th level were used to distribute the iterations. If the  $k$ -th level were used the iteration space partitioning would exactly match the data partitioning, case (a).

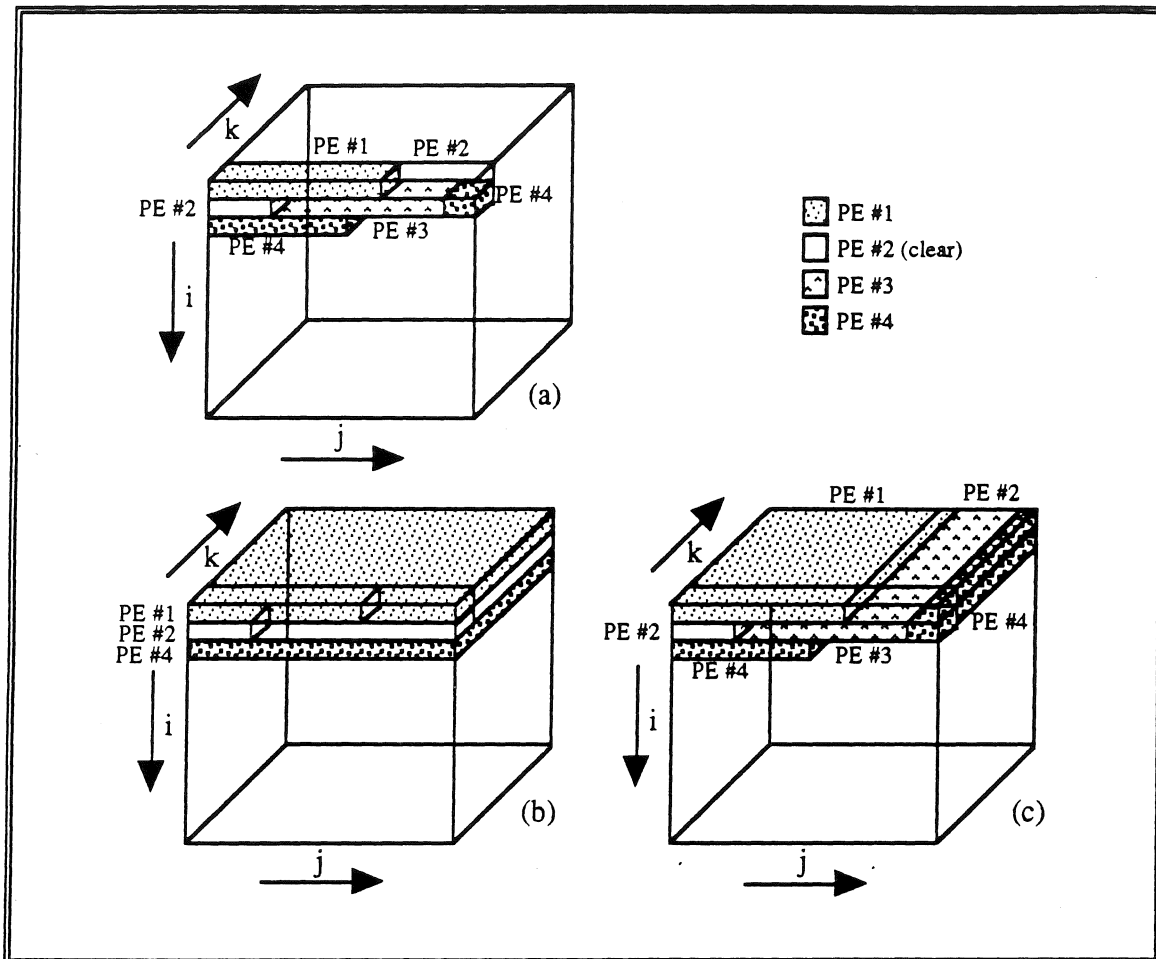


FIGURE 2.21. PARTITIONING A 3D ITERATION SPACE.

This would seem to indicate that the lower the level the better the partitioning. However, the upper levels must communicate their values all the way down to the inner-most level. This causes excessive communication. While distributing at the outer-most level can cause

miss matches, this can be overcome via array caching. Seeing that the sooner the iterations are distributed the fewer the number of broadcasts necessary, PODS always distributes as soon as possible.

These give rise to the distribution scheme below.

1. Given an array A:  
partition and distribute as described in Section 2.4, Array Partitioning and Distribution, above.
2. Given a loop L:  
if L does not contain an array write, then do not distribute  
else distribute the outer-most level of the nest possible.
3. Once the level has been chosen, use the first element in that level  
to determine the iteration space partitioning.

The reason a certain level of nesting cannot be distributed is dependent on the loop-carried dependencies at that level. This is explained in detail in LCD Effects Section below.

DDE can be greatly increased by array caching. In PODS, once a page is read into local memory from a remote PE it is held in a software cache which is replaced using a Least-Recently-Used algorithm. Array caching is explained in detail in Chapter 3.

DDE of for-loops is achieved in PODS by generating only those loop variables which make the array accesses local. This is performed by range filters. The operation of range filters is explained in detail in the next section.

### 2.5.2. Range Filters

In this section, the concept of range filters is explained in detail, and explains how each PE restricts loop execution to its own portion of an array.

#### Objective and Usage

The objective of the range filter construct is to control which iterations of a distributed loop are to be executed by a given PE. The diagram below shows a simplified dataflow of the simple array filling loop in the upper right hand corner. Contrast this with the diagram in Figure 2.23; the same loop after the range filter has been inserted. In PODS the loop nest level in which the range filter is inserted is defined to be the *distributed loop*.

A dataflow diagram with a 2-dimensional range filter is shown in Figure 2.23. The items added to Figure 2.22 are bolded. The range filter replaces the predicate and needs the array  $A$  and the outer index  $i$  from the  $i$ -loop to determine what  $j$ 's a given PE is responsible for. The range filter takes these and the current index  $j$ , and produces the next index for which this PE is responsible. Also notice that the L operators in the  $i$ -loop are now DIST-L operators.

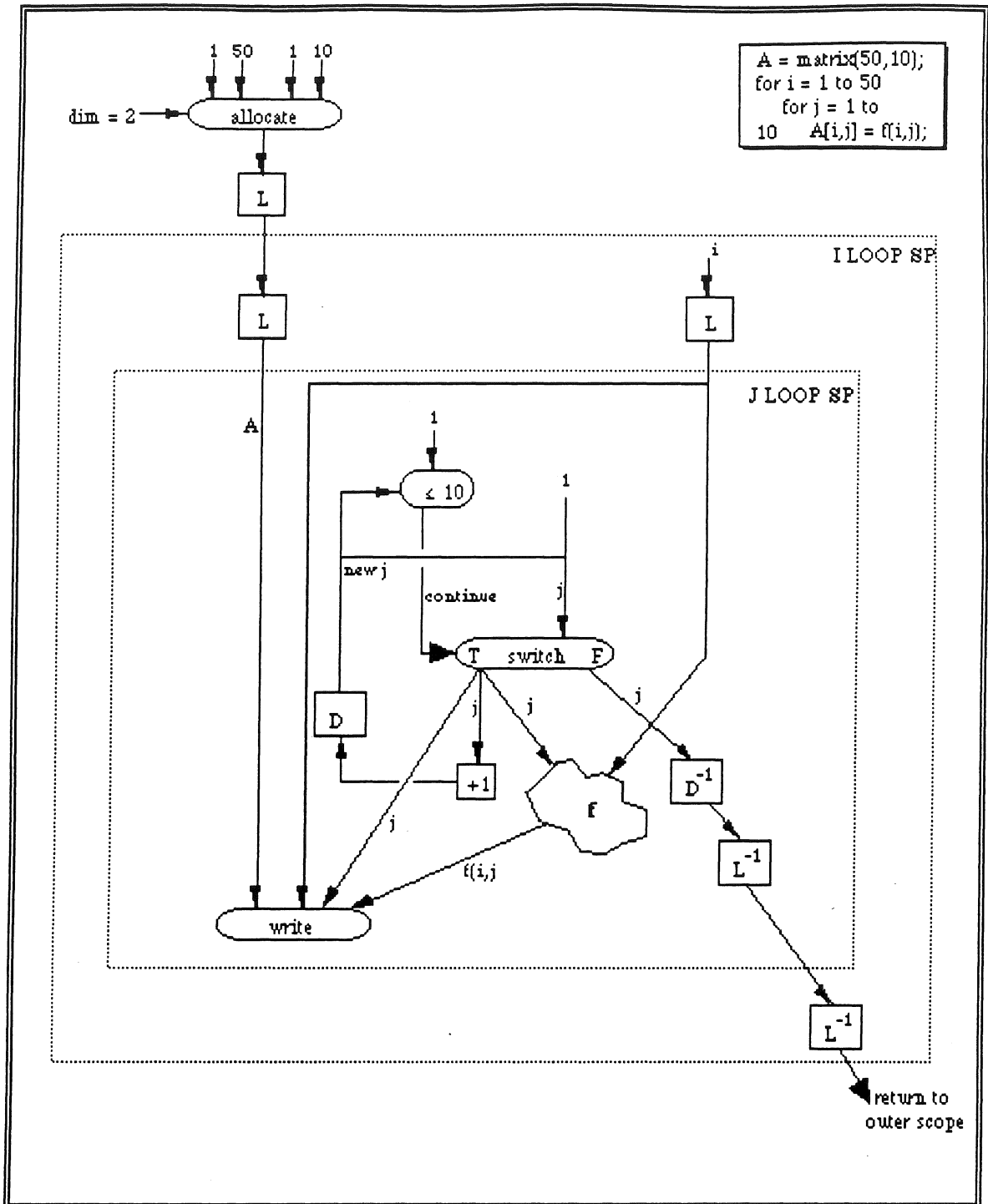


FIGURE 2.22. SIMPLE 2-D ARRAY FILL.

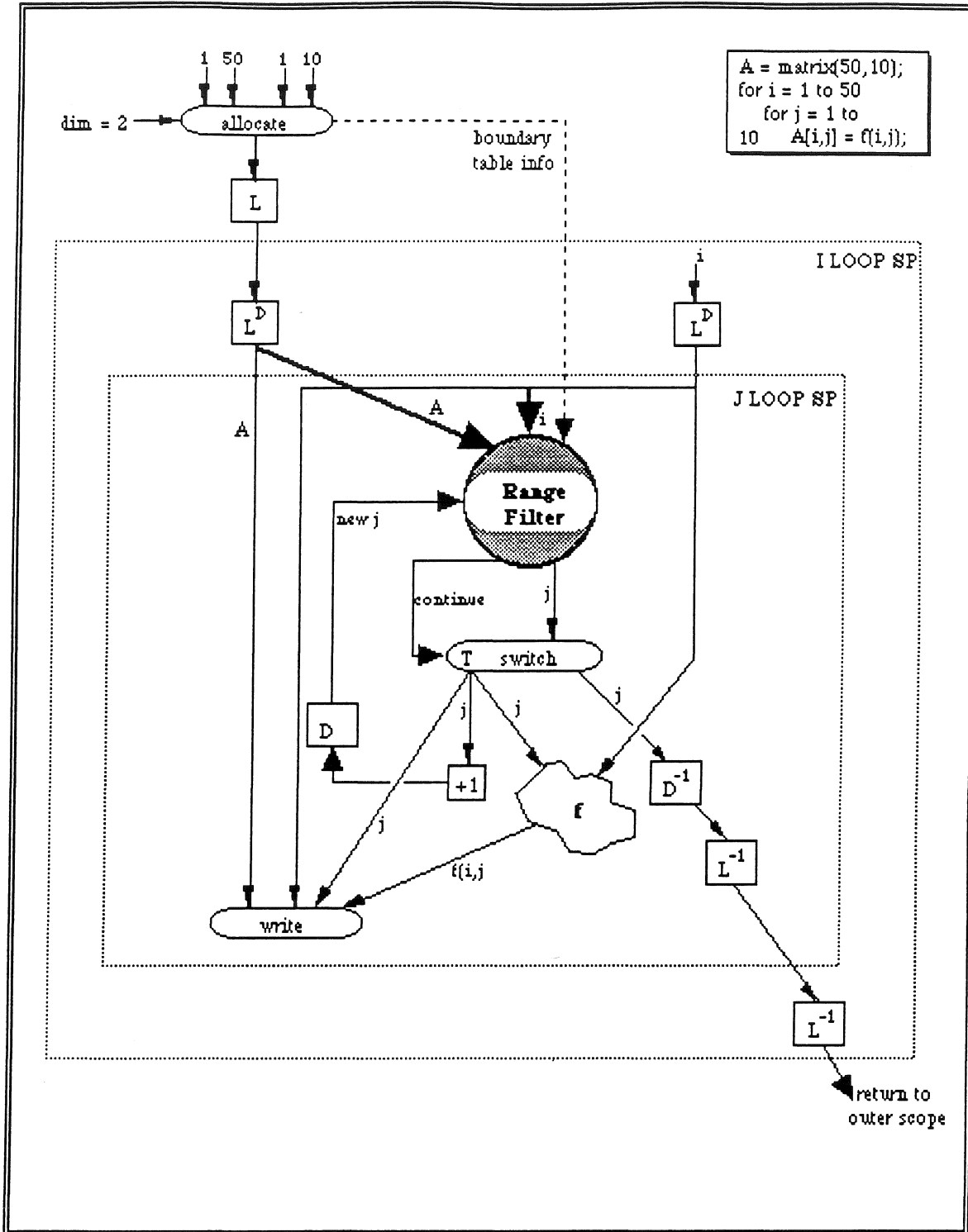


FIGURE 2.23. 2-D ARRAY FILL WITH RANGE FILTER.

### Boundary Table

Boundary tables are generated at allocation time and referenced by the range filter to determine the boundaries of its area-of-responsibility. In PODS, grouped ranges are used because they generate fewer superpage boundaries than interleaved ranges in general.

In the table below, an array header for PE #1 with a 8 x 4 array (page size of 6) is shown. The values *beginning\_rangeX\_dim1* and *beginning\_rangeX\_dim2* are the beginning values for a given range interval in each of the two dimensions; similarly for *ending\_rangeX\_dim1* and *ending\_rangeX\_dim2*. A range interval is the area-of-responsibility for a given PE and in a given dimension; there is one range interval for each entry in a boundary table. For example, range interval 1 runs from 1 to 1 in the *i* direction, and from 1 to 4 in the *j* direction.

Field Name	Value
beginning_offset	1
ending_offset	6
number_of_dimensions	2
size_dim1	8
size_dim2	4
ELEMENT_SPACE	space allocated for this array on this PE
<b>beginning_range1_dim1</b>	<b>1</b>
<b>ending_range1_dim1</b>	<b>1</b>
<b>beginning_range1_dim2</b>	<b>1</b>
<b>ending_range1_dim2</b>	<b>4</b>
<b>beginning_range1_dim1</b>	<b>2</b>
<b>ending_range1_dim1</b>	<b>2</b>
<b>beginning_range1_dim2</b>	<b>1</b>
<b>ending_range1_dim2</b>	<b>2</b>
NULL	

TABLE 2.3. EXAMPLE BOUNDARY TABLE FOR A GIVEN PE.

The boundary table fields are bolded. For different numbers of PEs (four in this example) different distributions are produced. The page size comes into play because pages are used in caching and remote accesses. In this example the page size of 6 splits the array into a non-rectangular area for PE #1.



### Master Array

In Figures 2.22 and 2.23 above only one array is being written into inside the loop. However there can be more than one. In PODS, only one array, the *master* array, controls the partitioning for that loop. Currently the first array written into is chosen as the master array. Later on a more intelligent algorithm could be used, but this approach has produced acceptable results.

### Algorithm

The algorithm for the range filter is fairly straight forward. It is important to note, however, that the general algorithm is parameterized. The general algorithm functions by repeatedly extracting range intervals from the array boundary table. While within the range, the filter passes indices for elements within that range. The filter also keeps the loop alive by sending a continue token to the loop switch until all ranges have been exhausted. In the figure below, *m* is just some variable used to count the intervals; *i* and *j* are the loops indices, and *continue* is the signal to the loop body telling it whether to continue or not.

There are three new PODS instructions required to implement a RANGE\_FILTER: INTERVAL\_COUNT (retrieve the number of range intervals for this array); and B\_RANGE /E\_RANGE (retrieve the beginning and ending values for the specified range interval). These new instructions simply read entries from the array header (generated at allocation time). With RANGE\_FILTER, each PE has the same code; only the local boundary tables are different.

```

1  m = interval count of master array
2  if m < 0 then exit
3  if (ci*i+ki) is not in interval m then decrement m and
   goto 2
4  set j to the minimum of the loop end and (end of the
   interval-kj)/cj
5  if (cj*j+kj) is not in the interval or the first element
   of this dimension is not owned then decrement m and goto
   2
6  if j is within the loop bounds then set continue to TRUE
   and send j and continue into the loop body
   else decrement m and goto 2
7  if continue is TRUE do the loop body else goto 10
8  true part of loop body
9  if new_j is within loop bounds set continue to TRUE,
   send j and continue into the loop body, and goto 5
   else set continue to FALSE, send j and continue into the
   loop body, and goto 7 (with j set to new_j)
10 false part of loop body

```

FIGURE 2.24. ALGORITHM FOR SECOND LEVEL, DESCENDING RANGE FILTER FOR  $A[C_I * I + K_I, C_J * J + K_J]$ .

The algorithm shown in Figure 2.24 is for a descending loop with a stepsize of 1 writing into array  $A[c_i * i + k_i, c_j * j + k_j]$ . Array A is the master array in this case. Step #7 above is the SWITCH which between the true and false parts of the loop body.

For different levels of distribution (distribute the first level of nested loop vs. other levels) or directions (ascending vs. descending), different range filters are used, see Appendix A: Range Filter Algorithms. The selection of the algorithm is done at compile time, so no more run-time overhead is used than necessary.

In the case where the distribution is done a level above the lowest level, the RANGE\_FILTER checks only the first element in a range interval to see if that element belongs to it. This prevents other PEs with range intervals in the same index (e.g., PEs #1 & #2 for  $i = 2$  below) from both trying to execute a particular iteration. The figure below shows the partitioning for a 8 x 4 array, page size of 6 (same as the boundary table example above).

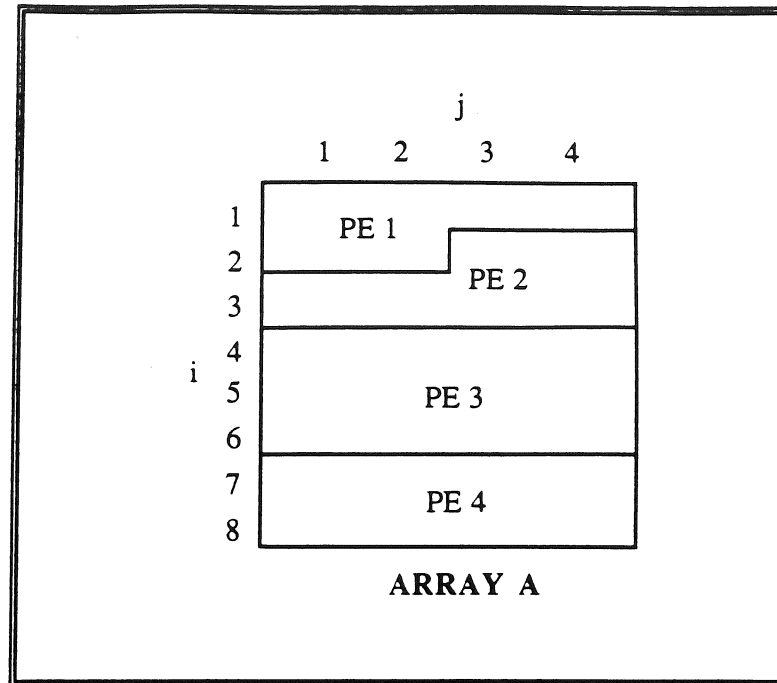


FIGURE 2.25. NON-RECTANGULAR ARRAY PARTITIONING EXAMPLE.

If the RANGE\_FILTER were at the  $i$  level, then each PE would be responsible for distinct rows of  $i$ , i.e. PE #1 has rows 1 and 2, PE #2 has only row 3, PE #3 has rows 4, 5, and 6, and finally PE #4 has rows 6 and 7.

### 2.5.3. LCD Effects

LCDs have a major affect on the policies for code distribution. This section discusses those effects.

If a for-loop performs a reduction it will have LCDs. If the for-loop fills an array it *may* have loop-carried dependencies. These LCDs prevent iterations from running in parallel. In PODS these LCD for-loops are executed in place just as they would on a sequential processor. This is the case where PODS degenerates into a sequential machine for the sequential code. The reason for this is the extreme cost of communication on distributed memory machines.

For distributed memory MIMD machines the ratio of the communication time to execution time can be as great as 400, as in the iPSC/2 [iPSC89]. This means that the LCD distance,  $D$ , times the number of overlapping instructions,  $N$ , must be at least 400. i.e.

$$D * N \geq \text{communication time} / \text{execution time.}$$

A distance 4 LCD means that iteration  $i$  must wait for iteration  $i-4$ . In order to see this better, consider a loop body with 100 overlapping instructions. If  $D$  is less than 4 then it is better to execute the for-loop on one PE rather than distribute the loop. If it were to be distributed, the for-loop iterations would be grouped and assigned to PEs via DDE. For example, iterations 1 through 4 to PE #1, 5 through 8 to PE #2, etc. For-loops with larger LCD distances or larger instruction overlap may be able to perform better when distributed, this is a current topic of research.

In order to see how communication delay and overlapping execution interact, consider the Figure 2.26. In the first cast (Non-Distributed) the loop is executed on one PE, causing no communication delay. The second case (Distributed with Fast Communication) performs the best. Its completion time (indicated by the dark horizontal lines) is the earliest of the three. Notice how the amount overlapping instructions must be comparable to the delay time for any benefit to occur. The third and final case (Distributed with Slow Communication) shows what would happen if loops with LCDs were distributed when communication is costly, e.g. the iPSC/2.

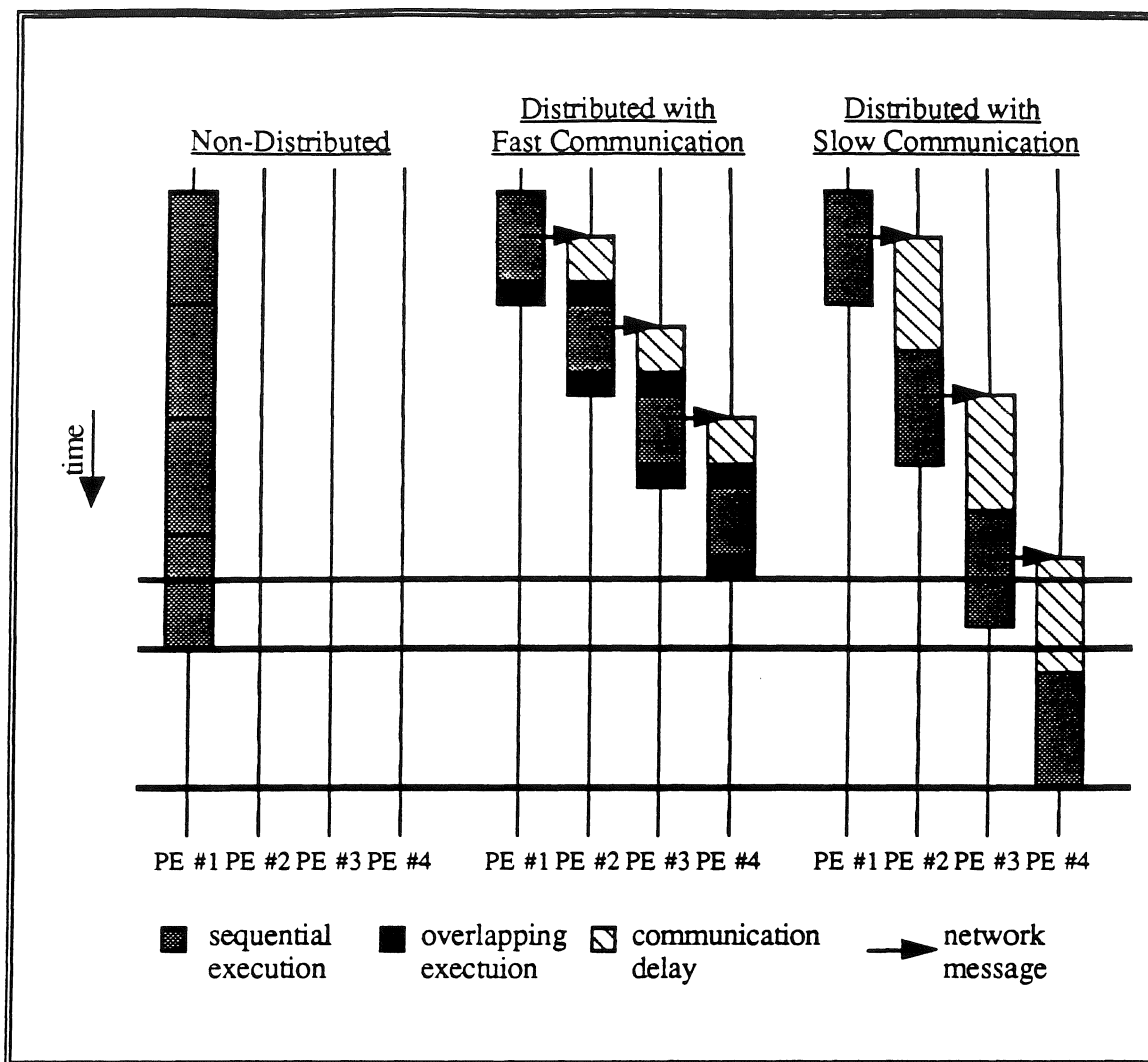


FIGURE 2.26. EFFECTS OF COMMUNICATION SPEED ON OVERLAPPING ITERATIONS.

Considering the abstract case shown in Figure 2.20, given LCDs in all  $i_1$  through  $i_k$ , PODS distributes  $i_{k+1}$ . To understand why this is better, consider the three different shapes of arrays: rectangular, long and narrow, and short and wide. When the array is rectangular, like in Figure 2.20 (e) and (f),  $PE_j$  and  $PE_{j+1}$  can pipeline. This overlapping execution will increase as the work at each element increases. If there is very little work at each element then it is run sequentially. Note that this usually only occurs when  $i_{k+1}$  is the innermost level of the nested loop. When the array is long and narrow, the same rules apply except

even more work is needed at each element for distribution to show a gain. Finally, if the array is short and wide, like Figure 2.20 (b) and (c), multiple wavefronts occur thus providing some parallelism.

So, for the scope of this discussion, the communication costs of distributed memory architectures is too high for for-loops with LCDs to be distributed. The communication cost overwhelms any efficiency gains from the overlapping iterations. Thus PODS does not distribute loops with LCDs. One outcome from this is that all distributed loops are array filling loops.

Based on the above, PODS needs a compiler which will reduce the number of LCDs so that the loops can be distributed. Scalar expansion is one optimization which does this. Any state-of-the-art vectorizing compiler will have (see Padua and Wolfe [P&W86]) scalar expansion. Since the ID Nouveau compiler is not intended for a machine which is aided by scalar expansion (GITA), it does not scalar expand. PODS, on the other hand, is aided by scalar expansion and would have this and other optimizations (e.g., loop interchange, and loop fission).

PODS also needs a LCD Detector, which will detect when LCDs occur. The LCD Detector performs two major phases. The first phase finds the loop bodies and the second traces these looking for array writes (I-structure STOREs) which use values from the same array (I-structure FETCHs). The first phase performs the following steps:

1. Find all D operators.
2. Search backward from each D until the same D or another D is found. Do not search beyond the SWITCH operator.
3. If found D is the same, then the path search forms the loop variable path, else it forms the loop body path.

The loop bodies are now traced using the following:

1. Find all I-structure STORE operators.
2. Trace up the data dependency arcs from each to find all I-structure FETCH operators which feed this I-structure STORE.
3. Trace up the data dependency arcs from each index input to find the source the index value.
4. If any I-structure FETCH uses the same array as the I-structure STORE and their index input paths differ from each other, then there is a LCD.

#### **2.5.4. Remote Array Accesses**

Remote array accesses will occur due to the distribution of array data. No new reads nor writes are added to the original ID program. Remote reads and writes are discussed in this section. Array caching affects remote reads, but this is not part of the model and therefore discussed in Chapter 3, PODS Logical Architecture. Also in Chapter 3 is a discussion of the Array Manager which actually performs these operations.

##### **Remote Reads**

As a simple example of remote array reading, consider a multiprocessor with 4 PEs. Using a page size of 32 elements, and 3 arrays A, B, and C, each of size 100. PE 0, 1, and 2 will each contain a single page of each array. PE 3 will contain a partial page (4 elements) of each array. Consider the following loop:

```
For i <- 1 To 100 Do
{
  A[i] = B[101-i] + C[i]
}
```

FIGURE 2.27. REMOTE READ CODE EXAMPLE.

All four processors begin executing simultaneously—PE 0 fills A(1..32), PE 1 fills A(33..64), PE 2 fills A(65..96), and PE 3 fills A(97..100). Note that for most of the loop, each processor must access elements of array B that lie on a different processor than the executing processor. Each one of these remote accesses entails a transfer of data from the producing PE to the consuming PE, an operation that is relatively expensive on all current distributed memory multiprocessors. It will never be possible to remove the need for remote accesses from distributed computations, so PODS must instead use a technique for diminishing their effect on the overall computation time. The technique PODS uses is called *remote access caching*.

Remote access caching takes advantage of the fact that in PODS, no array element may be written to multiple times. As a result of this, PEs can cache data that has been recently accessed without considering cache coherency problems. In the partitioning scheme defined above, each PE contains some number of pages of each array. To accomplish remote access caching, PODS defines a remote access as the retrieval and local storing of the entire page containing the remote datum. That is, when a particular element is fetched from a remote PE, the entire page containing that element is sent back to the requesting PE when the requested element becomes available. Due to locality of reference in many algorithms, it is likely that the same PE will need another element from that page in the near future, so if the cache is checked first a remote access will often be avoided. Of course, if the next requested element was not available at the time the page was cached, then another remote access, transferring the same page, will be required. Note that the term "cache"



used here does not refer to a specialized hardware device, used to reduce access time to main memory. Rather, it is a "software cache" used to reduce access time to remote memory modules.

### Remote Writes

In the previous discussion, it was mentioned that occasionally the program structure makes remote writes unavoidable. A remote write is an array write where the data collector is on a different PE than data storage. When this occurs a message must be sent to the remote PE with the value, the array ID, and the indices. As an example of why this might happen, examine the following code segment:

```
For i <- 1 To 100 Do
{
  A[i] = B[i]
  C[i+10] = B[i+5] * 2
}
```

FIGURE 2.28. REMOTE WRITE CODE EXAMPLE.

To see why the PODS data partitioning method causes remote writes in this case, consider that a write to C may occur at a location not owned by the PE executing the loop. For example, suppose  $i$  is 25. PE #0 is responsible for writing A(25), however PE #1 is responsible for writing C(35). Without loop fission, it is necessary either for PE #0 to remotely write to C(35) or for PE #1 to remotely write to A(25). This is not a single assignment violation, but it *is* inefficient. In this case loop fission could solve the problem, however, in general, there is no simple solution. To avoid using different mapping functions for A and C, PODS allows remote writes instead.

Remote writes are also necessary for another reason. In ideal circumstances all data is written to locally, but program structure can sometimes cause remote array writes. Note

that it is not always possible to determine, at compile time, which element is being updated by an assignment statement. Consider the loop below:

```

For k <- 1 To n Do
{
  A[f(k)] = B[g(k)]
}

```

FIGURE 2.29. IMPOSSIBLE COLLECTOR WRITES.

The functions  $f$  and  $g$  make it impossible to determine which element a given  $k$  will be assigning at compile time. In this case each PE must calculate  $f(k)$  for all  $k$ 's to determine if that element of  $A$  is inside its area-of-responsibility. It should also be noted that arrays are single assignment and that the  $f(k)$  must be well behaved (one-to-one) over the range of  $k$ , otherwise a single assignment run-time error will occur.

#### 2.5.5. For-Loop Distribution Algorithm

Now that DDE has been introduced, the effects of LCDs have been discussed, and the mechanisms for distribution have been explained, the actual for-loop distribution algorithm can be presented.

There are three primary mechanisms for achieving distribution. The data distribution mechanism (ALLOCATE operator) has already been discussed. For PODS to distribute SPs they need to be spawned on multiple PEs. It is the DIST-L operator which performs this. When PODS determines that a certain level of a nested loop is to be distributed, its parent SP gets DIST-L operators, and it gets the third primary mechanism: the RANGE\_FILTER.

At compile time the program is analyzed to determine which for-loops will be distributed. Those for-loops which are distributed will be augmented with RANGE\_FILTER code. The task of the RANGE\_FILTER is to produce only those loop variables which make the array

accesses local. At load time, each PE will be given a copy of the entire program (all PEs are homogeneous). At run-time tokens are sent to different PEs to start execution of a particular for-loop SP.

Since arrays are partitioned row-major the code will be row-major as well. If it is not row-major it will run, but inefficiencies will occur. In order to efficiently execute a row-major nested loop the outer-most level of the nest should be distributed. This reduces communication cost and context switching, and allows the array caching to operate efficiently. Given these observations and the previous principles, the algorithm for-loop distribution determination is as presented below.

#### **Algorithm: Loop Distribution**

1. Starting with the outer-most code-block, repeat the following until all sets of nested loops are marked (depth-first traversal) as either distributed or local .
  - a. Consider the next inner code-block. If this code-block does not have an LCDs, then mark it and all descendent SPs will be local.
  - b. If this inner SPs has a LCD, then goto step 2.
  - c. If this is the inner most SPs , then consider the next unmarked SPs (depth-first) and goto step 2.
2. In each marked SP a range filter replaces the predicate.
3. In the parent of each marked SP the L operators are changed into DIST\_L operators.

The outer-most SPs of an entire program cannot be distributed. This is because every program must start somewhere; i.e., there is a single first instruction in every program. If it is desirable, due to LCDs, to distribute this outer-most SP, then a dummy SP is set up to drive the original SP.

### 2.5.6. Examples

#### LCD Examples

In a two level nested loop there are four basic cases which involve LCDs: (1) no LCDs in either  $i$  nor  $j$ ; (2) LCD in  $i$ ; (3) LCD in  $j$ ; and (4) LCDs in both. PODS handles each of these cases efficiently.

In the following examples the same array and filling loop will be used, however the filling function (FUNC) will be changed to add or subtract LCDs as necessary. Consider a simple nested loop which fills an 8 x 4 array A.

```

For i <- 1 To 8 Do
  For j <- 1 To 4 Do
  {
    A[i, j] = FUNC(x)
  }

```

FIGURE 2.30. SIMPLE ARRAY FILLING EXAMPLE CODE.

For the above code there would be two SPs, one for the  $i$  loop and one for the  $j$  loop. Since there are no LCDs, either level can be distributed. Assume the array is partitioned as shown in Figure 2.31 below, and assume that the communication delay is a short five time units, and that a context switch is one time unit.

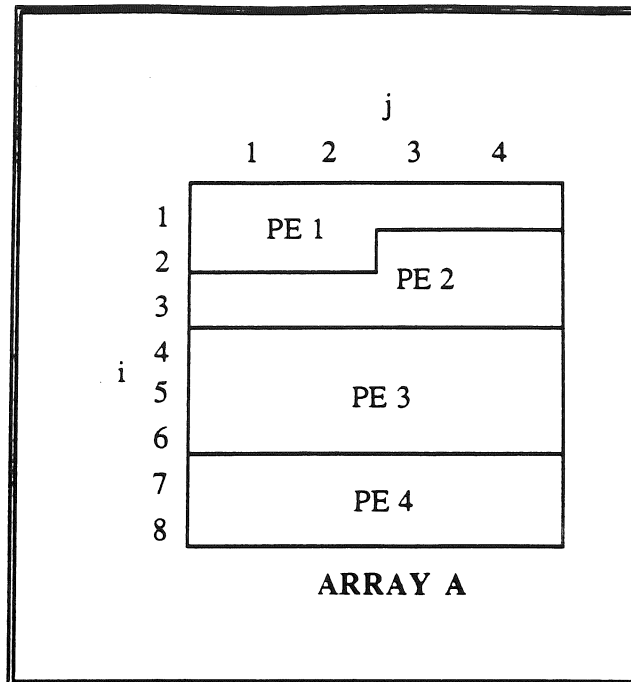


FIGURE 2.31. SIMPLE ROW-MAJOR ARRAY PARTITIONING.

### Case 1: No LCDs

FUNC has no LCDs, e.g.  $A[i,j] = B[i,j]$ . If  $i$  were distributed the execution would be as shown in Table 2.4. The pairs of numbers in the table show when  $A[i,j]$  is being written; this only occurs in the  $j$  SP. The operations in italics are for the  $i$  SP. This assume PE 1 starts out generating the  $i$ 's needed and then broadcasts them to all of the PEs (including itself). When a PE gets an  $i$  value it starts the  $j$  SP. There are times when there is nothing in this PEs area-of-responsibility; thus the for  $i=x : \emptyset$ . The **initial bc** comes from the parent SP which contains the DIST\_L operators, this is how all of the initial parameters get broadcast. Note that  $i$  is not broadcast in this case.

time	PE 1	PE 2	PE 3	PE 4
0	initial bc			
1	<i>context sw</i>	comm delay	comm delay	comm delay
2	<i>gen i=1</i>	comm delay	comm delay	comm delay
3	<i>gen i=2</i>	comm delay	comm delay	comm delay
4	context sw	<i>gen i=3</i>	<i>gen i=4</i>	<i>gen i=7</i>
5	1, 1	context sw	<i>gen i=5</i>	<i>gen i=8</i>
6	1, 2	3, 1	<i>gen i=6</i>	context sw
7	1, 3	3, 2	context sw	7, 1
8	1, 4	3, 3	4, 1	7, 2
9	context sw	3, 4	4, 2	7, 3
10	2, 1		4, 3	7, 4
11	2, 2		4, 4	context sw
12	2, 3		context sw	8, 1
13	2, 4		5, 1	8, 2
14			5, 2	8, 3
15			5, 3	8, 4
16			5, 4	
17			context sw	
18			6, 1	
19			6, 2	
20			6, 3	
21			6, 4	

TABLE 2.4. EFFECTS OF OUTER LOOP DISTRIBUTION WITH NO LCDS.

Notice that PE #1 will take over elements 2,3 and 2,4 as the iteration space partitioning extends the area-of-responsibility based upon the first element. The communication delays will overlap with the operations and context switches so that the multiple *i*-loop communication delays do not delay the execution multiple times. Now if *j* were distributed the execution would be as follows in Table 2.5. The **parent sp** does not have DIST\_L operators like above, it has regular L operators (which do not broadcast). Here the *i*'s are broadcast from the *i* SP (in *italics*).

time	PE 1	PE 2	PE 3	PE 4
0	parent sp			
1	context sw			
2	gen i=1			
3	gen i=2	comm delay	comm delay	comm delay
4	gen i=3	comm delay	comm delay	comm delay
5	gen i=4	comm delay	comm delay	comm delay
6	gen i=5	for i=1: Ø	for i=1: Ø	for i=1: Ø
7	gen i=6	context sw	context sw	context sw
8	gen i=7	2,3	for i=2: Ø	for i=2: Ø
9	gen i=8	2,4	context sw	context sw
10	1, 1	context sw	for i=3: Ø	for i=3: Ø
11	1, 2	3, 1	context sw	context sw
12	1, 3	3, 2	4, 1	for i=4: Ø
13	1, 4	3, 3	4, 2	context sw
14	context sw	3, 4	4, 3	for i=5: Ø
15	2, 1	context sw	4, 4	context sw
16	2, 2	for i=4: Ø	context sw	for i=6: Ø
17	context sw	context sw	5, 1	context sw
18	for i=3: Ø	for i=5: Ø	5, 2	7, 1
19	context sw	context sw	5, 3	7, 2
20	for i=4: Ø	for i=6: Ø	5, 4	7, 3
21	context sw	context sw	context sw	7, 4
22	for i=5: Ø	for i=7: Ø	6, 1	context sw
23	context sw	context sw	6, 2	8, 1
24	for i=6: Ø	for i=8: Ø	6, 3	8, 2
25	context sw	context sw	6, 4	8, 3
26	for i=7: Ø		context sw	8, 4
27	context sw		for i=7: Ø	
28	for i=8: Ø		context sw	
29			for i=8: Ø	

TABLE 2.5. EFFECTS OF INNER LOOP DISTRIBUTION WITH NO LCDS.

Note that every PE is doing something, thus distributing additional levels of the nest would do nothing to speed up execution. In this case the  $j$  loop distribution must wait for each  $j$  to be generated. After the initial communication delays each PEs will start checking the  $i$  values they receive. If  $i$  is in the range (as determined by the RANGE\_FILTER ) then  $j$  values will be generated, if not, the SP completes. This example graphically shows that outer level distribution is better than inner level (execution time of 21 vs. 29), as described in Section 2.6.1 above.

**Case 2: LCD in  $i$** 

FUNC uses  $i$  in such a way that there is a LCD, e.g.  $A[i,j] = A[i-1, j]$ . In this case PODS would not allow  $i$  to be distributed, and the RANGE\_FILTER would go in the  $j$ th level (i.e. be distributed). As in Case 1 when  $j$  was distributed, the iterations must wait for  $i$  to be generated. Since the LCD is in  $i$  the loop would execute as shown in Table 2.6 (execution time of 45). Note that Table 2.6 obeys the LCD restriction on  $i$ . The block's in Table 2.6 mean that the necessary array elements have not yet been written.



time	PE 1	PE 2	PE 3	PE 4
0	parent sp			
1	context sw			
2	gen i=1			
3	gen i=2	comm delay	comm delay	comm delay
4	gen i=3	comm delay	comm delay	comm delay
5	gen i=4	comm delay	comm delay	comm delay
6	gen i=5	for i=1: Ø	for i=1: Ø	for i=1: Ø
7	gen i=6	context sw	context sw	context sw
8	gen i=7	block	for i=2: Ø	for i=2: Ø
9	gen i=8	block	context sw	context sw
10	1, 1	block	for i=3: Ø	for i=3: Ø
11	1, 2	block	context sw	context sw
12	1, 3	block	block	for i=4: Ø
13	1, 4	comm delay	block	context sw
14	context sw	comm delay	block	for i=5: Ø
15	2, 1	comm delay	block	context sw
16	2, 2	2, 3	block	for i=6: Ø
17	context sw	2, 4	block	context sw
18	for i=4: Ø	context sw	block	block
19	context sw	3, 1	block	block
20	for i=5: Ø	3, 2	comm delay	block
21	context sw	3, 3	comm delay	block
22	for i=6: Ø	3, 4	comm delay	block
23	context sw	context sw	4, 1	block
24	for i=7: Ø	for i=4: Ø	4, 2	block
25	context sw	context sw	4, 3	block
26	for i=8: Ø	for i=5: Ø	4, 4	block
27		context sw	context sw	block
28		for i=6: Ø	5, 1	block
29		context sw	5, 2	block
30		for i=7: Ø	5, 3	block
31		context sw	5, 4	block
32		for i=8: Ø	context sw	block
33			6, 1	block
34			6, 2	comm delay
35			6, 3	comm delay
36			6, 4	comm delay
37			context sw	7, 1
38			for i=7: Ø	7, 2
39			context sw	7, 3
40			for i=8: Ø	7, 4
41				context sw
42				8, 1
43				8, 2
44				8, 3
45				8, 4

TABLE 2.6. EFFECTS OF INNER LOOP DISTRIBUTION WITH LCDS.

Comparing this to Table 2.7 below, the execution time is shorter with inner loop distribution than with no distribution (sequentially). As the work per element grows or the array dimensions increase, this advantage will grow.

### Case 3: LCD in $j$

FUNC uses  $j$  in such a way that there is a LCD, e.g.  $A[i,j] = A[i, j-1]$ . In this case PODS would distribute  $i$ , and the RANGE\_FILTER would go in the  $i$ th level. As in Case 1 when  $i$  was distributed, the iterations will run in parallel right away. And since the LCD is in  $j$  the loop would execute exactly like the first part of Case 1 (execution time of 21). Note that Table 2.4 obeys the LCD restriction on  $j$ .

### Case 4: LCD in $i$ and $j$

In this case FUNC would be something like  $A[i,j] = A[i-1, j-1]$ . Since there are LCDs in each level, PODS would not distribute this loop at all and the execution would be as shown below in Table 2.7 (total execution time of 49). The load balance in this case is also very poor.

time	PE 1	PE 2	PE 3	PE 4
0	parent sp			
1	context sw			
2	gen i=1			
3	gen i=2			
4	gen i=3			
5	gen i=4			
6	gen i=5			
7	gen i=6			
8	gen i=7			
9	gen i=8			
10	context sw			
11	1, 1			
12	1, 2			
13	1, 3			
14	1, 4			
15	context sw			
16	2, 1			
17	2, 2			
18	2, 3			
19	2, 4			
20	context sw			
...	...			
43	7, 3			
44	7, 4			
45	context sw			
46	8, 1			
47	8, 2			
48	8, 3			
49	8, 4			

TABLE 2.7. EFFECTS OF NO DISTRIBUTION DUE TO LCDS.

It is interesting to note that Cases 2 and 3 are still executed in parallel by PODS even with the LCDs. In general Case 2 could generate multiple diagonal wavefronts, while Case 1 would execute with a horizontal sweep. The diagrams below illustrate the different execution patterns. The numbers in each cell are the time each cell would be filled. By tracing lines through equal times the wavefront can be seen.

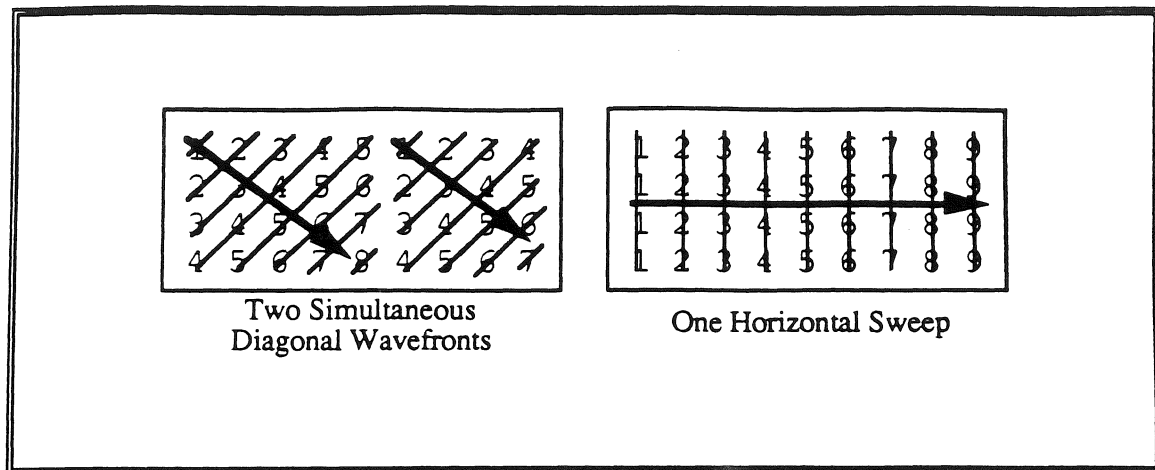


FIGURE 2.32. LCD EXECUTION WAVEFRONTS.

### Matrix Multiply

To better understand the distribution algorithm reconsider the Matrix Multiply code shown previously in Figure 2.15. There are three code-blocks in this function which turn into SPs: one for  $i$ -loop (MM-0); one for  $j$ -loop (MM-1); and one for  $k$ -loop (MM-2). This function has no LCDs in the  $i$  or  $j$  loops, only in the  $k$ -loop. Using the loop distribution determination algorithm above, the outer-most code-block (the  $i$ -loop) cannot be distributed without setting up a dummy parent. The next inner code-block (the  $j$ -loop) has no LCD and will thus be distributed. All descendent code-blocks (only the  $k$ -loop) will be local.

The files below are the exact inputs that were used to run Matrix Multiply on the PODS Simulator.

MM-0		#args	args (value, port)	dest [i, p]	route (r) {c}
#	opcode				
0	PROMPT	0		-> [12,0]	{B}
1	PROMPT	0		-> [13,0]	[7,0] [2,0] [5,0]
				[4,0]	[3,0] {A}
2	UPPER_BOUND	2	(0.00,1)	-> [6,2]	[9,1] [11,0]
3	LOWER_BOUND	2	(1.00,1)	-> [6,3]	[14,0]
4	UPPER_BOUND	2	(1.00,1)	-> [6,4]	[15,0]
5	LOWER_BOUND	2	(0.00,1)	-> [6,1]	[16,0]
6	ALLOCATE	5	(2.00,0)	-> [8,0]	
7	LOWER_BOUND	2	(0.00,1)	-> [9,0]	[10,1]
8	FORKJUMP	2	(1.00,1)	-> [17,0]	
9	LE	2	(STKY,1)	-> [10,0]	
10	SWITCH	5	(1.00,2) (11.00,3) (2.00,4)	-> [18,0]	[19,0] [21,0] {I}
11	DIST_LOPERATOR	1	(STKY,0)	-> (12)	
12	DIST_LOPERATOR	1	(STKY,0)	-> (14)	
13	DIST_LOPERATOR	1	(STKY,0)	-> (15)	
14	DIST_LOPERATOR	1	(STKY,0)	-> (10)	
15	DIST_LOPERATOR	1	(STKY,0)	-> (11)	
16	DIST_LOPERATOR	1	(STKY,0)	-> (13)	
17	DIST_LOPERATOR	1	(STKY,0)	-> (16)	
18	DIST_LOPERATOR	1		-> (1)	
19	PLUS	2	(1.00,1)	-> [20,0]	{NEXT-I}
20	D	2	(-11.00,1)	-> [9,0]	[10,1] {I}
21	DINV	1		->	

In SP MM-0 the PROMPT instructions acquire the A and B matrices used in the Matrix Multiply. The UPPER\_BOUND and LOWER\_BOUND instructions access the array headers to setup the loop boundaries. ALLOCATE then remotely distributes the C array and feeds a FORKJUMP operator. This FORKJUMP is necessary for the array manager to have a place to return the array identifier it just allocated. The LE, SWITCH, PLUS, D, and DINV are the standard dataflow operators. The new PODS operator is the DIST\_LOPERATOR, which performs the standard L operator dataflow operations, but also sends its tokens to *all* PEs. This is how *i* gets distributed.

In SP MM-1 below, there is the local equivalent of the DIST\_LOPERATOR, the LOCAL\_LOPERATOR, which sends its tokens only to itself. LOCAL\_LOPERATORs are only used when the operations have already been distributed, and more distribution would just create network overhead without additional parallelism. MM-1 also has a range filter inserted into it, from instruction 0 to 18 and 29 to 30.

MM-1		#args	args (value, port)	dest [i, p]	route (r) {c}
0	INTERVAL_COUNT	1	(STKY, 0)	->	[1, 1]
1	LT	2	(0.00, 0) (STKY, 1)	->	[2, 0]
2	SWITCH	5	(0.00, 1) (1.00, 2) (29.00, 3) (3.00, 4)	->	[3, 0] [5, 0] [8, 1] [31, 0]
3	B_RANGE	3	(STKY, 1) (0.00, 2)	->	[4, 1]
4	GE	2	(STKY, 0)	->	[7, 0]
5	E_RANGE	3	(STKY, 1) (0.00, 2)	->	[6, 0]
6	GE	2	(STKY, 1)	->	[7, 1]
7	AND	2		->	[8, 0]
8	SWITCH	5	(1.00, 2) (9.00, 3) (3.00, 4)	->	[9, 0] [10, 0] [16, 1] [17, 0]
9	E_RANGE	3	(STKY, 1) (1.00, 2)	->	[11, 1]
10	B_RANGE	3	(STKY, 1) (1.00, 2)	->	[11, 0] [12, 1]
11	LE	2	(STKY, 1)	->	[12, 0] [16, 0]
12	SWITCH	5	(1.00, 2) (4.00, 3) (3.00, 4)	->	[13, 1] [15, 0] [19, 1]
13	LE	2	(STKY, 0)	->	[14, 0]
14	SWITCH	5	(STKY, 1) (1.00, 2) (-3.00, 3) (0.00, 4)	->	[11, 0] [12, 1]
15	LE	2	(STKY, 1)	->	[16, 0] [19, 0]
16	SWITCH	5	(STKY, 0) (STKY, 1) (3.00, 2) (1.00, 3) (0.00, 4)	->	[17, 0]
17	PLUS	2	(1.00, 1)	->	[18, 0]
18	FORKJUMP	2	(-17.00, 1)	->	[1, 0] [2, 1]
19	SWITCH	5	(1.00, 2) (12.00, 3) (3.00, 4)	->	[20, 0] [26, 0] [27, 3] [31, 0]
{J}					
20	LOCAL_LOPERATOR	1		->	(7)
21	LOCAL_LOPERATOR	1	(STKY, 0)	->	(2)
22	LOCAL_LOPERATOR	1	(STKY, 0)	->	(3)
23	LOCAL_LOPERATOR	1	(STKY, 0)	->	(4)
24	LOCAL_LOPERATOR	1	(STKY, 0)	->	(5)
25	LOCAL_LOPERATOR	1	(STKY, 0)	->	(6)
26	PLUS	2	(1.00, 1)	->	[28, 0] {NEXT-J}
27	WRITE_ARRAY	4	(STKY, 1) (STKY, 2)	->	
28	D	2	(1.00, 1)	->	[11, 0] [12, 1] [29, 1] {J}
29	GE	2	(STKY, 0)	->	[30, 0]
30	SWITCH	5	(0.00, 1) (-11.00, 2) (-19.00, 3)	->	[19, 0]
31	DINV	1		->	

SP MM-2 is a simple local loop which performs a reduction-like operation. MM-2's LCD causes it to be run on one PE and not distributed. The LOCAL\_LINV operator routes the sum (S) back to its parent SP which is on the same PE since it is a local operation. This route uses route list 9 which is programmed into every Routing Unit.

MM-2

#	opcode	#args	args (value, port)	dest [i, p]	route (r) (c)
0	LE	2	(STKY,1)	->	[2,0] [1,0]
1	SWITCH	5	(1.00,2) (1.00,3) (3.00,4)	->	[3,2] [5,1] [4,0] [10,0]
{TRIGGER}					
2	SWITCH	5	(0.00,1) (1.00,2) (8.00,3) (1.00,4)	->	[7,0] [11,0] (S)
3	READ_ARRAY	3	(STKY,0) (STKY,1)	->	[6,0]
4	PLUS	2	(1.00,1)	->	[8,0] (NEXT-K)
5	READ_ARRAY	3	(STKY,0) (STKY,2)	->	[6,1]
6	MULT	2		->	[7,1]
7	PLUS	2		->	[9,0] (NEXT-S)
8	D	2	(1.00,1)	->	[0,0] [1,1] (K)
9	D	2	(-9.00,1)	->	[2,1] (S)
10	DINV	1		->	
11	DINV	1		->	[12,0]
12	LOCAL_LINV	1		->	(9)

The routing file below is the "program" that the Routing Unit follows for sending tokens to different SPs. Notice that route list 9, used by MM-2, sends the sum to MM-1, instruction 27, port 0. Checking MM-1 we see that instruction 27 is the WRITE\_ARRAY instruction which is filling array C.

## DISPLAYING ROUTES

#	destinations [sp, inst, port]
1	-> [1, 25, 0] [1, 27, 2] [1, 4, 0] [1, 6, 1]
2	-> [2, 0, 0] [2, 1, 1]
3	-> [2, 0, 1]
4	-> [2, 5, 0]
5	-> [2, 3, 0]
6	-> [2, 3, 1]
7	-> [2, 5, 2]
9	-> [1, 27, 0]
10	-> [1, 13, 0] [1, 14, 1]
11	-> [1, 15, 1] [1, 29, 0]
12	-> [1, 22, 0]
13	-> [1, 21, 0]
14	-> [1, 23, 0]
15	-> [1, 24, 0]
16	-> [1, 27, 1] [1, 0, 0] [1, 3, 1] [1, 5, 1] [1, 9, 1] [1, 10, 1]

Figure 2.33 illustrates the distribution of the three Matrix Multiply SPs across four PEs. The curved lines represent broadcasts, the straight lines represent execution time, and the bold lines correspond to the comments on the right-hand side of the figure. For this example assume the Matrix Multiply starts out on PE #2. There SP 0 begins execution, and encounters the "ALLOCATE C" instruction. This instruction initiates a broadcast

message to the other PEs. Upon receipt of this message, each PE allocates its portion of the array. Next, SP 0 generates and broadcasts the first value for  $i$ . Note that SP 0 does not have a range generator, thus it will generate *all*  $i$ -indices.

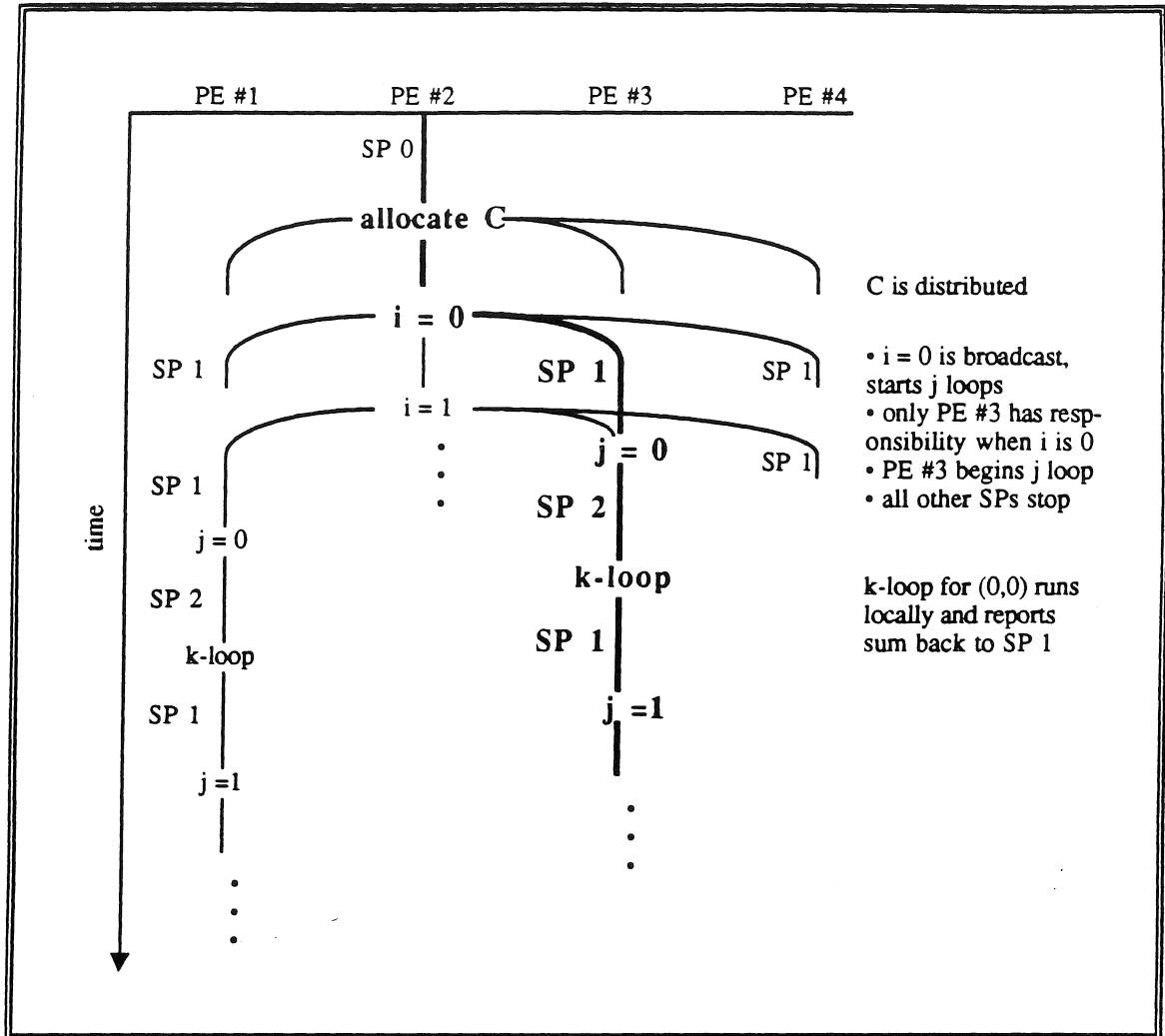


FIGURE 2.33. EXAMPLE EXECUTION TRACE FOR MATRIX MULTIPLY ON 4 PEs.

Each remote PE that receives an activating token (value 0) instantiates SP 1. SP 1 *does* have a range filter, so it will process only those indices for which the current PE is responsible. Thus a number of PEs quickly execute essentially empty SPs because they have no elements for which they are responsible when  $i$  is 0. In this case, PE 3 is the only PE with operations to perform when  $i$  is 0. PE #3 executes SP #1, which spawns the



$k$ -loop locally (the fact that the loop is local was determined at compile time). The  $k$ -loop is a simple loop that generates a vector dot product and returns the result to its parent SP. The  $j$ -loop may now continue with the  $j$  values for which it is responsible when  $i$  is 0.

In parallel with the execution of the first iteration of the  $i$ -loop, the original SP 0 continues generating and broadcasting successive values for  $i$ . This will cause new ready SPs to queue up in remote PEs. As other SPs block waiting for tokens, these new SPs will be selected for execution by the scheduler.

Once the  $k$ -loop starts, it will access remote pages from different PEs as necessary. This is where the existence of the remote access cache becomes important — a large number of reads will access the local array cache rather than causing a remote read.

Thus the SPs are efficiently distributed across the PEs. The distribution of Matrix Multiply across the set of PEs is efficient and uses little overhead.

## 2.6. Functional Distribution

In PODS, functional distribution is not a primary concern. The APPLY operator is used to spawn function calls on a single remote PE, and the INVERSE\_APPLY is used to report any answers. Both of these operators are similar to the original ID operators, as described previously.

PODS distributes functions at run time. Since all communication into and out of a function go through the calling SP, this decision does not have to be broadcast to the other SPs. Functional distribution occurs in two steps: the first step is to determine whether to distribute the function or not, and the second is to determine where to send it to if it is to be distributed.

Currently all function calls in PODS are distributed. In the future the loading of the PE and the size of the function may be used to determine whether functions should be local or distributed.

Once it has been determined that a function will be distributed, where it will be sent to must be decided. In order to randomly distribute the work load the simple hash function below is used to generate the ID of the target PE.

$$\text{Target PE ID} = (\text{iteration} + \text{SP ID} + \text{Calling PE ID}) \bmod (\text{number PEs})$$

This will place different iterations on different PEs; necessary for calls inside of loops. By using the calling PE's ID the same functions called from different PEs will not all end up on the same PEs. Finally, the SP number adds to the randomness, particularly at the beginning of a program.

This approach provides a fairly random distribution, which in turn tends to generate a balanced work load. Given more information, a more complex and possibly better distribution function may be used, but the simple approach achieves acceptable results without wasting interconnect bandwidth in order to maintain global state information.

## 2.7. Deadlock Handling

Once SPs are formed they are checked for deadlock. Deadlock can occur when dynamic arcs are present in such a way the actual instruction execution order depends on the indices used.

Iannucci [Ian88] handles deadlock in such a way that the execution of very small SPs must be efficient. This is not possible on currently available distributed memory MIMD machines. PODS instead produces a partitioning then checks it for deadlock. If it is deadlock-free then it will run efficiently. If it has deadlocks then the programmer is given

the choice to either change the offending code, or have the partitioner split the SP to remove the deadlocks.

PODS uses a combination of deadlock avoidance and detection. In PODS, unnecessary deadlocks occur only when an array read is placed before its array write. In order to understand simple deadlock, consider the SP fragment below. The READ will request an I-structure read and the value would be sent to the '+ 1'. But since the WRITE has not yet occurred (if A[i] already has a value then a single-assignment violation will occur), the '+ 1' will block and will never unblock — causing deadlock.

```

0  reg0 <- read(A, i)
1  reg0 <- reg0 + 1
...
8  reg0 <- somevalue
9  write(reg0, A, i)
...
```

In order to avoid this PODS places array writes before any reads of the same array. This is only limited by the static data dependencies. If  $A[i] = A[i+1] + 1$  (a LCD), then the array read of A[i+1] will be before the array write into A[i]. This is not a problem. In the example above, PODS would avoid the deadlock by ordering the instructions as follows.

```

0  reg0 <- somevalue
1  write(reg0, A, i)
2  reg0 <- read(A, i)
3  reg0 <- reg0 + 1
...
```

However, this will not always work. If another array write to the same array occurs in the same SP then deadlock can occur. Once this has been detected, PODS splits the SP just after the array write to avoid the possible deadlock. This will avoid the deadlock because array writes do not have an output dependency arc. Thus, putting instructions after an array write adds an implicit dependency arc out of the array write. Splitting just after the array write will remove this added arc, thus returning the dataflow graph to its original, deadlock-free state.

Consider the example below. This is the example Iannucci used to describe MDS. What follows is how PODS would handle it.

```

{ a = vector(0,2);
  a[0] = 0;
  a[1] = a[i] + 1;
  a[2] = a[j] - 2;

  in a[1] - a[2];
}

```

FIGURE 2.34. ID NOUVEAU DEADLOCK CODE EXAMPLE.

The unchecked PODS SP would look something like:

```

SP 0
0 write(0, A, 0)
1 reg0 <- read(A, i)
2 reg0 <- reg0 + 1
3 write(reg0, A, 1)
4 reg0 <- read(A, j)
5 reg0 <- reg0 - 2
6 write(reg0, A, 2)
7 reg0 <- read(A, 1)
8 reg1 <- read(A, 2)
9 return(reg0 - reg1)

```

This can cause an unnecessary deadlock if  $i = 2$  and  $j = 0$ . In the code above (with  $i = 2$  and  $j = 0$ ), instruction #2 blocks awaiting the read from instruction #1. This deadlock is unnecessary because a different ordering would not deadlock. By moving the bolded instructions #4 - #6 above to instructions #1 - #3 below,  $i = 2$  and  $j = 0$  would not cause a deadlock. to form another ordering. However, the code below would block on  $i = 0$  and  $j = 1$ , where the code above would not. Both of these orderings cause *unnecessary* deadlocks because they can be removed; a necessary deadlock would occur if  $i = 1$  or  $j = 2$  (see Figure 2.34 above).

```

SP 0
0 write(0, A, 0)
1 reg0 <- read(A, j)
2 reg0 <- reg0 - 2
3 write(reg0, A, 2)
4 reg0 <- read(A, i)
5 reg0 <- reg0 + 1
6 write(reg0, A, 1)
7 reg0 <- read(A, 1)
8 reg1 <- read(A, 2)
9 return(reg0 - reg1)

```

PODS would recognize that there are three array writes to the same array in the same SP.

Therefore, the SP must be split after every write. This will form the following SPs.

```

SP 0
0 write(0, A, 0)

SP 1
0 reg0 <- read(A, i)
1 reg0 <- reg0 + 1
2 write(reg0, A, 1)

SP 2
0 reg0 <- read(A, j)
1 reg0 <- reg0 - 2
2 write(reg0, A, 2)

SP 3
0 reg0 <- read(A, 1)
1 reg1 <- read(A, 2)
2 return(reg0 - reg1)

```

This will remove the dynamic arcs caused by placing instructions after an array write; array writes do not have output dependency arcs. These unnecessary dependency arcs are what cause the deadlock. These types of situations are possible but unlikely. In none of the Livermore Loops, nor Matrix Multiply, nor in any of SIMPLE does code like this occur. Iannucci has designed a completely safe system, however it cannot run efficiently without special purpose hardware. PODS has been designed for the most likely cases (scientific code), but can still operator on the abnormal cases (though not as efficiently as regular code). A detection method more complex than the simple array write test is currently being

investigated. It is based upon the LCD algorithm. This would allow PODS to create even larger SPs.

## CHAPTER 3

### PODS Logical Implementation

This chapter discusses the logical implementation of a Process-Oriented Dataflow System. The logical implementation describes the functional units and their tasks in PODS. The remote array caching scheme is also presented. Once these are covered the logical architecture is examined. Finally, the supporting software suite is presented.

#### 3.1. System Overview

The driving force behind the PODS logical implementation design was the desire to support the programmer with automatic, but efficient, parallelization of his code. To achieve this the logical implementation had to execute the partitioned code with as little global information as possible. Global information is the root cause of many computational bottlenecks. And since PODS is to be used on MIMD machines with relatively slow communications; communications over the network have to be kept to a minimum.

With the above goals in mind, the logical PE design was constrained to contain a conventional von Neumann CPU at its core. The support units would provide additional power to perform specialized tasks. It is envisioned that these unit would be placed on a single circuit board to form a complete PE. Over time the support units changed in number and functionality until the complete set below was finalized.

- Execution Unit — main unit, performs all ALU functions, a standard microprocessor (e.g., Intel 80386).
- Matching Store — support unit, handles matching of incoming tokens.

- Routing Unit — support unit, processes messages between PEs, similar to the Direct-Connect Model in the iPSC/2.
- Array Manager — support unit, handles array manipulation requests and remote caching.
- Memory Manager — support unit, manages SP memory and loads SPs.

In order to produce partitioned code, a system software suite was built. The suite consists of the ID World Compiler, the PODS Translator, the PODS Partitioner, and the PODS Simulator. The ID World Compiler was graciously supplied by MIT [Nik87b] and the other three programs were designed and built here at UC, Irvine. A parallel programmer would write in ID Nouveau, compile the program, run it through the translator, and then the partitioner to produce PODS code. In the future a PODS compiler is envisioned that would replace the first three programs, and would be tailored for a specific MIMD architecture, like the iPSC/2.

The PODS instruction set is designed along the lines outlined in Bic's original paper [Bic87]. It was designed to perform the required tasks (internal and external token passing) as efficiently as possible. Though it was not tailored to a specific von Neumann CPU, the tasks required are not beyond the standard von Neumann CPU.

Underlying all of the instructions is the remote array caching scheme. This is a software caching scheme designed to exploit the locality of reference in most programs. This is critical for slow communication MIMD systems.



### 3.2. Logical PE Architecture

The logical implementation describes the functional units and their tasks necessary in PODS [Roy90]. The design was constrained to contain a conventional von Neumann CPU at its core. The support units would provide additional power to perform specialized tasks. It is envisioned that these unit would be placed on a single circuit board to form a complete PE. This logical implementation is currently being modified to run directly on an iPSC/2. The way in which the tasks are performed is changing, but the tasks are still the same.

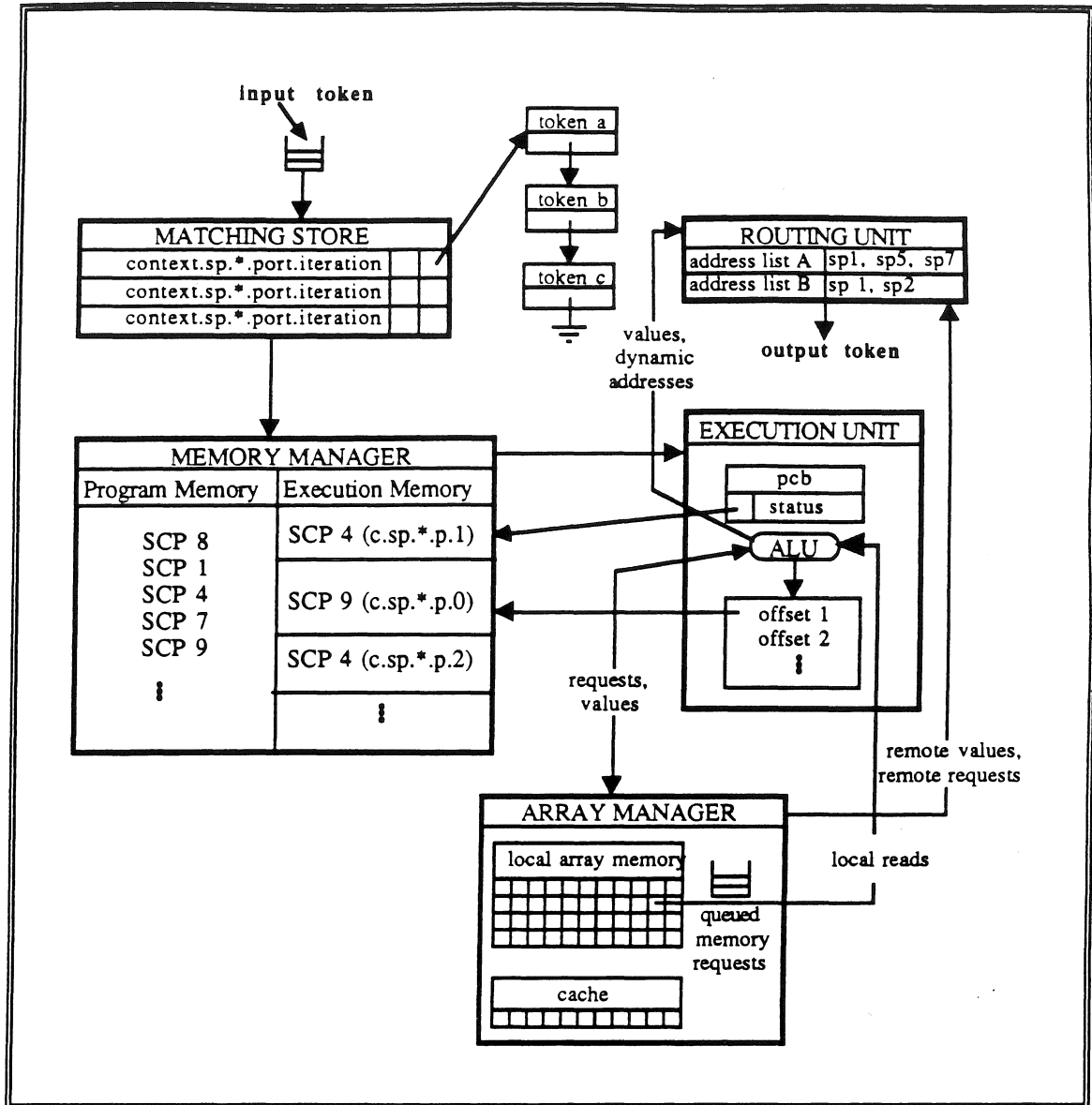


FIGURE 3.1. LOGICAL UNITS OF A PODS PE.

Figure 3.1 shows how the functional units within a PE interact. When an input token arrives it is run through the Matching Store. When the required tokens are present the Memory Manager will load the SP from the Program Memory into Execution Memory. Once in Execution Memory the Execution Unit will begin operating on it as it percolates to the top of the ready list. The key is to keep the Execution Unit operating as much as

possible and to keep the number of context switches to a minimum. In order to support this the Execution Unit calls upon the Array Manager and the Routing Unit to handle specialized tasks.

Each of the tasks of the functional units is explained below.

### **3.2.1. Execution Unit**

The Execution Unit is a simple von Neumann machine which automatically blocks the executing process when a necessary operand is not available. This unit is the most heavily used and is the most complex. PODS is designed such that this unit can be a standard off-the-shelf microprocessor, e.g. Intel 80368. This will allow PODS to make use of advancements in microprocessor technology, e.g. Intel i860.

The Execution Unit uses the state transitions described in Chapter 1. In order to execute one PODS instruction the following tasks need to be performed:

1. check if all operands are available — if not block
2. perform basic instruction to produce value
3. pass value internally to needy instructions
4. if necessary, send message to Routing Unit with route list and value.
5. increment or set program counter as directed by instruction

These steps can easily be performed by an off-the-shelf microprocessor, and many optimizations can be performed. For example, many instructions will never block since all of their operands are generated locally with the SP. Most instructions do not have routes

attached, only internal offsets for value passing. Value passing is performed by using registers. See Bic's [Bic90] for a detailed discussion of the Execution Unit's functions.

### 3.2.2. Routing Unit

The Routing Unit is loosely based upon the Direct-Connect Module in the iPSC/2. However, it must perform a number of tasks other than just making the connection. All of these tasks involve the use of the Routing Table.

The Routing Table is built at compile time and holds the static information needed to send a token from one SP to another. The figure below shows the structure of a Routing Table (note that is *not* limited to only 3 entries as shown). The Routing Table is only dependent upon the program, and is built by the PODS Translator.

unique route ID 1	(sp inst port)	(sp inst port)	(sp inst port)
unique route ID 2	(sp inst port)	(sp inst port)	NULL
unique route ID 3	(sp inst port)	NULL	NULL
unique route ID 4	(sp inst port)	(sp inst port)	(sp inst port)

FIGURE 3.2. ROUTING TABLE.

Each PE has a copy of the Routing Table. It is of a fixed size because it only holds static information, the dynamic information will be in the token's tag. To send a route the Execution Unit simply sends a local message to the Routing Unit. This message contains the route ID, the token's value, the token's tag, and whether this is to be a distributed or local or hash route. This is shown below in Figure 3.3.

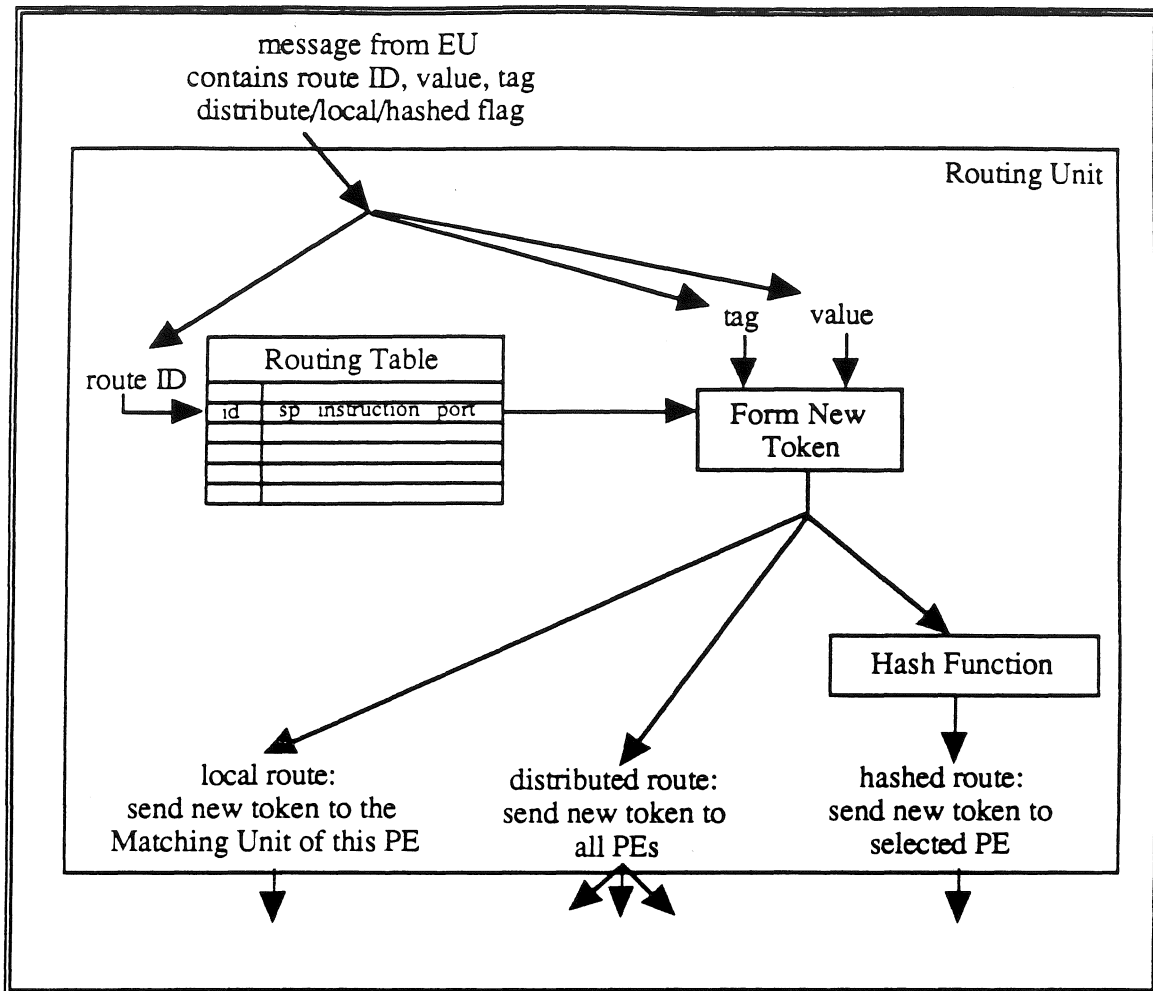


FIGURE 3.3. ROUTING UNIT BLOCK DIAGRAM.

If the route is local the destination PE is this one, and the network need not be accessed. In the future, the Execution Unit may take on this local responsibility, but that would put more burden on the Execution Unit.

If the route is a hashed route, then the Routing Unit must take the token's context, combine it with the destination (sp inst port) from the Routing Table, and run it through the hash function to determine where this particular SP is located. It is possible that this SP will be on this PE, but the Routing Unit is the only unit which can determine that.

If the route is to be distributed, then each PE is sent a message with the token in it. This is how an SP is distributed. Its parent SP calls the Routing Unit with a token and calls for it to be distributed. This will cause every PE to receive a copy of the token, and every PE will start up the appropriate SP. These distributed SPs have range filters which limit the indices which are actually generated.

As an example, consider a token with the following: value = 1, context = (2,3), iteration number = 4, and route ID = 5. If this token were to be distributed, and route ID 5 contained (1, 0, 0) (1, 1, 1), then every PE would get two messages. The first message would be destined for context (2,3), iteration number 4, SP 1, instruction 0, port 0 and have the value 1. The second would be for context (2,3), iteration number 4, SP 1, instruction 1, port 1 and have the value 1.

In an actual implementation these messages would be batched together to reduce communication costs.

### 3.2.3. Array Manager

The Array Manager handles all array accesses, except local array reads. The Execution Unit will issue a request to the Array Manager to read, write, or allocate an array. This will not cause a context switch, the Execution Unit will keep on processing until a needed value is not available. This causes a *shadow* to occur between the time the value is requested and the time it is needed. In the future this shadow can be exploited to execution as many instructions as possible before reaching the needy instruction.

When a request for an array read is received, the Array Manager determines whether the element is:

1. cached and present — return value

2. local or cached, and not present — enqueue request
3. remote — send remote request to routing unit

If the element was local and present then the Execution unit would have read it directly. To enqueue a request a flag is set in the memory location of the cell to indicate that there are requests which will need to be serviced when the cell is written. This is much like Arvind's I-structures, [ANP87a, ANP89].

When a remote read is needed, the Routing Unit will send the request to the appropriate PE (based upon the global partitioning). If the value is present then the entire page is returned. This page is then cached in the PE's software cache for that array. In this way the remote caching scheme is implemented, and further reads by this SP will most likely have some locality-of-reference. The single assignment restriction prevents writes from needing to be replicated across the network and this allows a simple caching mechanism to operate without cache coherency problems.

When an array write is requested, the Array Manager performs a similar set of tests, but the cache is never directly written. The cache will be updated when the page is brought over from the remote PE. When the value is actually written into the cell the queued read requests are dequeued and the value is sent to all of the requesting SPs, be they local or remote.

To allocate an array, every PE needs to know that space should be reserved. To do this the Array Manager on the PE where the ALLOCATE operator is fired, called the host PE, will assign the array a unique ID. This ID is then sent to all of the other PEs so that they will reserve the requested space and use the same ID. This ID is then returned to the requesting SP so that it will be used as a reference to the array from any PE.

#### **3.2.4. Memory Manager**

The Memory Manager is quite simple. It has only one task, to load SPs from program memory to execution memory. In an actual implementation, this would simply be a SP frame manager with no copying of instructions, and would probably be part of the Matching Unit.

SP's are loaded as soon as all of the tokens for the first instruction are present in the Matching Unit. There is no reason to load the SP earlier, since the SP cannot start executing until then. There is also no reason to load it any later, as the second instruction may be fed by the first.

#### **3.2.5. Matching Unit**

The task of the Matching Unit is to receive tokens and determine which SP they are destined for. Logically two tokens match if their dynamic parts and SP numbers match. This will uniquely identify a specific SP. In an actual implementation this is implemented as a hash table lookup based upon the SP ID, and the frame pointer. This hash table can be handled by a small, quick, microprocessor like the AMD 29000.

### **3.3. Remote Array Caching**

This remote array caching scheme was presented previously in [BNR89b]. For that paper the Livermore Loops benchmark programs were run and a cache size equal to 5% of the array size was found to be sufficient. This scheme has not changed significantly since that time.

Single assignment is essential to this remote array caching scheme, and a little explanation is in order. Single assignment principles allow the implementation of a simple automatic synchronization mechanism. Each memory cell has two states—undefined or defined. If a



cell is undefined, it may also have a queue of read requests associated with it. Hardware enforces the write-before-read requirement. Some examples of architectures that have this type of write-once/read-many memory access mechanism include HEP [Smi85, Smi81] and I-structure memory in dataflow [A&C86, ANP87a, ANP89].

Prior to execution, an array is either undefined or filled with initialization data (if specified in the program). Each PE may write only into undefined array cells. Race conditions are avoided by this single assignment policy. There will never be a race condition for writes to memory cell, since only one PE may write to any particular cell and writing more than once results in a run-time error.

Thus the single assignment rule automatically enforces synchronization in a distributed manner; no explicit synchronization mechanisms are necessary—a major issue in other programming paradigms.

In PODS remote writes are kept to a minimum by the partitioning described in Chapter 2. However, remote reads still occur quite often, since any instruction may read any data item. If data is mapped onto the reading PE, the access is local, otherwise it is remote; the PE must request the value from the responsible PE by sending a message. Remote reads are synchronized just like local reads—if the data item is not available, the request is queued, and if the data item is available, the *page* containing that item is sent back. During this remote read the requesting PE can perform other useful work. The requesting PE may resume executing this SP when the page arrives. This is where the benefits of array caching come in, and array caching is greatly simplified because of the single assignment principle.

Since the central idea in single assignment programming is to permit only one write to any element, by requiring single assignment we can guarantee that a page fetched from a remote PE and cached locally will not need any further updates during the lifetime of the array,

ignoring for now the possibility of partially filled pages. Given this, each PE may safely cache a remotely fetched page in a local data cache, preventing future accesses of the same remote page. The cache used will be of fixed size and a least-recently-used (LRU) replacement strategy is employed.

Without single assignment, partitioning data among PEs is possible, but it would require excessive communication overhead to allow any instruction to write to any location of an array. In addition, array caching would be nearly useless as each write performed would require the update of all remote caches containing the modified page. The machine could broadcast or multicast these updates to avoid the inefficiencies of individual messages, but the broadcasts would still strain the network facilities. Not only that, but without single assignment the caches would be inconsistent for the duration of the page modification broadcast (cache coherency problem). If no cache approach is taken, no page modification broadcasts will be necessary, and there will be no inconsistency problems. But, the use of caching leads to considerable decreases in total remote accesses performed.

It has been shown [BNR89b] that a software cache size of 5% of the array size is sufficient to reduce the number of remote read significantly. Tests with scientific code have shown that the percentage of remote reads can be reduce to less than 10% of the total number of reads in most cases. Figure 3.4 below shows the effects of different size caches on percentage of remote reads for a number of the Livermore Loops scientific benchmark programs [LLL83]. Notice that nearly *all* of the kernels drop below 10% when caching is used. The only exception is Matrix Multiply; this is because it reads one entire column of one matrix and one entire row of the other in order to write one element. PODS uses a 5% array cache.

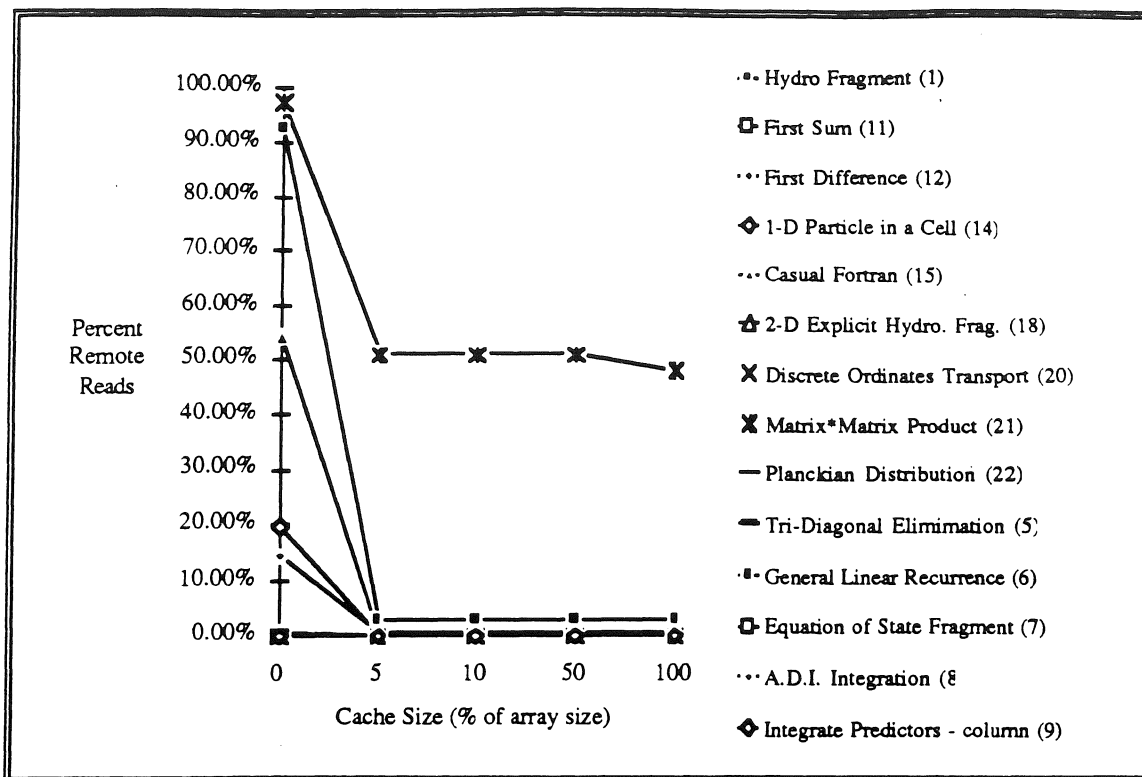


FIGURE 3.4. EFFECTS OF CACHE SIZE ON PERCENTAGE OF REMOTE READS.

As can be seen in Figure 3.5 below, the percentages of remote accesses are usually less than 5% when a 5% cache size is used, independent of the number of PEs. This caching can have anywhere from a minimal effect to an extremely large effect. Large reductions, such as 1/20th of the original remote reads, have been observed. Scientific code demonstrates significant reductions (see [BNR89b]).

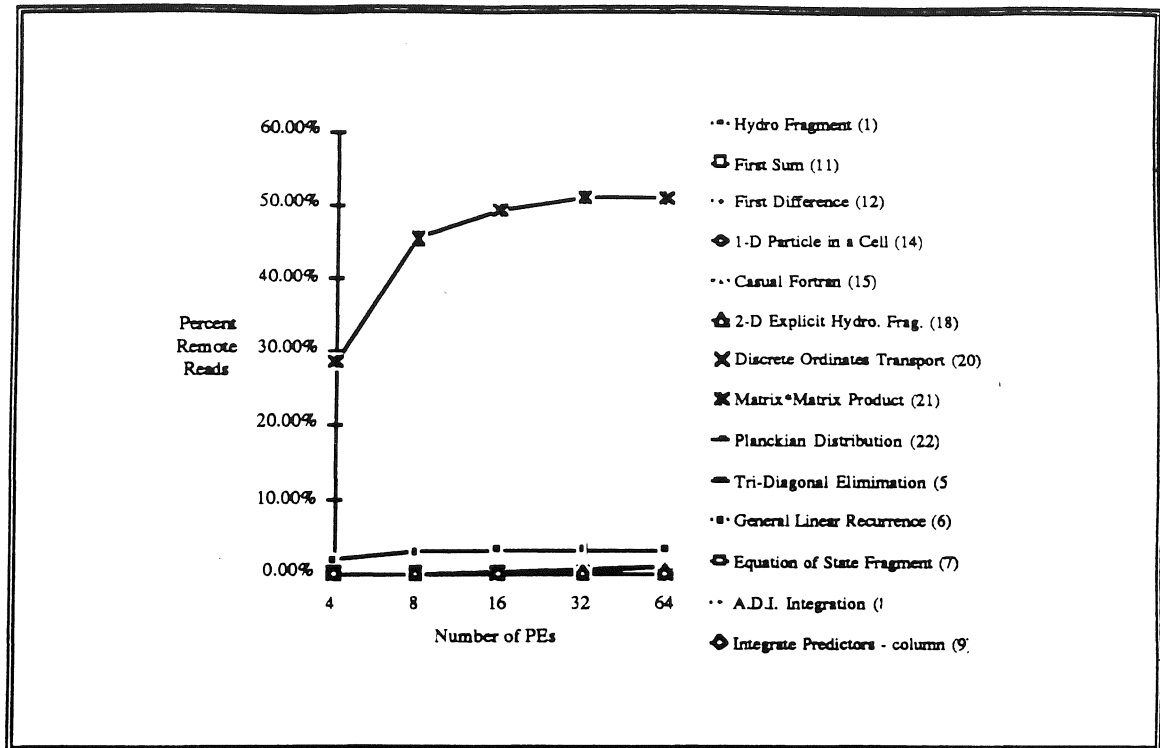


FIGURE 3.5. REMOTE READS FOR THE LIVERMORE LOOPS USING REMOTE CACHING.

At a high-level this approach is similar to that taken by Callahan and Kennedy in [C&K88]. They describe a number of the software oriented issues involved in distributing arrays across distributed memories. Unlike this approach, they allow a completely general distribution function for allocating array elements. This is very powerful, but forces the programmer to *explicitly* program in the decomposition and can lead to expensive run-time calculations. This differs from the automatic parallelization goal of PODS.

### 3.4. Software Support

In order to actually use PODS a number of support programs are necessary. These are shown in Figure 3.6 below.

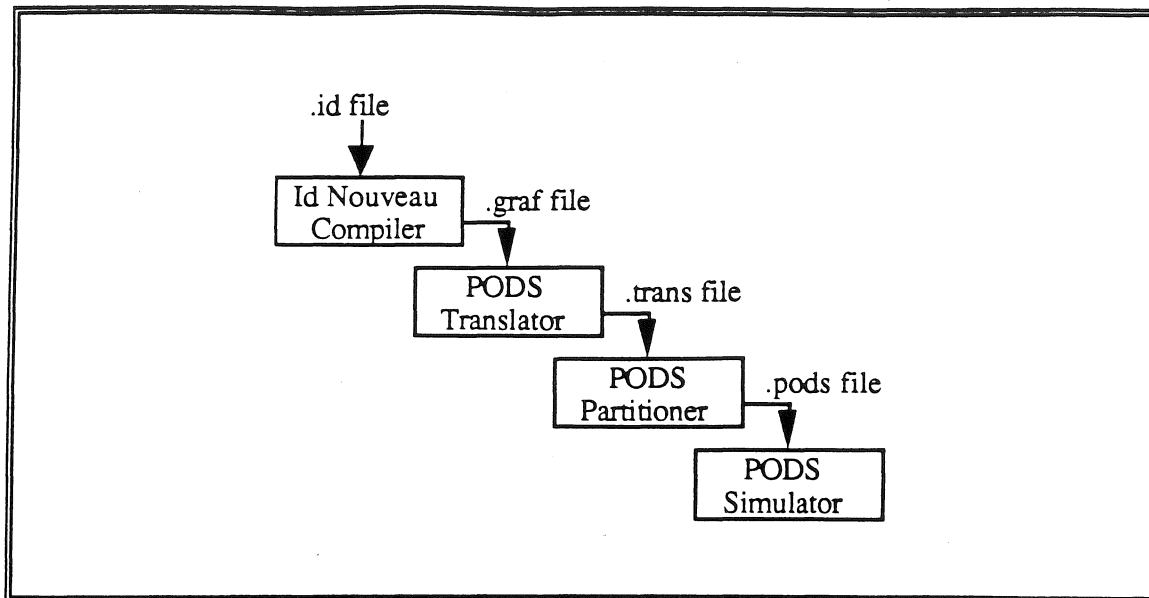


FIGURE 3.6. PODS PROGRAMMING SYSTEM.

#### 3.4.1. ID World and GITA Compiler

ID World is a software environment written at MIT [Nik87b] in LISP. As a part of the environment there is a GITA compiler which can produce dataflow graphs for the GITA.

The compiler itself [Tra86] makes use of peephole and other optimizations upon the code. The idea here was to leverage previous work in the field until the needs of PODS were better understood. In the future a direct PODS compiler is in order.

#### 3.4.2. Translator

The PODS Translator takes a set of .graf files which make up a program and converts the GITA code into PODS code. This is usually a one-to-one translation. In order for PODS to properly execute the dataflow graphs they must be ordered.

SPs, being small segments of sequential code, have to worry about supplying tokens. An operator should only send tokens to instructions which come later in the SP. The exception

to this rule is the D operator, which sends data back to the beginning of a loop. As Iannucci has pointed out [Ian88] it is not always possible to properly order a dataflow program so that the instructions are in a set, correct order. This is due to the dynamic arcs which can occur. In Chapter 2 this is discussed in the context of deadlock avoidance, and the PODS Partitioner is the program which ensures this.

Specifically the tasks of the PODS Translator are:

1. **Instruction Translation** — most GITA instructions get converted directly over to a PODS instruction one-to-one. Sometimes groups of GITA instructions make one PODS instruction. This is a format change only.
2. **Removal of Unnecessary Instructions** — for GITA a number of IDENT instructions are inserted for synchronization purposes; these are unnecessary in PODS because of the synchronization imposed by a program counter.
3. **Building of Routing Table** — for every dependency arc which goes from one .graf file to another, an entry into the Routing Table is needed.
4. **Ordering Instructions** — by following the dependency arcs the PODS instructions are placed in order such that no instructions depend upon the input from a later instruction. This handles the static dependency problem, the dynamic dependency problem is handled in the PODS Partitioner (deadlock avoidance).

### 3.4.3. Partitioner

The PODS Partitioner breaks apart the program into static SPs. It is primarily responsible for implementing the distribution scheme discussed in Chapter 2.

To break apart the dataflow graph the Partitioner starts with the code-blocks generated by the GITA compiler. From there deadlock detection is used and the SPs are split as necessary. Once it has been determined that an SP will be distributed, the Partitioner then adds the range filters and the DISTRIBUTE versions of the L operators. The .pods files are produced and the program is now ready to be run or simulated. Figure 3.7 shows the Partitioner Block Diagram.

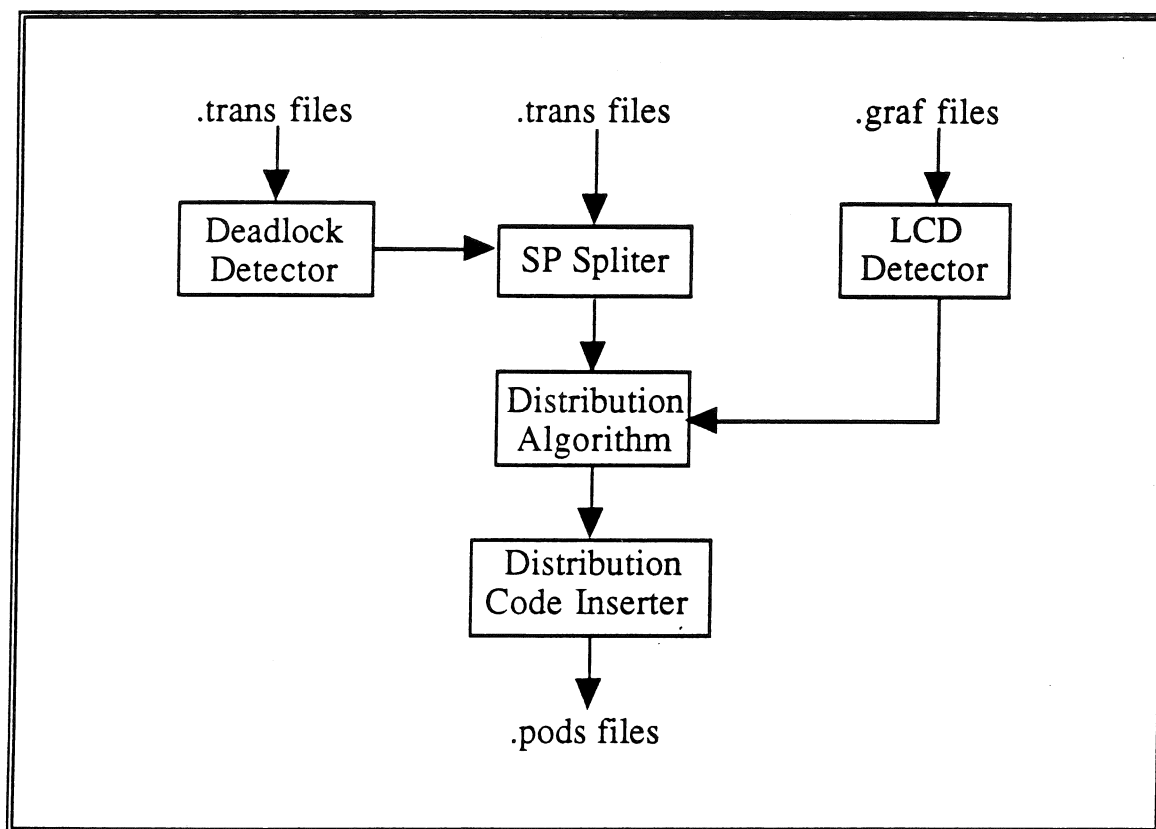


FIGURE 3.7. PODS PARTITIONER BLOCK DIAGRAM.

The Deadlock Detector uses the scheme described in Chapter 2 and informs the SP Splitter where deadlocks may occur. The SP Splitter breaks up the SPs as directed. This deadlock prevention is not necessary very often; it was not necessary anywhere in SIMPLE nor Matrix Multiply. The LCD Detector feeds the Distribution Algorithm Unit the loop-carried dependency status of each code-block. The Distribution Algorithm Unit then executes the algorithm discussed in Chapter 2. Finally the Distribution Code Inserter places the appropriate range filters into the code and annotates the L operators with either DISTRIBUTE or LOCAL.

The LCD Detector is simple because of the dataflow nature of ID Nouveau (see Section 2.5.3, LCD Effects) and the .graf files it generates. The LCD Detector is written in 'C' and follows the algorithm outlined in Chapter 2. The SP Splitter simply break a given SP up after every write to the problem array; the problem array is specified by the LCD Detector.

Specifically the tasks of the PODS Partitioner are:

1. Deadlock Detection and Avoidance — performed by the Deadlock Detector and SP Splitter; uses algorithm discussed in Section 2.7, Deadlock Handling.
2. LCD Detection — performed by the LCD Detector; uses algorithm discussed in Section 2.5.3, LCD Effects.
3. SP Distribution Determination — used output from LCD Detector to apply distribution algorithm discussed in Section 2.5.5, For-Loop Distribution Algorithm.



4. **Distribution Code Insertion** — inserts proper range filter and DISTRIBUTE or LOCAL versions of L operators; uses approach outlined in Section 2.5.2, Range Filters.

#### **3.4.4. Simulator**

The PODS Simulator is the subject of Chapter 4.

## CHAPTER 4

### PODS Simulations

In this chapter the PODS Simulator and two example programs, Matrix Multiply and SIMPLE, are examined. The results of multiple test cases are analyzed and discussed.

In the PODS Programming System, the simulator is the last program in the support software suite. The PODS Simulator was designed and build to test the logical implementation of PODS. Each PE is simulated down to the instruction level, with different functional units operating in parallel (see Chapter 3 for a description of the functional units). The PODS Simulator takes in a program and executes it step by step as if the program were actually running on PODS. In this manner the system can be measured and monitored as if running real programs.

In order to compare the results of PODS simulations to the outside world, the PODS Simulator is set-up as if it were executing on Intel 386 microprocessors in a hypercube configuration. This is not an exact simulation of Intel's iPSC/2, but timing comparisons to programs on iPSC/2 systems are valid. The major, real-world program described herein is the SIMPLE benchmark [CH&R] developed by Lawrence Livermore Laboratory. This code is indicative of the large scale scientific code which is executed on supercomputers today.

#### 4.1. Overview

##### 4.1.1. Simulator Approach

The PODS Simulator is an event-driven simulator which uses SMPL at its core. MacDougall has written an excellent book [Mac87] which describes SMPL and its proper

usage. In the PODS Simulator, as in any simulation, certain assumptions are necessary. These have been kept to a minimum and are based on known or measured statistics.

There is a hardware configuration file which holds the following hardware parameters:

- **NUMBER\_OF\_PE** — the number of processing elements to simulate
- **PAGE\_SIZE** — the size of an array page (set at 32 array elements)
- **BROADCAST\_NET** — whether a broadcast type of message is available or not (set to true)
- **CACHE\_PERCENT** — the size of the software cache for each array (set at 5%)

The hardware configuration file also holds the following timing parameters:

- **NETWORK\_TIME** — the time for a message to propagate over the network
- **SINGLE\_ROUTE\_TIME** — the time to build a single message token into a batch inside the Routing Unit
- **MS\_SETUP\_TIME** — the time for the Matching Unit to find if a token has a match
- **MM\_SETUP\_TIME** — the time for the Memory Manager to wake-up when a new SP is to be loaded

- SINGLE\_CONTEXT\_SWITCH\_TIME — the time for a fast context switch

The values for these, and other timing parameters, is discussed in the next section.

#### 4.1.2. Timing Assumptions

In order to estimate the amount of time a CISC would take to perform a given operation, the PODS Simulator is sized to the Intel iPSC/2's PEs. These are Intel 80386/80387 CPU's at 16 MHz. with Direct-Connect Modules for communication. All timing is done in  $\mu$ seconds. Each functional unit's timing is described below

##### Execution Unit

This is the ALU and associated units. Its timing is based upon three calculations: (1) the time it takes to perform a fast context switch; (2) the time to perform a local array read; and (3) the time of each normal operation. Time for each normal operation was measured on the iPSC/2 with the following results:

iPSC/2 Instruction	Execution time ( $\mu$ sec)
integer add	0.300
integer subtraction	0.300
bitwise logical	0.558
floating point negate	0.555
floating point compare	5.803
floating point power	96.418
floating point abs	12.626
floating point square root	18.929
floating point multiply	7.217
floating point division	10.707
floating point addition	6.753
floating point subtraction	6.757

TABLE 4.1. MEASURED TIMES OF OPERATIONS ON IPSC/2.

The time for a local array read is based on the pseudo code in Figure 4.1 below.

```

offset = size_dim2 * i + j
if (offset < beginning_offset) goto REMOTE_READ
if (offset ≥ ending_offset) goto REMOTE_READ
if (element not present) goto ENQUEUE_READ
value = array[offset]

```

FIGURE 4.1. 2-D ARRAY READ PSEUDO-CODE.

The time for a local array read (assuming the value is present) is: 1 integer multiply + 1 integer add + 3 integer compares + 1 local read. This works out to be 2.7  $\mu$ seconds.

The time for a fast context switch is based on the 80386 CALL ptr16:32 instruction. This is a full 32 bit indirect procedure call. The worst case for this is 21 clock cycles or 1.312  $\mu$ seconds at 16 Mhz.

### Array Manager

The Array Manager handles all array operations except local array reads (which are performed by the Execution Unit). The Array Manager handles the following tasks in the indicated times.

- FreeArray: number\_arrays \* memory\_read\_time
- ArrayWrite: memory\_write\_time + number\_queued\_reads \* message\_time
- CachedRead: memory\_read\_time + message\_time if not present
- RemoteRead: memory\_read\_time + enqueued\_read\_time or message\_time

- ReceivePage:  $\text{page\_size} * \text{memory\_write\_time}$
- SendPage:  $\text{page\_size} * \text{memory\_read\_time} + \text{message\_time}$
- AllocateArray:  $100.0 \mu\text{seconds} + \text{message\_time}$

where

- $\text{memory\_read\_time}$  is the time for a local read =  $0.3 \mu\text{seconds}$
- $\text{memory\_write\_time}$  is the time for a local write =  $0.4 \mu\text{seconds}$
- $\text{message\_time}$  is the time for a signal from one functional unit to another on the same PE =  $1.0 \mu\text{seconds}$
- $\text{enqueued\_read\_time}$  is the time to push an early read onto a stack =  $3 * \text{memory\_read\_time} + 5 * \text{memory\_write\_time} = 2.9 \mu\text{seconds}$

### Routing Unit

This is basically the Direct-Connect Module with some extra operations. This unit is responsible for taking a token, forming the message, and sending it over the network to the correct PE and SP. Dunigan [Dun88] has done some extensive testing of the iPSC/2 and found that the communication can effectively be expressed using the following equations:

if ( $\text{message\_length} \leq 100$  bytes) then  $390 \mu\text{sec}$

if ( $\text{message\_length} > 100$  bytes) then  $697 + 0.4 * \text{message\_length} \mu\text{sec}$

The extra operations calculate the SP and PE to which the token will be sent. When the Routing Unit receives a token to route, a simple table look-up is used to find the destination SPs. This is then used in a hash function to find the destination PE. Since tokens are less

than 100 bytes, and they are batched together in groups of 20, the simulation uses an estimate of 19.5  $\mu$ seconds for each token added to a batch.

### Memory Manager

The Memory Manger simply grabs execution memory frames from free memory. This is a list manager, one list for free SP frames and one for used SP frames. To perform its operations the Memory Manger need only add or delete from a linked list. This is a constant time operation which takes approximately 3 memory references or 0.9  $\mu$ seconds.

### Matching Store

The Matching Store must search the hash table for the appropriate SP. This is a simple hash search which takes 15  $\mu$ seconds.

### Network

The Network is simply the physical propagation time. The Routing Unit handles all of the transmission setup. The iPSC/2 has a theoretical 100 Mbyte per second bandwidth. Assuming each message is approximately 100 bytes, the time for 1 hop is 1  $\mu$ second. The network time is set to 2.5  $\mu$ seconds, simulating an average of 2.5 hops. The Network can only handle so many messages at a time, this is estimated to be half the number of PEs.

## **4.2. Measures of Effectiveness (MOEs)**

The motivation behind the following Measures of Effectiveness (MOEs) is to gauge how well PODS will perform on a real system with real-world problems, and how does this compare to what is available today.

**Functional Unit Balance** — how well balanced are the functional units which make up the PE? This is measured by SMPL as the fraction of the time which a given facility is

busy, i.e. the utilization. Since PODS PEs contain parallel functional units, the balance between the units is important. If one of the support units, e.g. the Routing Unit, were very heavily loaded then the Execution Unit may be waiting for it. This would point to possible improvements in the logical architecture design.

**Execution Unit Utilization** — what percent of the time are the Execution Units operating? Do some PEs sit idle awaiting the outcome of other PEs? Ideally utilization should be 100% for each PE, this is never actually possible. This is measured by SMPL as the accumulated busy time of the each execution unit, divided by the total run time.

**Execution Unit Load Balance** — how equally distributed is the work load? Ideally each PE will put in the same amount of work. This shows if there are any "hot-spots" where some PEs are doing all the work while others are idle.

**Parallelization Overhead** — how many of the instructions executed are "work" instructions and how many are due to parallelization. This shows how much additional overhead there is in the parallel version of the program. In the PODS Simulator the dynamic work instructions as well as the total dynamic instructions are counted. Work instructions are those which must be executed no matter how many PEs are used. i.e. All instructions except the range filter instructions.

**Efficiency Comparison** — how efficient is the parallel version on one PE when compared to a real sequential version (usually 'C' or FORTRAN). Usually the parallel version will be less efficient because of the additional tasks which must be performed for multiple PEs even though there is only one operating. Also, commercial systems have additional optimizations which research systems do not. If this comparison shows that the parallel system is within 100% of the sequential version on one PE, then the parallel system is not grossly inefficient, and the scalability results can be considered to have a valid base



time. For Matrix Multiply and SIMPLE, 'C' versions were compiled using the Intel supplied compiler and timed on the iPSC/2 host.

**Scalability** — how much do problems speed-up as the number of PEs is increased?

Ideally linear speed-up is possible. However, overhead and program dependencies prevent this from being achieved. This can be seen by plotting the number of PEs vs the speed-up, where speed-up is defined to be the time of a single PE run divided by the time of the multiple PE run. This is the most important measure of effectiveness of a parallel processing system.

### 4.3. Example Programs

The results presented here are for two different programs. The first program is for matrix multiplication and is discussed in detail in Chapter 2. The second program is SIMPLE, a benchmark program written by Crowley et. al. [CH&R] at Lawrence Livermore Laboratory. This benchmark was designed to test computer systems performance on the type of large scientific programs which the laboratory runs. It is used here to show how well PODS executes large scientific programs. For more detail on SIMPLE see [P&R90].

#### 4.3.1. Matrix Multiply

A detailed discussion of the Matrix Multiply example is contained in Chapter 2. However, a brief discussion here is also in order.

##### Discussion

Consider the implementation of Matrix Multiply in ID Nouveau shown in Figure 4.2. The code follows the basic sequential Matrix Multiply algorithm below, very closely.

$$C[i, j] = \sum_{k=1}^n A[i, k] * B[k, j]$$

The use of Next s in line #9 creates a LCD while performing a reduction operation. The array write into C in line #7 controls the partitioning, i.e., array C is the master array.

```

%%% Matrix Multiply
1 Def mm A B = {(l1,u1), (l2,u2) = 2D_bounds A;
2 C = i_matrix ((l1,u1), (l2,u2));
3 In
4 { For i <- l1 To u1 Do
5   { For j <- l2 to u2 Do
6     s = 0;
7     C[i,j] =
8       { For k <- l1 To u1 Do
9         Next s = s + A[i,k] * B[k,j];
10      Finally s
11    }
12  };
13 Finally C
14 }
15 };

```

FIGURE 4.2. MATRIX MULTIPLY ID NOUVEAU SOURCE CODE.

This function contains a number of items worth noting: (1) there are three different SPs (one for each for-loop nest level); (2) a new array, C, must be dynamically allocated and distributed efficiently; (3) there is a loop-carried dependency in the innermost loop (the sum variable, s); (4) the two input arrays, A and B, have different access patterns; and (5) the sizes of the input arrays are not known at compile time. These attributes make the Matrix Multiply algorithm an interesting test case.

### Results

**Functional Unit Balance.** Figure 4.3 below shows the average utilization for the different functional units when the 16 x 16 case is run. Notice that all of the support units are not being heavily utilized. Thus the Execution Unit is not being slowed by the support units. This shows that the support units are truly operating in a support function and are

not performing extensive operations. This bodes well for HyperPODS, where all PE functions will be performed by one CPU.

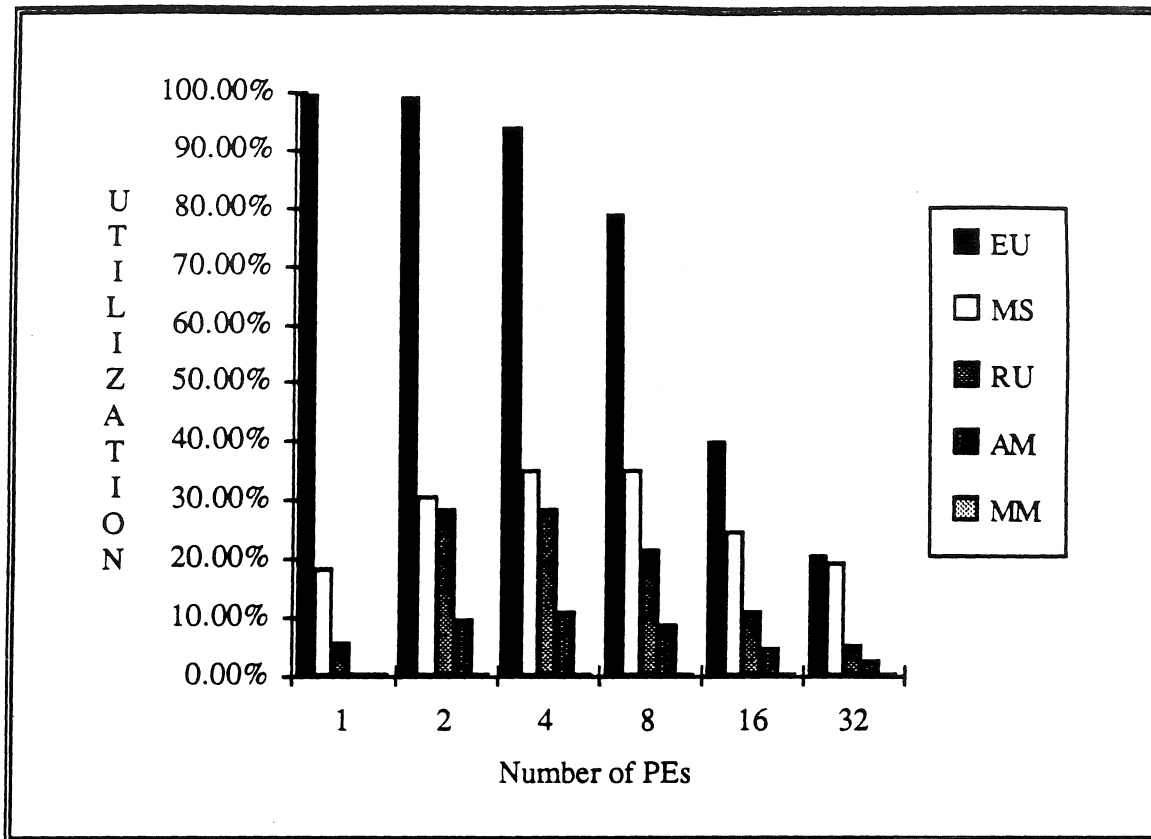


FIGURE 4.3. UTILIZATION FOR EACH FUNCTIONAL UNIT (16 X 16 MM).

The Execution Unit has the highest utilization until the parallelism drops below that necessary to keep all of the PEs active. The important case above is the 8 PE situation. This is where the problem size meets the available PEs. In this case the Execution Unit utilization is more than double that of the most loaded support unit (78% vs 35% for the Matching Store).

**Execution Unit Utilization.** Since the Execution Unit is the major unit doing the work done by a PE, as shown above, its utilization is critical. For Matrix Multiply the Execution Unit utilization increases as the problem size increases. This is true in general and is due to

the increase in the parallelism in larger problems. As Figure 4.4 shows below, PODS is only able to spread the available parallelism so far, and as more PEs are made available PODS is unable to fully utilize all of them.

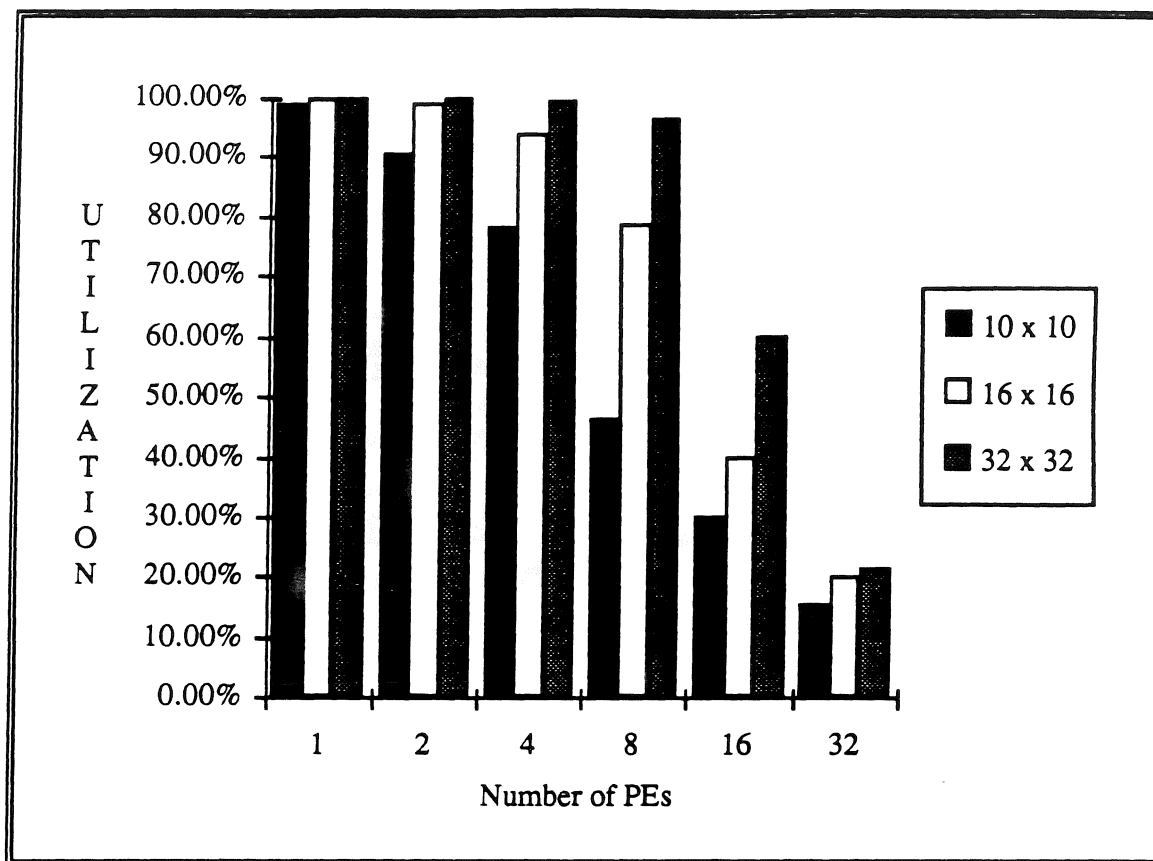


FIGURE 4.4. AVERAGE EXECUTION UNIT UTILIZATION FOR MATRIX MULTIPLY.

This inability to work all of the Execution Units fully will show up in the scalability of the program. When the average utilization nears 80% this is usually the end of the speed-up. For a the 10 x 10 case this occurs at 4 PEs, for 16 x 16 at 8 PEs, and for 32 x 32 at 16 PEs. The scalability results below bear this out. This 80% number is only indicative of Matrix Multiply-like problems. SIMPLE, being much more complex does not exhibit this problem.

**Execution Unit Load Balance.** Load balance is more of an issue when Execution Unit utilization is less than 80%. For utilizations greater than 80%, most of the PEs must be working about the same or the utilization would be lower. For this reason it is more interesting to consider the load balance for the medium sized problem, 16 x 16 arrays, than the large problem.

Figure 4.5 shows each Execution Unit's utilization for the 16 x 16 case on 8 PEs. Contrast this to Figure 4.6 where most of the work is being performed on only half of the PEs.

This is where the iteration level parallelism is completely used up. This is what causes the flat speed-up curve at from 8 PEs on up to 32 PEs for the 16 x 16 Matrix Multiply (see Figure 4.7 below).

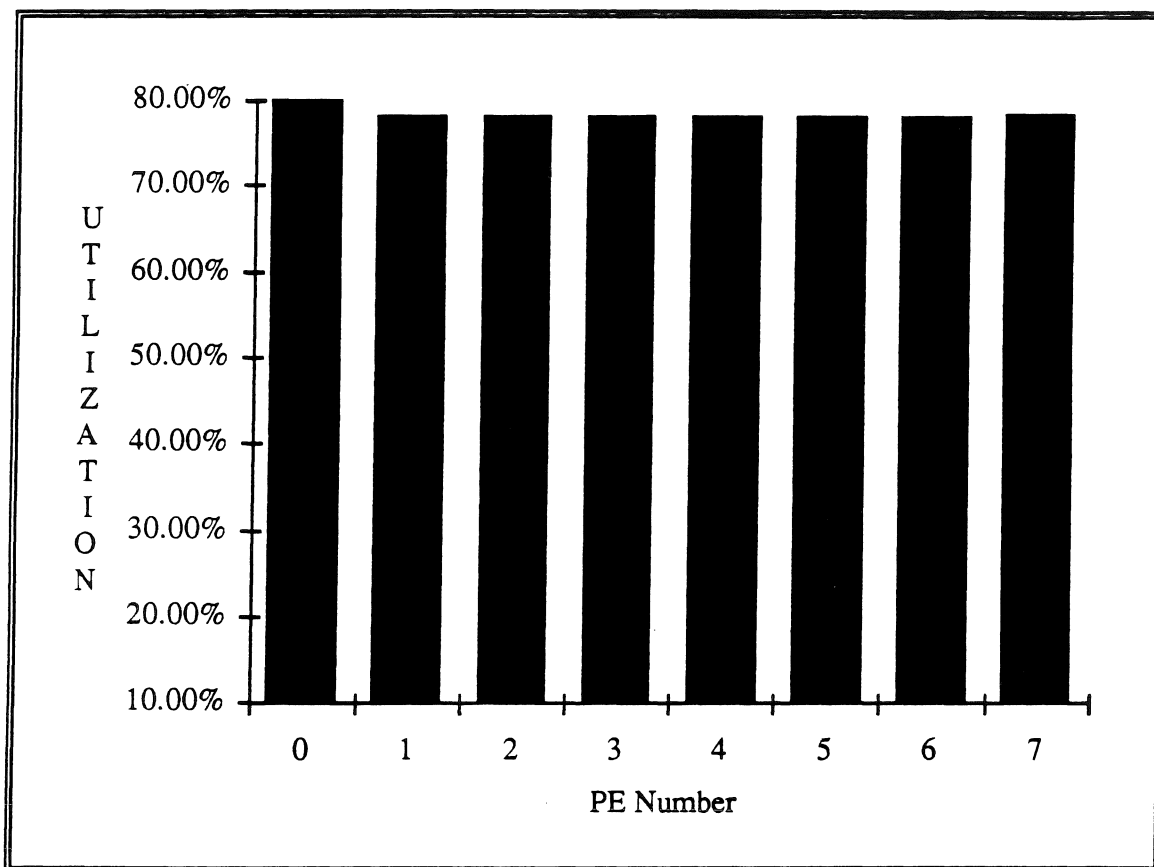


FIGURE 4.5. UTILIZATION FOR EACH EXECUTION UNIT (16 X 16 MM ON 8 PEs).

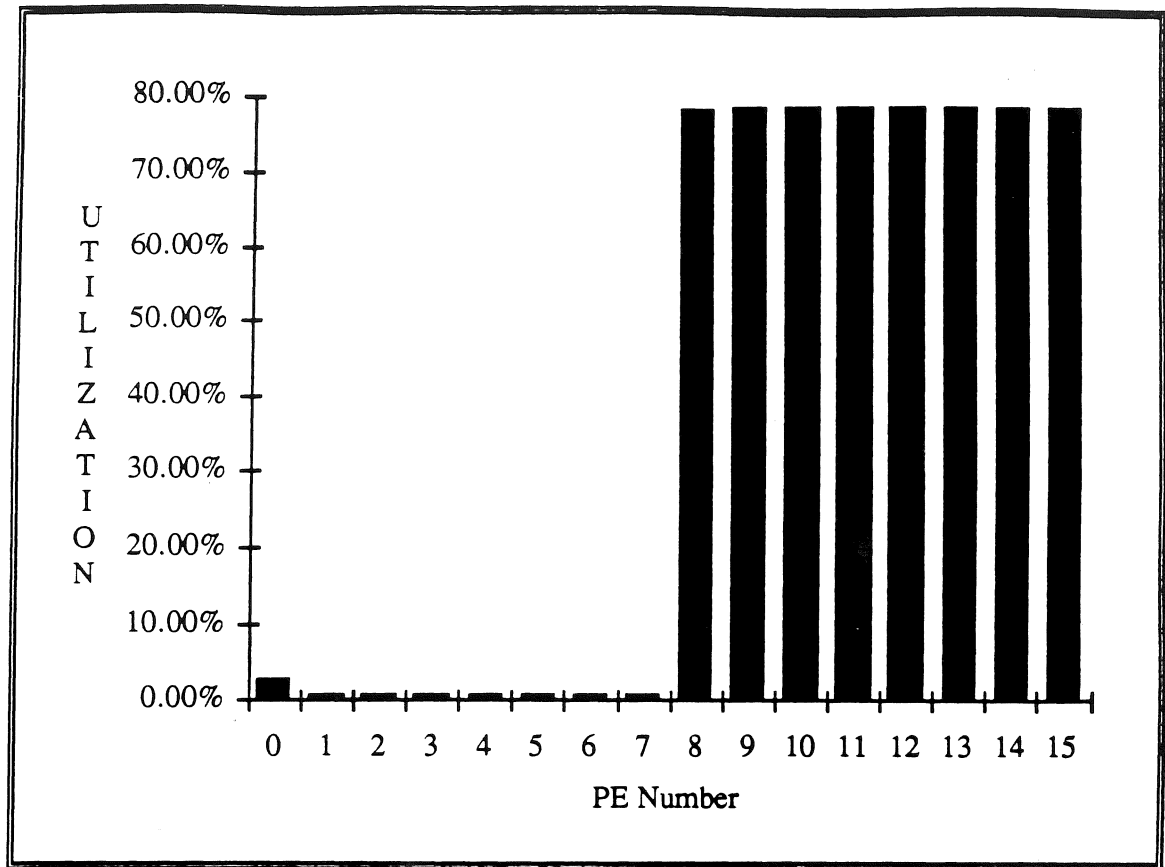


FIGURE 4.6. UTILIZATION FOR EACH EXECUTION UNIT (16 X 16 MM ON 16 PES).

**Parallelization Overhead.** For Matrix Multiply the amount of overhead due to parallelization decreases as the problem size increases. The table below shows dynamic instruction counts for different problem sizes. All of these counts are for the 32 PE system (worst case).

Problem Size	Work Instructions	Total Instructions	Percent Overhead
10 x 10	10,851	15,072	28.01%
16 x 16	43,083	50,460	14.62%
32 x 32	336,011	362,028	7.19%

TABLE 4.2. PERCENT OVERHEAD INSTRUCTIONS FOR MATRIX MULTIPLY.

This indicates that, for Matrix Multiply-like algorithms, the amount of parallelization overhead in PODS is acceptable at large input sizes. This is one reason that speed-up increases (see scalability below) as the problem size increases.

**Efficiency Comparison.** A 16 x 16 Matrix Multiply, written in 'C' and compiled for a single iPSC/2 PE, takes 0.1 seconds to execute. The PODS Simulator estimates that the program would run in 0.190 seconds. This is within 100% of the commercial 'C' version, and shows that PODS is not grossly inefficient, even on one PE.

It is also interesting to compare the number of dynamic work instructions the two systems execute. The standard C compiler on the iPSC/2 produces code which executes 51,893 instructions, while PODS executes 43,083. This ratio of about 1.2:1 holds true for all of the Matrix Multiply cases. This means that PODS executes about the same number of the same size instructions as a commercial system. The reason PODS is slower on one PE, is because of the multiple PE tasks it is performing.

**Scalability.** Figure 4.7 shows the speed-up of different size Matrix Multiply runs. For comparison the speed-up predicted for Iannucci's hybrid system is plotted [Ian88].

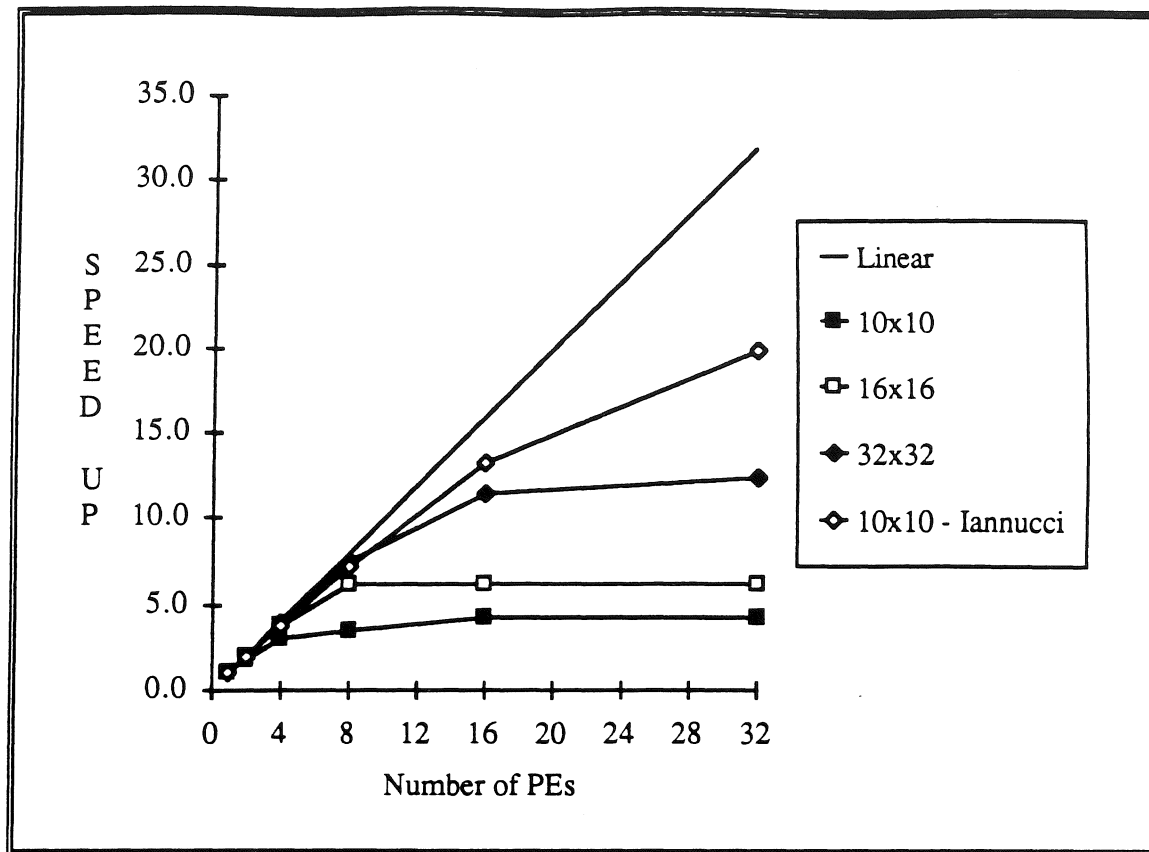


FIGURE 4.7. SPEED-UP OF MATRIX MULTIPLY.

Iannucci's machine is finer grain and is able to exploit more of the parallelism in the small 10 x 10 problem. PODS does not reach this type of performance until the 32 x 32 problem is run. Since Iannucci's machine requires a new CPU design and system architecture, it is impossible to know how well it compares to a commercial system. Leaving open the question of absolute run times and true scalability. It will be interesting to see how cost effective the system will be once it is built.

#### 4.3.2. SIMPLE

Simulating the execution of all of SIMPLE on the PODS Simulator is not possible due to memory limitations. So SIMPLE was broken up into its component routines. The major routines were run through the translator then through the partitioner, and finally simulated



on the PODS Simulator. These major routines are: VELOCITY\_POSITION, HYDRODYNAMICS, and CONDUCTION. All of the other procedures are either run only once (e.g. GENERATOR) or are called by one of the above. This breaking up of SIMPLE is appropriate because the routines feed each other in a sequential fashion. There may be some parallelism which is not being exploited, but it is minimal.

The most important routine is CONDUCTION, both VELOCITY\_POSITION, and HYDRODYNAMICS are much easier to parallelize. VELOCITY\_POSITION has no LCDs, no function calls, and runs in parallel very well. HYDRODYNAMICS has only 5 SPs (CONDUCTION has 15 SP) and is basically one big nested loop; it is not nearly as complex as CONDUCTION. CONDUCTION is the most difficult to parallelize because of: (1) the sweep phases where every element is recalculated twice, based upon its neighbors; (2) the complexity of 15 SP plus multiple function calls; and (3) the large number of LCDs with both incrementing and decrementing for-loops. These LCD's prevent iteration level parallelism from being distributed efficiently. For these reasons CONDUCTION is examined in detail in the discussion section, while the final results for all of SIMPLE added together is presented below in the results section.

### Discussion

The original ID code for SIMPLE was written at MIT based upon the Lawrence Livermore version. This original ID code was then updated to ID Nouveau. CONDUCTION is a complex routine with multiple function calls and code blocks.

The sweep operations in CONDUCTION cause LCD to occur in the inner-most nest of the loops. Figure 4.8 shows one of the sweep blocks (there are two nearly identical sweeps) inside of CONDUCTION. Notice that the local arrays *a* and *b* are allocated at the outer level (lines #3 and #4), filled in the next inner nest (lines #11 - #13), and then used to fill the *theta\_bar* arrays (lines #16 and #17). Both of these last two loops have LCDs. In lines

#12 and #13  $b[l-1]$  is used to produce  $b[l]$ , generating a LCD with distance 1. In lines #16 and #17  $\theta_{bar}[k,l+1]$  is used to produce  $\theta_{bar}[k,l]$ . This generates a LCD with distance 1 because the for-loop is decrementing (see `downto` in line #15).

```

% Alternating direction sweeps
% z_sweep
1  theta_bar = i_matrix ((kmn+1,kmx), (lmn+1,lmx+1));
2      {for k <- kmn+1 to kmx do
3          a = i_array (lmn,lmx);
4          b = i_array (lmn,lmx);
5
6          % a[lmn],b[lmn] are not used because cbb[* ,lmn]=0
7          a[lmn] = 0;
8          b[lmn] = 0.0;
9
10         {for l <- lmn+1 to lmx do
11             y = sigma[k,l]+cbb[k-1,l]
12                +cbb[k-1,l-1]*(1-a[l-1]);
13             a[l] = cbb[k-1,l]/y;
14             b[l] = (sigma[k,l]*theta_hat[k,l]
15                    +cbb[k-1,l-1]*b[l-1])/y
16         };
17
18         %%% back substitution
19         theta_bar[k,lmx+1] = 0;
20
21         % theta[k,lmx+1] is not used because a[lmx]=0
22
23         {for l <- lmx downto lmn+1 do
24             theta_bar[k,l] = a[l]*theta_bar[k,l+1]
25                + b[l]
26         };
27     };

```

FIGURE 4.8. SWEEP FOR-LOOPS IN CONDUCTION CODE.

These sweep operations can severely limit parallelism in some systems. In PODS the outer nest of the for-loop (either  $k$  or  $l$ ) is distributed across the available PEs. Once this is done then no future distribution is necessary.

In another part of CONDUCTION there is a nested for-loop with LCDs at all levels: lines #30 - #32 for the outer level, and lines #20 and #21 for the inner level. This for-loop is shown in Figure 4.9. This for-loop would be modified by a scalar expanding compiler.

```

1  delta_theta_max, internal_eps =
2  {
3      delta_theta = 0; internal_eps = 0;
4      in
5      {for k <- kmn+1 to kmx do
6          y, col_internal_eps =
7          {
8              delta_theta_col = 0; col_internal_eps=0;
9              in
10             {for l <- lmn+1 to lmx do
11                 i = table_look_up
12                     theta_table theta_transp[l,k]
13                     index1[k,l] 3;
14                 j = index2[k,l];
15                 last_index1[k,l] = i;
16                 eps_k_l = polynomial theta_transp[l,k]
17                     rho[k,l] i j T_Coefficients;
18                 p[k,l] = polynomial theta_transp[l,k]
19                     rho[k,l] i j P_Coefficients;
20                 next col_internal_eps =
21                     col_internal_eps + mass[k,l]*eps_k_l;
22                 eps[k,l] = eps_k_l;
23                 y = abs(theta_hat[k,l] -
24                     theta_transp[l,k])/theta_transp[l,k];
25                 next delta_theta_col =
26                     if y > delta_theta_col
27                         then y else delta_theta_col
28                 finally delta_theta_col, col_internal_eps}
29             };
30             next delta_theta = if y > delta_theta then y
31                 else delta_theta;
32             next internal_eps = internal_eps + col_internal_eps
33             finally delta_theta, internal_eps }
34         };

```

FIGURE 4.9. ORIGINAL CONDUCTION CODE WITH MULTIPLE LCDS.

The above code was replaced with the following in Figure 4.10. The lines in bold below were added or modified ( lines #1, #2, #28, #29, and #31 - #42).

```

%%% changed by jmar
1  vect_cie = i_array (kmn, kmx);
2  vect_y = i_array (kmn, kmx);
3  {for k <- kmn+1 to kmx do
4      y, col_internal_eps =
5      {
6          delta_theta_col = 0; col_internal_eps=0;
7      in
8      {for l <- lmn+1 to lmx do
9          i = table_look_up
10             theta_table theta_transp[l,k]
11             index1[k,l] 3;
12          j = index2[k,l];
13          last_index1[k,l] = i;
14          eps_k_l = polynomial theta_transp[l,k]
15                  rho[k,l] i j T_Coefficients;
16          p[k,l] = polynomial theta_transp[l,k]
17                  rho[k,l] i j P_Coefficients;
18          next col_internal_eps = col_internal_eps
19                  + mass[k,l]*eps_k_l;
20          eps[k,l] = eps_k_l;
21          y = abs(theta_hat[k,l] -
22                  theta_transp[l,k])/theta_transp[l,k];
23          next delta_theta_col =
24                  if y > delta_theta_col
25                  then y else delta_theta_col
26          finally delta_theta_col, col_internal_eps}
27      };

28      vect_y[k] = y;
29      vect_cie[k] = col_internal_eps;
30 };

%%% added by jmar
31 delta_theta_max, internal_eps =
32 {
33     delta_theta = 0; internal_eps = 0;
34 in
35 {for k <- kmn+1 to kmx do
36     next delta_theta = if vect_y[k]>delta_theta
37                       then vect_y[k]
38                       else delta_theta;
39     next internal_eps = internal_eps +
40                       vect_cie[k];
41     finally delta_theta, internal_eps}
42 };

```

FIGURE 4.10. SCALAR EXPANDED CONDUCTION CODE FRAGMENT.

This scalar expansion does not change the output in any way and is a standard compiler optimization.

Another interesting point is that three different subroutines are called: POLYNOMIAL, TABLE\_LOOK\_UP, and BOUNDARY\_HEAT\_FLOW. With POLYNOMIAL and TABLE\_LOOK\_UP being called many times inside the inner for-loop. These function calls are spun off onto other processors to allow more parallelism to be exploited.

Once the scalar expansion is done, all of the for-loops, except the one added by the expansion (lines #31 - #42), are distributed at the first level of the nest. This allows CONDUCTION iterations to be spread across all available PEs, thus producing excellent speed-up.

The 22 SPs which PODS forms for CONDUCTION are shown in Table 4.3 below along with some statistics for each SP.

SP Name	Static PODS Instructions	Distribution Comments
CONDUCTION.pods	94	Main SP, drives others
CONDUCTION-0.pods	20	LCD prevents distribution, added by scalar expansion
CONDUCTION-1.pods	39	Distributed For-Loop SP
CONDUCTION-1-0.pods	12	Local For-Loop SP
CONDUCTION-1-1.pods	26	Local For-Loop SP
CONDUCTION-2.pods	40	Distributed For-Loop SP
CONDUCTION-2-0.pods	12	Local For-Loop SP
CONDUCTION-2-1.pods	26	Local For-Loop SP
CONDUCTION-3.pods	29	Distributed For-Loop SP
CONDUCTION-4.pods	37	Distributed For-Loop SP
CONDUCTION-4-0.pods	38	Local For-Loop SP
CONDUCTION-5.pods	31	Distributed For-Loop SP
CONDUCTION-5-0.pods	27	Local For-Loop SP
CONDUCTION-6.pods	28	Distributed For-Loop SP
CONDUCTION-6-0.pods	27	Local For-Loop SP
BHF.pods	22	Main SP of Procedure
BHF-0.pods	20	Small SP, local to BHF
BHF-1.pods	20	Small SP, local to BHF
TLU.pods	19	Main SP of Procedure
TLU-1.pods	9	Small SP, local to TLU
TLU-0.pods	10	Small SP, local to TLU
POLY.pods	40	Procedure SP

TABLE 4.3. SP STATISTICS FOR CONDUCTION.

### Results

These results are for all of the SIMPLE routines added together. This is valid because each of the routines feeds the next one. If there is some iteration level parallelism available between routines, then the results will be better than shown here. This was necessary due to the performance limitations of the PODS simulator.

**Functional Unit Balance.** Smaller problem sizes stress the distribution of work between functional units more than larger ones. This is because larger problems have more available parallelism and an unbalance PE may not show a drop in utilization. The worst case, 16 x 16, utilization is shown in Figure 4.11.

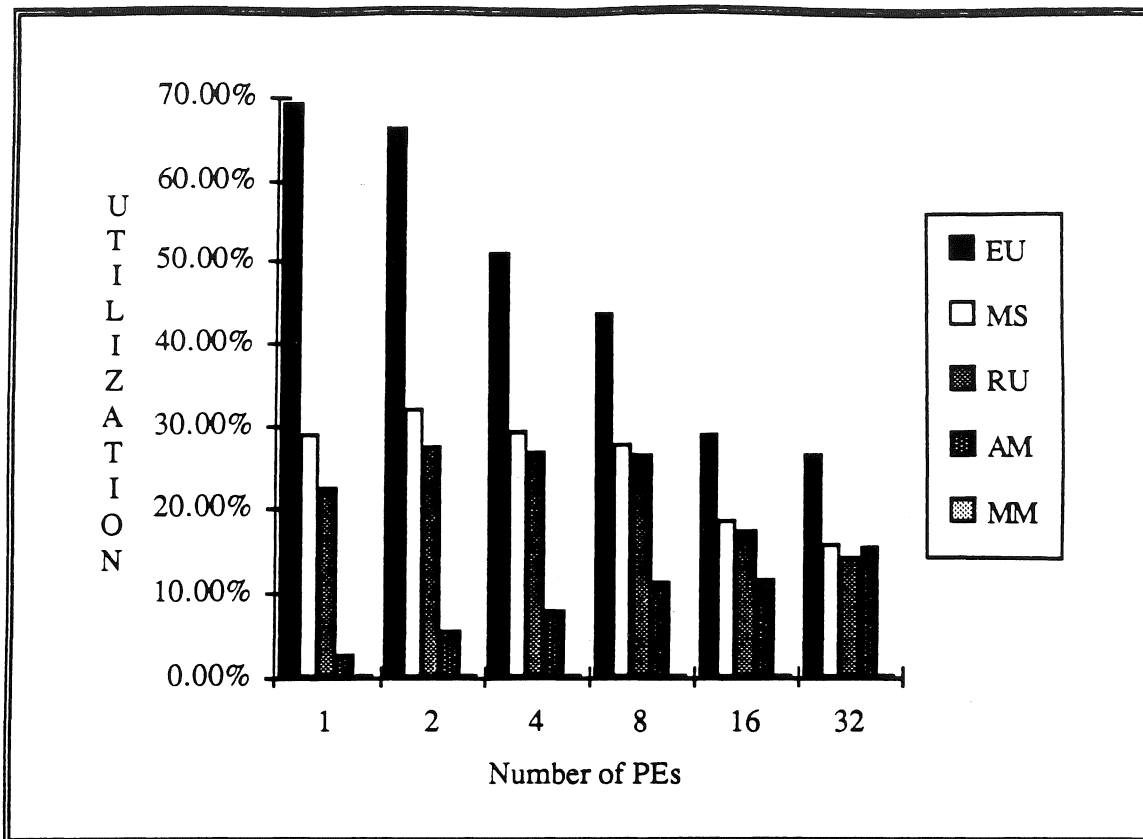


FIGURE 4.11. UTILIZATION FOR EACH FUNCTIONAL UNIT (16 X 16 SIMPLE).

The support units once again act in a support role, never reaching any significant utilization until the available parallelism has been used up, at around 8 PEs. Even at 32 PEs the support units do not have any bottlenecks, the only change is that the Execution Units utilization has dropped to a level comparable to the support units.

**Execution Unit Utilization.** For a 64 x 64 SIMPLE the utilization starts out at approximately 70% for 1 PE and goes down to 50% for 32 PEs (see Figure 4.12). Once again on small problems (16 x 16) the Execution Unit utilization is much lower than on large problems (64 x 64).

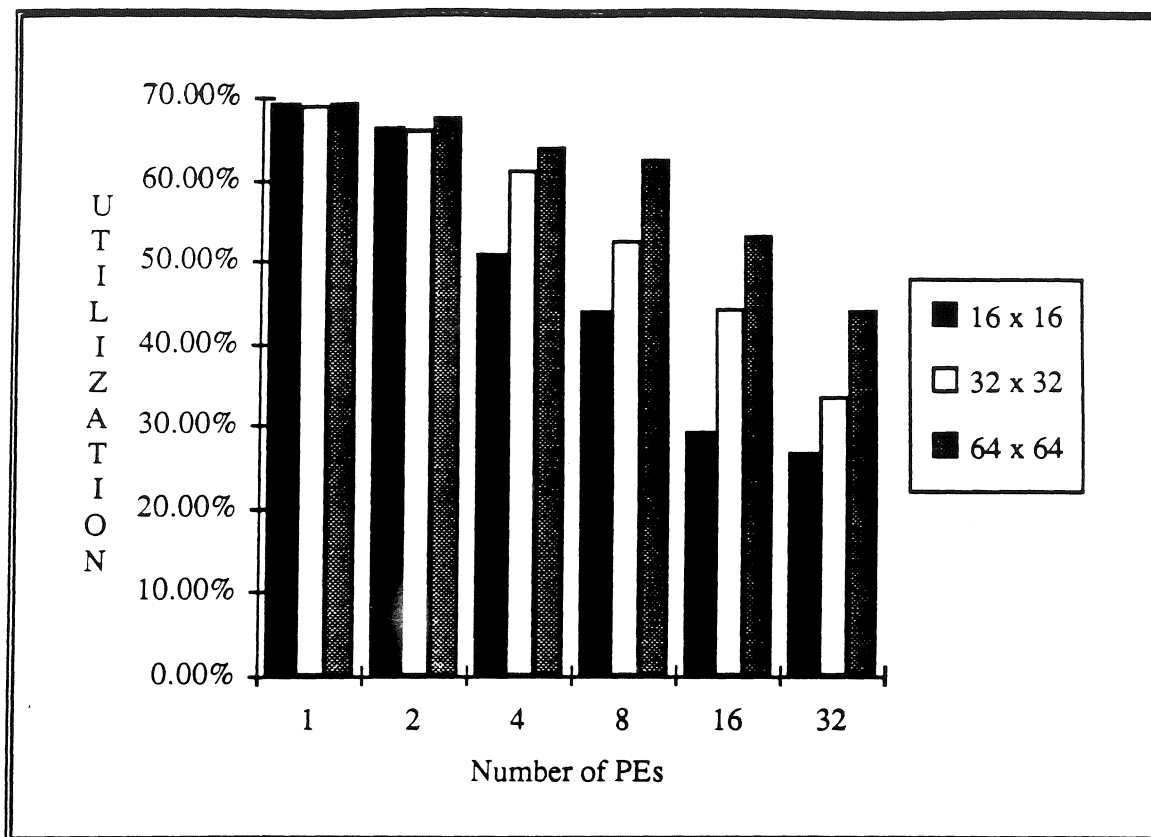


FIGURE 4.12. EXECUTION UNIT UTILIZATION FOR SIMPLE.

It is interesting that SIMPLE continues to speed-up even with Execution Units which are 50% idle (see Figure 4.16 below). This differs from the Matrix Multiply example above, which stopped speeding-up when utilization drops below 80%. This is due to the complexity difference between SIMPLE and Matrix Multiply.

**Execution Unit Load Balance.** SIMPLE, being much more complex than Matrix Multiply, spread its load much better. Even in the worst case (16 x 16 on 32 PEs), where little speed-up is begin gained, every PE contributes to the final solution (see Figure 4.13).



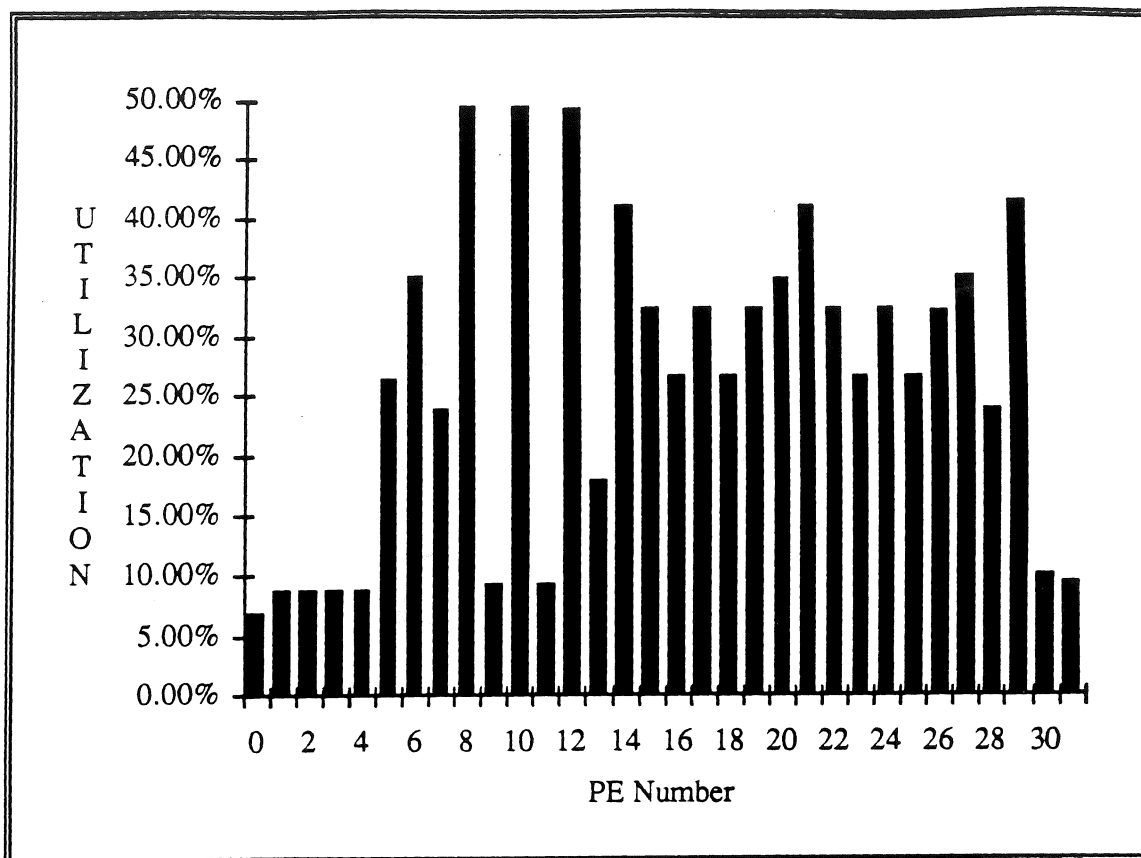


FIGURE 4.13. EXECUTION UNIT UTILIZATION (16 X 16 SIMPLE ON 32 PEs).

When a medium sized problem is run the load balance is better, see Figure 4.14.

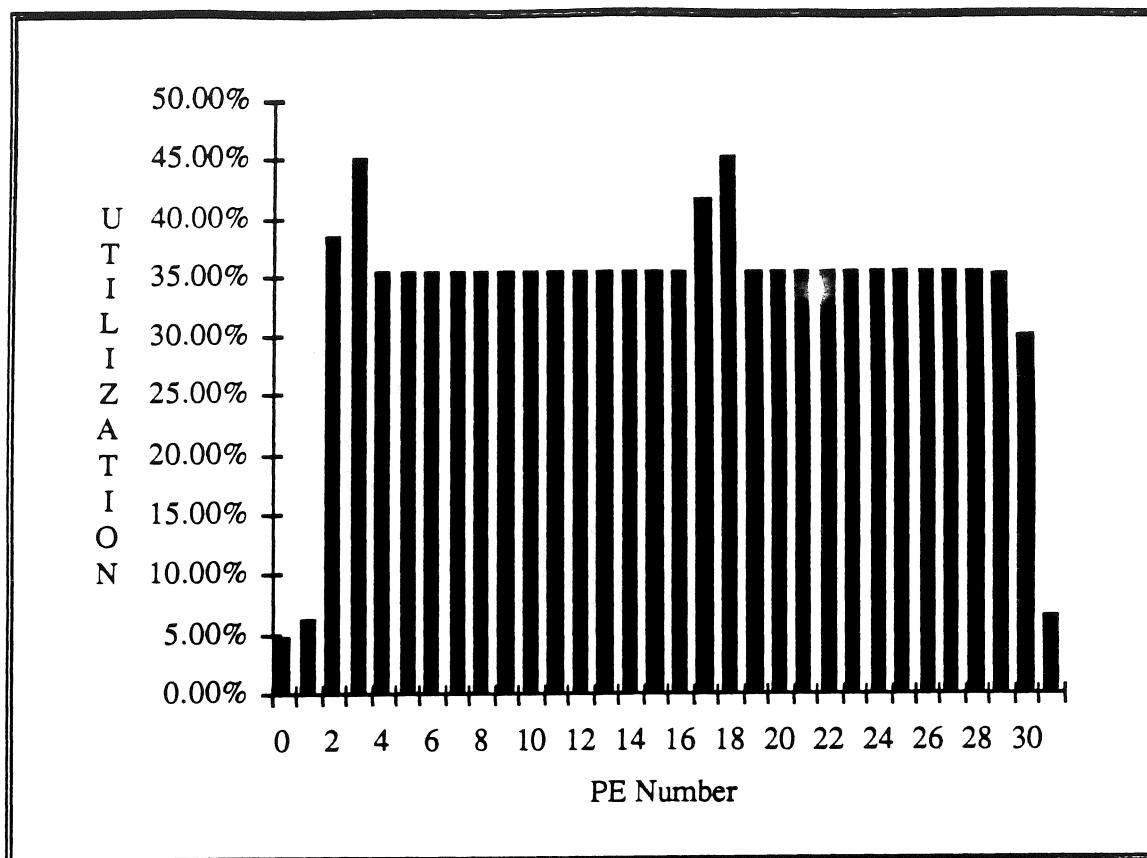


FIGURE 4.14. EXECUTION UNIT UTILIZATION (32 x 32 SIMPLE ON 32 PEs).

Finally, when a large problem is run the load balance is quite flat on 32 PEs. This is a much more realistic size problem for scientific programs.

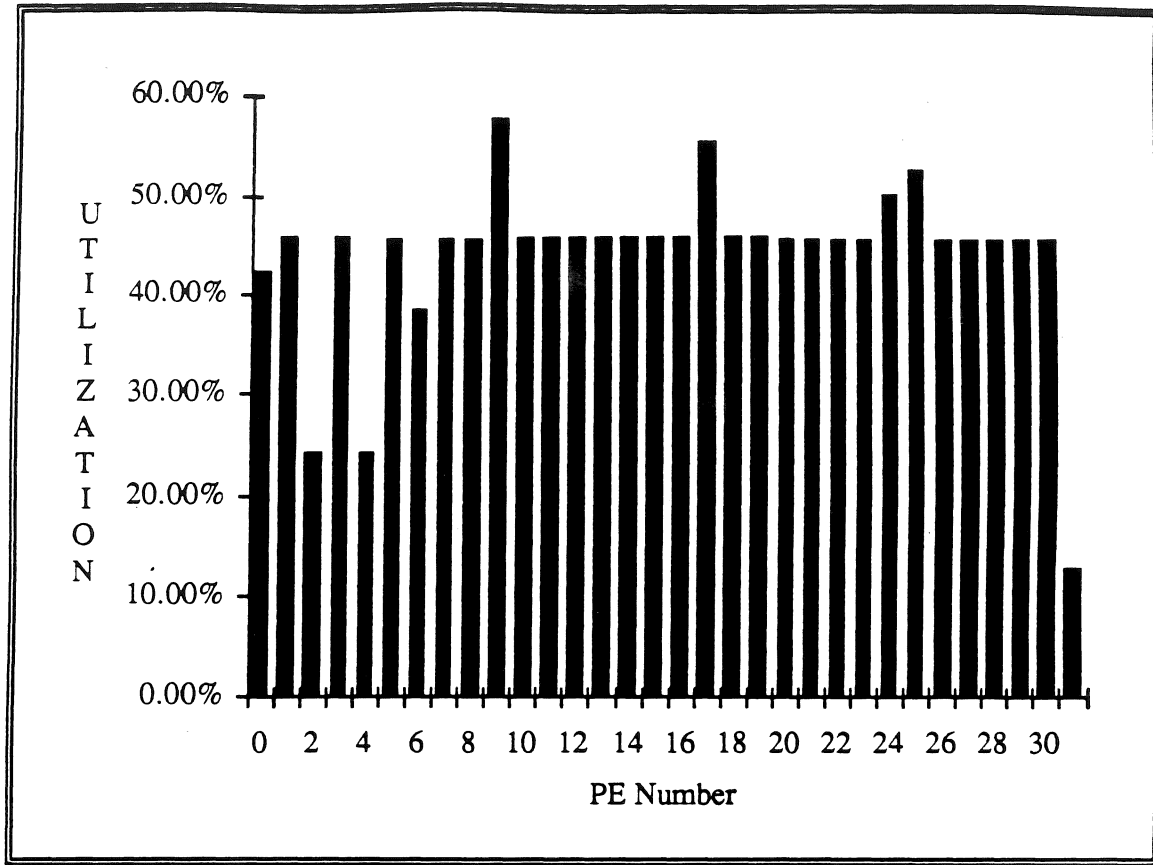


FIGURE 4.15. EXECUTION UNIT UTILIZATION (64 X 64 SIMPLE ON 32 PEs).

**Parallelization Overhead.** The table below shows dynamic instruction counts for different problem sizes. All of these counts are for the 32 PE system (worst case).

Problem Size	Work Instructions	Total Instructions	Percent Overhead
16 x 16	49,714	56,839	12.54%
32 x 32	215,546	240,288	10.30%
64 x 64	907,711	993,322	8.62%

TABLE 4.4. PERCENT OVERHEAD INSTRUCTIONS FOR SIMPLE.

The percentage of overhead in SIMPLE is smaller than for Matrix Multiply. This is due to the size of the for-loop bodies being larger in SIMPLE (see CONDUCTION code above).

Keeping the parallelization overhead low is central to efficient parallel processing.

**Efficiency Comparison.** For a 32 x 32 input CONDUCTION takes 0.9 seconds on a single iPSC/2 PE. This was measured by compiling the standard 'C' version of SIMPLE, then running one iteration of the main loop, and subtracting the setup time (mainly the GENERATE routines). CONDUCTION is used here rather than the total SIMPLE because of the function calls and other operations between the major routines which do not appear in the total. This would cause the single iPSC/2 PE time to be inflated compared to the PODS time. However, the PODS Simulator still estimates that the program would run in 1.72 seconds. This is once again within 100% of the commercial version, and shows that PODS is not grossly inefficient. This has been found to be true on all of the test cases.

**Scalability.** This is the true test of a parallel system — how well does it speed-up for real-world type problems. Figure 4.16 shows the speed-up of different size SIMPLE runs. For comparison the speed-up Pingali and Rogers obtained for a 64 x 64 run is also plotted. [P&R90]

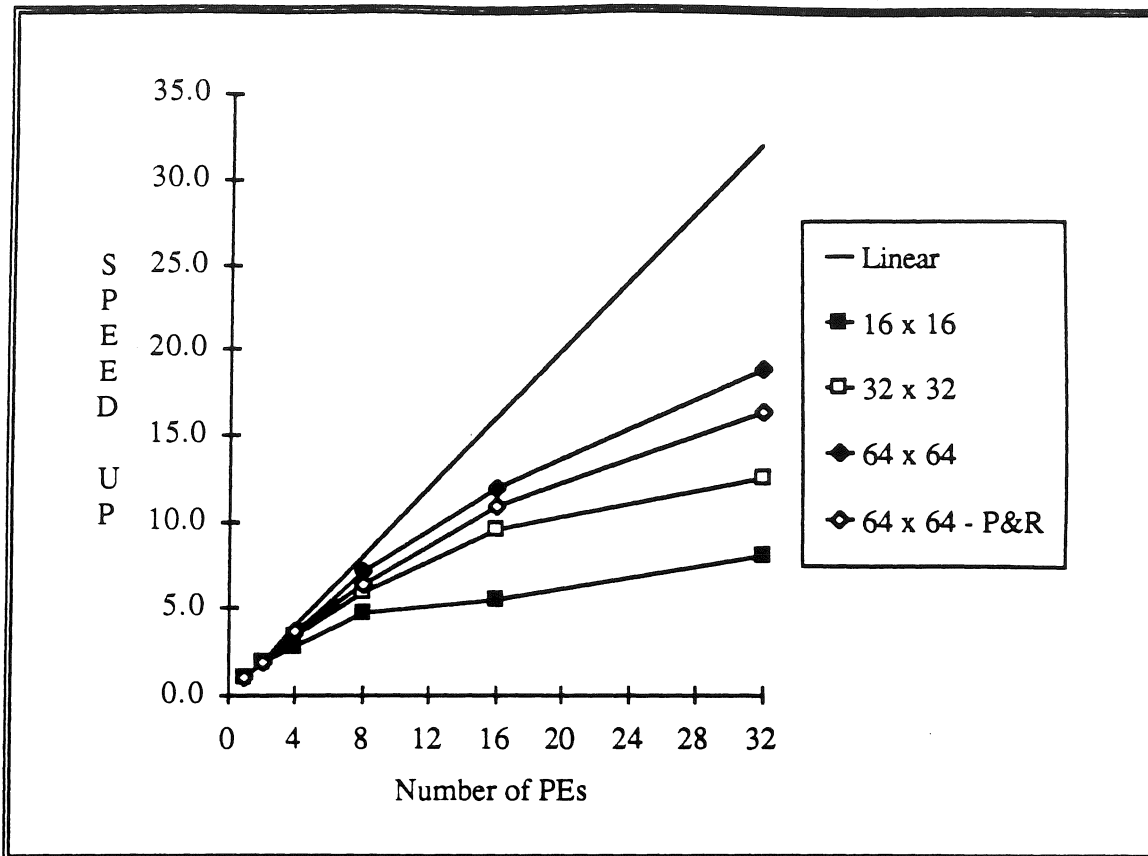


FIGURE 4.16. SPEED-UP OF SIMPLE.

For the small 16 x 16 case, PODS tops out at a speed-up of 8.1. Eventually the parallelization overhead would cause this small problem to even run slower as the number of PEs increased. There is not yet a way for PODS to determine when a problem is so small that it should not be spread across all of the available PEs. PODS either runs the SP in place or distributes it across all PEs.

For the 32 x 32 case, speed-up tops out at 12.4. This order-of-magnitude speed-up is quite acceptable and is comparable to the speed-up obtained by Pingali & Rogers on the 64 x 64 case.

The 64 x 64 problem size is much more likely for scientific coding and is thus a better gauge for the success of PODS in parallelizing scientific code. For the 64 x 64 case, PODS

is able to spread the work efficiently across all of the PEs, achieving a speed-up of 18.9 on 32 PEs. It is unlikely that a greater speed-up would occur on 64 PEs since the average Execution Unit utilization is 44%. And based upon the 16 x 16 case, once the Execution Unit utilization drops below 40% little speed-up is possible for SIMPLE. This speed-up is better than Pingali & Rogers' 64 x 64.

The reason PODS performs better is due to the remote caching in PODS. Pingali & Rogers send the data values to the PE where they are needed. This causes a large number of individual messages to be sent, thus their extreme interest in batching messages. In PODS the individual messages are also batched, however array data is passed a page at a time. The remote caching allows PEs to access array elements as if they were local. Using this locality of reference, PODS is able to read over 187,000 data elements from caches in the CONDUCTION routine alone. This concept is heavily supported by the single assignment nature of ID Nouveau [Roy90]. Single assignment allows PODS to ignore cache coherency problems and to efficiently partition the arrays.

#### **4.4. Summary**

This chapter discussed the PODS Simulator and some results of interesting benchmark problems. The simulation is event-driven and is based on SMPL. The timing assumptions were based on the iPSC/2 computer system. The simulator is like an emulator in that it actually executes the code at the instruction level. Each different type of instruction takes different amount of simulated time. Thus a reasonable estimated of the actual run-time was achieved.

Different measures of effectiveness were used to evaluate PODS on the classic Matrix Multiply problem and on the more complex SIMPLE hydrodynamics problem. In all cases the parallelization overhead was low and the support units did not slow down the Execution Unit. It is important to note that the single PE time for PODS was not grossly inefficient

when compared to commercial 'C' systems when run on the same size CPU. This gives the speed-up computations a solid base execution time from which to work.

For Matrix Multiply (a small problem) the Execution Unit utilization was high (80% and greater) until the available iteration level parallelism was used up. When this occurred the load balance was driven way down. Half the PEs were being utilized at 80% and half at less than 3%. This unequal load balance caused the speed-up to end abruptly. As the problem size was increase this unequal load balance was staved off until greater and greater numbers of PEs were made available. This points to a future enhancement — PODS needs to know how many PEs to distribute a problem across. Currently PODS decides to distribute or not to distribute, there is no algorithm for gauging when a problem is so small that all of the available PEs should not be used.

For comparison purposed the 10 x 10 Matrix Multiply speed-up predicted for Iannucci's Hybrid Architecture is included. Iannucci is able to achieve impressive speed-up on small problems because of the finer grain. PODS is designed to exploit iteration level parallelism, and there is not that much available on the 10 x 10 Matrix Multiply. Iannucci's system requires new hardware components while PODS is designed for off-the-shelf components. It will be interesting to see how cost effective it is once it is built.

The more complex SIMPLE hydrodynamics program showed how well PODS performs on scientific programs. Being much more complex, SIMPLE contains much of the iteration level parallelism PODS is designed to exploit. The Execution Unit utilization was not as high for SIMPLE as it was for Matrix Multiply. This is to be expected, the simple, regular nature of Matrix Multiply is much easier to distribute evenly. However, with SIMPLE there is not the abrupt load imbalance that Matrix Multiply encounters. The complexity in SIMPLE allows speed-ups to continue raising even though the Execution Unit utilization is only 50%.

Pingali and Rogers have run the ID version of SIMPLE on an iPSC/2. Their results were quite good, but PODS is able to achieve an even greater speed-up. This is due to the individual message passing which they use. Pingali and Rogers' static scheduling allows one PE to know when another PE needs the value just calculated. They then send this value to the needy PE. Recognizing early on that this would cause numerous messages, they batched messages together in order to reduce communication costs. In PODS, individual messages are also batched, however, array references are handled differently. The remote caching of array values allows the locality of reference to be exploited. This can be a major source of speed-up, on the larger SIMPLE runs over 187,000 cached array reads occur out of the 210,000 total reads. This, in conjunction with the efficient distribution of work, allows PODS to achieve even greater speed-ups.



## CHAPTER 5

### Conclusions

This chapter presents the related research projects at other universities and some of the advantages and disadvantages of single assignment, followed by a summary of the conclusions found in this research. The areas for future research are discussed as well.

#### 5.1. Related Work

All of these research project recognize the need to integrate the Dataflow and von Neumann models of computation. Different compiler technology and hardware are used with various levels of success.

##### 5.1.1. Iannucci's Hybrid Architecture

The Dataflow / von Neumann Hybrid Architecture proposed by Iannucci [Ian88] differs from PODS in that it requires a new CPU specifically designed for the architecture, where PODS uses off-the-shelf components. A compiler is used to partition the program into *scheduling quantum*s [Ian88]. Scheduling quantum are collections of dataflow instructions subject to sequential execution. The Method of Dependency Sets is used to generate these scheduling quantum without deadlock.

Like PODS this approach executes only one thread at a time, while blocking others which are awaiting values. Given that the scheduling quantum are usually less than five instructions long, the need for a fast context switch is high. In PODS the average SP is over 25 instructions long. Iannucci's model predicts that 23,569 instructions would be executed for a 10 x 10 Matrix Multiply [Ian88]. For the same program PODS only executes 15,072 instructions; thus each PODS instruction does 1.5 times the work. Together these reduce the need for a fast context switch significantly.

Iannucci's ability to exploit a fair amount of parallelism from a 10 x 10 Matrix Multiply (nearly 20 times speed-up) is impressive. It will be interesting to see how cost effective the new architecture with the its new CPU will be.

### 5.1.2. Gao's Hybrid Machine

At McGill University Guang Gao has been working on a hybrid machine which basically adds control-flow to dataflow [Gao90]. This is achieved with a signal graph which is similar to the PODS routing table. However, Gao does not use the concept of sequential threads. Instead his granularity is a single instruction. He makes use of the pipelined architectures available for von Neumann execution, but the next instruction is not necessarily stored right after the present one. A signal graph indicates which instruction will be loaded next. This had advantages and disadvantages.

The flexibility of this approach is very high. Depending upon the signal graph the system will function as a dataflow machine or as a von Neumann machine. This can change back and forth from instruction to instruction. The amount of overhead this incurs is unknown. There is also the problem of a completely new hardware architecture, which may make this approach intractable from a cost standpoint.

Another difference from PODS is the use of SISAL [MSS85] rather than ID Nouveau. SISAL has a number of good concepts, however, any parallel architecture will have a difficult time supporting the dynamic arrays, the update operator, and the recursive function calling required. These force the memory manager to be highly efficient at allocating and deallocating space. Additionally, the overall machine performance depends on a careful layout of these dynamic arrays to reduce memory contention, a difficult problem at best.

### 5.1.3. Alfalfa

The Alfalfa system [G&H89] is mainly concerned with different dynamic scheduling techniques and does not address the problem of distributing large data structures, such as arrays. They achieve some impressive results for problems involving little to no data communication, however, for Matrix Multiply, they see poor speed-up results. They claim that this is due to the slow message passing time of the iPSC, but PODS shows that a data cache combined with simple scheduling can overcome the long latencies associated with accessing remote data.

### 5.1.4. Decoupled Multilevel Dataflow Model

The Decoupled Multilevel Dataflow Model at USC [E&G90] is a macro-dataflow project aimed at the exploitation of virtual space, multilevel memory hierarchies, and RISC design principles. The variable resolution (different size macro operators) allows programs to be matched with the system. With vector and larger operators the standard von Neumann optimizations can be used.

This system uses SISAL as Gao does and will have some of the same difficulties. The problem is compounded by the need for vector extensions to SISAL so that the programmer can tell the system what to vectorize. This places the additional burden of specifying parallelism on the programmer.

The amount of overhead the system incurs, and the cost effectiveness of building a new CPU have yet to be determined. It is possible that this variable resolution will be very effective at matching a programs inherent parallelism to the processor's capabilities.

### **5.1.5. Dynamic Structured Dataflow**

The Dynamic Structured Dataflow project [Got90] at the Israel Academy of Sciences Foundation for Basic Research is working on an execution model with arbitrarily fine granularity. This approach is similar to the original PODS concept of SCSs, but here the scheduling and resource allocation is done dynamically, where the original PODS attempted to determine the best groupings at compile time. The current PODS uses iteration level parallelism rather than SCS threads.

In Dynamic Structured Dataflow, the need for a fast context switch is very high, and a fair amount of effort has been put into the Parallel Work Conveyor [G&K80] which satisfies this requirement. Currently the project is working on an architectural specification and simulator. It will be interesting to see how large of a granularity the system produces and how well the Parallel Work Conveyor operates. These will be very important to efficient execution.

### **5.1.6. Pingali and Rogers' Compiler**

At Cornell Pingali and Rogers have been working on a compiler [P&R90, R&P88, R&P89] which will take ID Nouveau and compile it into 'C' for execution on an Intel iPSC/2. Their language (ID Nouveau) and architecture (iPSC/2) are the same as the current PODS, however from there on the approaches differ significantly.

In PODS there is an underlying execution model which is very different from that used in standard von Neumann processors. Pingali and Rogers have stayed with the standard von Neumann model. This places PODS closer to true dataflow, and, as such, is better able to exploit irregular parallelism.

Pingali and Rogers exploit the locality of data reference in large programs, however they do not have anything analogous to the remote array caching in PODS. It is conceivable that this could be added to their compiler. It is unclear how this would affect their speed-up.

One of the most critical elements of their work is the batching of messages. In their system a PE knows when and where to send a data value to another PE. This would create a large number of messages if it were not for the batching which is used. It would be interesting to incorporate some of their ideas into PODS.

Their performance on an iPSC/2 running SIMPLE is quite good. This seems to be due to the clear and concise nature of a compiler which takes ID Nouveau and produces 'C' code for a parallel machine. This approach has stimulated the desire to build such a compiler for PODS.

## **5.2. Advantages and Disadvantages of Single Assignment**

The proper use of single assignment is central to PODS. The main advantage of single assignment is its ability to implicitly expose parallelism. With single assignment only the definitional data dependencies restrict parallelism. There are no extra dependencies based upon storage location naming. This is critical for parallel program synchronization, otherwise innocuous timing bugs can occur.

Exposing this much parallelism can cause resource overloading. The reality of physical machines requires that the parallelism be throttled by the operating system. This throttling can take significant overhead. This disadvantage is minimized in PODS by the large granularity of the SPs.

An oft criticized feature of implicit parallelism is the inability of a programmer to override the synchronization when he knows a better way. This lack of control is unsettling to

many parallel programmers. This is because the current state of parallel programming art requires the programmer to take control or take his chances. See [Kar87] for a look at parallel programming today.

Another danger is too much copying of intermediate array elements. If an update or replace array operator is available, grossly inefficient programs can be written. Aids for detecting this type of inefficiency are needed.

In an architectural sense single assignment has some problems. The fact that memory is finite means that memory locations will have to be written over. i.e. a variable's definition (its one and only assignment) will not exist forever. This presents the problem of knowing *when* a variable is no longer needed by *any* of the processors.

The final factor is the ease (or difficulty) to program in a single assignment language. See [ANP87a] for a convincing argument as to the ease of single assignment programming.

The combination of single assignment, areas-of-responsibility, and caching leads to low communication overhead and well-balanced loads when applied to the majority of the Livermore Loops [BNR89b, LLL83], Matrix Multiply, and SIMPLE [CH&R]. Single assignment permits the exploitation of large numbers of PEs automatically.

Synchronization problems are solved through the adoption of the single assignment policy. By segmenting array writes using the area-of-responsibility concept, all PEs perform roughly the same number of remote accesses. These two concepts allow caching to be implemented without extensive communication, and caching is central to reducing remote array accesses.

### 5.3. Summary

This dissertation has discussed the Process-Oriented Dataflow System and its suitability for running scientific programs on distributed-memory MIMD machines. The partitioning and

distribution algorithms, along with their underlying principles, have been examined and discussed. The logical implementation which was used in the simulations has been presented along with the support software suite. The remote array caching scheme used has been described. The event-driven simulation was explained and the results of experiments with Matrix Multiply and SIMPLE were examined.

It has been found that PODS can achieve speed-ups of nearly 20 times on large versions of SIMPLE. This surpasses the speed-up of other approaches on similar architectures. This speed-up is sufficient to warrant recoding of large scientific programs from FORTRAN or C to ID Nouveau; usually a 10 times speed-up is considered large enough. When large scientific programs are written, they are usually written by scientists, not computer programmers. ID Nouveau will be easier for scientists to use because of its declarative nature. Combine this with the automatic parallelization in PODS and this approach is much more productive for parallel scientific programming.

The basic PODS model of execution with its ability to “degenerate” to a von Neumann machine as necessary, has the following advantages:

- the number of tokens through matching store and across the routing network in general is reduced due to the use of SPs.
- instruction fetch/execution is as efficient as in a typical von Neumann architecture, especially when loops run in-place.
- programmers may ignore such parameters as the number of available PEs — the automatic partitioning allows a higher level of abstraction.

- SPs are long and execute an average of 70 instructions before a context switch — reducing context switches greatly increase the efficiency and scalability of the system.

The mechanism for distributing arrays in PODS not only allows for larger arrays than normally available in such machines, but it also takes advantage of locality of reference. The remote array caching scheme further enhances the locality.

Both SIMPLE and Matrix Multiply have been used as performance measures. Matrix Multiply is a good measure because it has several interesting properties:

- there are multiple code-blocks
- a new array must be dynamically allocated and distributed
- there is a loop-carried dependency in the innermost loop
- the two input arrays, A and B, have different access patterns
- the sizes of the input arrays are not known at compile time

Matrix multiply also forms the basis for many important scientific algorithms such as: LU decomposition, convolution, and the Fast-Fourier Transform. SIMPLE is a good measure because of its size (nearly 1000 FORTRAN instructions) and complexity (numerous SPs and function calls with many dynamic array). SIMPLE was also designed as a benchmark program by one of the largest users of supercomputers, Lawrence Livermore Laboratory.

In summary, PODS allows MIMD machines to exploit vector and data parallelism efficiently, while still providing the flexibility of distributed-memory MIMD machines.



## 5.4. Future Research

This is the first step in the development of a new approach to parallel processing. To further understand the advantages and disadvantages of this approach, a variety of issues need to be examined:

- Reduction operators are not fully exploited. How can vector to scalar operations be implemented? Current ideas include a mechanism to allow collection of subrange results.
- How well can scientific programmer use ID Nouveau and PODS?
- How well does PODS execute non-scientific code?
- Should the programmer be able to specify *any* partitioning parameters?
- How well does PODS run on real hardware?

To investigate these issues two major projects are in the works: the first is HyperPODS, an implementation of PODS on an Intel iPSC/2; the second is a PODS compiler which would take ID Nouveau and compile it directly for a particular implementation of PDS (e.g., HyperPODS).

### 5.4.1. HyperPODS

HyperPODS is currently being build using the logical implementation described herein. So far the logical implementation has served well, but changes will undoubtedly be necessary. The issues below will have to be addressed:

- Register Allocation — the passing of tokens internal to an SP will be done through registers.
- Presence Bits — these are not supported in the hardware, but are necessary for I-structures.
- Blocking of an SP — this will have to be done at certain instructions and not others. The efficiency of this is important to context switch times.
- Matching Store — this support unit is the most utilized. It must be efficient.
- Routing Unit — the batching of messages will have to be done in the CPU and the interaction with the Direct-Connect Module is critical to the scalability of the system.
- Array Manager — the enqueueing and dequeuing of reads will require dynamic memory allocation.
- Resource Limitations — PODS may have to be throttled down to prevent deadlock. The exact implementation of this is unclear.

These are just a few of the research issues which the PODS team will be addressing in HyperPODS.

#### **5.4.2. PODS Compiler**

The development of HyperPODS and the success of Pingali and Rogers has lead to renewed interest in a PODS Compiler. The GITA Compiler currently in use is written in LISP and takes up a large amount of memory when executing. More importantly there are

optimizations which PODS can use which are not in the GITA Compiler (e.g., scalar expansion). The PODS Compiler would replace the GITA Compiler and the PODS Translator. The PODS Partitioner could be incorporated, but this is not necessary.

Once HyperPODS and a PODS Compiler for it are finished, the complete PODS system can be sent to beta-test sites at facilities which have an iPSC/2 and are interested in getting more scientific programs to be parallel. This will be the true test of the PODS concept.

## CHAPTER 6

### References

- [A&O85] S. Allan, R. Oldehoeft. HEP SISAL: Parallel Functional Programming. In *Parallel MIMD Computation: the HEP Supercomputer and Its Applications*. J. S. Kowalik, Eds. (MIT Press, Cambridge, MA, 1985), pp. 123-150.
- [A&K87] R. Allen, K. Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Trans. Prog. Lang. and Sys.* V9, n4 (1987), pp. 491-542.
- [A&C86] Arvind, D. E. Culler. Dataflow Architectures. *Annual Reviews in Computer Science VI*, (1986), pp. 225-253.
- [A&E88] Arvind, K. Ekanadham. Future Scientific Programming on Parallel Machines. *J. Parallel Dist. Comp.* V5, n5 (1988), pp. 460-493.
- [AGP78] Arvind, K. P. Gostelow, W. Plouffe. An Asynchronous Programming Language and Computing Machine. *Technical Report 114a* (December 1978), Department of Information and Computer Science, University of California, Irvine.
- [A&N87] Arvind, R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *MIT Technical Report Computation Structures Group Memo 271* (March 1987), Laboratory for Computer Science, MIT.

- [ANP87a] Arvind, R. S. Nikhil, K. K. Pingali. I-Structures: Data Structures for Parallel Computing. *MIT Technical Report Computation Structures Group Memo 269* (February 1987), Laboratory for Computer Science, MIT.
- [ANP87b] Arvind, R. S. Nikhil, K. K. Pingali. ID Nouveau Reference Manual Part II: Operational Semantics. *MIT Technical Report* (April 1987), Laboratory for Computer Science, MIT.
- [ANP89] Arvind, R. S. Nikhil, K. K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM TOPLAS VII, n4* (1989), pp. 598-632.
- [Bab84] R. G. Babb. Parallel Processing with Large-Grain Data Flow Techniques. *IEEE Computer* (July 1984), pp. 55-61.
- [Bic87] L. Bic. A Process-Oriented Model for Efficient Execution of Dataflow Programs. *Proc. 7th Int'l Conf. on Distributed Computing Systems* (1987), pp. 744-758.
- [Bic90] L. Bic. A Process-Oriented Model for Efficient Execution of Dataflow Programs. *Journal of Dist. and Parallel Computing VMarch*, (1990), pp. 15-38.
- [BNR89a] L. Bic, M. D. Nagel, J. M. A. Roy. Automatic Data/Program Partitioning Using the Single Assignment Principle. *Technical Report #89-09* (January 1989), University of California, Irvine.

- [BNR89b] L. Bic, M. D. Nagel, J. M. A. Roy. Automatic Data/Program Partitioning Using the Single Assignment Principle. *Supercomputing '89* (1989), pp. 551-556.
- [BNR90a] L. Bic, M. D. Nagel, J. M. A. Roy. Executing Matrix Multiply on a Process Oriented Dataflow Machine. *Technical Report 90-08* (April 1990), Department of ICS, University of California, Irvine.
- [BNR90b] L. Bic, M. D. Nagel, J. M. A. Roy. On Array Partitioning in PODS. In *Advanced Topics in Data-Flow Computing*. J. L. Gaudiot, L. Bic, Eds. (Prentice Hall, Englewood Cliffs, New Jersey, 1990), pp. 305-325.
- [B&E87] R. Buehrer, K. Ekanadham. Incorporating Data Flow Ideas into von Neumann Processors for Parallel Execution. *IEEE Trans. Comp. VC-36, n12* (1987), pp. 1515-1521.
- [Bur88] D. Burns. Loop-Based Concurrency Identified as Best at Exploiting Parallelism. *Computer Technology Review* (Winter 1988), pp. 19-23.
- [C&K88] D. Callahan, K. Kennedy. Compiling Programs for Distributed-Memory Multiprocessors. *Jour. of Supercomputing V2*, (1988), pp. 151-169.
- [CH&R] W. P. Crowley, C. P. Henderson, T. E. Rudy. The SIMPLE Code. *UCID 17715* (February 1978), Lawrence Livermore Laboratory.
- [DFL89] D. DeForest, A. Faustini, R. Lee. Hyperflow. *The Third Conference on Hypercube Concurrent Computers and Applications* (1989), pp. 482-488.

- [Den75] J. B. Dennis. First Version of a Dataflow Procedure Language. *Machine Tech. Memorandum 61* Cambridge, MA. M.I.T.
- [Dun88] T. H. Dunigan. Performance of a Second Generation Hypercube. *Technical Report ORNL/TM-10881* (November 1988), Oak Ridge National Laboratory.
- [E&G90] P. Evtipidou, J. L. Gaudiot. The USC Decoupled Multilevel Data-Flow Execution Model. In *Advanced Topics in Data-Flow Computing*. J. L. Gaudiot, L. Bic, Eds. (Prentice Hall, Englewood Cliffs, New Jersey, 1990), pp. pp. 347-380.
- [Gao90] G. R. Gao. A Flexible Architecture Model for Hybrid Data-Flow and Control-Flow Evaluation. In *Advanced Topics in Data-Flow Computing*. J. L. Gaudiot, L. Bic, Eds. (Prentice Hall, Englewood Cliffs, New Jersey, 1990), pp. 327-346.
- [G&H89] B. Goldberg, P. Hudak. Implementing Functional Programs on a Hypercube Multiprocessor. *The Third Conference on Hypercube Concurrent Computers and Applications* (1989), pp. 489-504.
- [G&K80] A. Gottlieb, C. P. Kruskal. A Data Motion Algorithm. *Technical Report Ultracomputer Note 7* (January 1980), Courant Institute of Mathematical Sciences.

- [Got90] I. Gottlieb. Work Distribution in the DSDF Architecture. In *Advanced Topics in Data-Flow Computing*. J. L. Gaudiot, L. Bic, Eds. (Prentice Hall, Englewood Cliffs, New Jersey, 1990), pp. 381-409.
- [H&B84] K. Hwang, F. A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, New York, New York, 1984.
- [Ian88] R. A. Iannucci. A Dataflow/von Neumann Hybrid Architecture. *Dissertation* (1988), MIT.
- [IEEE89] IEEE. The Computer Spectrum: A perspective on the Evolution of Computing. *IEEE Computer* (November 1989), pp. 57-68.
- [Ins91] IEEE. Intel's Newest Supercomputer. In *The Institute*, Eds., 1991, pp. 6.
- [iPSC89] *IPSC User's Guide*, Intel, Portland, Oregon, 1989.
- [K&T88] M. Kallstrom, S. S. Thakkar. Programming Three Parallel Computers. *IEEE Software* (January 1988), pp. 11-22.
- [Kap86] I. Kaplan. A Large-Grain Dataflow Architecture. *Workshop on Future Directions in Computer Architecture and Software* (1986), pp. 131-138.
- [Kar87] A. H. Karp. Programming for Parallelism. *IEEE Computer* (May 1987), pp. 43-57.



- [K&B88] A. H. Karp, R. G. B. II. A Comparison of 12 Parallel Fortran Dialects. *IEEE Software* (September 1988), pp. 52-67.
- [LLL83] L. L. N. Laboratory. FORTRAN KERNELS: MFLOPS, V22/DEC/86 mf328 (Regents of the University of California, Livermore, CA., 1983).
- [Lan65] P. J. Landin. A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part I. *Comm. ACM V8, n2* (1965), pp. 89-101.
- [L&G86] J. W. Liu, A. Grimshaw. A Distributed System Architecture Based on Macro Dataflow Model. *Workshop on Future Directions in Computer Architecture and Software* (1986), pp. 155-162.
- [Mac87] M. H. MacDougall. *Simulating Computer Systems: Techniques and Tools*, MIT Press, Cambridge, MA, 1987.
- [MSS85] J. R. McGraw, *et al.* SISAL, Streams and Iteration in a Single Assignment Language. *Language Reference Manual, Ver. 1.2 M-146* (1985), Lawrence Livermore National Laboratory.
- [Nik87a] R. S. Nikhil. ID Nouveau Reference Manual Part I: Syntax. *MIT Technical Report* (April 1987), Laboratory for Computer Science, MIT.
- [Nik87b] R. S. Nikhil. ID World Reference Manual (for Lisp Machines). *MIT Technical Report* (April 1987), Laboratory for Computer Science, MIT.

- [Nik88] R. S. Nikhil. ID Reference Manual - Version 88.1. *MIT Technical Report Computation Structures Group Memo 284* (August 1988), Laboratory for Computer Science, MIT.
- [N&A89] R. S. Nikhil, Arvind. Can Dataflow Subsume von Neumann Computing? *16th Int'l Computer Architecture Conference* (1989), pp. 262-272.
- [P&W86] D. A. Padua, M. J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Comm. of ACM V29, n12* (1986), pp. 1184-1201.
- [P&B90] C. M. Pancake, D. Bergmark. Do Parallel Languages Respond to the Needs of Scientific Programmers? *IEEE Computer* (December 1990), pp. 13 - 23.
- [Pap88] G. M. Papadopoulos. Implementation of a General-Purpose Dataflow Multiprocessor. *Technical Report TR-432* (August 1988), MIT Laboratory for Computer Science.
- [P&R90] K. Pingali, A. Rogers. Compiler Parallelization of SIMPLE for a Distributed Memory Machine. *TR 90-1084* (January 1990), Department of Computer Science, Cornell University.
- [R&P88] A. Rogers, K. Pingali. Process Decomposition Through Locality of Reference. *Technical Report TR 88-935* (August 1988), Department of Computer Science, Cornell University.

- [R&P89] A. Rogers, K. Pingali. Compiling Programs for Distributed Memory Architectures. *4th Hypercube Concurrent Computers & Applications Conference* (1989), pp. 529-542.
- [Roy90] J. M. A. Roy, M. D. Nagel, L. Bic. Partitioning Declarative Programs into Communicating Processes. *Supercomputing '90* (1990), pp. 846-855.
- [S&H87] B. Shirazi, A. R. Hurson. A Large/Fine Grain Parallel Dataflow Model and its Performance Evaluation. *1987 National Computer Conference* (1987), pp. 119-126.
- [Smi85] B. Smith. The Architecture of HEP. In *Parallel MIMD Computation: the HEP Supercomputer and Its Applications*. J. S. Kowalik, Eds. (MIT Press, Cambridge, MA, 1985), pp. 41-58.
- [Smi81] B. J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *Society of Photo-Optical Instrumentation Engineers V298 Real-Time Signal Processing IV*, (1981), pp. 241-248.
- [Tra86] K. R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. *MIT Technical Report NEED* (August 1986), Laboratory for Computer Science, MIT.
- [U&Z89] T. Ungerer, E. Zehendner. A parallel Programming Language Directed Towards Top-Down Software Development. *International Conference on Parallel Processing* (1989), pp. 122-125.

- [Veg88] S. R. Vegdahl. Architectures That Support Functional Programming Languages. In *Computer Architecture: Concepts and Systems*. V. M. Milutinovic, Eds. (North-Holland, New York, NY, 1988), pp. 405-453.
- [W&A85] W. W. Wadge, E. A. Ashcroft. *Lucid, the Dataflow Programming Language*, Academic Press, London, 1985.
- [Z&U87] E. Zehendner, T. Ungerer. The ASTOR Architecture. *7th International Conference on Distributed Computing Systems* (1987), pp. 424-430.

## Appendix A: Range Filter Algorithms

This appendix presents the different range filter algorithms used in PODS. There are three base algorithms and three parameterizations used to generate a specific range filter.

When a level of a nest (say  $i_a$ ) is distributed, the range filter needs to consider all of the indices *above* it,  $i_1$  to  $i_{a-1}$ . This produces three different base algorithms in the current PODS. The first base algorithm is the most common, and uses only the first level index, see Figure A.1 below. This range filter is the most common because PODS will distribute the outermost level whenever possible.

```
1  m = 0
2  if m > interval count of master array then exit
3  set i to the maximum of the beginning of the interval
   and the loop beginning
4  if i is not in the interval or the first element of this
   dimension is not owned then increment m and goto 2
5  if i is within the loop bounds then set continue to TRUE
   and send i and continue into the loop body
   else increment m and goto 2
6  if continue is TRUE do the loop body else goto 9
7  true part of loop body
8  if new_i is within loop bounds set continue to TRUE,
   send i and continue into the loop body, and goto 4
   else set continue to FALSE, send i and continue into the
   loop body, and goto 6 (with i set to new_i)
9  false part of loop body
```

FIGURE A.1. BASE RANGE FILTER ALGORITHM FOR OUTERMOST LEVEL DISTRIBUTION.

The general algorithm functions by repeatedly extracting ranges from the array boundary table. While within the range, the filter passes indices for elements within that range. The filter also keeps the loop alive by sending a continue token to the loop switch until all ranges have been exhausted. In the figure above,  $m$  is just some variable used to count the

intervals;  $i$  is the loop index, and *continue* is the signal to the loop body telling it whether to continue or not.

The next base algorithm is for loops which are distributed at the second outermost level. In this case the range filter must consider two indices,  $i$  and  $j$ . Figure A.2 below shows this algorithm. Notice that it is only slightly different the first case; line #3 is added and  $j$  rather than  $i$  is checked in lines #4 - #10.

```

1  m = 0
2  if m > interval count of master array then exit
3  if i is not in interval m then increment m and goto 2
4  set j to the maximum of the beginning of the interval
   and the loop beginning
5  if j is not in the interval or the first element of this
   dimension is not owned then increment m and goto 2
6  if j is within the loop bounds then set continue to TRUE
   and send j and continue into the loop body
   else increment m and goto 2
7  if continue is TRUE do the loop body else goto 10
8  true part of loop body
9  if new_j is within loop bounds set continue to TRUE,
   send j and continue into the loop body, and goto 5
   else set continue to FALSE, send j and continue into the
   loop body, and goto 7 (with j set to new_j)
10 false part of loop body

```

FIGURE A.2. BASE RANGE FILTER ALGORITHM FOR SECOND OUTERMOST LEVEL DISTRIBUTION.

The final case handles the situation when the third level of a nest is distributed. Once again this is a simple extension of the first case: adding additional lines to check the additional levels (lines #3 and #4) and checking  $k$  rather than  $i$ . This algorithm can easily be extended to handle further levels once PODS handles arrays with more than three dimensions.

```

1  m = 0
2  if m > interval count of master array then exit
3  if i is not in interval m then increment m and goto 2
4  if j is not in interval m then increment m and goto 2
5  set k to the maximum of the beginning of the interval
   and the loop beginning
6  if k is not in the interval or the first element of this
   dimension is not owned then increment m and goto 2
7  if k is within the loop bounds then set continue to TRUE
   and send k and continue into the loop body
   else increment m and goto 2
8  if continue is TRUE do the loop body else goto 11
9  true part of loop body
10 if new_k is within loop bounds set continue to TRUE,
   send k and continue into the loop body, and goto 6
   else set continue to FALSE, send k and continue into the
   loop body, and goto 8 (with k set to new_k)
11 false part of loop body

```

FIGURE A.3. BASE RANGE FILTER ALGORITHM FOR THIRD OUTERMOST LEVEL DISTRIBUTION.

Once the base algorithm is selected the three parameterizations are applied. These are:

1. Loop direction parameterization, 1 to n vs. n downto 1.
2. Indices parameterization,  $A[i, j]$  vs.  $A[c_i*i+k_i, c_j*j+k_j]$ .
3. Stepsize parameterization, step by 1 vs. step by C.

These parameterizations are independent of each other. The first, loop direction parameterization is quite simple. Lines #1, #2, and #3 need to be replaced as shown in bold in Figure A.4 below. In this way the intervals are accessed in descending order.

Note that the interval counter  $m$  is decremented rather than incremented.

```

1  m = interval count of master array
2  if m < 0 then exit
3  set i to the minimum of the end of the interval
   and the loop end
4  if i is not in the interval or the first element of this
   dimension is not owned then decrement m and goto 2
5  if i is within the loop bounds then set continue to TRUE
   and send i and continue into the loop body
   else decrement m and goto 2
6  if continue is TRUE do the loop body else goto 9
7  true part of loop body
8  if new_i is within loop bounds set continue to TRUE,
   send i and continue into the loop body, and goto 4
   else set continue to FALSE, send i and continue into the
   loop body, and goto 6 (with i set to new_i)
9  false part of loop body

```

FIGURE A.4. RANGE FILTER ALGORITHM FOR STEPSIZE -1.

The second parameterization is for complex indices like  $A[c_i*i+k_i, c_j*j+k_j]$ . The range filter for this situation needs different index check conditions. Figure A.5 shows the algorithm for a second level distribution (along  $j$ ) writing into  $A[c_i*i+k_i, c_j*j+k_j]$ .



```

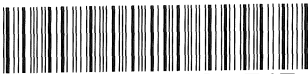
1  m = 0
2  if m > interval count of master array then exit
3  if (c1*i+k1) is not in interval m then increment
   m and goto 2
4  set j to the maximum of the loop beginning and
   (beginning of the interval-kj)/cj
5  if (cj*j+kj) is not in the interval or the first
   element of this dimension is not owned then
   increment m and goto 2
6  if j is within the loop bounds then set continue to TRUE
   and send j and continue into the loop body
   else increment m and goto 2
7  if continue is TRUE do the loop body else goto 10
8  true part of loop body
9  if new_j is within loop bounds set continue to TRUE,
   send j and continue into the loop body, and goto 5
   else set continue to FALSE, send j and continue into the
   loop body, and goto 7 (with j set to new_j)
10 false part of loop body

```

FIGURE A.5. SECOND LEVEL DISTRIBUTION RANGE FILTER FOR  $A[C_1*I+K_1, C_J*J+K_J]$ .

The lines in bold (lines #3 - #5) have different check conditions than those in Figure A.2; this is the only change.

The third parameterization is also quite simple. This handles the case where the stepsize is not 1 nor -1, but some constant  $c$ . Note that this stepsize is important only on the level of the nest which is distributed. Figure A.6 shows the algorithm for a third level distribution with stepsize  $c$ . Note that line #5, in bold, is the only modification.



```
1  m = 0
2  if m > interval count of master array then exit
3  if i is not in interval m then increment m and goto 2
4  if j is not in interval m then increment m and goto 2
5  set k to the (first multiple of C + start of
   loop) > start of interval m
6  if k is not in the interval or the first element of this
   dimension is not owned then increment m and goto 2
7  if k is within the loop bounds then set continue to TRUE
   and send k and continue into the loop body
   else increment m and goto 2
8  if continue is TRUE do the loop body else goto 11
9  true part of loop body
10 if new_k is within loop bounds set continue to TRUE,
   send k and continue into the loop body, and goto 6
   else set continue to FALSE, send k and continue into the
   loop body, and goto 8 (with k set to new_k)
11 false part of loop body
```

FIGURE A.6. RANGE FILTER FOR THIRD LEVEL DISTRIBUTION WITH STEPSIZE C.

As an example consider the loop range: for  $k = 2$  to 30 stepsize 3. Valid values of  $k$  are: 2, 5, 8, 11, 14, 17, 20, 23, 26, and 29. If an interval  $m$ , for a given PE, ran from 6 to 16 inclusive, then  $k$  would start out at 8, and stop at 14.

The three basic algorithms plus the three parameterizations allow PODS to insert the proper range filter at compile time.