MIT/LCS/TM-490

COMPILE-TIME LOOP SPLITTING FOR DISTRIBUTED MEMORY MULTIPROCESSORS

Donald O. Tanguay, Jr.

November 1993

Compile-time Loop Splitting for Distributed Memory Multiprocessors

by

Donald O. Tanguay, Jr.

Submitted to the
Department of Electrical Engineering and Computer Science

May 17, 1993

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering

Abstract

In a distributed memory multiprocessor, a program's task is partitioned among the processors to exploit parallelism, and the data are partitioned to increase referential locality. Though the purpose of partitioning is to shorten the execution time of an algorithm, each data reference can become a complex expression based upon the data partitions. As an attempt to minimize the computation needed for array references, loop splitting can further divide a partitioned loop into segments that allow the code hoisting and strength reduction optimizations. This thesis introduces two methods of loop splitting, *rational* and *interval*. While rational splitting divides the loop into equal-length GCD segments, interval splitting specifies segments as an explicit list of intervals. These two methods have been implemented and studied. Under our execution model, the loop in the algorithms analyzed executes an average of 2 to 3 times faster after loop splitting.

Keywords: loop splitting, code hoisting, strength reduction, peeling, distributed memory multiprocessors, partitioning, array referencing.

Thesis Supervisor: Anant Agarwal
Title: Associate Professor

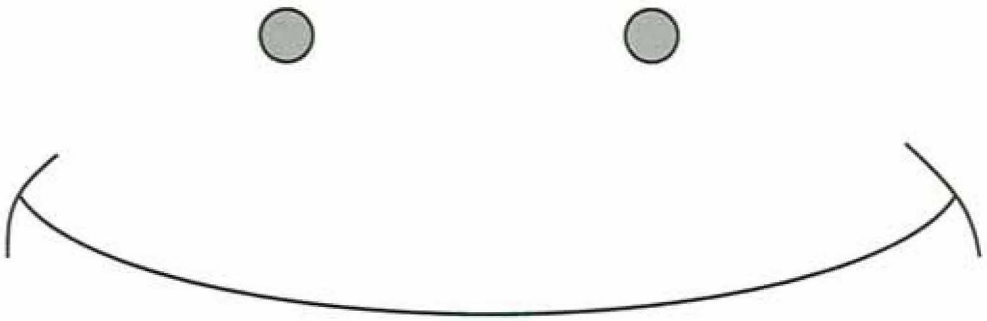
Acknowledgments

There are several people who have contributed to my thesis experience and to whom I extend much appreciation.

I thank Gino Maa for his in-depth explanations at 5 AM about everything from data placement to the operation of CD players; David Kranz for the valuable talks on loop splitting and fixing his bugs; and Anant Agarwal for being an ideal.

More personally, I thank Kate Nguyen for being concerned and reading the final draft; Hooman Davoudiasl for late-night company, comedy relief, and the IHOP experience; Andy Choi for friendship and financial support (before the thesis prize); and Ravi Soundararajan for still being my friend.

I made this happen, and I enjoyed every bit of it.



Contents

1 Introduction 9

1.1 Overview 10

1.2 Previous Application 11

2 Array Management in Multiprocessors 12

2.1 Introduction 12

2.2 Partitioning 14

2.2.1 Task Partitioning 14

2.2.2 Data Partitioning 18

2.3 Optimal Partitioning 19

2.3.1 Optimal Task Partitioning 19

2.3.2 Optimal Data Partitioning 21

2.4 The Problem 21

2.5 Array Referencing 22

2.6 Simple Optimization of Array Calculation 25

2.7 Summary 27

3 Overview of Loop Splitting 28

3.1 Introduction 28

3.2 Two Loop Transformations 29

3.2.1 General Loop Splitting 29

3.2.2 Peeling 30

3.3 Two Compiler Optimizations 31

3.3.1 Code Hoisting 31
3.3.2 Strength Reduction 31
3.4 Loop Splitting for Compiler Optimizations 32
3.4.1 Optimizing Code 33
3.4.2 Accounting for Integer Offsets 35
3.5 Loop Splitting for Array Referencing Optimizations 37
3.6 Summary 38

4 Loop Splitting Analysis **39**
4.1 Introduction 39
4.2 Implemented Methods 41
4.2.1 Rational Splitting 41
4.2.2 Interval Splitting 41
4.3 Methodology 42
4.4 Results 43
4.5 Interpretation 45

5 Conclusions **47**
5.1 Summary 47
5.2 Future Work 48

A Performance Benchmarks **51**
A.1 Vector Add 5000 52
A.2 Matrix Add 100x50 53
A.3 Matrix Multiplication 40x40 55
A.4 Matrix Transposition 100x50 57

B Loop Splitting Code **60**
B.1 Normal 61
B.2 Rational 63
B.3 Interval 69

List of Figures

2-1 The distributed memory multiprocessor model: each node is a processor-memory pair. 13

2-2 Partitioning a 100-iteration 1-D loop: **(a)** the iteration space, **(b)** the partitioned iteration space. 15

2-3 Partitioning a 100x100 2-D loop: **(a)** the iteration space, **(b)** the partitioned iteration space with blocking, and **(c)** the partitioned iteration space with striping. 17

2-4 Three examples of data partitioning for a 100x100 array on four processors: **(a)** blocking, **(b)** striping by rows, and **(c)** striping by columns. The numbers designate assignment of the array portions to the virtual processors. 18

2-5 The data footprint of a processor **(a)** for the expression $A[i][j] = A[i+1][j+1] + A[i+1][j+2]$ and **(b)** for the more general expression in which $di+$ and $dj+$ are the largest positive offsets for the induction variables i and j, respectively; and $di-$ and $dj-$ are the smallest negative offsets. The white area represents data for which a value is calculated, while the shaded areas are the additional data needed for the calculations. 20

2-6 Referencing a single array cell on a distributed memory multiprocessor. 24

2-7 Array reference expressions for one-, two-, and three-dimensional arrays. . . . 25

2-8 The same references with all the divisions replaced by interval invariants and all modulos replaced by an incrementing counter. 26

2-9 The 2-D and 3-D reference expressions after further compiler optimization. . . 26

3-1 The 1-D and 2-D array reference examples of Section 2.5. 37

4-1 The performance improvement in addition on 5000-element vectors and 100x50-element matrices. 43

4-2 The performance improvement in multiplication of 40x40 matrices and transposition of a 100x50 matrix: without additional compiler optimizations **(top)** and with the optimizations **(bottom)**. 44

List of Tables

2.1 A 1-D loop nest before and after partitioning. The processor identifier, ranging from 0 to 3, is denoted by pid. 16

2.2 Two examples of task partitioning for a 2-D loop nest: the programmer's original code, the code after blocking, and the code after striping. 17

2.3 Reduction in operations for array references by an optimizing compiler. 26

3.1 Transformation of a 1-D loop nest into two different code sequences illustrating arbitrary loop splitting. The loop body is a function of the index variable *i*. . . 29

3.2 Ten separate loops of equal iteration length can be reduced in code to two nested loops. 30

3.3 The original loop with a prologue and epilogue both peeled away and grouped into loops themselves. 30

3.4 A loop before and after code hoisting. 31

3.5 The same code after strength reduction on the multiplication and the modulo function. 32

3.6 Eight code sequences showing various forms of optimization; the right column is generalization of the example on the left, where *a* and *b* are positive integer constants. 33

3.7 A one-dimensional loop erroneously transformed by loop splitting. 35

3.8 The previous example with correct loop splitting, showing peeled iterations; the general loop splitting transformation. 36

3.9 The optimized general 1-D array reference. 38

Chapter 1

Introduction

Distributed memory multiprocessors have a clear advantage over uniprocessor machines — more than one processor allows multiple tasks to be performed simultaneously. However, distributed memory multiprocessors also have several advantages over shared memory machines. For example, scalability allows a network to be constructed proportional to a specific problem size, and nonuniform access time allows referential locality to minimize latencies of memory accesses. However, the distributed memory of the machine introduces drawbacks as well, such as the problems of finding and exploiting locality, distributing task and data of a program, and addressing data.

The addressing complications are manifested in the involved expressions associated with array referencing. To reference a particular array cell, the expression must calculate not only the memory location of the cell but also which processor's memory contains the cell. These two address specifications are functions of the array distribution and the network configuration.

Fortunately, possibilities exist for simplifying the array reference expressions in loops. When the data needed by a processor is located on only one processor, parts of these reference expressions are loop invariant (i.e., have a constant value for the duration of the loop). To avoid recomputing these invariant parts of the address on every access, a compiler can perform code transformations to simplify the expressions. Once performed, these transformations allow even further code improvements by an optimizing compiler. The end result is a less computationally expensive loop body that reduces the execution time of the loop.

Unfortunately, the data needed by a processor is often located on more than one processing element so that no loop invariants exist for optimization. However, because arrays are often both accessed and distributed in segments of contiguous array cells, *intervals* of a loop access data from a single processor and have their own invariants. Thus, each such interval has its own invariants. By dividing the loop into these intervals, the code transformations can still be performed, albeit on a smaller scale.

A compiler can isolate these intervals by performing a loop transformation called *loop splitting*. Loop splitting divides a loop into subloops, which in entirety have the same effect as the single loop. These subloops can then be reduced in computation.

In the context of distributed memory multiprocessors, this thesis explores the improvement of array references allowed by the loop splitting transformation. More specifically, this paper examines program speedup resulting from loop splitting, the code transformations *code hoisting* and *strength reduction*, and the subsequent compiler optimizations.

1.1 Overview

Section 2 describes array management in distributed memory multiprocessors. This topic includes partitioning of task and data as well as alignment for minimal execution time. Then, the method and complexity of array reference expressions are presented to illustrate the problem this thesis attempts to ameliorate.

Section 3 provides an overview of loop splitting. First, the relevant loop transformations (*general loop splitting* and *peeling*) and compiler optimizations (*code hoisting* and *strength reduction*) are presented. Next, these elements are brought together by describing the loop splitting transformation for compiler optimizations. Then, to prepare for Section 4, this section presents the loop splitting framework for optimizing array reference expressions on a distributed memory multiprocessor.

Section 4, the crux of this thesis, describes in detail the loop splitting study. This includes the methodology and performance results of several experiments. The results are then interpreted.

Section 5 concludes by summarizing both the interpretations of the study and the contributions of this thesis.

1.2 Previous Application

Loop splitting has been used in other contexts besides simplification of array references for distributed memory multiprocessors. One previous area of application is improvement of register use in vector processors.

Vector processors, such as the Cray-1 ([Rus78]), are specialized for vector operations. Vectorization of a loop removes the loop code and replaces the scalar operations of the loop body with stand-alone vector operations. Though the operations are consequently faster, if any of the operations share data, multiple memory references must be made for such data because the values were overwritten by previous vector computation. To increase register reuse (reduce the redundant memory traffic), the original loop is partitioned into vector-length strips to produce a loop with smaller vector operations, appropriate for the size of the register set. The operations are small enough to avoid flushing the common data from the vector register set before it is needed again by a subsequent operation.

Chapter 2

Array Management in Multiprocessors

Before describing the details of loop splitting, this thesis presents some background information on multiprocessor array management. Specifically, the multiprocessors discussed here are distributed memory machines. This section helps to give perspective on both the complexities of array references and the benefits of loop splitting in reducing these expressions.

2.1 Introduction

Multiprocessors are designed with the hope that many computers can perform the job of one computer in less time. Ideally hidden from the user, a program is run on a communication network of processor-memory pairs, as seen in Figure 2-1. However, to utilize more than one processor, a system of distributing the job among the processing elements must first be devised. In the most ideal case, a load-balanced partition – an equitable division of work among all processors – is found; and a multiprocessor terminates a program in $1/P^{th}$ the time of a uniprocessor, where P is the number of processors. However, partitioning problems, data dependencies, processor communication, and

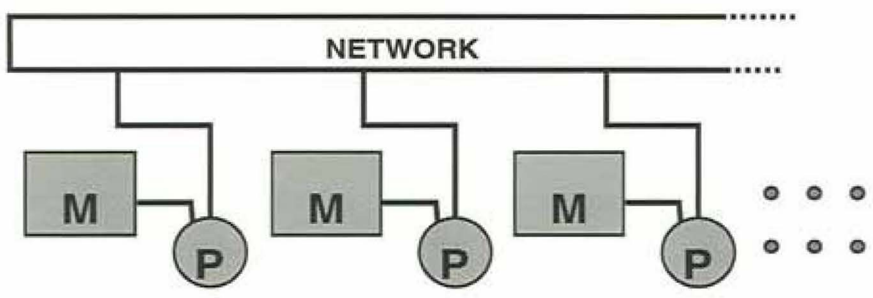


Figure 2-1: The distributed memory multiprocessor model: each node is a processor-memory pair.

inherently sequential code[1] inhibit such a speedup by incurring unequal work distribution and communication overhead. Even so, the decrease in execution time can still be remarkable, particularly in loop- and array-oriented scientific codes.

As in any algorithm, a parallelizable algorithm, such as matrix addition, has two types of specifications — that of the task to be performed, and that of the data on which to perform the task. Unlike a sequential algorithm, however, the task and data of the parallel algorithm are divided among the processors with regards to the *task partitioning* and *data partitioning*, respectively. Each processor computes a unique section of the matrix addition, and contains a unique portion of the matrices in its memory.

Through analysis of a program, a procedure called task and data partitioning determine the optimal task and data partitions. First, the task partition is obtained by dividing the work equally among processors while maximizing the reuse of data in each division. This decreases the execution time by reducing the number of non-cache references. Next, the data partition attempts to choose the size and shape of the data tile, which when placed in the memory of a processor maximizes the probability that a cache miss is satisfied in local memory. This lowers the execution time by reducing the number of network requests. Finally, an alignment step determines which specific task and data partition is assigned to each processor.

Though the above steps are performed with the hope of reducing the execution time per processor, accessing array elements on a multiprocessor requires more calculation than

[1]A simple example of nonparallelizable code is a sequence of I/O instructions, such as printing the directions of a program's usage. In such a case, only one processor executes while the others stand idle.

on a uniprocessor. In removing the uniprocessor, scalable multiprocessors also remove the single memory address space. Array referencing must not only consider where in memory an array is located, but also *in which processor's* memory, thus requiring more calculation.

Virtual memory can eliminate the additional addressing calculations by providing the abstraction of a uniform, linear address space. However, this abstraction results in an efficiency loss because an unnecessarily large number of memory references may need to be satisfied by a remote node.

The remainder of this section describes in more detail what has been outlined above. First, Section 2.2 presents task and data partitioning, while introducing terms relevant to loop splitting. Then, Section 2.3 describes the procedure for determining the optimal task and data partitions. Section 2.4 describes the problem with array referencing on a distributed memory multiprocessor. Finally, Section 2.5 examines a method of multiprocessor array referencing, the complex expressions involved, and the possibility of reducing these expressions.

2.2 Partitioning

While a uniprocessor executes an entire program, a processing element (PE) in a distributed memory multiprocessor executes only a portion of the program. Dividing a program into portions for the PEs is called partitioning, of which there are two types – task partitioning and data partitioning.

2.2.1 Task Partitioning

Performed for both distributed and shared memory multiprocessors, task partitioning assigns a portion of work to each processor such that the sum of all processor tasks is equivalent to the original single task. Two types of task partitioning exist: dynamic and static. Dynamic task partitioning is determined at the time of execution, where processes spawn other processes until all processors are busy. This is achieved by *fork* operations, which are later *join*ed to merge the results of two processors onto one processor. The fork and join operations are well-suited to recursive algorithms.

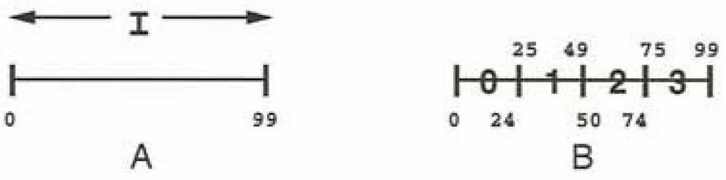


Figure 2-2: Partitioning a 100-iteration 1-D loop: **(a)** the iteration space, **(b)** the partitioned iteration space.

Static task partitioning, on the other hand, is determined at compilation time, where beforehand the total task is divided among processors as equitably as possible. This type of task distribution is suitable for iterative algorithms and is attained by using a parallelizable loop construct, such as the FORALL[2] instruction. Because the task partitioning is a distribution of the loop's iterations across processors, it is often termed the *iteration space partitioning*. This thesis is concerned with only static task partitioning, and further mention of task partitioning will be done without ambiguity of type.

After some definitions, task partitioning is examined through two examples. A loop construct, such as FORALL, is often used in clusters; a group of associated FORALL instructions is called a *loop nest*, since the cluster represents one or more nested loops. An N-dimensional loop nest contains N nested FORALL instructions. Two loops are *perfectly nested* if both share exactly the same loop body. Two loops are *imperfectly nested* if one contains the common body as well as other code.

Example 1: A 1-D Loop Nest As a simple example of task partitioning, consider a 1-D loop nest of 100 iterations performed on a four-processor machine. Figure 2-2 shows the iteration space and a possible partitioning among the processors. Processor Zero computes iterations 0 through 24, Processor One computes iterations 25 through 49, and so on. Each processor calculates 25 iterations; this is an ideal task partition because it distributes the task equitably.

The code written by the programmer is shown on the left of Table 2.1 in C syntax, and the task-partitioned code of a parallel compiler is depicted on the right. To simplify the code, this thesis uses the (noninclusive) expression **for**(*var*, init, limit, step) in place of the

[2]The DOALL instruction is another example.

Original	**Partitioned**

forall(*i*, 0, 100, 1){
... body ... }
	low = pid * 25;
	high = low + 25;
	for(*i*, low, high, 1){
	... body ... }

Table 2.1: A 1-D loop nest before and after partitioning. The processor identifier, ranging from 0 to 3, is denoted by pid.

normal C syntax expression **for**(*var*=init; *var* < limit; *var*+=step).

The code on the right is generalized code; regardless of what processor executes the code, the processor will perform only its assigned portion of the task. This enables a compiler to generate one code sequence to be executed by all processors. However, other methods exist to generalize the code. As an example, instead of computing high and low interval boundaries, the code could use a CASE statement to assign intervals based on the processor ID. This is particularly useful when the intervals are irregular and hard to generalize into equations; of course, the CASE statement may be long if there are many processors. These and other issues lead to the different methods of loop splitting and will be discussed later.

Figure 2-2 and Table 2.1 help to introduce two new terms. The length of the intervals in each dimension is called the *task spread* of that dimension. In the context of the 1-D loop nest above, the dimension I has a task spread of 25 iterations. Furthermore, the *tile size*, the product of the task spreads, is the total number of iterations each processor must perform for the loop nest. In the 1-D example, the tile size[3] is 25 iterations. This thesis focuses on rectangular tiles; however, the results generalize to parallelograms as well.

Example 2: A 2-D Loop Nest A more complex example is a pair of perfectly nested loops with iteration intervals of 0 to 99 for both dimensions. Pictorial representations of this loop nest are contained in Figure 2-3, which shows blocking and striping of the iteration space.

Table 2.2 displays the code[4] of these two possible partitions. The code with blocking

[3]In this thesis, the size of a processor's task will be called the tile size (implying two dimensions) regardless of dimension.

[4]All code examples in this thesis uses a C syntax; thus, '*' (asterisk) represents multiplication, '/' integer

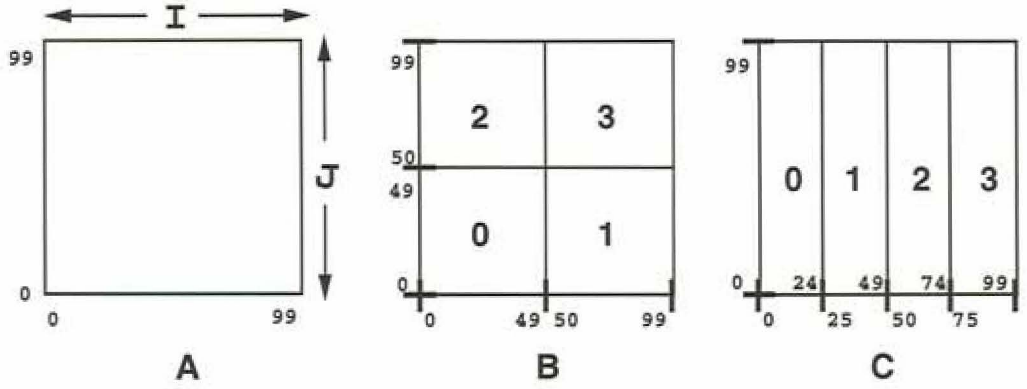


Figure 2-3: Partitioning a 100x100 2-D loop: **(a)** the iteration space, **(b)** the partitioned iteration space with blocking, and **(c)** the partitioned iteration space with striping.

Original	With Blocking	With Striping
forall(*i*, 0, 100, 1){	i_low = (pid % 2) 50;	i_low = pid * 25;
forall(*j*, 0, 100, 1){	i_high = i_low + 50;	i_high = i_low + 25;
... body ... }}	j_low = (pid / 2) * 50;	**for**(*i*, i_low, i_high, 1) {
	j_high = j_low + 50;	**for**(*j*, 0, 100, 1) {
	for(*i*, i_low, i_high, 1) {	... body ... }}
	for(*j*, j_low, j_high, 1) {	
	... body ... }}	

Table 2.2: Two examples of task partitioning for a 2-D loop nest: the programmer's original code, the code after blocking, and the code after striping.

divides both dimensions into two intervals, while the code with striping divides one dimension into four intervals. The first partitioned loop nest has a task spread of 50 for both the I and J dimensions; and the second partitioned loop nest has an I-dimension task spread of 25 and a J-dimension task spread of 100. In both methods of partitioning, the tile size is 2500 iterations.

Because every processor in each example has the same tile size (total task divided by the number of processors), these are *ideal* partitions and ensure load-balancing. However, where multiple partitions are possible (in loop nests of dimension > 1), communication cost becomes a major criterion in determining the *optimal* task partitioning – that leading to the shortest execution time. Section 2.3 discusses this issue in more detail.

division, and '%' the modulo or remainder function.

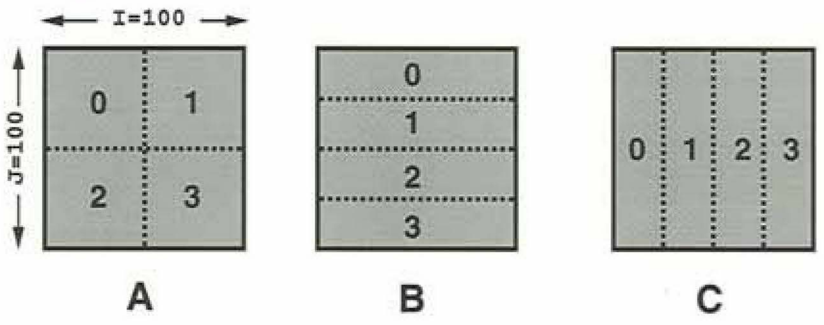


Figure 2-4: Three examples of data partitioning for a 100x100 array on four processors: **(a)** blocking, **(b)** striping by rows, and **(c)** striping by columns. The numbers designate assignment of the array portions to the virtual processors.

2.2.2 Data Partitioning

While the task is divided among processors for all parallel machines, the data of a program is divided only on a distributed memory multiprocessor. In a distributed memory system, every memory address is local to some processor node; and every processor node has its own piece of memory, from which all nodes may read and to which all may write. The data structure relevant to data partitioning is the array and is distributed among the memories without duplication. Figure 2-4 shows several data distributions for a 2-D array, each with a different mapping of the array onto the processor memories.

Figure 2-4 helps to define more terms. First, the *data spread* is the dimension length of the data portion located in a processor's memory. In Figure 2-4a both the *i_spread* and *j_spread* are 50 array elements each; in Figure 2-4b, *i_spread* is 100 elements and *j_spread* is 25; and in Figure 2-4c *i_spread* is 25 elements and *j_spread* is 100. Second, the *virtual network* is the arrangement of the virtual processors on which alignment maps the data. This virtual network is then mapped onto the (possibly different) real network of physical processors by a procedure called *placement*. The dimensions of the virtual network play an important role in array referencing (explained in Section 2.5) and are called *x_proc*, where *x* is the dimension name. In Figure 2-4a, both the *i_proc* and *j_proc* are 2 virtual processors; in Figure 2-4b *i_proc* is 1 and *j_proc* is 4; and in Figure 2-4c *i_proc* is 4 processors and *j_proc* is 1.

Like task partitioning, the *ideal* data partitioning distributes equal amounts of data to the processors, and the *optimal* data partitioning leads to the shortest execution time. Also like task partitioning, many ideal data partitions may exist, but very few optimal partitions do and may be difficult to obtain. The next section outlines the process of obtaining optimal task and data partitions.

2.3 Optimal Partitioning

As observed in the previous section, many possible methods of task and data partitioning exist for a specific loop nest. This section describes the process of obtaining the optimal task and data partitions, those leading to the minimal execution time, while introducing more terms relevant to array referencing and loop splitting.

2.3.1 Optimal Task Partitioning

As a whole, the array cells needed by a particular processor is termed the processor's *data footprint*. This represents the data required for the processor's task and aids in determining the optimal task partition. An example data footprint on a 2-D array A is depicted in Figure 2-5 for the expression $A[i][j] = A[i+1][j+1] + A[i+1][j+2]$. Here, the white region in the center represents the array cells for which Processor 12 computes values; this corresponds to Processor 12's task. The shaded regions represent additional data, for which Processors 4, 5, and 13 are responsible, needed to perform Processor 12's task.

Justification for the additional data is seen by studying the reference expression. Consider the single array cell in the center of Figure 2-5a. If this cell is $A[2][5]$, then $A[2][6]$ and $A[2][7]$ are added to assign its new value. These two cells are the pair next to the single cell. Of course, in general, cell $A[i][j]$ needs cells $A[i+1][j+1]$ and $A[i+1][j+2]$ to obtain a new value. From this, it is obvious that the array cells along the border of Processor 12's task require values outside the central white region, namely, the cells for which Processors 4, 5, and 13 compute values.

The optimal task partition, represented by the *aspect ratio* I/J, is found by minimizing the interprocessor communication (the shaded region of Figure 2-5) while maintaining the tile size constraint ($I \times J$ = total task size $\div$ number of processors). Such

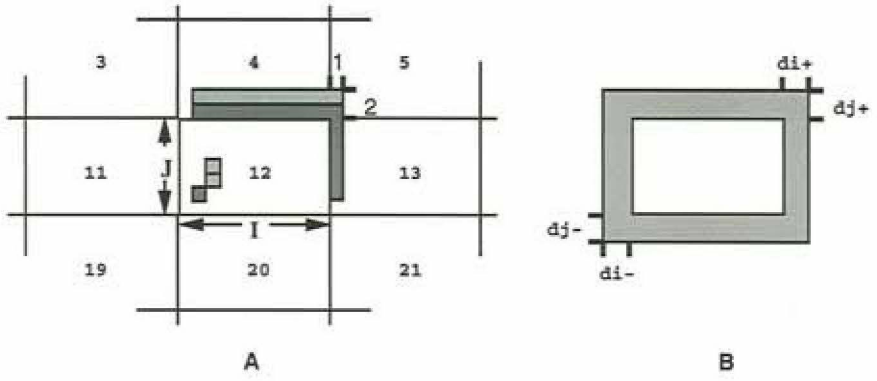


Figure 2-5: The data footprint of a processor **(a)** for the expression $A[i][j] = A[i+1][j+1] + A[i+1][j+2]$ and **(b)** for the more general expression in which $di+$ and $dj+$ are the largest positive offsets for the induction variables i and j, respectively; and $di-$ and $dj-$ are the smallest negative offsets. The white area represents data for which a value is calculated, while the shaded areas are the additional data needed for the calculations.

a partitioning groups closely associated iterations on one processor, thereby increasing the temporal locality by maximizing data reuse. When an iteration needs a particular array cell, the cell is cached and available to later iterations on the same processor. Because a network or memory access occurs only once per unique array cell, and because the suggested tile dimensions minimize the number of different array references; such a task partition minimizes the total access time and is optimal.

The details of optimal task partitioning are contained in [AKN92], but determining the optimal aspect ratio for a 2-D loop nest will quickly be presented here.

The derivation of the optimal (to a first approximation) aspect ratio is rather simple. Finding the I and J resulting in minimal communication we compute their ratio I/J. This is performed in the following manner.

The tile size is $k = I \times J$. Communication (to a first approximation) is the number of rows and columns of nonlocal data. Where α is the number of rows and β is the number of columns, the total communication in a multiprocessor with caches is

$$c = \alpha I + \beta J = \alpha I + \beta \frac{k}{I} = \alpha \frac{k}{J} + \beta J.$$

To obtain the I and J that minimize communication, we calculate the derivative of communication with respect to the variable and find where it has zero value:

$$\frac{dc}{dI} = \alpha - \beta\frac{k}{I^2} = 0 \Rightarrow I = \sqrt{\frac{\beta k}{\alpha}} \tag{2.1}$$

$$\frac{dc}{dJ} = \beta - \alpha\frac{k}{J^2} = 0 \Rightarrow J = \sqrt{\frac{\alpha k}{\beta}} \tag{2.2}$$

The optimal aspect ratio is the ratio of equation 2.1 to equation 2.2, which becomes

$$\frac{I}{J} = \frac{\sqrt{\frac{\beta k}{\alpha}}}{\sqrt{\frac{\alpha k}{\beta}}} = \sqrt{\frac{\beta^2}{\alpha^2}} = \frac{\beta}{\alpha} = \frac{\text{\# columns of communication}}{\text{\# rows of communication}}.$$

Thus, the optimal aspect ratio for Figure 2-5a is $I/J = 1/2$, and the ratio for Figure 2-5b is $\frac{I}{J} = \frac{di_+ + di_-}{dj_+ + dj_-}$.

2.3.2 Optimal Data Partitioning

When only one loop nest and one array exist, the optimal data partition is exactly the optimal task partition. Such a data partition reduces most of the network accesses to local memory accesses by locating directly on a processor the cells for which it is responsible to compute values. In Figure 2-5a, the white portion would be in Processor 12's memory, and the shaded region would be the remote data needed from Processors 4, 5, and 13.

In general, however, the optimal data partition is harder to obtain. *Alignment* is the process of attempting to place the data footprint accessed by a task on the same processor as the task. Details of obtaining the optimal data partition parameters can be found in [AKN92].

2.4 The Problem

Most programs are not of the single loop nest, single array type. Instead, multiple loop nests with multiple arrays make alignment difficult. The resulting poor alignment induces fragmentation of a processor's array references, causing accesses across several memory modules over the execution of the loop nest.

For example, in the context of the previous 2-D loop nest, a task partition with blocking (Figure 2-3b) and a data partition with striping by rows (Figure 2-4b) have poor alignment. Even with optimal placement (virtual processors 0, 1, 2, 3 map to real processors 3, 2, 1, 0, respectively), only half of a processor's data footprint is in its local memory. Processor Zero (with virtual PID 3) would require data in both local memory and the memory of Processor One (with virtual PID 2). For Processor Zero to reference the data, it must not only specify the address of an array cell but also the processor's memory in which it is located.

Thus, the very definition of a distributed memory multiprocessor leads to complications in referencing the program's dispersed data. The complication appears in the form of involved reference expressions, which adversely affect the execution time of a program. The next section describes these expressions and illustrates the possibilities for simplification.

2.5 Array Referencing

Methods of array referencing on distributed memory multiprocessors vary; the method described here is used by the Alewife effort ([A*et al.*91]) at MIT. This thesis concerns itself with rectangular task and data partitioning and barrier synchronization of parallel loop nests, both of which are provided by the Alewife compiler.

Array referencing in Alewife is implemented in software. The general expression to access an array cell is

$$\text{aref}\,(\text{aref}\,(\mathit{array},\ \mathit{pid}),\ \mathit{offset})\,,$$

where aref is the array reference procedure, *array* is the name of the array, *pid* is the unique ID of the processor whose memory contains the cell, and *offset* is the offset into that memory. The reference procedure aref has two arguments – a list structure and an offset into the structure, and returns the element in the structure located at the offset. In the general expression above, the inner aref determines the memory segment in which the cell resides from both *array*'s dope vector (a list of pointers to memory segments) and the processor ID. The outer aref uses the memory segment and *offset* to obtain the actual location of the array element.

Figure 2-6 illustrates this array reference scheme with a 2-D array distributed over a network of processors. To reference a particular array cell, *pid* indexes the dope vector to determine the memory segment. *Offset* is then added to the address of the memory segment to locate the array cell. This procedure is traced by the path shown in bold. In general, the array cells are mapped to contiguous memory addresses so that, regardless of the array dimension, one offset identifies a cell's address. The dope vector itself lies in the memory of one processor and has potential of becoming a bottleneck.

As example reference expressions, the code fragments of Figure 2-7 are array references to cells A[i], B[i][j], and C[i][j][k]. I and J are loop index variables so that the first expression is contained in at least a 1-D loop nest, the second is in at least a 2-D loop nest, and the third is in at least a 3-D loop nest. Further, the constants i_spread, j_spread, i_proc, etc. represent values described in the previous sections.

The validity of the first expression shall be justified to give a sense of how *pid* and *offset* are calculated. In the 1-D situation, the array is divided into P sections of i_spread contiguous elements, where P is the number of processors. A procedure given I can identify the segment with $I /$ i_spread and find the offset into the section with $I \%$ i_spread. Because each section is contained in a unique processor, calculating the section holding an element is tantamount to calculating the virtual processor node holding that element. Thus, $I /$ i_spread is the virtual processor ID, and $I \%$ i_spread is the offset into that processor's memory segment holding the section of the array.

The second and third expressions (and any N-dimensional reference for rectangularly partitioned arrays) have the same structure – compute the processor ID and calculate the offset in the processor's memory. Of course, the expression to find the ID and the offset becomes more complicated with an increase in the dimension of the array.

```
aref (aref (A, I / i-spread) , I % i-spread);

aref (aref (B, (I / i-spread) + (i-proc * J / j-spread)),
      (I % i-spread) + i-spread * (J % j-spread));

aref (aref (C, (I / i-spread) + i-proc * ((J / j-spread) + (j-proc * K / k-spread))),
      (I % i-spread) + i-spread * ((J % j-spread) + j-spread * (K % k-spread)));
```

Figure 2-7: Array reference expressions for one-, two-, and three-dimensional arrays.

2.6 Simple Optimization of Array Calculation

When the interval of a loop is sufficiently[5] small (one iteration, if necessary), some reductions in calculation are possible. First, all divisions with an induction variable as the dividend have a constant value for the duration of the loop. This allows the constant to be computed once outside the loop so that all iterations require no computation for the value. Second, any modulo functions with an index variable as the first argument can be replaced with a new variable whose value increments or decrements on successive iterations. This is allowed by the nonperiodicity of the modulo expression (i.e., it does not flip back to zero), guaranteed if the interval is sufficiently small. Section 3 explains these reductions in calculation in greater detail.

In such a loop with a sufficiently small number of iterations, the three previous array references become those in Figure 2-8, in which the `div-` and `rem-` variables have replaced the division and modulo functions, respectively.

However, an optimizing compiler can reduce the calculation further. The newly introduced constants from the division reduction can propagate to form more constants. For example, the in the 2-D reference, (i-n * *J* / j-spread) becomes a product of two constants (i-n * div-b-j), which itself is an interval invariant. Thus, the entire expression can be removed from the loop body. With further propagation of constants, the second and third expressions become those in Figure 2-9 so that all array references have a loop invariant as the processor ID. Table 2.3 summarizes the total reduction in operations for

[5]To be specified later.

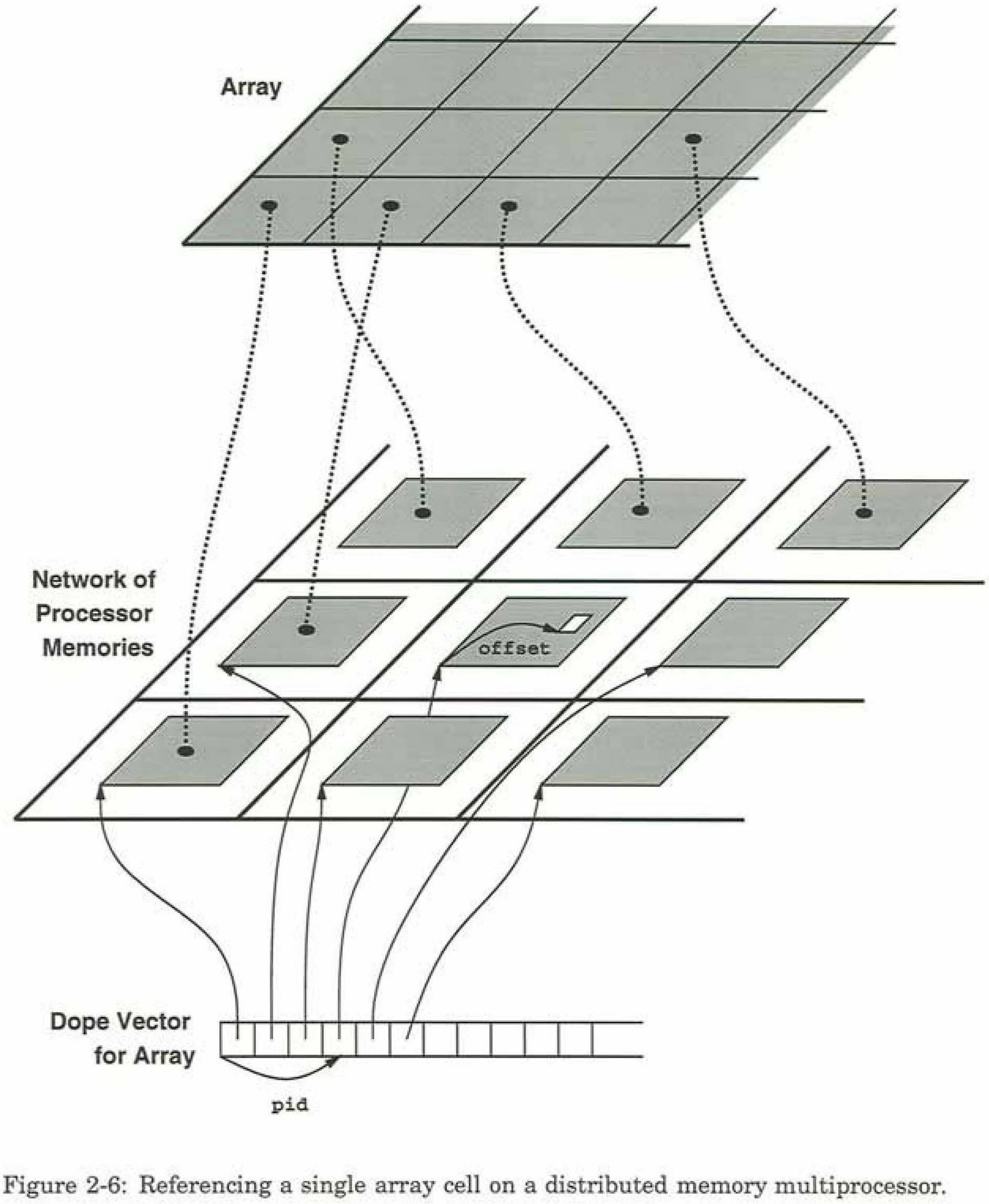


Figure 2-6: Referencing a single array cell on a distributed memory multiprocessor.

```
aref (aref (A, div-a-i), rem-a-i);

aref (aref (B, div-b-i + (i-n * div-b-j)),
      rem-b-i + (i-spread * rem-b-j));

aref (aref (C, div-c-i + i-proc * (div-c-j + (j-proc * div-c-k))),
      rem-c-i + i-spread * (rem-c-j + j-spread * rem-c-k));
```

Figure 2-8: The same references with all the divisions replaced by interval invariants and all modulos replaced by an incrementing counter.

```
aref (aref (B, pid-B-ij), rem-b-i + (i-spread * rem-b-j));

aref (aref (C, pid-C-ijk), rem-c-i + i-spread * (rem-c-j + j-spread * rem-c-k);
```

Figure 2-9: The 2-D and 3-D reference expressions after further compiler optimization.

each reference.

An optimizing compiler could even further reduce the calculation by replacing the multiplication by *i_spread* in the second reference (*j_spread* in the third reference) with an addition of *i_spread* (*j_spread*) on each iteration. This type of optimization, called *strength reduction*, is described in Section 3.3.2.

In the context of rectangular partitioning, "sufficiently small" interval values are those that keep the loop nest occupied with a single processor. If this condition is met, the processor ID is constant, and the offset into the processor's memory can be determined with monotonically increasing or decreasing counters. Thus, appropriate intervals can

Number of Operations Per Array Reference								
Reference	**Before Optimization**				**After Optimization**			
	mod	÷	×	+	mod	÷	×	+
1-D	1	1	0	0	0	0	0	0
2-D	2	2	2	2	0	0	1	1
3-D	3	3	4	4	0	0	2	2

Table 2.3: Reduction in operations for array references by an optimizing compiler.

reduce the array referencing computation to offset calculation alone, which is less taxing than before. The key, then, is to determine a loop partition such that the above conditions are met.

2.7 Summary

The task and data of a parallel loop nest are divided on a distributed memory multiprocessor. The task is divided among processors while minimizing the communication between tasks (thus maximizing the reuse of data within a task). The data is then distributed by assigning to each processor the data its task requires while minimizing communication among processors; this creates a virtual network of processors. Placement then maps the virtual network onto the real processors of the physical multiprocessor. In this way, alignment and placement attempt to minimize the processor communication, in turn minimizing execution time.

However, array reference expressions require complex calculations, which increase dramatically with array dimension. If the intervals of a loop nest are chosen sufficiently small, however, these complex calculations can be reduced in execution cost – the divisions can be removed from inside the loop and the modulos can be replaced by counters.

The next section describes loop splitting – a technique to obtain a loop nest with "sufficiently small" intervals that still computes the original calculations.

Chapter 3

Overview of Loop Splitting

Now that array referencing on multiprocessors has been presented, this section continues the background presentation by defining the term *loop splitting*. This includes describing the loop and code transformations comprising loop splitting as well as its application to simplifying array reference expressions.

3.1 Introduction

As supported by Section 2.5, array references can require involved calculations. For example, in a simple matrix transpose (2-D loop with body A[i][j] = B[j][i]), the assignment requires 4 each of modulos, divisions, multiplications, and additions. In matrix addition (A[i][j] = B[i][j] + C[i][j]), of the 6 modulos, 6 divisions, 6 multiplications, and 7 additions, only one addition was specified by the programmer. All of this array reference overhead significantly inhibits a loop nest's execution performance.

However, *loop splitting* is an attempt to minimize this overhead by altering the loop structure. By modifying the intervals of the loop nest and introducing another nesting level, not only are the array reference operations fewer in number, but they are also no longer performed on every iteration. This reduction in calculation leads to faster program execution time.

This section presents sufficient descriptions of the components of loop splitting. Section 3.2 describes the two loop transformations composing loop splitting, and Section

Original	**After Arbitrary Loop Splitting**	
for(i,0,100,1)	**for**(i,0,50,1)	**for**(i,0,10,1)
{body(i);}	{body(i);}	{body(i);}
	for(i,50,100,1)	**for**(i,10,30,1)
	{body(i);}	{body(i);}
		for(i,30,100,1)
		{body(i);}

Table 3.1: Transformation of a 1-D loop nest into two different code sequences illustrating arbitrary loop splitting. The loop body is a function of the index variable i.

3.3 describes the two compiler optimizations allowed by loop splitting. Section 3.4 then presents these two transformations and two optimizations together as the one transformation *loop splitting*. Finally, Section 3.5 explains the application of loop splitting to improvement of the array references described in Section 2.5.

3.2 Two Loop Transformations

This section introduces the two loop transformations performed in compile-time loop splitting for distributed memory multiprocessors. The first transformation is general loop splitting – the splitting of a loop into more than one subloop. The second is *peeling*, duplicating a loop body to remove iterations from a loop at the beginning or end of the iteration interval.

3.2.1 General Loop Splitting

The general loop splitting transformation is weakly defined as dividing one loop into several loops. These loops as a whole accomplish the same task as the single loop. Dividing the loop into subloops can be arbitrary, as Table 3.1 shows. Here, one loop has been separated into two equal-length loops and into three unequal ones. Obviously, this arbitrary style of loop splitting replicates the loop body for each division, endangering a program with code growth.

This thesis, however, defines loop splitting so that it must create subloops of equal length. These equal-length loops avoid the code expansion by allowing a generalization of each loop. A generalized loop, as the body of an appropriate outer loop, can represent the

Ten 1-D Loop Nests	One 2-D Loop Nest

for(i,0,10,1) {body(i);}
for(i,10,20,1) {body(i);}
for(i,20,30,1) {body(i);}
...
for(i,90,100,1) {body(i);}
	for(I_step,0,100,10) {
	for(i,I_step,I_step+10,1)
	{body(i);}}

Table 3.2: Ten separate loops of equal iteration length can be reduced in code to two nested loops.

Peeled Iterations	Grouped Into Loops

body(i=0);
body(i=1);
for(i,2,97,1) {body(i);}
body(i=97);
body(i=98);
body(i=99);
	for(i,0,2,1) {body(i);}
	for(i,2,97,1) {body(i);}
	for(i,97,100,1) {body(i);}

Table 3.3: The original loop with a prologue and epilogue both peeled away and grouped into loops themselves.

subloops in entirety. For example, the code on the left of Table 3.2, comprised of ten subloops of equal length (10 iterations), can be represented by the much smaller code on the right. With this definition of loop splitting, code expansion is no longer a concern, and the extra loop overhead is negligible provided the subloops have a moderate iteration size.

3.2.2 Peeling

This thesis defines peeling as the removal of a small number of iterations from either end of a loop's iteration interval. These peeled iterations result in a prologue (the first few iterations) and an epilogue (the last few iterations) around the modified loop. Table 3.3 shows the original loop of Table 3.1 with two iterations peeled into a prologue and three peeled into an epilogue. Of course, if the number of peeled iterations is large enough, loop overhead may be favored over code expansion so that the prologue and epilogue become loops themselves, as Table 3.3 also illustrates.

For the remainder of this thesis, the term *loop splitting* will refer not to the general transformation of Section 3.2.1, but specifically to the combination of general loop splitting and peeling to allow for parallel compiler optimizations, which Section 3.4 describes.

Original	After Code Hoisting

c = ...;
for(*i*,40,60,1)
  {A[*i*] = 6*c + 10**i* + *i*%35;}
	c = ...;
	temp = 6*c;
	for(*i*,40,60,1)
	  {A[*i*] = temp + 10**i* + *i*%35;}

Table 3.4: A loop before and after code hoisting.

3.3 Two Compiler Optimizations

Before continuing the discussion of loop splitting, this section introduces two compiler optimizations – *code hoisting* and *strength reduction*. These two code improvements are precisely the ones allowed by loop splitting and are the motivation for performing this loop transformation.

3.3.1 Code Hoisting

The *code hoisting* compiler optimization for loops, also known as *factoring loop-invariants*, avoids redundant computations by pulling an expression out of a loop body to precede the loop code. This expression must have a constant value for the entire duration of the loop. After code hoisting, the expression's value is computed only once and is used by every iteration of the loop.

Table 3.4 shows a simple example of code hoisting for loops. The original code, on the left, computes the value 6*c one hundred times, while the optimized code on the right computes it once and assigns the value to a temporary variable. This variable is then used in place of the expression in the loop body.

Obviously, as the number of iterations in a loop nest increases and as the expression removed grows in calculation complexity, code hoisting significantly reduces the execution time of a loop.

3.3.2 Strength Reduction

The *strength reduction* compiler optimization replaces an expensive (with regards to time) operation with a less expensive one. For example, in loops multiplications with a constant can be replaced by additions with the constant. Because multiplication on a processor is

Reduction of Multiplication	**Reduction of Modulo**

c = ...;
I_mult_10=400;
for(i,40,60,1)
  {A[i] = 6*c + I_mult_10 + i%35;
  I_mult_10+=10;}
	c = ...;
	I_rem_35 = 5;
	for(i,40,60,1)
	  {A[i] = 6*c + 10*i + I_rem_35;
	  I_rem_35++; }

Table 3.5: The same code after strength reduction on the multiplication and the modulo function.

typically three to ten times slower than addition, such a strength reduction can lead to noticeable speedups in loop execution time.

The left of Table 3.5 displays the original code of Table 3.4 after strength reduction on the multiplication. The expression 10*i in the loop has been replaced by the variable *I_mult_*10, whose value is increased by ten on every iteration. Thus, by modifying the loop variables, an expensive multiplication has been reduced to a faster addition.

Figure 3.5 also shows strength reduction in the much more complex modulo function, which requires repeated divisions until a remainder is found. Here, the modulo is replaced by a faster variable increment. Because divisions are slower than additions and modulos require multiple divisions, this reduction is a significant improvement on execution time. However, there is one very important condition for performing this optimization; *the modulo function must never reset back to zero.* Here, the modulo's range of values is 5 to 24. If the initial i value were 0, however, the modulo expression would have values from 0 to 34 and then from 0 to 24. In such a case, this strength reduction cannot be performed because incrementing a variable can never lead to cyclic values.

Though the acyclic requirement may seem stringent, strength reduction on modulo functions will always be available in loop splitting for parallel compilers of distributed memory multiprocessors, as explained in the next section.

3.4 Loop Splitting for Compiler Optimizations

Now that the relevant loop transformations and code optimizations have been introduced, loop splitting for compiler optimizations is examined. Specifically, loop splitting allows the code hoisting and strength reduction optimizations of the previous section. This section

Optimization	Example	Generalization

None	**for**(*i*,0,100,1)
	{A[*i*] = *i*/20 + *i*%30;}
		for(*i*,LL,UL,1)
		{A[*i*] = *i*/a + *i*%b;}

Code Hoisting of the Division Expression	**for**(I_step,0,100,20){
	I_div_20 = I_step/20;
	for(*i*,I_step,I_step+20,1)
	{A[*i*] = I_div_20 + *i*%30;}}
		for(I_step,LL,UL,a){
		I_div_a = I_step/a;
		for(*i*,I_step,I_step+a,1)
		{A[*i*] = I_div_a + *i*%b;}}

Strength Reduction of the Modulo Expression	**for**(I_step,0,100,30){
	I_rem_30 = 0;
	for(*i*,I_step,I_step+30,1)
	{A[*i*] = *i*/20 + I_rem_30;}
	I_rem_30++;}
		for(I_step,LL,UL,b){
		I_rem_b = 0;
		for(*i*,I_step,I_step+b,1)
		{A[*i*] = *i*/a + I_rem_b;}
		I_rem_b++;}

Both	**for**(I_step,0,100,10){
	I_div_20 = I_step/20;
	I_rem_30 = I_step%30;
	for(*i*,I_step,I_step+10,1)
	{A[*i*] = I_div_20 + I_rem_30;}
	I_rem_30++;}
		for(I_step,LL,UL,gcd(a,b)){
		I_div_a = I_step/a;
		I_rem_b = I_step%b;
		for(*i*,I_step,I_step+gcd(a,b),1)
		{A[*i*] = I_div_a + I_rem_b;}
		I_rem_b++;}

Table 3.6: Eight code sequences showing various forms of optimization; the right column is generalization of the example on the left, where a and b are positive integer constants.

describes the realization of these optimizations with loop splitting and presents a method of allowing induction variables with constant offsets.

3.4.1 Optimizing Code

To clearly explain how loop splitting can help improve code, this section fully analyzes a detailed loop splitting example.

The top of Table 3.6 shows code segments for a one-dimensional loop that assigns array cells with an expression composed of a division and a modulo function. Both terms in the expression are functions of i, the induction variable of the loop, which is monotonically increasing. This thesis restricts its focus to induction variables with step size +1, typical of array-modifying loop nests.

If a compiler were to perform loop splitting with an interval length of 20, as in the code sequence second from the top, the division expression becomes a loop-invariant of the inner loop and can be hoisted to the outer loop. The division is then reduced from 100 to 5 executions. The interval length can be any factor of 20; however, an increase in interval size leads to a decrease in calculation so that 20 iterations is the optimal interval size.

If, instead, a compiler were to simplify only the modulo function, the largest loop splitting interval would be 30. The third code sequence of Table 3.6 shows the resulting code after strength reduction, where the modulo function has been replaced with addition by one.

In order to optimize *both* expressions with loop splitting, the loop splitting interval in the example must be yet a different length. Upon inspection of the original code, one can see the largest interval length is 10 iterations. The bottom left code sequence of Table 3.6 depicts this fully optimized case.

In general, in order to perform code hoisting on all divisions and strength reduction on all modulos in a loop, the loop splitting interval must be a common divisor of all the divisors and modulo bases, thereby guaranteeing the requirements of code hoisting and strength reduction. Of course, the greatest common divisor (gcd) would remove the most computation. The bottom right of Table 3.6 illustrates this.

Before continuing this section, it is helpful to realize the consequences of the loop splitting interval length. The second and third code sequences of Table 3.6 are optimally split for the division and modulo expressions, respectively, because they minimize the number of calculations for each. However, to allow optimization of both expressions, the fourth code sequence re-introduces redundant computations. Here, ten divisions and modulos are performed though the minimum computations are five divisions and three modulos.

The additional five divisions and seven modulos represent the trade-offs associated with choosing which expressions to optimize. In the best case, all expressions to be optimized lead to the same loop splitting interval length, and all are optimized to the minimal computation needed. However, in the worst case, they lead to a greatest common divisor of 1, where the resulting code is "fully optimized" but performs the same number of divisions and modulos as the original code. Indeed, this extreme case is actually worse than the original code due to the extra loop overhead of the outer loop! Deciding the interval length and which expressions to optimize leads to different loop splitting methods, of which two are presented in Section 4.

Original	After Loop Splitting

for(*i*,0,100,1)
  {A[*i*] = (*i*+1)/10 + (*i*-2)%10;}
	for(I_step,0,100,10){
	  I_div_10 = I_step/10;
	  I_rem_10 = I_step%10;
	  for(*i*,I_step,I_step+10,1)
	    {A[*i*] = I_div_10 + I_rem_10;}}

Table 3.7: A one-dimensional loop erroneously transformed by loop splitting.

3.4.2 Accounting for Integer Offsets

The previous section showed the problem of choosing both the loop splitting interval and which divisions (of the form i/a) and modulos (of the form $i\%a$) to optimize. However, loop splitting experiences a further complication. Because integer offsets of induction variables are very common, loop splitting must handle the more general expression $i+c$, where c is an integer constant. This section illustrates the code modification loop splitting requires to optimize expressions of the form $(i+c)/a$ and $(i+c)\%b$.

As noted earlier, expressions of the form i/a and $i\%b$ can be fully optimized by loop splitting with an interval length of a and b, respectively. In such a case, the value of i/a is constant through all a iterations. However, an expression $(i+c)/a$ would have two different values: one for $|c\%a|$ iterations and another for $a - |c\%a|$ iterations. More specifically, the values are $val - 1$ and val for $c < 0$, and val and $val + 1$ for $c > 0$, where $val = i/a + c/a$. Of course, when $c = 0 \pmod{a}$, the interval has only one value; and when $c = 0$, the situation is exactly that described in the previous section.

The example of Table 3.7 illustrates this claim. The original code on the left has been erroneously transformed into the code on the right by loop splitting, where the interval was computed as gcd(10,10) = 10. Two errors can be seen. First, the division expression has the wrong value on the last iteration of an interval. For example, when $i = 9$ (the last iteration of the first interval) the value used is 0; it should be $(i+1)/10 = (9+1)/10 = 1$ instead. Because the last iteration is treated the same as all others in the loop structure, the loop code cannot account for this discrepancy between the last iteration and the first 9 iterations.

Second, the modulo function has the wrong values for *every* iteration. The computed value is always 2 (mod 10) higher than the correct value. For example, the very first

Original	Generalization
for(I_step,0,100,10){	OriginalLoop
i=I_step; A[*i*] = (*i*+1)/10 + (*i*-2)%10;	{body;}
i=I_step+1; A[*i*] = (*i*+1)/10 + (*i*-2)%10;	
I_div_10 = I_step/10;	IntervalLoop {
I_rem_10 = I_step%10;	HeadLoop
for(*i*,I_step+2,I_step+9,1)	{body;}
{A[*i*] = I_div_10 + I_rem_10;	BodyLoop
I_rem_10++;}	{optimized body;}
i=I_step+9; A[*i*] = (*i*+1)/10 + (*i*-2)%10;}	TailLoop
	{body;}}

Table 3.8: The previous example with correct loop splitting, showing peeled iterations; the general loop splitting transformation.

iteration ($i = 0$) uses the value 0 in place of the modulo function, where $-2\%10 = 8$ is the correct value. Because the range of values for the modulo in the interval is from 8 to 9 and then from 0 to 7, the restriction of Section 3.3.2 (a modulo cannot flip to zero in a loop splitting interval) has been violated. Clearly, the loop splitting interval cannot account for the offsets correctly.

One solution is to optimize only expressions without offsets. When many such division and modulo expressions exist, this option may be appealing, especially if optimizing the expressions with offsets reintroduces redundant computations by reducing the interval length. Of course, if all expressions have offsets, none are optimized.

A better, more general solution is to divide the interval into segments of continuous iterations with common values. In the example of Table 3.7, the ten-iteration interval is divided into three segments – iterations 1 and 2, iterations 3 through 9, and iteration 10. The first group of peeled iterations accounts for the modulo's negative offset, and the third group accounts for the division's positive offset. Table 3.8 shows the same loop with the first, second, and last iterations peeled away. All iterations now have the correct value, though only the middle segment has been optimized.

In general, many different offset values may exist, leading to the appearance of many small segments at both ends of the original interval. This thesis groups these little segments into two segments, one representing the maximum positive offset, and the other representing the minimum negative offset. Because they contain different sets of values,

```
aref (aref (A, I / i-spread) , I % i-spread);

aref (aref (B, (I / i-spread) + (i-proc * J / j-spread)),
    (I % i-spread) + i-spread * (J % j-spread));
```

Figure 3-1: The 1-D and 2-D array reference examples of Section 2.5.

these offset segments cannot be optimized.

Because offsets are very often small relative to the interval length, the simplicity of loop code outweighs the negligible cost of loop overhead; therefore, a loop represents each offset segment. While the *head loop* represents the initial iterations of the loop splitting interval and accounts for the negative offsets of the induction variable, the *tail loop* represents the final iterations of the interval and accounts for the positive offsets. The optimized loop representing all other iterations is called the *body loop*. Table 3.8 also depicts this general loop structure.

3.5 Loop Splitting for Array Referencing Optimizations

Now that sufficient background has been presented regarding both array referencing and loop splitting for compiler optimizations, this section explains how the loop splitting transformation can be applied to reduce the calculation complexity of distributed memory array references.

Section 2.5 described the expressions needed to reference an array; the two examples presented have been duplicated in Figure 3-1. In these expressions, the divisors and modulo bases are spreads of the data partition for the array. From the claims of Section 3.4, the loop splitting interval required for optimization is the greatest common factor of the spreads. Table 3.9 shows the original and optimized reference code for a general 1-D array reference A[i].

Because the data spread is the length of one dimension of a processor's array chunk, intervals based only on the spreads group iterations focused on only one chunk. In other words, spread-based intervals group iterations that reference the memory of only one

Source Code	Reference Code	After Loop Splitting
for(*i*,LL,UL,1) {temp = A[*i*];}	**for**(*i*,LL,UL,1) {temp = aref (aref (A, *i*/i_spread), *i*%i_spread);}	**for**(I_step,LL,UL,i_spread){ I_div = I_step/i_spread; I_rem = I_step%i_spread; **for**(*i*,I_step,I_step+i_spread,1) {temp = aref (aref (A, I_div), I_rem); I_rem++;}}

Table 3.9: The optimized general 1-D array reference.

processor. As mentioned in Section 3.5, the processor ID becomes constant (removing much of the computation), and the offset into that processor's memory can be computed with increments to variables rather than modulo functions (simplifying the remaining computation).

3.6 Summary

In this thesis, loop splitting is a combination of both splitting a loop into equal-sized intervals and peeling iterations from the intervals to account for induction variable offsets. Code hoisting and strength reduction are the two compiler optimizations allowed by loop splitting. In the context of array referencing on a distributed memory multiprocessor, the loop splitting intervals are determined by the data partitioning of the arrays. Such intervals allow a compiler to improve the code, reducing the execution time for array references.

Although loop splitting improves array references, the extent of the improvement is unclear. The next section attempts to clarify this issue by describing two loop splitting methods and the methodology of a loop splitting study. Experimental results are presented and interpreted.

Chapter 4

Loop Splitting Analysis

From the descriptions in the preceding sections, it seems clear that loop splitting can only improve program performance; but, it is unclear just how much. This section describes two loop splitting implementations and provides a quantitative analysis of their improvement on loop performance.

4.1 Introduction

When the task and data partitions are identical and all references have no induction variable offsets, the optimal code is easy to produce. In this case, all references in a processor's task are associated with data at a single processor[1]. Every iteration of the task has the same reference constants, and all remainder operations can be modeled with monotonically increasing or decreasing counters. No actual loop splitting needs to be performed, but code hoisting the divisions and reducing the remainders in strength must be done if an optimizing compiler were to improve code quality.

The optimal code becomes harder to produce, however, when the task and data partitions are different. Here, a processor's task is composed of subtasks, each of which corresponds to accessing data at a particular processor and has dimensions equal to the data spreads. Each subtask has different division constants and remainder behavior. Now, loop splitting is required in order to reduce the number of reference calculations, and the

[1]With proper placement, these two processors are one and the same.

loop splitting interval size for each dimension of the loop nest has the value of the data spread.

The presence of different data partitions in the same loop nest also complicates loop splitting. An example of how different data partitions are possible is adding two matrices A and B, of which B was computed previously as a matrix product. While A's distribution is influenced solely by the addition (a two-dimensional loop nest), B's partition is influenced by both the addition and the multiplication (a three-dimensional loop nest). In such a case, either array references are selectively optimized or the loop splitting interval becomes a function of the different data partitions.

As described in Section 3.4.2, induction variable offsets also complicate loop splitting. In such a situation, there is communication between different segments. In other words, in some of the iterations, data is located on more than one processor. In such iterations, array references do not all have division constants in common, and so these reference values are computed on *every* iteration, as in the original code. This problem is addressed by the head and tail loops of peeling.

With the above factors, it is hard to predict how well loop splitting will improve code. Therefore, to obtain a better understanding experiments were performed to determine the value of loop splitting. Code quality is a function of code size, loop overhead, register use, and number of operations, among other contributors. This thesis, however, uses execution time as the sole determinant of code quality. From the execution times, the performance improvement of the loop code was determined. This improvement was calculated as the ratio of the execution time before loop splitting to the execution time after loop splitting.

The following subsections determine the quantitative benefits of loop splitting in the presence of the above complications. First, two implemented methods of loop splitting are introduced, and the procedure for performance analysis is described. Then, the experimental results are presented, and the benefits of loop splitting are interpreted from the analysis.

4.2 Implemented Methods

Before proceeding to describe the experiments, this section presents the two methods of loop splitting used in the study – rational splitting and interval splitting.

4.2.1 Rational Splitting

Rational splitting is the method outlined in Section 3.4.1, where the first detailed explanation of loop splitting was presented. The following is a more specific description.

Each loop in the original loop nest is divided into subloops; however, these subloops have the same number of iterations. In order to optimize all references, the subloop size must obey all constraints imposed by the data spreads in the references. To have such a quality, the loop splitting interval must be a common divisor of the data spreads. The trivial common divisor is one, which is the subloop size of the original loop code. However, to minimize the amount of redundant computation, the number of subloops must also be minimized (their size maximized). A size equal to the largest common divisor achieves this so that the optimal subloop size for rational loop splitting is the greatest common divisor (gcd) of the data spreads. Thus, a single number (the gcd) specifies how the loop is split; this minimal information keeps code size small.

Unfortunately, as mentioned earlier, it is likely that the data spreads will have a gcd of one. In such a case, no references are optimized, and the resulting code is slower than the original due to the extra loop overhead on every iteration.

4.2.2 Interval Splitting

Interval splitting is a different approach to the problem of multiple data partitions. Where rational splitting uses a single number to divide a loop into subloops, interval splitting uses a list of numbers, explicitly specifying the subloops. With this list, the compiler has already decided exactly how to divide the loop for each processor so that *all* redundant computations are avoided.

Unfortunately, because each processor can have a unique list of intervals, as the number of processors increases the code size also increases. With a 256 processor network, for example, the case statement assigning the interval lists contains 256 lists. On a thousand-node network, very simple code can become alarmingly large.

4.3 Methodology

To study the effects of loop splitting on multiprocessor performance, the preceding transformations were added to the Alewife precompiler. The following describes the analysis procedure used to perform the experiments.

First, a statically parallel program was transformed by the precompiler and then compiled by the Orbit optimizing compiler ([Kra88]) so that full advantage was taken of the optimizing opportunities provided by loop splitting (as described in Section 3.5). The resulting object code was then executed by Asim ([Nus91]), the simulator for the Alewife multiprocessor architecture, which tabulated the total number of execution cycles for the program. To isolate the loop execution time, the simulation time when starting the loop was subtracted from the simulation time at completion.

To keep the results focused on loop splitting, Asim was used with the following multiprocessor model. First, every memory reference was considered a cache hit. This not only prevented the memory latency from affecting loop performance but also allowed comparisons among different programs with different references by making memory access time constant (subsequent memory accesses are not favored over the first one, which would normally be a cache miss). Second, all network requests were resolved immediately, removing the effects of communication (network topology, routing, placement) from the results.

The experimental plots are improvement of loop performance against the number of processors. Improvement was computed as the ratio of the original loop's execution time to that of the loop after loop splitting. For comparison and generality, improvement was calculated with the original and transformed codes both optimized by the Orbit compiler and unoptimized. The number of processors was the variable in the experiments because it determined the task and data partitions, in turn determining the data spreads that affect the loop splitting intervals.

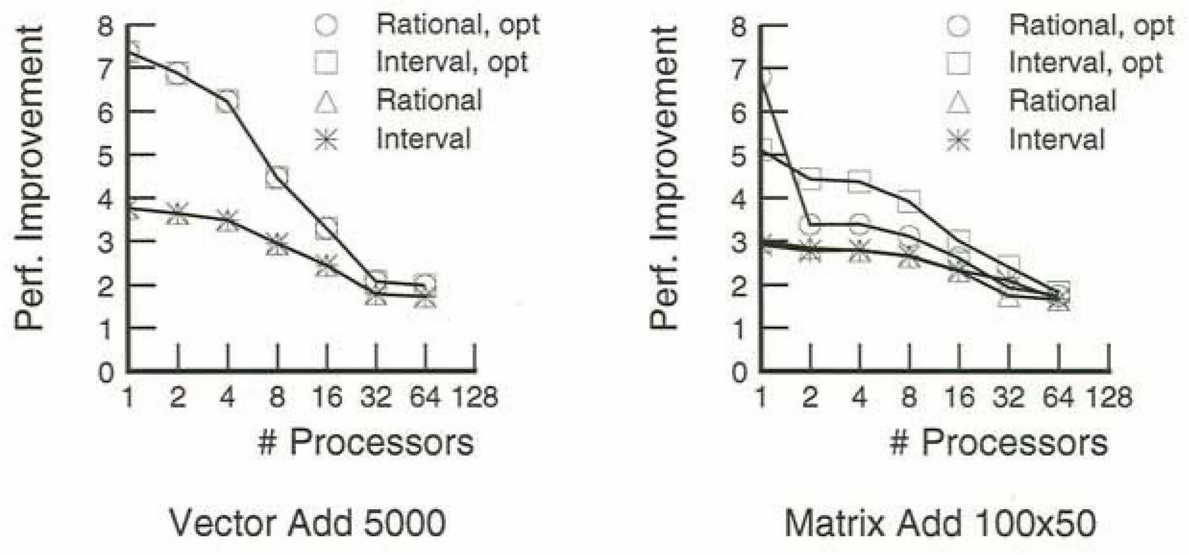


Figure 4-1: The performance improvement in addition on 5000-element vectors and 100x50-element matrices.

The programs used to analyze loop splitting effects were chosen to be representative of the various routines found in scientific code. These programs are vector addition, matrix addition, matrix multiplication, and matrix transposition. Appendix A contains the source code for these programs as well as the object code of the precompiler after the different states of transformation.

4.4 Results

The simulation results of the experiments on vector and matrix addition are shown in Figure 4-1, and the results of the matrix multiplication and transposition are shown in Figure 4-2.

Several interesting phenomena can be discerned from the graphs. First, while vector and matrix addition show a rather steady decrease in the performance gain due to loop splitting, matrix multiplication and transposition exhibit sawtooth behavior as their performance decreases with an increase in processors. The cause of local minima in performance is the presence of incongruent data spreads, requiring more calculation than well-behaved ones.

Second, Figure 4-1 also shows that the two loop splitting methods have more improvement when an optimizing compiler is used on the code both before and after loop splitting. This makes sense because loop splitting not only reduces the reference

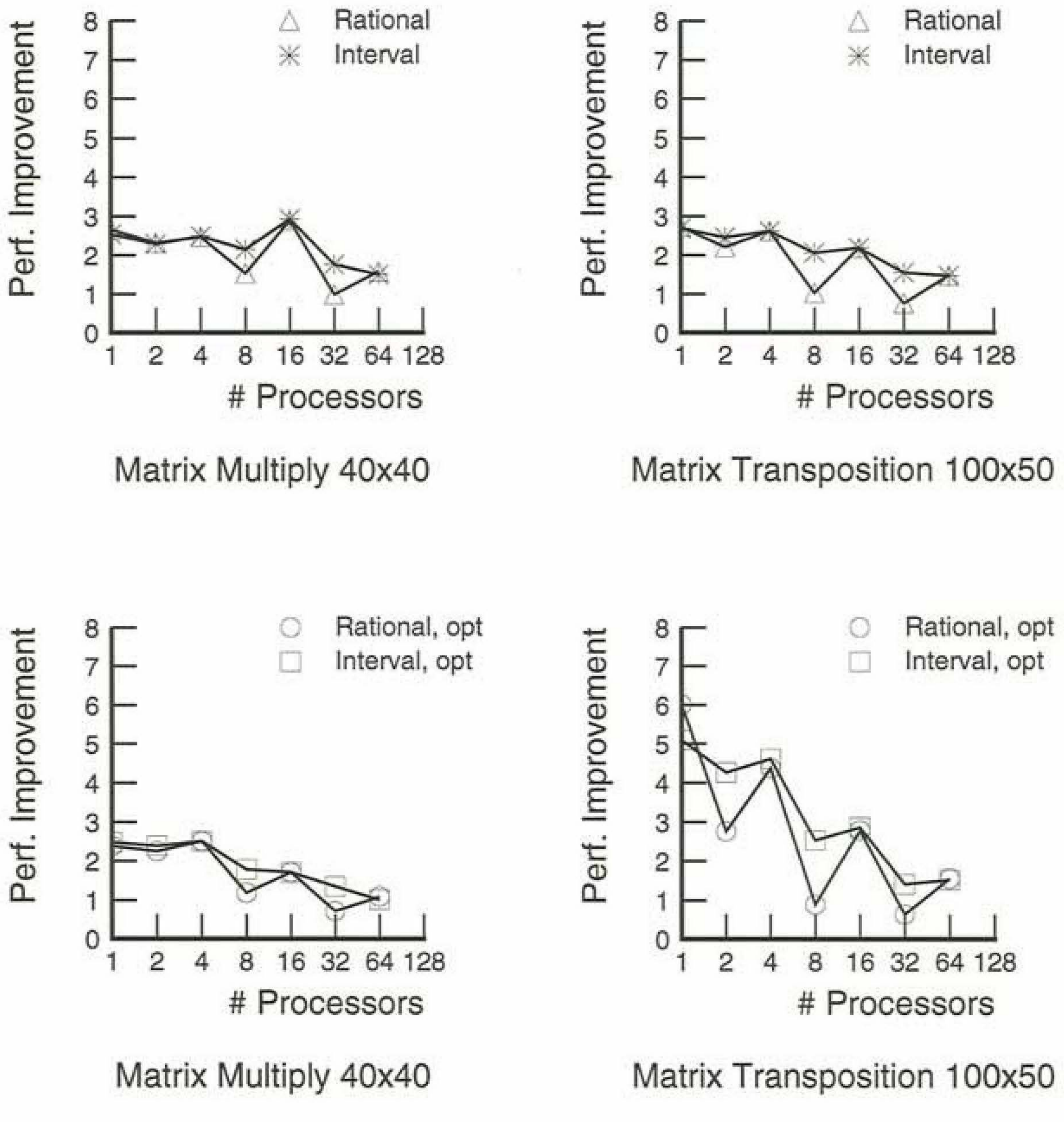


Figure 4-2: The performance improvement in multiplication of 40x40 matrices and transposition of a 100x50 matrix: without additional compiler optimizations **(top)** and with the optimizations **(bottom)**.

calculations, but also allows more opportunities of which an optimizing compiler can take advantage.

Third, with or without an optimizing compiler, the rational and interval splitting methods produce the same performance gain in the vector and matrix addition. This is due to the fact that all the arrays have the same partition in these programs.

In Figure 4-2, the optimized and unoptimized cases are separated to show the results more clearly. We can see that the interval splitting method behaves as an upper bound on loop splitting performance. It is also apparent that 2, 8, and 32 processors lead to incongruent data spreads because the improvement from loop splitting degrades at those points, especially at 8 and 32 processors where rational splitting often performs worse than the original code (improvement is less than one). At these numbers of processors, interval splitting is much more resistant to the incongruency by retaining a higher performance improvement.

4.5 Interpretation

From the plots in the previous section, several interpretations have been made.

First, in general loop splitting clearly improves loop execution time. However, if performed incorrectly with incongruent data spreads, the performance gain can actually become a performance loss. Therefore, it is important to produce code with partitions that are multiples of each other or are the same. If this cannot be done, then interval splitting should be performed since it resists the incongruencies better by avoiding any redundant computations.

Second, incongruent data spreads are not uncommon. None of the preceding experiments were contrived to show how badly rational splitting can perform. Therefore, in order to reduce the calculation in array referencing, a compiler with loop splitting should take care not to create data partitions that are very similar but not exactly the same, since those are the types that require more calculation from interval splitting (proportional to the number of different partitions) and cause rational splitting to perform worse than the original code (by having a gcd of 1).

Third, loop splitting of any type should be performed with further compiler optimizations following. Since the loop splitting produces new constants, the potential for additional optimization should be used.

Chapter 5

Conclusions

This section summarizes the results and contributions of this thesis, and suggests a possible path for future research.

5.1 Summary

Array referencing on a distributed memory multiprocessor requires costly calculations. An array cell is not only identified by its position in memory, but also in which processor's memory.

To reduce the calculation needed for array referencing in a static parallel program, loop splitting can be employed to divide a single loop into subloops. Once these subloops expose invariants, code hoisting and strength reduction improve the code quality. These code transformations allow further calculation reduction by an optimizing compiler.

We have introduced two methods of loop splitting that improve all array references of a loop – rational splitting and interval splitting. While rational splitting creates subloops of length equal to the greatest common divisor of the data spreads, interval splitting explicitly defines the subloops in order to remove all redundant computations.

Implementations of the above methods were created and used to analyze the improvement of loop execution time. The results on four different benchmarks in two compiler settings were presented. The following summarizes the interpretations of the study.

Overall, loop splitting is an effective technique to lower the computation cost of array references on a distributed memory multiprocessor. In most cases, it is easy to improve the execution time of a loop to 2 to 3 times faster.

The rational splitting method has code size independent of the number of processors but has good probability of reintroducing much of the array reference computations. In some cases, where data partitions were similar but not identical, rational splitting performs worse than the original code. Interval splitting, on the other hand, minimizes the number of computations but has the potential of large code size with a high number of processors. However, incongruent data partitions still greatly diminish its effectiveness.

Two solutions to the problems above seem apparent. First, where data partitions are similar, they can be forced identical, greatly improving the gains of loop splitting by avoiding data spreads with a gcd of 1. Because the partitions were similar, the change will not have a major effect on communication, load balancing, etc. Second, interval splitting should be used only on a relatively small number of processors, such as 256 and lower, depending on how large a reasonable-size program is defined. In a real multiprocessor with more than 256 nodes, communication would likely be the major contributor to execution time, in which case loop splitting would hardly be helpful.

5.2 Future Work

The research of this thesis is by no means complete. To extend the study, several suggestions are made below.

Currently, the Alewife compiler does not support 3-D arrays. Implementing n-dimensional arrays would help evaluate the growth of array reference complexity with array dimension.

In addition to the two methods described in this thesis, other techniques of loop splitting are possible. One such example that has been considered is *single partition* splitting. This method selects a single representative partition to dictate the loop splitting intervals. While every reference to arrays with this partitioning is fully optimized (no redundant computations), all other references remain unchanged.

This method was not implemented due to many difficulties. One problem with this method is deciding the rules for choosing the "representative" partition. It is not hard to produce problematic programs in which a particular heuristic either optimizes the minimal number of references, or optimizes none at all. Also, the references not optimized become slower than they previously were either due to the computation needed to reconstruct behavior of the induction variables before loop splitting or due to the extra registers simulating that behavior.

Though single partition splitting possesses these difficulties, an analysis of its performance would offer concrete evidence of its effectiveness. Therefore, perhaps an implementation of the single partition method should be created, and the resulting performance improvement compared with the two methods analyzed.

Further, results with more aggressive benchmarks, such as Wave or a 3-D relaxation, may prove interesting; and, the performance of loop splitting should be evaluated in a real system with communication, caches, etc.

Bibliography

[A*et al.*91] A. Agarwal *et al.* The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. An extended version of this paper has been submitted for publication, and appears as MIT/LCS Memo TM-454, 1991.

[AKN92] Anant Agarwal, David Kranz, and Venkat Natarajan. Automatic Partitioning of Parallel Loops for Cache-Coherent Multiprocessors. Technical Report MIT/LCS TM-481, Massachusetts Institute of Technology, December 1992. A version of this report appears in ICPP 1993.

[Kra88] David A. Kranz. *ORBIT: An Optimizing Compiler for Scheme*. PhD thesis, Yale University, February 1988. Technical Report YALEU/DCS/RR-632.

[Nus91] Dan Nussbaum. ASIM Reference Manual. ALEWIFE Memo No. 13, Laboratory for Computer Science, Massachusetts Institute of Technology, January 1991.

[Rus78] Richard M. Russel. The CRAY-1 Computer System. *Communications of the ACM*, 21(1):63–72, January 1978.

Appendix A

Performance Benchmarks

Each benchmark is shown as it would look in the four states of the loop splitting experiments on a multiprocessor of 8 PEs. The four states are: the programmer's source code, the precompiler's object code, the object code after rational splitting, and the object code after interval splitting. The code is shown in the language Mul-T, a parallel version of T (a dialect of Lisp). In the interest of saving space, only the loop is shown in the object code.

The reference functions jref and lref correspond to J-structures and L-structures, two methods of synchronization. They behave the same way as aref in terms of referencing an array cell.

A.1 Vector Add 5000

The 5000-element 1-D array is distributed among eight processors in groups of 625 array cells; therefore, the data spread of the data partitioning is 625.

Source Code

```
;;; C = A + B
(define (main)
  (define a (array (5000) 1))
  (define b (array (5000) 2))
  (define c (array (5000) 0))
  (format t "Beginning the loop!~%")
  (doall (i 0 4999)
         (set (aref c i) (+ (aref a i)
                            (aref b i))))
  (format t "Ending the loop!~%")
  (return c))
```

Object Code

```
(WAIT-SPAWN-COMPLETE
 (TREE-SPAWN 1
  (LAMBDA (TID ())
   (LET* ((I (FX-REM TID 8))
          (I-MIN (FX* I 625)))
    (%%DO
     (I_25 (FIXNUM-MAX I-MIN 0)
   (FIXNUM-MIN (FX+ I-MIN 625) 5000)
   1)
     ((LAMBDA (SET.2447_41)
(LET ((I.2448 I_25))
  (SET
   (AREF
    (JREF C (FX/ I.2448 625))
    (FX-REM I.2448 625))
   SET.2447_41))
SET.2447_41)
      (FIXNUM-ADD
       (LET ((I.2449 I_25))
 (AREF
  (JREF A (FX/ I.2449 625))
  (FX-REM I.2449 625)))
       (LET ((I.2450 I_25))
 (AREF
  (JREF B (FX/ I.2450 625))
  (FX-REM I.2450 625))))))))))
```

After Rational Splitting

```
(WAIT-SPAWN-COMPLETE
 (TREE-SPAWN 1
  (LAMBDA (TID ())
    (LET* ((I (FX-REM TID 8))
   (I-MIN (FX+ (FX* I 625) 0))
   (I-MAX
    (FIXNUM-MIN (FX+ I-MIN 625) 5000)))
      (LET* ((I-BMIN I-MIN)
     (I-BMAX (FX- I-MAX 1))
     (DIV-B-I625
      (JREF B (FX/ I-BMIN 625)))
     (DIV-A-I625
      (JREF A (FX/ I-BMIN 625)))
     (DIV-C-I625
      (JREF C (FX/ I-BMIN 625))))
(DO
    ((I-REM-625 (FX-REM I-BMIN 625)
(FX+ I-REM-625 1)))
    ((FX> I-REM-625 (FX-REM I-BMAX 625)))
  ((LAMBDA (SET.2366_41)
     (SET
      (AREF DIV-C-I625 I-REM-625)
      SET.2366_41)
     SET.2366_41)
   (FIXNUM-ADD
    (AREF DIV-A-I625 I-REM-625)
    (AREF DIV-B-I625 I-REM-625)))))))))
```

After Interval Splitting

```
(WAIT-SPAWN-COMPLETE
 (TREE-SPAWN 1
  (LAMBDA (TID ())
    (LET* ((I-INTERVALS
    (CASE (FX-REM TID 8)
      ((0) (LIST 0 625))
      ((1) (LIST 625 1250))
      ((2) (LIST 1250 1875))
      ((3) (LIST 1875 2500))
      ((4) (LIST 2500 3125))
      ((5) (LIST 3125 3750))
      ((6) (LIST 3750 4375))
      ((7) (LIST 4375 5000))
      (ELSE
       (FORMAT T "Case error~%")))))
      (LET* ((I-BMIN
      (FX+ (CAR I-INTERVALS) 0))
     (I-BMAX
      (FX- (CADR I-INTERVALS) 1))
     (DIV-B-I625
      (JREF B (FX/ I-BMIN 625)))
     (DIV-A-I625
      (JREF A (FX/ I-BMIN 625)))
     (DIV-C-I625
      (JREF C (FX/ I-BMIN 625))))
(DO
    ((I-REM-625 (FX-REM I-BMIN 625)
(FX+ I-REM-625 1)))
    ((FX> I-REM-625 (FX-REM I-BMAX 625)))
  ((LAMBDA (SET.2622_41)
     (SET
      (AREF DIV-C-I625 I-REM-625)
      SET.2622_41)
     SET.2622_41)
   (FIXNUM-ADD
    (AREF DIV-A-I625 I-REM-625)
    (AREF DIV-B-I625 I-REM-625)))))))))
```

A.2 Matrix Add 100x50

The 5000-element 2-D array is distributed among six processors in groups of 850 array cells; therefore, the data spread of the data partitioning is 50x17.

Source Code

```
;;; C = A + B
(define (main)
   (define a (array (100 50) 1))
   (define b (array (100 50) 2))
   (define c (array (100 50) 0))
   (format t "Beginning the loop!~%")
   (doall (i 0 99)
      (doall (j 0 49)
         (set (aref c i j) (+ (+ (aref a i j) (aref b i j))))))
   (format t "Ending the loop!~%")
   c)
```

Object Code

```
(WAIT-SPAWN-COMPLETE
 (TREE-SPAWN 1
  (LAMBDA (TID ())
    (LET* ((I (FX-REM TID 3))
   (J (FX/ TID 2))
   (I-MIN (FX* I 50))
   (J-MIN (FX* J 17)))
      (%%DO
       (I_25 (FIXNUM-MAX I-MIN 0)
     (FIXNUM-MIN (FX+ I-MIN 50) 100)
     1)
       (%%DO
(J_28 (FIXNUM-MAX J-MIN 0)
      (FIXNUM-MIN (FX+ J-MIN 17) 50)
      1)
((LAMBDA (SET.2552_44)
   (LET ((I.2553 I_25)
 (J.2554 J_28))
      (SET
       (AREF
        (JREF C
      (FX+ (FX/ I.2553 50)
  (FX* 2 (FX/ J.2554 17))))
        (FX+ (FX-REM I.2553 50)
    (FX* 50 (FX-REM J.2554 17))))
       SET.2552_44))
   SET.2552_44)
 (FIXNUM-ADD
  (LET ((I.2555 I_25)
(J.2556 J_28))
    (AREF
     (JREF A
   (FX+ (FX/ I.2555 50)
(FX* 2 (FX/ J.2556 17))))
     (FX+ (FX-REM I.2555 50)
  (FX* 50 (FX-REM J.2556 17)))))
  (LET ((I.2557 I_25)
(J.2558 J_28))
    (AREF
     (JREF B
   (FX+ (FX/ I.2557 50)
(FX* 2 (FX/ J.2558 17))))
     (FX+ (FX-REM I.2557 50)
  (FX* 50 (FX-REM J.2558 17))))))))))))))
```

After Rational Splitting

```
(WAIT-SPAWN-COMPLETE
 (TREE-SPAWN 1
  (LAMBDA (TID ())
    (LET* ((I (FX-REM TID 3))
   (J (FX/ TID 2))
   (I-MIN (FX+ (FX* I 50) 0))
   (J-MIN (FX+ (FX* J 17) 0))
   (I-MAX
    (FIXNUM-MIN (FX+ I-MIN 50) 100))
   (J-MAX
    (FIXNUM-MIN (FX+ J-MIN 17) 50)))
      (LET* ((I-BMIN I-MIN)
     (I-BMAX (FX- I-MAX 1)))
(DO
    ((I-REM-50 (FX-REM I-BMIN 50)
       (FX+ I-REM-50 1)))
    ((FX> I-REM-50 (FX-REM I-BMAX 50)))
  (LET* ((J-BMIN J-MIN)
 (J-BMAX (FX- J-MAX 1))
 (DIV-B-I50-J17
  (JREF B
(FX+ (FX/ I-BMIN 50)
     (FX* 2 (FX/ J-BMIN 17)))))
 (DIV-A-I50-J17
  (JREF A
(FX+ (FX/ I-BMIN 50)
     (FX* 2 (FX/ J-BMIN 17)))))
 (DIV-C-I50-J17
  (JREF C
(FX+ (FX/ I-BMIN 50)
     (FX* 2 (FX/ J-BMIN 17))))))
    (DO
((J-REM-17 (FX-REM J-BMIN 17)
   (FX+ J-REM-17 1)))
((FX> J-REM-17 (FX-REM J-BMAX 17)))
      ((LAMBDA (SET.2288_44)
 (SET
  (AREF DIV-C-I50-J17
(FX+ I-REM-50
     (FX* 50 J-REM-17)))
  SET.2288_44)
 SET.2288_44)
       (FIXNUM-ADD
(AREF DIV-A-I50-J17
      (FX+ I-REM-50
   (FX* 50 J-REM-17)))
(AREF DIV-B-I50-J17
      (FX+ I-REM-50
   (FX* 50 J-REM-17))))))))))))
```

After Interval Splitting

```
(WAIT-SPAWN-COMPLETE
 (TREE-SPAWN 1
  (LAMBDA (TID ())
    (IF (FX< TID 6)
(LET* ((I-INTERVALS
(CASE (FX-REM TID 3)
  ((0) (LIST 0 50))
  ((1) (LIST 50 100))
  (ELSE
   (FORMAT T "Case error~%"))))
       (J-INTERVALS
(CASE (FX/ TID 2)
  ((0) (LIST 0 17))
  ((1) (LIST 17 34))
  ((2) (LIST 34 50))
  (ELSE
   (FORMAT T "Case error~%")))))
  (LET* ((I-BMIN
  (FX+ (CAR I-INTERVALS) 0))
 (I-BMAX
  (FX- (CADR I-INTERVALS) 1)))
    (DO
((I-REM-50 (FX-REM I-BMIN 50)
   (FX+ I-REM-50 1)))
((FX> I-REM-50 (FX-REM I-BMAX 50)))
      (LET* ((J-BMIN
      (FX+ (CAR J-INTERVALS) 0))
     (J-BMAX
      (FX- (CADR J-INTERVALS) 1))
     (DIV-B-I50-J17
      (JREF B
    (FX+ (FX/ I-BMIN 50)
 (FX* 2
      (FX/ J-BMIN 17)))))
     (DIV-A-I50-J17
      (JREF A
    (FX+ (FX/ I-BMIN 50)
 (FX* 2
      (FX/ J-BMIN 17)))))
     (DIV-C-I50-J17
      (JREF C
    (FX+ (FX/ I-BMIN 50)
 (FX* 2
      (FX/ J-BMIN 17))))))
(DO
    ((J-REM-17 (FX-REM J-BMIN 17)
       (FX+ J-REM-17 1)))
    ((FX> J-REM-17
  (FX-REM J-BMAX 17)))
  ((LAMBDA (SET.1877_44)
     (SET
      (AREF DIV-C-I50-J17
    (FX+ I-REM-50
 (FX* 50 J-REM-17)))
      SET.1877_44)
     SET.1877_44)
   (FIXNUM-ADD
    (AREF DIV-A-I50-J17
(FX+ I-REM-50
     (FX* 50 J-REM-17)))
    (AREF DIV-B-I50-J17
(FX+ I-REM-50
     (FX* 50 J-REM-17))))))))))))))))
```

A.3 Matrix Multiplication 40x40

The 1600-element 2-D array is distributed among eight processors in groups of 280 array cells; therefore, the data spread of the data partitioning is 20x14.

Source Code

```
(define (main)
  (define x (lstruct (40 40) 0))
  (define y (array (40 40) 2))
  (define z (array (40 40) 2))
  (format t "Beginning the loop!~%")
  (doall (i 0 39)
    (doall (j 0 39)
      (doall (k 0 39)
        (set (lref x i j)
             (fx+ (lref x i j)
                  (fx* (aref y i k) (aref z k j)))))))
  (format t "Ending the loop!~%")
  (return x))
```

Object Code

```
(WAIT-SPAWN-COMPLETE
 (TREE-SPAWN 1
  (LAMBDA (TID ())
    (LET* ((I (FX-REM TID 2))
   (J (FX-REM (FX/ TID 2) 2))
   (K (FX/ TID (FX* 2 2)))
   (I-MIN (FX* I 20))
   (J-MIN (FX* J 20))
   (K-MIN (FX* K 20)))
      (%%DO
       (I_26 (FIXNUM-MAX I-MIN 0)
     (FIXNUM-MIN (FX+ I-MIN 20) 40)
     1)
       (%%DO
(J_29 (FIXNUM-MAX J-MIN 0)
      (FIXNUM-MIN (FX+ J-MIN 20) 40)
      1)
(%%DO
 (K_32 (FIXNUM-MAX K-MIN 0)
       (FIXNUM-MIN (FX+ K-MIN 20) 40)
       1)
 ((LAMBDA (SET.2678_56)
    (LET ((I.2679 I_26)
  (J.2680 J_29))
      (SET
       (LREF
(JREF X
      (FX+ (FX/ I.2679 20)
   (FX* 2 (FX/ J.2680 14))))
(FX+ (FX-REM I.2679 20)
     (FX* 20 (FX-REM J.2680 14))))
       SET.2678_56))
    SET.2678_56)
  (FIXNUM-ADD
   (LET ((I.2681 I_26)
(J.2682 J_29))
     (LREF
      (JREF X
    (FX+ (FX/ I.2681 20)
(FX* 2 (FX/ J.2682 14))))
      (FX+ (FX-REM I.2681 20)
  (FX* 20 (FX-REM J.2682 14)))))
  (FIXNUM-MULTIPLY
   (LET ((I.2683 I_26)
 (J.2684 K_32))
     (AREF
      (JREF Y
    (FX+ (FX/ I.2683 20)
 (FX* 2 (FX/ J.2684 14))))
      (FX+ (FX-REM I.2683 20)
   (FX* 20 (FX-REM J.2684 14)))))
   (LET ((I.2685 K_32)
 (J.2686 J_29))
     (AREF
      (JREF Z
    (FX+ (FX/ I.2685 20)
 (FX* 2 (FX/ J.2686 14))))
      (FX+ (FX-REM I.2685 20)
   (FX* 20 (FX-REM J.2686 14))))))))))))))))
```

After Rational Splitting

```
(WAIT-SPAWN-COMPLETE
 (TREE-SPAWN 1
  (LAMBDA (TID ())
   (LET* ((OFFSET (FX-REM TID 2))
  (TID
   (FX+
    (VREF '#(0 1) OFFSET)
    (FX- TID OFFSET))))
     (LET* ((I (FX-REM TID 2))
    (J (FX-REM (FX/ TID 2) 2))
    (K (FX/ TID 4))
    (I-MIN (FX+ (FX* I 20) 0))
    (J-MIN (FX+ (FX* J 20) 0))
    (K-MIN (FX+ (FX* K 20) 0))
    (I-MAX
     (FIXNUM-MIN (FX+ I-MIN 20) 40))
    (J-MAX
     (FIXNUM-MIN (FX+ J-MIN 20) 40))
    (K-MAX
     (FIXNUM-MIN (FX+ K-MIN 20) 40)))
      (LET* ((I-BMIN I-MIN)
     (I-BMAX (FX- I-MAX 1)))
  (DO
      ((I-REM-20 (FX-REM I-BMIN 20)
(FX+ I-REM-20 1)))
      ((FX> I-REM-20 (FX-REM I-BMAX 20)))
    (DO
        ((J-LOOP-START
  (FX* 14 (FX/ J-MIN 14))
  (FX+ J-LOOP-START 14)))
        ((FX>= J-LOOP-START J-MAX))
     (LET* ((J-BMIN
     (FIXNUM-MAX J-MIN
 (FX+ J-LOOP-START 0)))
    (J-BMAX
     (FX-
      (FIXNUM-MIN J-MAX
  (FX+ J-LOOP-START 14))
      1))
    (DIV-X-I20-J14
     (JREF X
   (FX+ (FX/ I-BMIN 20)
(FX* 2
     (FX/ J-BMIN 14))))))
       (DO
   ((J-REM-14 (FX-REM J-BMIN 14)
      (FX+ J-REM-14 1)))
   ((FX> J-REM-14 (FX-REM J-BMAX 14)))
 (DO
     ((K-LOOP-START
       (FX* 2 (FX/ K-MIN 2))
       (FX+ K-LOOP-START 2)))
     ((FX>= K-LOOP-START K-MAX))
   (LET* ((K-BMIN
   (FIXNUM-MAX K-MIN
       (FX+ K-LOOP-START 0)))
  (K-BMAX
   (FX-
    (FIXNUM-MIN K-MAX
(FX+ K-LOOP-START
     2))
    1))
  (DIV-Z-K20-J14
   (JREF Z
 (FX+ (FX/ K-BMIN 20)
      (FX* 2
   (FX/ J-BMIN 14)))))
  (DIV-Y-I20-K14
   (JREF Y
 (FX+ (FX/ I-BMIN 20)
      (FX* 2
   (FX/ K-BMIN 14))))))
    (DO
 ((K-REM-20 (FX-REM K-BMIN 20)
    (FX+ K-REM-20 1))
  (K-REM-14 (FX-REM K-BMIN 14)
    (FX+ K-REM-14 1)))
 ((FX> K-REM-14
       (FX-REM K-BMAX 14)))
       ((LAMBDA (SET.2210_56)
  (SET
   (LREF DIV-X-I20-J14
 (FX+ I-REM-20
      (FX* 20 J-REM-14)))
   SET.2210_56)
  SET.2210_56)
(FIXNUM-ADD
 (LREF DIV-X-I20-J14
       (FX+ I-REM-20
```

```
    (FX* 20 J-REM-14)))
 (FIXNUM-MULTIPLY
  (AREF DIV-Y-I20-K14
(FX+ I-REM-20
     (FX* 20 K-REM-14)))
  (AREF DIV-Z-K20-J14
(FX+ K-REM-20
     (FX* 20 J-REM-14))))))))))))))))))
```

After Interval Splitting

```
(WAIT-SPAWN-COMPLETE
 (TREE-SPAWN 1
  (LAMBDA (TID ())
   (LET* ((OFFSET (FX-REM TID 2))
  (TID
   (FX+
    (VREF '#(0 1) OFFSET)
    (FX- TID OFFSET))))
     (IF (FX< TID 8)
 (LET* ((I-INTERVALS
 (CASE (FX-REM TID 2)
   ((0) (LIST 0 20))
   ((1) (LIST 20 40))
   (ELSE
    (FORMAT T "Case error~%"))))
(J-INTERVALS
 (CASE
     (FX-REM (FX/ TID 2) 2)
   ((0) (LIST 0 14 20))
   ((1) (LIST 20 28 40))
   (ELSE
    (FORMAT T "Case error~%"))))
(K-INTERVALS
 (CASE (FX/ TID 4)
   ((0) (LIST 0 14 20))
   ((1)
    (LIST 20 28 40))
   (ELSE
    (FORMAT T "Case error~%")))))
   (LET* ((I-BMIN
   (FX+ (CAR I-INTERVALS) 0))
  (I-BMAX
   (FX- (CADR I-INTERVALS) 1)))
     (DO
 ((I-REM-20 (FX-REM I-BMIN 20)
    (FX+ I-REM-20 1)))
 ((FX> I-REM-20 (FX-REM I-BMAX 20)))
       (%%DO-INT (J-INT J-INTERVALS)
 (LET* ((J-BMIN
 (FX+ (CAR J-INT) 0))
(J-BMAX
 (FX- (CADR J-INT) 1))
(DIV-X-I20-J14
 (JREF X
       (FX+
(FX/ I-BMIN 20)
(FX* 2
     (FX/ J-BMIN 14))))))
 (DO
     ((J-REM-14
(FX-REM J-BMIN 14)
(FX+ J-REM-14 1)))
     ((FX> J-REM-14
   (FX-REM J-BMAX 14)))
   (%%DO-INT
    (K-INT K-INTERVALS)
    (LET* ((K-BMIN
    (FX+ (CAR K-INT) 0))
   (K-BMAX
    (FX- (CADR K-INT)
 1))
   (DIV-Z-K20-J14
    (JREF Z
  (FX+
   (FX/ K-BMIN 20)
   (FX* 2
 (FX/ J-BMIN
     14))))))
   (DIV-Y-I20-K14
    (JREF Y
  (FX+
   (FX/ I-BMIN 20)
   (FX* 2
 (FX/ K-BMIN
     14))))))
(DO
  ((K-REM-20
    (FX-REM K-BMIN 20)
    (FX+ K-REM-20 1))
   (K-REM-14
    (FX-REM K-BMIN 14)
    (FX+ K-REM-14 1)))
  ((FX> K-REM-14
 (FX-REM K-BMAX 14)))
 ((LAMBDA (SET.2367_56)
    (SET
    (LREF
     DIV-X-I20-J14
     (FX+ I-REM-20
  (FX* 20 J-REM-14)))
     SET.2367_56)
    SET.2367_56)
  (FIXNUM-ADD
   (LREF DIV-X-I20-J14
 (FX+ I-REM-20
      (FX* 20
  J-REM-14)))
   (FIXNUM-MULTIPLY
    (AREF
     DIV-Y-I20-K14
     (FX+ I-REM-20
  (FX* 20 K-REM-14)))
    (AREF
     DIV-Z-K20-J14
     (FX+ K-REM-20
  (FX* 20 J-REM-14)))))))))))))))))))))
```

A.4 Matrix Transposition

100x50

The 5000-element 1-D array is distributed among six processors in groups of 850 array cells; therefore, the data spread of the data partitioning is 25x34.

Source Code

```
;;; B = A transpose
(define (main)
   (define a (array (100 50) 9))
   (define b (array (50 100) 0))
   (format t "Beginning the loop!~%")
   (doall (x 0 49)
      (doall (y 0 99)
         (set (aref b x y) (aref a y x))))
   (format t "Ending the loop!~%")
   (return b))
```

Object Code

```
(WAIT-SPAWN-COMPLETE
 (TREE-SPAWN 1
  (LAMBDA (TID ())
   (LET* ((I (FX-REM TID 2))
  (J (FX/ TID 2))
  (I-MIN (FX* I 25))
  (J-MIN (FX* J 34)))
     (%%DO
      (X_20 (FIXNUM-MAX I-MIN 0)
    (FIXNUM-MIN (FX+ I-MIN 25) 50)
    1)
      (%%DO
       (Y_23 (FIXNUM-MAX J-MIN 0)
     (FIXNUM-MIN (FX+ J-MIN 34) 100)
     1)
        ((LAMBDA (SET.2802_34)
  (LET ((I.2803 X_20)
(J.2804 Y_23))
     (SET
      (AREF
       (JREF B
     (FX+ (FX/ I.2803 25)
 (FX* 2 (FX/ J.2804 34))))
       (FX+ (FX-REM I.2803 25)
   (FX* 25 (FX-REM J.2804 34))))
      SET.2802_34))
  SET.2802_34)
(LET ((I.2805 Y_23)
      (J.2806 X_20))
  (AREF
   (JREF A
 (FX+ (FX/ I.2805 50)
      (FX* 2 (FX/ J.2806 17))))
   (FX+ (FX-REM I.2805 50)
(FX* 50 (FX-REM J.2806 17))))))))))))
```

After Rational Splitting

```
(WAIT-SPAWN-COMPLETE
 (TREE-SPAWN 1
  (LAMBDA (TID ())
   (LET* ((OFFSET (FX-REM TID 2))
  (TID
   (FX+
    (VREF '#(0 1) OFFSET)
    (FX- TID OFFSET))))
     (LET* ((X (FX-REM TID 2))
    (Y (FX/ TID 2))
    (X-MIN (FX+ (FX* X 25) 0))
    (Y-MIN (FX+ (FX* Y 34) 0))
    (X-MAX
     (FIXNUM-MIN (FX+ X-MIN 25) 50))
    (Y-MAX
     (FIXNUM-MIN (FX+ Y-MIN 34) 100)))
       (DO
   ((X-LOOP-START (FX* 1 (FX/ X-MIN 1))
  (FX+ X-LOOP-START 1)))
   ((FX>= X-LOOP-START X-MAX))
 (LET* ((X-BMIN
 (FIXNUM-MAX X-MIN
     (FX+ X-LOOP-START 0)))
(X-BMAX
 (FX-
  (FIXNUM-MIN X-MAX
      (FX+ X-LOOP-START 1))
  1)))
   (DO
       ((X-REM-17 (FX-REM X-BMIN 17)
  (FX+ X-REM-17 1))
(X-REM-25 (FX-REM X-BMIN 25)
  (FX+ X-REM-25 1)))
       ((FX> X-REM-25 (FX-REM X-BMAX 25)))
     (DO
 ((Y-LOOP-START
   (FX* 2 (FX/ Y-MIN 2))
   (FX+ Y-LOOP-START 2)))
 ((FX>= Y-LOOP-START Y-MAX))
       (LET* ((Y-BMIN
       (FIXNUM-MAX Y-MIN
   (FX+ Y-LOOP-START 0)))
      (Y-BMAX
       (FX-
(FIXNUM-MIN Y-MAX
    (FX+ Y-LOOP-START 2))
1))
      (DIV-A-Y50-X17
       (JREF A
     (FX+ (FX/ Y-BMIN 50)
  (FX* 2
       (FX/ X-BMIN 17)))))
      (DIV-B-X25-Y34
       (JREF B
     (FX+ (FX/ X-BMIN 25)
  (FX* 2
       (FX/ Y-BMIN 34))))))
 (DO
      ((Y-REM-50 (FX-REM Y-BMIN 50)
(FX+ Y-REM-50 1))
       (Y-REM-34 (FX-REM Y-BMIN 34)
(FX+ Y-REM-34 1)))
      ((FX> Y-REM-34
    (FX-REM Y-BMAX 34)))
    ((LAMBDA (SET.2898_34)
       (SET
        (AREF DIV-B-X25-Y34
      (FX+ X-REM-25
  (FX* 25 Y-REM-34)))
        SET.2898_34)
       SET.2898_34)
     (AREF DIV-A-Y50-X17
  (FX+ Y-REM-50
        (FX* 50 X-REM-17))))))))))))))))
```

After Interval Splitting

```
(WAIT-SPAWN-COMPLETE
 (TREE-SPAWN 1
  (LAMBDA (TID ())
   (LET* ((OFFSET (FX-REM TID 2))
  (TID
   (FX+
    (VREF '#(0 1) OFFSET)
    (FX- TID OFFSET))))
     (IF (FX< TID 6)
 (LET* ((X-INTERVALS
 (CASE (FX-REM TID 2)
   ((0) (LIST 0 17 25))
   ((1)
    (LIST 25 34 50))
   (ELSE
    (FORMAT T "Case error~%"))))
(Y-INTERVALS
 (CASE (FX/ TID 2)
   ((0) (LIST 0 34))
   ((1)
    (LIST 34 50 68))
   ((2) (LIST 68 100))
   (ELSE
    (FORMAT T "Case error~%")))))
   (%%DO-INT (X-INT X-INTERVALS)
     (LET* ((X-BMIN
     (FX+ (CAR X-INT) 0))
    (X-BMAX
     (FX- (CADR X-INT) 1)))
       (DO
   ((X-REM-17
     (FX-REM X-BMIN 17)
     (FX+ X-REM-17 1))
    (X-REM-25
     (FX-REM X-BMIN 25)
     (FX+ X-REM-25 1)))
   ((FX> X-REM-25
 (FX-REM X-BMAX 25)))
 (%%DO-INT
  (Y-INT Y-INTERVALS)
```

```
   (LET* ((Y-BMIN
   (FX+ (CAR Y-INT) 0))
  (Y-BMAX
   (FX- (CADR Y-INT) 1))
  (DIV-A-Y50-X17
   (JREF A
(FX+
 (FX/ Y-BMIN 50)
 (FX* 2
       (FX/ X-BMIN 17)))))
 (DIV-B-X25-Y34
  (JREF B
(FX+
 (FX/ X-BMIN 25)
 (FX* 2
       (FX/ Y-BMIN 34))))))
     (DO
((Y-REM-50
   (FX-REM Y-BMIN 50)
   (FX+ Y-REM-50 1))
  (Y-REM-34
   (FX-REM Y-BMIN 34)
   (FX+ Y-REM-34 1)))
((FX> Y-REM-34
       (FX-REM Y-BMAX 34)))
       ((LAMBDA (SET.2487_34)
  (SET
   (AREF DIV-B-X25-Y34
(FX+ X-REM-25
      (FX* 25
   Y-REM-34)))
   SET.2487_34)
  SET.2487_34)
        (AREF DIV-A-Y50-X17
      (FX+ Y-REM-50
   (FX* 50 X-REM-17))))))))))))))))
```

Appendix B

Loop Splitting Code

This appendix contains the code implementing the two loop splitting methods discussed. The code is written in T, a dialect of Lisp, in which the Alewife precompiler is implemented.

The first section holds the code for the normal array referencing expressions. The second section is the code that implements the rational splitting method. While some of the procedures replace those in the original code, most are new. The last section contains the code implementing the interval splitting method. This section is much smaller than the previous one because it contains only the procedures different from those in rational splitting, either new or modified.

B.1 Normal

```
;;;
;;; /donald/splitting/lower/loop.t
;;;
;;; This is the original loop scheme (no loop splitting).
;;;

(herald loop)

;;; Old code generation stuff (Pre-Donald)

(define (generate-doall doall body)
  (xcase (%doall-rank doall)
    ((1)
     (generate-1d-doall doall body))
    ((2)
     (xcase (car (%doall-partition doall))
       ((rectangular)
 (generate-2d-rectangular-doall doall body))))
    ((3)
     (xcase (car (%doall-partition doall))
       ((rectangular)
 (generate-3d-rectangular-doall doall body))))))

 (define (generate-1d-doall doall body)
 (format t "In generate-1d-doall: limits = ~s~%"
 (%doall-ivar-limits doall))
   (let ((init (car (%doall-ivar-inits doall)))
 (limit (car (%doall-ivar-limits doall)))
 (stride (car (%doall-ivar-strides doall))))
     (destructure (((#f (i-var i-spread i-n)) (%doall-partition
       doall)))
 (format t "i-spread = ~s~%" i-spread)
     `(wait-spawn-complete
       (tree-spawn 1
 (lambda (tid #f)
   (let* ((I (fx-rem tid ,i-n))
  (I-min (fx* I ,i-spread)))
     (%%do (,i-var
    (fixnum-max I-min ,init)
    (fixnum-min (fx+ I-min ,(+ i-spread init)) ,(fx+ limit 1))
    ,stride)
   ,@(body))))))))

 ;;; We vary first in the I direction and then in the J direction

 (define (generate-2d-rectangular-doall doall body)
 (format t "limits = ~s~%" (%doall-ivar-limits doall))
   (destructure (((i-init j-init) (%doall-ivar-inits doall))
 ((i-limit j-limit) (%doall-ivar-limits doall))
 ((i-stride j-stride) (%doall-ivar-strides doall))
 ((#f (i-var i-spread i-n) (j-var j-spread j-n))
  (%doall-partition doall)))
     `(wait-spawn-complete
       (tree-spawn 1
 (lambda (tid #f)
   (let* ((I ,(if (fxn= i-n 1)
'0
`(fx-rem tid ,i-n)))
(J ,(cond ((fx= j-n 1)
   '0)
  ((fx= i-n 1)
   `(fx-rem tid ,j-n))
  (else
   `(fx/ tid ,i-n))))
(I-min (fx* I ,i-spread))
(J-min (fx* J ,j-spread)))
   (%%do (,i-var
  (fixnum-max I-min ,i-init)
  (fixnum-min (fx+ I-min ,(+ i-spread i-init))
      ,(fx+ i-limit 1))
  ,i-stride)
 (%%do (,j-var
(fixnum-max J-min ,j-init)
(fixnum-min (fx+ J-min ,(+ j-spread j-init))
    ,(fx+ j-limit 1))
,j-stride)
,@(body))))))))

;;; We vary first in the I direction. then in the
;;; J direction, then in K

(define (generate-3d-rectangular-doall doall body)
  (destructure (((i-init j-init k-init) (%doall-ivar-inits doall))
((i-limit j-limit k-limit) (%doall-ivar-limits
          doall))
((i-stride j-stride k-stride) (%doall-ivar-strides
          doall))
((#f (i-var i-spread i-n)
     (j-var j-spread j-n)
     (k-var k-spread k-n))
 (%doall-partition doall)))
    `(wait-spawn-complete
      (tree-spawn 1
(lambda (tid #f)
  (let* ((I ,(if (fxn= i-n 1)
  '0
  `(fx-rem tid ,i-n)))
  (J ,(select j-n
      ((1) '0)
      ((*nprocs*)
       `(fx-rem tid ,j-n))
      (else
       `(fx-rem (fx/ tid ,i-n) ,j-n))))
   (K ,(select k-n
      ((1) '0)
      ((*nprocs*)
       `(fx-rem tid ,k-n))
      (else
       `(fx/ tid (fx* ,i-n ,j-n)))))
  (I-min (fx* I ,i-spread))
  (J-min (fx* J ,j-spread))
  (K-min (fx* K ,k-spread)))
     (%%do (,i-var
    (fixnum-max I-min ,i-init)
   (fixnum-min (fx+ I-min ,(+ i-spread i-init))
       ,(fx+ i-limit 1))
  ,i-stride)
         (%%do (,j-var
        (fixnum-max J-min ,j-init)
        (fixnum-min (fx+ J-min ,(+ j-spread j-init))
    ,(fx+ j-limit 1))
        ,j-stride)
(%%do (,k-var
  (fixnum-max K-min ,k-init)
  (fixnum-min (fx+ K-min ,(+ k-spread k-init))
      ,(fx+ k-limit 1))
 ,k-stride)
      ,@(body)))))))))

(define (xform-indices type array indecis doall set? value)
  (xcase (%array-rank array)
    ((1) (1d-array-ref array (car indecis) type set? value))
    ((2) (2d-array-ref array (car indecis) (cadr indecis) type
     set? value))))

(define (1d-array-ref array i type set? value)
  (destructure (((#f (#f i-spread i-n)) (%array-partition array)))
    (let* ((i-temp (generate-symbol 'i))
   (index '(,type (jref ,(%array-name array) (fx/ ,i-temp
                                                          ,i-spread))
  (fx-rem ,i-temp ,i-spread))))
      '(let ((,i-temp ,i))
 ,(if set?
      '(set ,index ,value)
      index)))))

(define (2d-array-ref array i j type set? value)
  (destructure (((#f (#f i-spread i-n) (#f j-spread j-n))
 (%array-partition array)))
    (let* ((i-temp (generate-symbol 'i))
   (j-temp (generate-symbol 'j))
   (index '(,type (jref ,(%array-name array)
                                (fx+ (fx/ ,i-temp ,i-spread)
                                    (fx* ,i-n (fx/ ,j-temp ,j-spread))))
  (fx+ (fx-rem ,i-temp ,i-spread)
       (fx* ,i-spread (fx-rem ,j-temp
       ,j-spread))))))
      '(let ((,i-temp ,i)
     (,j-temp ,j))
 ,(if set?
      '(set ,index ,value)
      index)))))

;;; fixed version:  Used to divide the array size by the number
;;; of processors, now it multiplies the data spreads to get the
;;; size of the filled-array.  Fixed another bug dealing with
;;; make-jstruct.

(define (generate-make-array-type array maker init? val)
  (let* ((sizes (map cadr (cdr (%array-partition array))))
 (size (ceiling (*-reduce sizes)))
 (name (%array-name array))
```

```
 (len (if (null? (%array-mapping array))
  1
  (vector-length (%array-mapping array)))))
    '(let ((,name (make-jstruct ,*nprocs*)))
       (wait-spawn-complete
(tree-spawn 1
  (lambda (tid #f)
    (if (fx< tid ,*nprocs*)
,(if (= len 1)
     '(set (jref ,name tid)
   ,(if init? (list maker size val)
                                           (list maker size)))
     '(if (fx>= tid ,(fx* (fx/ *nprocs* len) len))
  (set (jref ,name tid)
       ,(if init? (list maker size val)
    (list maker size)))
  (let* ((offset (fx-rem tid ,len))
(tid (fx+ (vref (quote
                                               ,(%array-mapping array))
offset)
  (fx- tid offset))))
     (set (jref ,name tid)
 ,(if init? (list maker size val)
     (list maker size))))))))))
     ,name)))
(define (pcomfile file n . ofile)
  (set *loops* '())
  (set *data* '())
  (let ((waif (anal (t->waif (shuffle-top (front-end file))) nil n)))
    ;;; walk loop list and fill aspect ratios in loops and data
    (walk preliminary-partition-doall *loops*)
    ;;; walk data and do data partition
    (walk data-ratio *data*)
    (format t "Partitioning done.")
    (walk (lambda (x) (do-data-partition x n)) *data*)
    (walk (lambda (x) (final-partition-doall x n)) *loops*)
    (align-partitions *loops* *data*)
    (if (not ofile)
      (waif->t-port waif (terminal-output))
      (waif->t waif (car ofile)))))
```

B.2 Rational

```
;;;
;;;     Source: loop2.t
;;;     Author: donald tanguay

;;; This stuff is loaded over the original loop.t.  In other words, loop.t has
;;; some critical routines which this does not define.  This file has only the
;;; routines that are different from loop.t.

(herald loop2)

(define (xform-indices type array indecis doall set? value)
  (let ((c-vars ;(if (null? doall)
                ;     '()
                      (bgen-constant-vars (%doall-blockgen-private doall))))
  (xcase (%array-rank array)
    ((1) (1d-array-ref array (car indecis) c-vars type set? value doall))
    ((2) (2d-array-ref array (car indecis) (cadr indecis)
                         c-vars type set? value doall)))))



(define (1d-array-ref array i const-vars type set? value doall)
  (destructure (((#f (#f i-spread i-n)) (%array-partition array)))
  (let* ((body?     (memq? (get-letter i) const-vars))
(constant (make-constant-symbol (list (%array-name array)
      (get-letter i))
(list i-spread)))
(index (if body?
    '(,type       ;; body, so improved refs
      ,constant
      ,(concatenate-symbol (get-letter i) '-rem- i-spread))
    '(,type            ;; not in body
      (jref ,(%array-name array) (fx/ ,i ,i-spread))
      (fx-rem ,i ,i-spread)))))
     (if set?
'(set ,index ,value)
index))))



(define (2d-array-ref array i j const-vars type set? value doall)
  (destructure (((#f (#f i-spread i-n) (#f j-spread j-n))
 (%array-partition array)))
     (let* ((i-body?     (memq? (get-letter i) const-vars))
    (j-body?     (memq? (get-letter j) const-vars))
    (both-body? (and i-body? j-body?))
          (i-bmin     (concatenate-symbol (get-letter i) '-bmin))
          (j-bmin     (concatenate-symbol (get-letter j) '-bmin))
    (index  (cond
       (both-body?          ;;; Fully optimized references
        '(,type
 ,(make-constant-symbol (list (%array-name array)
       (get-letter i) (get-letter j))
(list i-spread j-spread))
 (fx+ ,(concatenate-symbol (get-letter i) '-rem- i-spread)
```

```
        (fx* ,i-spread ,(concatenate-symbol (get-letter j)
 '-rem- j-spread)))))
        (else                  ;;; no constants, only normal references
         '(,type
(jref ,(%array-name array)
         (fx+ (fx/ ,i ,i-spread)
   (fx* ,i-n (fx/ ,j ,j-spread))))
(fx+ (fx-rem ,i ,i-spread)
        (fx* ,i-spread (fx-rem ,j ,j-spread))))))))
        (if set?
 '(set ,index ,value)
 index))))


(define (generate-doall doall body)
(format t "~%Generating doall...~%")
  (let ((len (if (null? (%doall-mapping doall))
                1
 (vector-length (%doall-mapping doall)))))
    '(wait-spawn-complete
      (tree-spawn 1
(lambda (tid #f)
  ,(if (= len 1)
       (general-generate-doall doall body)
       '(let* ((offset (fx-rem tid ,len))
       (tid (fx+ (vref (quote ,(%doall-mapping doall))
       offset)
 (fx- tid offset))))
  ,(general-generate-doall doall body))))))))


;;; fixed version:  Used to divide the array size by the number of processors,
;;; now it multiplies the data spreads to get the size of the filled-array
;;; Fixed another bug dealing with make-jstruct.

(define (generate-make-array-type array maker init? val)
  (let* ((sizes (map cadr (cdr (%array-partition array))))
 (size (ceiling (*-reduce sizes)))
 (name (%array-name array))
 (len (if (null? (%array-mapping array))
  1
  (vector-length (%array-mapping array)))))
    '(let ((,name (make-jstruct ,*nprocs*)))
       (wait-spawn-complete
(tree-spawn 1
  (lambda (tid #f)
    (if (fx< tid ,*nprocs*)
,(if (= len 1)
     '(set (jref ,name tid)
   ,(if init? (list maker size val) (list maker size)))
     '(if (fx>= tid ,(fx* (fx/ *nprocs* len) len))
  (set (jref ,name tid)
       ,(if init? (list maker size val)
    (list maker size)))
  (let* ((offset (fx-rem tid ,len))
(tid (fx+ (vref (quote ,(%array-mapping array))
offset)
  (fx- tid offset))))
```

```
    (set (jref ,name tid)
  ,(if init? (list maker size val)
       (list maker size)))))))))
       ,name)))


(define (pcomfile file n . ofile)
  (set *loops* '())
  (set *data* '())
  (let ((waif (anal (t->waif (shuffle-top (front-end file))) nil n)))
    ;;; walk loop list and fill aspect ratios in loops and data
    (walk preliminary-partition-doall *loops*)
    ;;; walk data and do data partition
    (walk data-ratio *data*)
    (format t "Partitioning done.")
    (walk (lambda (x) (do-data-partition x n)) *data*)
    (walk (lambda (x) (final-partition-doall x n)) *loops*)
    (align-partitions *loops* *data*)
    (if (not ofile)
        (waif->t-port waif (terminal-output))
        (waif->t waif (car ofile)))))


(herald part-doall)


;;; General doall generation. (One type of array partition only --- this
;;; means the doall can reference more than one array as long as they all
;;; have the same partition.)  Unless it is preceded by 'data-' then
;;; all variables are assumed to be associated with the task.  For example,
;;; spread-list is the list of spreads associated with the task.

(define (general-generate-doall doall body)
  (let* ((part-list         (make-task-partition-lists doall))
         (var-list          (car part-list))
         (tspread-list      (cadr part-list))
         (n-list            (caddr part-list))
         (sym-list          (map get-letter var-list))
         (init-list         (%doall-ivar-inits doall))
         (limit-list        (%doall-ivar-limits doall))
         (stride-list       (%doall-ivar-strides doall))
         (index-list        (make-index-lists doall))
         (snt              (make-spread-and-n-table sym-list doall index-list))
         (interval-table (car snt))
         (interval-list  (map (lambda (x)
                               (table-entry interval-table x))
                             sym-list))
         (n-table           (cadr snt))
  (spread-list-table (caddr snt))
  (array-table       (cadddr snt))
         (plnlot            (offset-magnitude-extractor sym-list doall))
         (pos-list          (car plnlot))
         (neg-list          (cadr plnlot))
         (offset-table      (caddr plnlot)))

    (format t "index-list = ~s~%" index-list)
    (labels
      (((do-gen dim vlist symlist initlist limitlist stridelist neglist
```

```
                  poslist tslist intlist cvars donelist indexlist)
           (if (=0? dim)
            (block (set (bgen-constant-vars (%doall-blockgen-private doall)) cvars)
           (body)) ;;; make loop body
     (let ((loop-start (concatenate-symbol (car symlist) '-loop-start))
               (x-min      (concatenate-symbol (car symlist) '-min))
               (x-max      (concatenate-symbol (car symlist) '-max))
               (hmin (concatenate-symbol (car symlist) '-hmin))
               (hmax (concatenate-symbol (car symlist) '-hmax))
               (bmin (concatenate-symbol (car symlist) '-bmin))
               (bmax (concatenate-symbol (car symlist) '-bmax))
               (tmin (concatenate-symbol (car symlist) '-tmin))
               (tmax (concatenate-symbol (car symlist) '-tmax)))
          (labels (((make-loop)
               (format t "In make-loop.~%")
               '( ;;; Head
                  ,@(if (<0? (car neglist))
                         '((let ((,hmin (fixnum-max ,x-min ,loop-start))
                                 (,hmax (fixnum-min ,x-max (fx+ ,loop-start
          ,(abs (car neglist))))))
           (%%do (,(car vlist) ,hmin ,hmax ,(car stridelist))
      ,@(do-gen (- dim 1) (cdr vlist)
                              (cdr symlist) (cdr initlist) (cdr limitlist)
                              (cdr stridelist) (cdr neglist) (cdr poslist)
                              (cdr tslist) (cdr intlist) cvars donelist
                              indexlist))))
                         '())
                  ;;; Body
                  ,(block (set cvars (cons (car symlist) cvars))
                      (let ((let-code (make-let-constants indexlist
                                            cvars n-table doall array-table))
         (do-rem-code (make-do-rem (car symlist)
           spread-list-table)))
                            (block (set indexlist (remove-refs indexlist donelist))
      '(let* ((,bmin (fixnum-max ,x-min (fx+ ,loop-start
     ,(abs (car neglist)))))
      (,bmax (fx- (fixnum-min ,x-max
      (fx+ ,loop-start
            ,(- (car intlist) (car poslist)))) 1))
      ,@let-code)
        (do ,@do-rem-code
     ,@(do-gen (- dim 1) (cdr vlist)
     (cdr symlist) (cdr initlist) (cdr limitlist)
     (cdr stridelist) (cdr neglist) (cdr poslist)
     (cdr tslist) (cdr intlist) cvars donelist
     indexlist))))))

                  ;;; Tail
                  ,@(block (set cvars (delq (car symlist) cvars))
                       (if (>0? (car poslist))
      '((let ((,tmin (fixnum-max ,x-min (fx+ ,loop-start
        ,(fx- (car intlist) (car poslist)))))
          (,tmax (fixnum-min ,x-max (fx+ ,loop-start
         ,(car intlist)))))
      (%%do (,(car vlist) ,tmin ,tmax ,(car stridelist))
         ,@(do-gen (- dim 1) (cdr vlist)
                              (cdr symlist) (cdr initlist) (cdr limitlist)
                              (cdr stridelist) (cdr neglist) (cdr poslist)
                              (cdr tslist) (cdr intlist) cvars donelist
                              indexlist))))
                         '()))))

      ((make-simple-loop)
(block (set cvars (cons (car symlist) cvars))
                 (let ((let-code (make-let-constants indexlist
                                    cvars n-table doall array-table))
   (do-rem-code (make-do-rem (car symlist)
     spread-list-table)))
                    (block (set indexlist (remove-refs indexlist donelist))
 '(let* ((,bmin ,x-min)
 (,bmax (fx- ,x-max 1))
 ,@let-code)
    (do ,@do-rem-code
      ,@(do-gen (- dim 1) (cdr vlist)
(cdr symlist) (cdr initlist) (cdr limitlist)
(cdr stridelist) (cdr neglist) (cdr poslist)
(cdr tslist) (cdr intlist) cvars donelist
indexlist))))))))

         (set donelist (cons (car symlist) donelist))
  (if (and (every? (lambda (x) (= x (car tslist)))
   (table-entry spread-list-table (car symlist)))
   (= 0 (car poslist))
   (= 0 (car neglist)))
      '(,(make-simple-loop))
      '((do ((,loop-start
      (fx* ,(car intlist) (fx/ ,x-min ,(car intlist)))
      (fx+ ,loop-start ,(car intlist))))
    ((fx>= ,loop-start ,x-max))
                 ,@(make-loop))))))))

      ;;; body of general-generate-doall
      '(if (fx< tid ,(*-reduce n-list))
   '(let* (,@(make-top-let-statement doall init-list limit-list
     tspread-list n-list sym-list))
      ,@(do-gen (%doall-rank doall) var-list sym-list
init-list limit-list stride-list neg-list pos-list
tspread-list interval-list '() '() index-list))))))
```

```
   (,i-min (fx+ (fx* ,i ,i-spread) ,(car init-list)))
   (,i-max  (fixnum-min (fx+ ,i-min ,i-spread) ,(1+ (car limit-list)))))
  (let ((j-n           (cadr n-list))
      (j-spread      (cadr s-list))
(j          (cadr sym-list))
(j-min      (concatenate-symbol (cadr sym-list) '-min))
(j-max      (concatenate-symbol (cadr sym-list) '-max)))
(if (= 2 dim)
  '((,i ,(if (fxn= i-n 1)
             '(fx-rem tid ,i-n)
             '0))
   (,j ,(cond ((fx= j-n 1) '0)
              ((fx= i-n 1) '(fx-rem tid ,j-n))
              (else        '(fx/ tid ,i-n)))) ;DK 11/18/92
   (,i-min (fx+ (fx* ,i ,i-spread) ,(car init-list)))
   (,j-min (fx+ (fx* ,j ,j-spread) ,(cadr init-list)))
   (,i-max (fixnum-min (fx+ ,i-min ,i-spread) ,(1+ (car limit-list))))
   (,j-max (fixnum-min (fx+ ,j-min ,j-spread) ,(1+ (cadr limit-list)))))
  (let ((k-n           (caddr n-list))
      (k-spread      (caddr s-list))
(k          (caddr sym-list))
(k-min      (concatenate-symbol (caddr sym-list) '-min))
(k-max      (concatenate-symbol (caddr sym-list) '-max)))
   '((,i ,(if (fxn= i-n 1)
             '0
             '(fx-rem tid ,i-n)))
     (,j ,(select j-n
                ((1)        '0)
                ((*nprocs*) '(fx-rem tid ,j-n))
                (else       '(fx-rem (fx/ tid ,i-n) ,j-n))))
     (,k ,(select k-n
                ((1)        '0)
                ((*nprocs*) '(fx-rem tid ,k-n))
                (else       '(fx/ tid ,(* i-n j-n)))))
     (,i-min (fx+ (fx* ,i ,i-spread) ,(car init-list)))
     (,j-min (fx+ (fx* ,j ,j-spread) ,(cadr init-list)))
     (,k-min (fx+ (fx* ,k ,k-spread) ,(caddr init-list)))
     (,i-max (fixnum-min (fx+ ,i-min ,i-spread) ,(1+ (car limit-list))))
    (,j-max (fixnum-min (fx+ ,j-min ,j-spread) ,(1+ (cadr limit-list))))
     (,k-max (fixnum-min (fx+ ,k-min ,k-spread)
 ,(1+ (caddr limit-list))))))))))))
```

```
;;; A messy-looking procedure to make the top let-statement
(define (make-top-let-statement doall init-list limit-list s-list n-list
                              sym-list)
(format t "In make-top-let-statement.~%")
  (let ((dim            (%doall-rank doall)))
    (if (> dim 3)
        (format t "ERROR: ~s-dimensional doalls are unhandled.~%" dim)
        (let ((i-n            (car n-list))
              (i-spread      (car s-list))
              (i           (car sym-list))
              (i-min       (concatenate-symbol (car sym-list) '-min))
              (i-max       (concatenate-symbol (car sym-list) '-max)))
    (if (= 1 dim)
        '((,i      (fx-rem tid ,i-n))
```

```
;;; The first element of const-list is the array-name.  The rest are the
index variables.  Array-spread-list is the list of spread for the array.

(define (make-constant-symbol const-list array-spread-list)
;(format t "In make-constant-symbol.~%")
  (do ((clist (cdr const-list) (cdr clist))
     (alist array-spread-list (cdr alist))
     (sym (concatenate-symbol 'div- (car const-list))
          (concatenate-symbol sym '- (car clist) (car alist))))
    ((null? clist) sym)))

;;; Generates the code for constants.  The index-list is the list of array
;;; references which have not yet been satisfied by constants.  The(doall?impread-list))
;;; is the induction variables for which loop code has been generated.  Thmst init-code-list (append init-code-list
;;; these are the ones that constants can be made of.                 (make-do-rem-init symbol (car spread-list)))))
; array-table has the spreads needed for the constants (references)   (format t "  init-code-list = ~s~%" init-code-list)
                                                                      (list init-code-list exit-code)))
(define (make-let-constants index-list done-list n-table doall array-table)
(format t "In make-let-constants.~%")                               (define (make-do-rem-init symbol spread)
  (do ((index-list index-list (cdr index-list))                       (let ((rem (concatenate-symbol symbol '-rem- spread))
       (code '() code))                                             (bmin (concatenate-symbol symbol '-bmin)))
      ((null? index-list) code)                                         '((,rem (fx-rem ,bmin ,spread) (fx+ ,rem 1)))))
      (if (every? (lambda (x) (memq? x done-list)) (cdar index-list))
          (set code (append code (list
  (let* ((aspr-lst (table-entry array-table (caar index-list)))
 (const-var (make-constant-symbol (car index-list)                  ;;; After code has been generated for all doall dimensions used in a reference,
  (map car aspr-lst))))                                             ;;; that reference is then removed from the index-list.  The done-list is the
    '(,const-var                                                    ;;; list of those dimensions for which code has been generated.
             ,(case (length (cdar index-list))
                 ((1) (let* ((i-var (cadar index-list))             (define (remove-refs index-list done-list)
                             (i-bmin (concatenate-symbol i-var '-bmin)do ((new-list '() new-list)
                             (i-data-spread (caar aspr-lst)))          (index-list index-list (cdr index-list)))
                        '(jref ,(caar index-list)   ;;; array name    ((null? index-list) new-list)
                               (fx/ ,i-bmin ,i-data-spread))))        (if (not (every? (lambda (x) (memq? x done-list))
                 ((2) (let* ((i-var (cadar index-list))                                (cdar index-list)))   ;;; ignore the array-name
                             (j-var (caddar index-list))                  (set new-list (append new-list (list (car index-list)))))))
                             (i-bmin (concatenate-symbol i-var '-bmin))
                             (j-bmin (concatenate-symbol j-var '-bmin))
                             (i-data-spread (caar aspr-lst))
                             (j-data-spread (caadr aspr-lst))       ;;; Takes a variable and returns its first letter.  For example, I_21 -> I
                             (i-n (cdar aspr-lst)))                 ;;; This also handles cases with offsets, for example, if var is
                        '(jref ,(caar index-list)   ;;; array name ;;; FIXNUM-ADD I_11 '4 because of the offset of four, this can still extract
                               (fx+ (fx/ ,i-bmin ,i-data-spread)   ;;; the variable I.
                                    (fx* ,i-n (fx/ ,j-bmin ,j-data-spread))))))
                 (else (format t "ERROR: ~s-D refence in make-let-co(define (get-letter var)
                             (length (cdar index-list)))))))))))(format t "In get-letter: var = ~s~%" var)
                                                                     (if (atom? var)
                                                                       (string->symbol (char->string (char (symbol->string var))))
;;; This makes header for the loop before the body.  The first part of the(if (= 3 (length var))
;;; header defines the loop variables, their initial values, and their update(string->symbol (char->string (char (symbol->string (cadr var)))))
;;; expressions.  The loop variables here are the remainder variables.  The  (format t "ERROR: In get-letter, var = ~s~%" var))))
;;; second part of the header specifies the exit condition.  The exit condition
;;; here is just the first remainder variable reaching its final value for the
;;; interval.
                                                                    ; Ensures the only constants created are ones that will be used.  It
(define (make-do-rem symbol spread-list-table)                      ; returns a list of all the unique array references of the doall.
(format t "In make-do-rem: ivar=~s~%" symbol)
  (let* ((first-list (table-entry spread-list-table symbol))        (define (make-index-lists doall)
(spread-list '())                                                   (format t "In make-index-lists.~%")
(init-code-list '())                                                  (let ((i-list '()))
(exit-code '((fx> ,(concatenate-symbol symbol '-rem- (car first-list)) (map
 (fx-rem ,(concatenate-symbol symbol '-bmax)                              (lambda (arr-list)
 ,(car first-list))))))                                                      (let ((arr-name (%array-name (car arr-list))))
    (format t "  exit-code = ~s~%" exit-code)                                  (do ((arr-list (cdr arr-list) (cdr arr-list)))
    (do ((spread (car first-list) (car first-list)))                               ((null? arr-list))
((null? first-list))                                                               (let ((add-list (list arr-name)))
      (block (set first-list (delq spread first-list))                              (block
     (set spread-list (cons spread spread-list))))                                    (do ((ind-list (cdar arr-list) (cdr ind-list)))
    (format t "  spread-list = ~s~%" spread-list)                                        ((null? ind-list))
    (do ((spread-list spread-list (cdr spread-list)))                                    (let ((sym (string->symbol (char->string (char
                                (symbol->string (caar ind-list)))))))                    (set (table-entry ref-list-tbl var)
                          (set add-list (append add-list (list sym)))))                        (cons (cdar inds)
                 (if (not (mem? alikeq? add-list i-list))                                            (table-entry ref-list-tbl var)))))))
                      (set i-list (append i-list (list add-list)))))))))) (cdr arr-list)))
     (%doall-index-list doall))                                             (%doall-index-list doall))
(format t "  i-list = ~s~%" i-list)                                        (list (map (lambda (x) (table-entry pos-tbl x)) sym-list)
   i-list))                                                                       (map (lambda (x) (table-entry neg-tbl x)) sym-list)
                                                                                  ref-list-tbl)))


;;; makes the task partition easily accessible in three lists      ;;;
                                                                    ;;; lcm/smart.t
(define (make-task-partition-lists doall)                           ;;;
(format t "In make-task-partition-lists.~%")                        ;;; This is the code for the LCM Loops method of loop splitting.  This one
(format t "part-list = ~s~%" (%doall-partition doall))              ;;; procedure replaces the pick-the-spreads procedure of one/smart.t.  Of
  (do ((part-list   (cdr (%doall-partition doall)) (cdr part-list));;; course, the files for single partitioning must be loaded first.
       (var-list    '() (append var-list (list (caar part-list)))) ;;;
       (spread-list '() (append spread-list (list (cadar part-list))))
       (n-list      '() (append n-list (list (caddar part-list)))))(herald smart)
      ((null? part-list) (list var-list spread-list n-list))))
                                                                    ;;; Returns four very useful tables.
                                                                    ;;; interval-tbl: induction variable -> loop splitting interval
;;; This procedure returns a list of three items.  The first item is;a listofivar -> # processors in the associated dim of the virtual network
;;; the biggest positive offset in each dimension of the doall.  These spread-list-tbl: ivar -> list of the spreads this variable encounters
;;; are used to decide the division of the body and tail of the doall; array-tbl: array name -> list of the ivars used to reference the array
;;; second item is the equivalent for the negative offsets, containing the
;;; smallest values in each doall dimension.  These values are used(define (make-spread-and-n-table sym-list doall ref-list)
;;; the division between the head and body of the doall.  The third item is(interval-tbl    (make-symbol-table))
;;; table of lists.  Each entry of the table corresponds to a doall(dimension.      (make-symbol-table))
;;; Each entry is a list of all the offsets in that diemnsion, with(spread-list-tbl (make-symbol-table))
;;; duplications removed.  This table is used to make the variables(array-tbl body    (make-array-table doall)))
;;; of the doall which replace the FX-REM expressions of array references; Initialize the tables
                                                                       (map (lambda (x)
(define (offset-magnitude-extractor sym-list doall)                   (set (table-entry interval-tbl x) '0)
(format t "In offset-magnitude-extractor.~%")                         (set (table-entry n-tbl x) '0)
  (let* ((pos-tbl (make-symbol-table))         ;; tables of values    (set (table-entry spread-list-tbl x) '()))
         (neg-tbl (make-symbol-table))                                     sym-list)
         (ref-list-tbl (make-symbol-table)))   ;; tables of lists of values
    ;; Initialize the tables                                           ;;; Build the spread-list-tbl and n-tbl
    (map (lambda (x) (set (table-entry pos-tbl x) '0)) sym-list)       (map (lambda (ref)     ;;; ref = (A i j k)
    (map (lambda (x) (set (table-entry neg-tbl x) '0)) sym-list)      (do ((parts (table-entry array-tbl (car ref)) (cdr parts))
    (map (lambda (x) (set (table-entry ref-list-tbl x) '(0))) sym-list)          (indices (cdr ref) (cdr indices)))
    (map                                                                        ((or (null? parts) (null? indices))  ;;; quit when one empty
      (lambda (arr-list)                                                         (if (or parts indices)           ;;; is one not empty?
         (map (lambda (indices)                                        (format t "ERROR 1 in make-spread-and-n-table~%")))
                 (do ((inds (cdr indices) (cdr inds)))                  (let ((ind    (car indices))
                     ((null? inds))                                    (spread (caar parts))
                     (let ((var (get-letter (caar inds))))             (n      (cdar parts)))
                       ;;; keep only the biggest pos and smallest neg      (cond ((null? (table-entry n-tbl ind))
                       (if (table-entry pos-tbl var)                      (format t "ERROR: No n-tbl entry for ~s.~%" ind)
  (block                                                                   (set (table-entry (bgen-diff-table
   (if (> (cdar inds) (table-entry pos-tbl var))                     (%doall-blockgen-private doall))
       (set (table-entry pos-tbl var) (cdar inds)))                 ind) t))
   (if (< (cdar inds) (table-entry neg-tbl var))                        ((= 0 (table-entry n-tbl ind))
       (set (table-entry neg-tbl var) (cdar inds)))))                    (set (table-entry n-tbl ind) n))
                       ;;; keep all of the offsets in table-- no dups!  ((neq? n (table-entry n-tbl ind))
                       (if (not (memq? (cdar inds)                       (format t "WARNING: Different n's in partition.~%")))
                                   (table-entry ref-list-tbl var)))    (set (table-entry spread-list-tbl ind)
   (cons spread (table-entry spread-list-tbl ind))))))
ref-list)

   ;;; Pick lcm of spreads and put it in the interval-tbl
   (map (lambda (sym)
  (let ((spread-list (table-entry spread-list-tbl sym)))
    (set (table-entry interval-tbl sym)
 (pick-the-spread spread-list))))
        sym-list)

   ;;; Initialize the diff-table of the doall
   (map (lambda (sym)
  (set (table-entry (bgen-index-table (%doall-blockgen-private doall))
    sym) nil))
sym-list)


   ;;; Initialize the index-table of the doall
   (map (lambda (sym)
  (set (table-entry (bgen-index-table (%doall-blockgen-private doall))
    sym) '()))
sym-list)
   (map (lambda (ref)
  (let ((name (car ref)))
    (do ((parts (table-entry array-tbl name) (cdr parts))
 (indices (cdr ref) (cdr indices)))
((or (null? parts) (null? indices))
 (if (or parts indices)
     (format t "ERROR 1~%")))
      (let ((ind (car indices)))
(if (not (memq? name
(table-entry (bgen-index-table
      (%doall-blockgen-private doall))
     ind)))
    (set (table-entry (bgen-index-table
(%doall-blockgen-private doall))
      ind)
 (cons name
(table-entry (bgen-index-table
     (%doall-blockgen-private
      doall)) ind))))))))
ref-list)

   ;;; Calculate the index-table of the doall
   (map (lambda (ref)    ;;; ref = (A i j k)
  (let ((name (car ref)))
    (do ((parts (table-entry array-tbl name) (cdr parts))
 (indices (cdr ref) (cdr indices)))
((or (null? parts) (null? indices))   ;;; quit when one empty
 (if (or parts indices)               ;;; is one not empty?
     (format t "ERROR 1 in make-spread-and-n-table~%")))
      (let ((ind    (car indices))
    (spread (caar parts)))
(if (and (memq? name
(table-entry (bgen-index-table
      (%doall-blockgen-private
doall)) ind))
  (neq? spread (table-entry interval-tbl ind)))
```

```
     (block
      (set (table-entry (bgen-index-table
  (%doall-blockgen-private doall)) ind)
    (delq name (table-entry (bgen-index-table
     (%doall-blockgen-private
      doall))
    ind)))
      (set (table-entry (bgen-diff-table
  (%doall-blockgen-private doall))
ind)
    t)))))))
 ref-list)

     ;;; Check entries of the index-table of the doall
     (format t "Checking which arrays are in elts of index-table:~%")
     (map (lambda (sym)
    (format t "   ~s --> ~s~%" sym
    (table-entry (bgen-index-table
  (%doall-blockgen-private doall))
sym))
 sym-list)

     ;;; Check entries of the diff-table of the doall
     (format t "Checking which index vars have diff partitions:")
     (map (lambda (sym)
    (if (table-entry (bgen-diff-table
      (%doall-blockgen-private doall)) sym)
        (format t " ~s" sym)))
 sym-list)
     (format t "~%Done with smart.t~%~%")

     ;;; Return the tables
     (list interval-tbl n-tbl spread-list-tbl array-tbl)))



;;; Returns a lookup table for each array.  The keys are the array names.
;;; The values are lists of the pairs.  The pairs are the spread and n for
;;; each dimension.
;;; array_name --> '((sprd1 . n1) (sprd2 . n2) (sprd3 . n3) ... )

(define (make-array-table doall)
;(format t "In make-array-table.~%")
  (let ((arrays      (map car (%doall-index-list doall)))
(array-table (make-symbol-table)))
     (map (lambda (array)
    (do ((parts (cdr (%array-partition array)) (cdr parts))
                (spr-and-ns '() spr-and-ns))
        ((null? parts) (set (table-entry array-table
(%array-name array))
   spr-and-ns))
      (set spr-and-ns (append spr-and-ns (list
  (cons (cadar parts) (caddar parts)))))))
 arrays)
    ;;; Check the array table
    (format t "Checking the data partitions (spread . n):~%")
    (let ((names (map %array-name arrays)))
       (map (lambda (name)
    (format t "~s --> ~s~%" name (table-entry array-table name)))
  names))
  array-table))


;;; (pick-the-spread lst)
;;; Requires: lst is a list of integers
;;; Modifies: nothing
;;; Effects:  Returns the greatest common divisor of the integers in lst.

(define (pick-the-spread lst) ;;; return the greatest common divisor of spreads
  (cond ((null? lst) (format t "ERROR: Null list in pick-the-spread.~%"))
((= 1 (length lst)) (car lst))
(else (do ((num (car lst) (gcd num (car l)))
   (l (cdr lst) (cdr l)))
  ((null? l) num)))))
```

B.3 Interval

```
(herald part-doall)


;;; General doall generation.

(define (general-generate-doall doall body)
  (let* ((part-list          (make-task-partition-lists doall))
        (var-list            (car part-list))
        (tspread-list        (cadr part-list))
        (n-list              (caddr part-list))
        (sym-list            (map get-letter var-list))
        (init-list           (%doall-ivar-inits doall))
        (limit-list          (%doall-ivar-limits doall))
        (stride-list         (%doall-ivar-strides doall))
        (index-list          (make-index-lists doall))
        (snt                 (make-spread-and-n-table sym-list
                               doall index-list))
        (interval-table (car snt))
        (interval-list  (map (lambda (x)
                                   (table-entry interval-table x))
                                 sym-list))
        (n-table             (cadr snt))
 (spread-list-table (caddr snt))
 (array-table        (cadddr snt))
        (plnlot              (offset-magnitude-extractor sym-list doall))
        (pos-list            (car plnlot))
        (neg-list            (cadr plnlot))
        (offset-table        (caddr plnlot)))
    (format t "index-list = ~s~%" index-list)
    (labels
      (((do-gen dim vlist symlist initlist limitlist stridelist neglist
                poslist tslist intlist cvars donelist indexlist)
       (if (=0? dim)
        (block (set (bgen-constant-vars (%doall-blockgen-private doall)) cvars)
       (body)) ;;; make loop body
(let* ((no-do-int?  (table-entry (bgen-no-do-table
 (%doall-blockgen-private doall))
(car symlist)))
      (x-intervals (concatenate-symbol (car symlist) '-intervals))
      (x-int       (if no-do-int?
       x-intervals
       (concatenate-symbol (car symlist) '-int)))
             (x-min      (concatenate-symbol (car symlist) '-min))
             (x-max      (concatenate-symbol (car symlist) '-max))
             (hmin (concatenate-symbol (car symlist) '-hmin))
             (hmax (concatenate-symbol (car symlist) '-hmax))
             (bmin (concatenate-symbol (car symlist) '-bmin))
             (bmax (concatenate-symbol (car symlist) '-bmax))
             (tmin (concatenate-symbol (car symlist) '-tmin))
             (tmax (concatenate-symbol (car symlist) '-tmax)))
      (labels (((make-loop)
           (format t "In make-loop.~%")
           `( ;;; Head
             ,@(if (<0? (car neglist))
                   `((let ((,hmin (car ,x-int))
    (,hmax (fixnum-min (cadr ,x-int) (fx+ (car ,x-int)
   ,(abs (car neglist))))))
        (%%do (,(car vlist) ,hmin ,hmax ,(car stridelist))
 ,@(do-gen (- dim 1) (cdr vlist)
                             (cdr symlist) (cdr initlist) (cdr limitlist)
                             (cdr stridelist) (cdr neglist) (cdr poslist)
                             (cdr tslist) (cdr intlist) cvars donelist
                             indexlist))))
                     '())
               ;;; Body
               ,(block (set cvars (cons (car symlist) cvars))
                      (let ((let-code (make-let-constants indexlist
                                       cvars n-table doall array-table))
    (do-rem-code (make-do-rem (car symlist)
     spread-list-table)))
                        (block (set indexlist (remove-refs indexlist donelist))
 '(let* ((,bmin (fx+ (car ,x-int)
     ,(abs (car neglist))))
 (,bmax (fx- (cadr ,x-int) ,(fx+ 1
     (car poslist))))
 ,@let-code)
    (do ,@do-rem-code
,@(do-gen (- dim 1) (cdr vlist)
(cdr symlist) (cdr initlist) (cdr limitlist)
(cdr stridelist) (cdr neglist) (cdr poslist)
(cdr tslist) (cdr intlist) cvars donelist
indexlist)))))

               ;;; Tail
               ,@(block (set cvars (delq (car symlist) cvars))
                   (if (>0? (car poslist))
      '((let ((,tmin (fixnum-max
      (fixnum-min (cadr ,x-int)
  (fx+ (car ,x-int)
       ,(abs (car neglist))))
      (fx- (cadr ,x-int)
   ,(car poslist))))
     (,tmax (fixnum-min ,x-max (fx+ ,loop-start
    ,(car intlist)))))
 (%%do (,(car vlist) ,tmin ,tmax ,(car stridelist))
    ,@(do-gen (- dim 1) (cdr vlist)
                             (cdr symlist) (cdr initlist) (cdr limitlist)
                             (cdr stridelist) (cdr neglist) (cdr poslist)
                             (cdr tslist) (cdr intlist) cvars donelist
                             indexlist))))
                       '()))))

        ((make-simple-loop)
(block (set cvars (cons (car symlist) cvars))
                      (let ((let-code (make-let-constants indexlist
                                       cvars n-table doall array-table))
    (do-rem-code (make-do-rem (car symlist)
     spread-list-table)))
                        (block (set indexlist (remove-refs indexlist donelist))
 '(let* ((,bmin (fx+ (car ,x-int)
     ,(abs (car neglist))))
 (,bmax (fx- (cadr ,x-int) ,(fx+ 1
     (car poslist))))
 ,@let-code)
```

```
    (do ,@do-rem-code
      ,@(do-gen (- dim 1) (cdr vlist)
(cdr symlist) (cdr initlist) (cdr limitlist)
(cdr stridelist) (cdr neglist) (cdr poslist)
(cdr tslist) (cdr intlist) cvars donelist
indexlist)))))))

          (set donelist (cons (car symlist) donelist))
  (if (table-entry (bgen-no-do-table
    (%doall-blockgen-private doall))
    (car symlist))
      '(,(make-simple-loop))
      '((%%do-int (,x-int ,x-intervals)
  ,@(make-loop))))))))
      ;;; body of general-generate-doall
      '(if (fx< tid ,(*-reduce n-list))
(let* (,@(make-top-let-statement doall init-list limit-list sym-list
tspread-list n-list
   (map (lambda (x) (table-entry spread-list-table x))
sym-list)))
 ,@(do-gen (%doall-rank doall) var-list sym-list
                   init-list limit-list stride-list neg-list pos-list
                   tspread-list interval-list '() '() index-list))))))

;;; A messy-looking procedure to make the top let-statement

(define (make-top-let-statement doall init-list limit-list sym-list s-list
n-list ds-lists)
(format t "In make-top-let-statement.~%")
(format t "  ds-lists = ~s~%" ds-lists)
  (let ((dim            (%doall-rank doall)))
    (if (> dim 3)
        (format t "ERROR: ~s-dimensional doalls are unhandled.~%" dim)
        (let* ((i-init    (car init-list))
      (i-limit   (car limit-list))
      (i-sym     (car sym-list))
      (i-ints    (concatenate-symbol i-sym '-intervals))
      (i-n       (car n-list))
      (i-s       (car s-list))
      (i-ds      (car ds-lists))
      (i-intlist (make-interval-list i-init i-limit i-ds))
      (i-case    (make-case-intervals i-init i-limit i-s i-n i-ds
      doall i-sym)))
    (if (= 1 dim)
        '((,i-ints ,(if (fx= i-n 1)
'(list ,@i-intlist)
 (case (fx-rem tid ,i-n)
   ,@i-case
   (else
  (format t "Case error~%"))))))
(let* ((j-init  (cadr init-list))
       (j-limit (cadr limit-list))
       (j-sym   (cadr sym-list))
       (j-ints  (concatenate-symbol j-sym '-intervals))
       (j-n     (cadr n-list))
```

```
        (j-s      (cadr s-list))
        (j-ds     (cadr ds-lists))
        (j-intlist    (make-interval-list j-init j-limit j-ds))
        (j-case  (make-case-intervals j-init j-limit j-s j-n j-ds
      doall j-sym)))
     (if (= 2 dim)
`((,i-ints ,(if (fx= i-n 1)
`(list ,@i-intlist)
`(case (fx-rem tid ,i-n)
   ,@i-case
   (else
    (format t "Case error~%")))))
  (,j-ints ,(cond ((fx= j-n 1)
   `(list ,@j-intlist))
  ((fx= i-n 1)
   `(case (fx-rem tid ,j-n)
      ,@j-case
      (else
       (format t "Case error~%"))))
  (else
   `(case (fx/ tid ,i-n)
      ,@j-case
      (else
       (format t "Case error~%")))))))
(let* ((k-init  (caddr init-list))
       (k-limit (caddr limit-list))
       (k-sym   (caddr sym-list))
       (k-ints  (concatenate-symbol k-sym '-intervals))
       (k-n     (caddr n-list))
       (k-s     (caddr s-list))
       (k-ds    (caddr ds-lists))
       (k-intlist    (make-interval-list k-init k-limit k-ds))
       (k-case  (make-case-intervals k-init k-limit k-s k-n k-ds
     doall k-sym)))
  `((,i-ints ,(if (fx= i-n 1)
  `(list ,@i-intlist)
  `(case (fx-rem tid ,i-n)
     ,@i-case
     (else
      (format t "Case error~%")))))
    (,j-ints ,(select j-n
      ((1) `(list ,@j-intlist))
      ((*nprocs*)
       `(case (fx-rem tid ,j-n)
  ,@j-case
  (else
   (format t "Case error~%"))))
      (else
       `(case (fx-rem (fx/ tid ,i-n) ,j-n)
  ,@j-case
  (else
   (format t "Case error~%"))))))
    (,k-ints ,(select k-n
      ((1) `(list ,@k-intlist))
      ((*nprocs*)
       `(case (fx-rem tid ,k-n)
  ,@k-case
  (else
   (format t "Case error~%"))))
      (else
       `(case (fx/ tid ,(* i-n j-n))
  ,@k-case
  (else
   (format t "Case error~%"))))))))))))))
```

```
;;; The first element of const-list is the array-name.  The rest are the
;;; index variables.  Array-spread-list is the list of spread for the array.

(define (make-constant-symbol const-list array-spread-list)
  (do ((clist (cdr const-list) (cdr clist))
       (alist array-spread-list (cdr alist))
       (sym (concatenate-symbol 'div- (car const-list))
            (concatenate-symbol sym '- (car clist) (car alist))))
      ((null? clist) sym)))
```

```
;;; Generates the code for constants.  The index-list is the list of array
;;; references which have not yet been satisfied by constants.  The done-list
;;; is the induction variables for which loop code has been generated.  Thus,
;;; these are the ones that constants can be made of.
; array-table has the spreads needed for the constants (references)

(define (make-let-constants index-list done-list n-table doall array-table)
(format t "In make-let-constants.~%")
  (do ((index-list index-list (cdr index-list))
       (code '() code))
      ((null? index-list) code)
      (if (every? (lambda (x) (memq? x done-list)) (cdar index-list))
          (set code (append code (list
  (let* ((aspr-lst (table-entry array-table (caar index-list)))
 (const-var (make-constant-symbol (car index-list)
  (map car aspr-lst))))
    `(,const-var
            ,(case (length (cdar index-list))
               ((1) (let* ((i-var (cadar index-list))
                           (i-bmin (concatenate-symbol i-var '-bmin))
                           (i-data-spread (caar aspr-lst)))
                      `(jref ,(caar index-list)   ;;; array name
                          (fx/ ,i-bmin ,i-data-spread))))
               ((2) (let* ((i-var (cadar index-list))
                           (j-var (caddar index-list))
                           (i-bmin (concatenate-symbol i-var '-bmin))
                           (j-bmin (concatenate-symbol j-var '-bmin))
                           (i-data-spread (caar aspr-lst))
                           (j-data-spread (caadr aspr-lst))
                           (i-n (cdar aspr-lst)))
                      `(jref ,(caar index-list)   ;;; array name
                          (fx+ (fx/ ,i-bmin ,i-data-spread)
                               (fx* ,i-n (fx/ ,j-bmin ,j-data-spread))))))
               (else (format t "ERROR: ~s-D refence in make-let-constants.~%"
                          (length (cdar index-list)))))))))))))
```

```
;;; This makes header for the loop before the body.  The first part of the   (string->symbol (char->string (char (symbol->string (cadr var)))))
;;; header defines the loop variables, their initial values, and their update(format t "ERROR: In get-letter, var = ~s~%" var))))
;;; expressions.  The loop variables here are the remainder variables.  The
;;; second part of the header specifies the exit condition.  The exit condition
;;; here is just the first remainder variable reaching its final value for the
;;; interval.                                                 ; Ensures the only constants created are ones that will be used.  It
                                                              ; returns a list of all the unique array references of the doall.

(define (make-do-rem symbol spread-list-table)
  (let* ((first-list (table-entry spread-list-table symbol))       (define (make-index-lists doall)
(spread-list '())                                                  (let ((i-list '()))
(init-code-list '())                                                 (map
(exit-code '((fx> ,(concatenate-symbol symbol '-rem- (car first-list))   (lambda (arr-list)
 (fx-rem ,(concatenate-symbol symbol '-bmax)                               (let ((arr-name (%array-name (car arr-list))))
 ,(car first-list))))))                                                      (do ((arr-list (cdr arr-list) (cdr arr-list)))
    (format t "  exit-code = ~s~%" exit-code)                                     ((null? arr-list))
    (do ((spread (car first-list) (car first-list)))                              (let ((add-list (list arr-name)))
((null? first-list))                                                                (block
      (block (set first-list (delq spread first-list))                                (do ((ind-list (cdar arr-list) (cdr ind-list)))
     (set spread-list (cons spread spread-list))))                                       ((null? ind-list))
    (format t "  spread-list = ~s~%" spread-list)                                        (let ((sym (string->symbol (char->string (char
    (do ((spread-list spread-list (cdr spread-list)))                                                (symbol->string (caar ind-list)))))))
((null? spread-list))                                                                      (set add-list (append add-list (list sym)))))
      (set init-code-list (append init-code-list                                      (if (not (mem? alikeq? add-list i-list))
    (make-do-rem-init symbol (car spread-list)))))                                       (set i-list (append i-list (list add-list)))))))))
    (format t "  init-code-list = ~s~%" init-code-list)                      (%doall-index-list doall))
    (list init-code-list exit-code)))                                       i-list))

(define (make-do-rem-init symbol spread)
  (let ((rem (concatenate-symbol symbol '-rem- spread))
(bmin (concatenate-symbol symbol '-bmin)))                          ;;; makes the task partition easily accessible in three lists
    '((,rem (fx-rem ,bmin ,spread) (fx+ ,rem 1)))))
                                                                    (define (make-task-partition-lists doall)
                                                                      (do ((part-list   (cdr (%doall-partition doall)) (cdr part-list))
                                                                           (var-list    '() (append var-list (list (caar part-list))))
;;; After code has been generated for all doall dimensions used in a referespread-list '() (append spread-list (list (cadar part-list))))
;;; that reference is then removed from the index-list.  The done-list is thelist      '() (append n-list (list (caddar part-list)))))
;;; list of those dimensions for which code has been generated.          ((null? part-list) (list var-list spread-list n-list))))

(define (remove-refs index-list done-list)
  (do ((new-list '() new-list)                                      ;;; This procedure returns a list of three items.  The first item is a list of
       (index-list index-list (cdr index-list)))                    ;;; the biggest positive offset in each dimension of the doall.  These values
      ((null? index-list) new-list)                                 ;;; are used to decide the division of the body and tail of the doall.  The
      (if (not (every? (lambda (x) (memq? x done-list))            ;;; second item is the equivalent for the negative offsets, containing the
                       (cdar index-list)))   ;;; ignore the array-name smallest values in each doall dimension.  These values are used to decide
          (set new-list (append new-list (list (car index-list))))));; the division between the head and body of the doall.  The third item is a
                                                                    ;;; table of lists.  Each entry of the table corresponds to a doall dimension.
                                                                    ;;; Each entry is a list of all the offsets in that diemnsion, with
                                                                    ;;; duplications removed.  This table is used to make the variables in the body
;;; Takes a variable and returns its first letter.  For example, I_21;-ofIthe doall which replace the FX-REM expressions of array references.
;;; This also handles cases with offsets, for example, if var is
;;; FIXNUM-ADD I_11 '4 because of the offset of four, this can stilldefinectoffset-magnitude-extractor sym-list doall)
;;; the variable I.  SHOULD BE EXTENDED TO ALLOW FOR VARIABLES OF MORElet*HAN((pos-tbl (make-symbol-table))         ;; tables of values
;;; ONE LETTER?                                                          (neg-tbl (make-symbol-table))
                                                                         (ref-list-tbl (make-symbol-table)))   ;; tables of lists of values
(define (get-letter var)                                              ;; Initialize the tables
  (if (atom? var)                                                     (map (lambda (x) (set (table-entry pos-tbl x) '0)) sym-list)
      (string->symbol (char->string (char (symbol->string var))))     (map (lambda (x) (set (table-entry neg-tbl x) '0)) sym-list)
      (if (= 3 (length var))                                          (map (lambda (x) (set (table-entry ref-list-tbl x) '(0))) sym-list)
   (map
     (lambda (arr-list)
        (map (lambda (indices)
               (do ((inds (cdr indices) (cdr inds)))
                   ((null? inds))
                   (let ((var (get-letter (caar inds))))
                     ;;; keep only the biggest pos and smallest neg
                     (if (table-entry pos-tbl var)
 (block
  (if (> (cdar inds) (table-entry pos-tbl var))
      (set (table-entry pos-tbl var) (cdar inds)))
  (if (< (cdar inds) (table-entry neg-tbl var))
      (set (table-entry neg-tbl var) (cdar inds)))))
                     ;;; keep all of the offsets in table-- no dups
                     (if (not (memq? (cdar inds)
                                     (table-entry ref-list-tbl var)))
                         (set (table-entry ref-list-tbl var)
                              (cons (cdar inds)
                                    (table-entry ref-list-tbl var)))))))
       (cdr arr-list)))
      (%doall-index-list doall))
   (list (map (lambda (x) (table-entry pos-tbl x)) sym-list)
         (map (lambda (x) (table-entry neg-tbl x)) sym-list)
         ref-list-tbl)))


;;; Interval Splitting specific procedures


;;; This creates the list of CASE options for the case statement that
;;; assigns the interval lists to the processors.

(define (make-case-intervals init limit task-spread task-n data-spread-list
     doall sym)
  (let ((case-code '())
(interval-list (make-interval-list init limit data-spread-list))
(no-do-int? t))  ;;; assume true, prove false
    (do ((tid 0 (+ tid 1)))
((= tid task-n)
 (block (if no-do-int?
    (set (table-entry (bgen-no-do-table
       (%doall-blockgen-private doall))
      sym)
 t))
case-code))
      (let ((ints (get-intervals tid task-spread interval-list)))
(set case-code (append case-code
     '(((,tid) ,(append '(list) ints)))))
(set no-do-int? (and no-do-int? (= 2 (length ints))))))))



;;; This creates the entire list of intervals for a particular loop dimension.
;;; This one list is used to create the individual interval lists for each
;;; processor.

(define (make-interval-list init limit data-spread-list)
  (let ((first-list  (list init (+ 1 limit)))   ;; raw list
```

```
(second-list '()))                 ;; dups removed

    ;;; Place all intervals in list
    (do ((spr-list data-spread-list (cdr spr-list)))
((null? spr-list))
       (do ((interval 0 (+ interval (car spr-list))))
((> interval limit))
(if (> interval init)
    (set first-list (cons interval first-list)))))


    ;;; Remove duplicates
    (do ((interval (car first-list) (car first-list)))
((null? first-list))
       (block (set first-list (delq interval first-list))
 (set second-list (cons interval second-list))))


;;; return list sorted in increasing order
    (sort-list second-list <)))


;;; This returns the list of intervals for which the procesor tid is
;;; responsible.  The segment is taken from the complete interval-list
;;; computed earlier.

(define (get-intervals tid task-spread interval-list)
  (let* ((x-min          (+ (car interval-list) (* tid task-spread)))
 (x-max          (+ x-min task-spread))
this-interval (list x-min)))
    (do ((lst interval-list (cdr lst)))
((or (null? lst) (>= (car lst) x-max))
(if (null? lst) this-interval
    (append this-interval (list x-max))))
     (if (> (car lst) x-min)
  (set this-interval (append this-interval (list (car lst))))))))
```