COMPILATION AND RUN-TIME TECHNIQUES FOR DATA-PARALLEL PROGRAMS

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering

in

The Department of Electrical and Computer Engineering

by
Swaroop Vasudeva Dutta
B.Tech., Regional Engineering College, Warangal, India, 1994
December 1997

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to Dr. Ramanujam for his guidance throughout this work. His constant encouragement and motivation throughout the course of this thesis gave me the necessary drive to step closer to my research goals. I am also grateful to him for having provided financial assistance to me during my entire stay at LSU. His advise on aspects other than research has also helped me throughout. It was really a great pleasure working with him.

I would also like to thank Dr. Subhash C. Kak and Dr. R. Vaidyanathan for serving on my committee.

I also take this opportunity to thank Dr. Tumay, Director, Louisiana Transportation Research Center, for providing financial assistance during my stay. The minicone group also comprising of Dr. Kurup, William Tierney and Yigal Bynoe were especially understanding and supportive.

I thank my parents, my sister and my brother-in-law for having stood by my side throughout and providing every possible type of support to make my walk in life a lot easier and merrier. The very feeling that they were always there to support me gave me enormous confidence.

I thank Arun Venkatachar for being an ideal senior advising on important aspects. A special thanks to my roommate for a year and a half Senthil. Lastly I thank all my friends and colleagues for all their help in the course of my stay at LSU.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

Abstract

With the advent of data-parallel languages such as High Performance Fortran and the enormous compiler support for these languages, the difficult task of parallel programming is becoming easier. This thesis addresses several important problems in compilation of programs in such languages. Union and difference of regular sections (routinely required in code generation) are expensive operations. The results of these operations on regular sections exhibit a pattern that can be expressed as a finite-state machine; this thesis presents efficient techniques for constructing the pattern. Two methods are presented for generating the access sequence with two-level mapping of data onto processors supported by HPF; the first is based on hashing, and the second is an improvement over an existing technique as a result of replacing expensive integer operations. A large part of the efficiency of the node code derived by a compiler for HPF depends on the type of data structure used for node code generation. Techniques such as strip mining, compression and expansion of tables etc. using new efficient data structures are introduced. A key part in the address generation of complex subscripts is the address generation for coupled subscripts. Coupled subscripts have the deepest loop index common among the two dimensions, which makes the address generation difficult. Methods using hashing and search are employed once again to enumerate the common iterations in the various array dimensions. Dynamic redistribution plays an important part in HPF and determining the send and receive processor and data sets form the crux of the redistribution problem. Current approaches rely on multi-phase redistribution which incurs high communication overhead. This thesis presents an efficient single-phase table-based solution. Experimental results show the clear superiority of the solutions presented in this thesis for each of the five problems addressed.

CHAPTER 1

INTRODUCTION

Parallel processing appears to be the most promising option for the ever-increasing demand for faster and efficient computation. However to program such machines and to use them most efficiently, enormous support from languages, compilers and run-time systems is essential.

Current parallel machines can be divided broadly into two classes, namely shared memory machines and distributed memory machines. Shared memory machines have a global memory which is directly accessible to all the processors in the system. Programming such machines is easier as the processors have a single globally shared address space. Nevertheless, shared memory machines do not scale well to a large number of processors. In a distributed memory machine, as the name suggests, processor have their own local memory and cannot access the data in memories on other processors except through explicit data transfer from one processor to another. Consequently, programming such machines is a difficult task. However distributed memory systems are scalable and highly flexible. The sharing of data among processors in a distributed memory environment is accomplished through message-passing over the interconnection network among the processors. The messages constituting the data are transferred to the various processors through well established protocols already known to the processors. Here in the distributed memory environment the mode of communication is the network binding the processors together while in the case of the shared memory system the mode of communication is the shared memory which is shared among various processors. In this thesis work, we limit our attention to the message-passing machines programmed using the Single Program Multiple Data (SPMD) model. The SPMD model is very useful for a compiler to generate code for, since in a distributed memory message-passing machine. In the SPMD model, every processor executes the same piece of code albeit on different data; the SPMD model can be thought of as a loosely synchronous variant of the popular data-parallel model.

As stated earlier, programming distributed machines has remained a difficult task. However with a lot of research and evolution of parallel languages such as High Performance Fortran (HPF) [11], Fortran D [7], and Vienna Fortran [3], the support for programming such machines is increasing. Efficient compilers and run-time systems for these languages are essential to ease the burden of programming these machines [11]. Kennedy et al. [15] note:

> "Widespread use of data-parallel languages, such as HPF, will not come about until fast compilers and efficient run-time systems are developed. . . . Only after these problems are fully and efficiently solved, can the use of such novel features of HPF become commonplace."

The popularity of these languages can be traced to the fact that these languages retain the basic syntax and formalism of their sequential counterparts with added features for parallelism. For example, HPF includes directives such as `ALIGN` and `DISTRIBUTE` that distribute data among various processors. These directives act as hints to the compiler to distribute the data among various processors and generate the required message-passing code. The programmer need not worry about the low-level details of how data is distributed physically onto the processors. This eases the task of the programmer who does not have to concentrate on the low-level details of message passing and data distribution. These facts will and are making parallel systems popular.

1.1 DATA MAPPING

Most of this thesis work is related to code generation for HPF and data partitioning supported by such languages. In languages such as HPF, the user is allowed to specify the partitioning of data onto processors through compiler directives using which a compiler can generate node-code for data-parallel programs. HPF allows a two-level mapping of data to processors (referred to as abstract processors). Data objects are first aligned to an abstract discrete Cartesian grid called a template. This group of arrays are then distributed onto the physical processors perhaps through another layer of abstraction viewed as a rectilinear arrangement of abstract (virtual) processors.

The most general regular distribution is the *block-cyclic* or the CYCLIC(k) distribution, in which contiguous blocks of data of size k are assigned to processors in a round-robin fashion. For references to arrays with CYCLIC(k) distributions, separate computation of the processor number and the local address for each element accessed carries unacceptably high run-time overhead; hence, compilers and run-time systems must try to minimize the cost of generating memory addresses in computations over block-cyclically distributed arrays.

```
        REAL A(20)
!HPF$   TEMPLATE T(40)
!HPF$   PROCESSORS PROCS(5)
!HPF$   ALIGN A(j) WITH T(2*j)
!HPF$   DISTRIBUTE T(CYCLIC) ONTO PROCS
        DO i = 0, 15
             A(i) = ···
        ENDDO
```

The ALIGN directive specifies the alignment between the data objects and the template. This directive instructs the compiler to align the given array to the template. In the code example above, an element of the array $A(j)$ is aligned with $T(2j)$. This is an example of *stride* alignment. When the alignment stride is one, the mapping is referred to as a one-level mapping. Mappings with alignment strides greater than one are referred to as two-level mappings. The `DISTRIBUTE` directive specifies a mapping of data objects to abstract processors in a processor arrangement. This is the step following alignment in

HPF-style data mappings. A template may be distributed among the various processors either in *block*, *cyclic* or *block-cyclic* fashion. The block distribution specifies that if the array is of size N then it should be distributed evenly in blocks of $\frac{N}{p}$ among the p processors. In the case of cyclic distribution, template elements are assigned to processors in a round-robin fashion. With a *block-cyclic* distribution (referred to as CYCLIC(k)), blocks of template of size k are allocated to processors in a round-robin fashion.

The node-code that is generated for each processor by the HPF compiler should be highly optimized as this node code determines the execution time of each processor. The compiler has to apply several optimizations in the final node-code to be distributed among all the processors. The node-code should generate the sequence of local memory addresses for each array element accessed by a processor. Basically when the array is aligned to the template the accessed elements are in regular steps called the strides. However when the template is distributed among various processors then the accessed elements appear in a seemingly haphazard manner. Much recent research has been on the address sequence generation problem [1, 4, 12, 10, 15, 16].

Most of the previous efforts in compiling programs with CYCLIC(k) distributions have dealt with only fully parallel segments (HPF array statements or independent loops), where array accesses can be enumerated in arbitrary order. This is true for the linear algebra framework due to Ancourt et al. [1], the virtual processor scheme by Gupta et al. [10] and the method based on the intersection of array slices by Stichnoth, O'Hallaron and Gross [28]. Chatterjee et al. [4] exploit the periodicity of memory accesses to generate a table that can be used to traverse local regular section even if the code is not parallel. They show that for an array with the CYCLIC(k) distribution the table has at most k entries and present an $O(k \log k)$ table construction algorithm. Subsequently, Kennedy et al. [15] and Thirumalai and Ramanujam [34] have improved the complexity of address generation to $O(k)$. In addition, communication needs to occur for the case of non-local data. The *owner-computes* rule specifies that the processor will perform only those computations or assignments for which it is the left hand side owner. Access to right hand side variables or data is provided by inserting sends and receives. Communication generation has been studied by [17, 36, 39].

1.2 OVERVIEW OF THIS THESIS

The problem of address generation has received considerable attention. In this thesis, we present novel solutions for problems such as node code generation, address generation for coupled subscripts and two-level mapping, run-time redistribution, etc. In addition, many new efficient techniques are discussed that will help in solving the problems that are mentioned. In this section a brief overview of the research done in the various areas is described.

1.2.1 REGULAR SECTIONS

The first of the problems consists of finding the union and the intersection of regular sections. Regular sections are a set of array elements that are accessed in a regular manner, e.g., using a constant stride. Regular sections can be described by parametric descriptions called the *Regular Section Descriptors* or RSD's. RSD's define the upper and the lower bound of the section and the access step of the section. Computing the intersection, union or difference of two regular sections in the most efficient manner is a commonly occurring problem; union and difference are required for address sequence and node code generation for HPF programs. While regular sections are closed under intersection [6], they are not closed under *union* or *difference* operations; however it will be shown in the chapter on regular sections that the union or the difference of two regular sections is representable as FSM and can therefore be encoded as a pattern table. So far, researchers have relied on expensive run-time solutions for these. For example, Stichnoth [28] and Gupta et al. [10] considers the problem of address generation as the union and intersection of closed forms. Here the template space on each processor is divided into columns. Each column can be then regarded as a closed form with an access stride of $\frac{s}{\mathrm{gcd}\,(pk,s)}$. Here s represents the access stride for the array, p represents the number of processors and k represents the block size. In our work on the union and difference of regular sections, the FSM pattern is recognized and encoded in the form of a table. When the actual enumeration is done at run-time this table can be looked up and a simple computation done to determine the next element accessed. This is very efficient compared to methods that use bounds checking and guard evaluation which are pure overhead. Also with this scheme the global-to-local address transformation need not be done for the entire iteration space. Therefore the table-driven method for regular sections proves to be very effective.

1.2.2 TWO-LEVEL MAPPING

As the name suggests this problem deals with the address generation for two-level mapping. Earlier work on this has been done by Chatterjee et al. [4]. The basic algorithm presented in [4] was to apply the algorithm for the one-level mapping case twice. It converts the gaps in the two state machines derived into gaps in actual memory layout using very expensive integer arithmetic. Venkatachar [38] presented a variety of techniques for two-level mapping based on search, in addition to a novel technique called the ITABLE method. The ITABLE method involves expensive integer division operations that needs to be eliminated or optimized with some other less expensive alternative method. Using the properties of the operations in the itable method, we show how the integer division of the ITABLE method can be replaced by table lookups. We introduce hashing for the two-level mapping problem as well. In order to maintain simplicity and at the same time efficiency the simplest of the hashing methods was employed and it was also seen to perform efficiently. Similar to all the methods, this method too generates the addresses by the application of the one-level algorithm twice. So to explain simply whichever elements in both the levels hash onto the same space is assumed to be accessed and therefore forms a part of the pattern table for the two-level address generation.

1.2.3 Efficient node code using good data structures

An important contribution of this thesis work is the work on data structures. Choosing a good data structure for the efficient enumeration of the accessed elements is essential for good performance. Coelho et al. [6] observe:

> "The choice of data structure ranges from array sections to polyhedra for regular references, and includes lists for irregular applications. It should ne noted that more general the data structures are, the more expensive the operators are . . . Desirable features for such a data structure are: (1) minimality with respect to the describe set, to avoid guards; (2) minimality with respect to the number of operations involved . . .; (3) closure under common operators (union, intersection, difference); and (4) easy to scan in the programming model . . ."

In the case of the address generation in the one-level case the data structure most commonly used is the memory-gap table also popularly referred to as the ΔM table [4]. Another table proposed was the *Nextoffset* table by Kennedy et al. [15]. This table is an encoding of the FSM. In our work we propose six new data structures for the efficient enumeration of elements at run-time. These data structures have a space vs. time tradeoff as suggested by Knies et al. [18]. The main idea is to enumerate the elements using the table and at the same time to eliminate the expensive guards. The six data structures take into account various factors such as the size of the tables and the problem parameters. The guard comes into play when the table is traversed over the entire length and then the index needs to point to the beginning of the table again. This may be done by a conditional statement or a mod operation. It can also be done by a *Nextoffset* table that encodes the FSM. In this case the FSM is encoded as a table. This means that the entries in the *Nextoffset* table indicate the next index (i.e., column number from 0 to $k-1$) to jump to in the table. However the size of the table is always k independent of the actual number of entries in the memory gap table. Table lookups are known to be considerably better than conditional statements [18] provided the tables are of moderate size. In our work we propose a new scheme for eliminating the conditionals by having a new data structure in the form of a *reset* table. The reset table points to the next index to jump to in the ΔM table. It is more advantageous than the *Nextoffset* table as it is smaller and therefore has space as well as time efficiency.

1.2.4 Coupled subscripts

This case involves when the deepest loop index variables occurring in the two subscripts are identical. An example of a normalized loop with coupled subscripts is shown below:

```
DO j = 0, u
    A(l1 + js1, l2 + js2) = ...
ENDDO
```

In Kennedy et al. [16] it is shown how to handle the array references with coupled subscripts using the table-based method of [4]. In this approach the problem revolves around finding the starting element on each processor. This basically is finding the first common iteration among the two dimensions of the array which are distributed. That is, first, the iterations are enumerated in the two dimensions independently and then the common iterations are determined to form the ΔM table which is also two dimensional. Kennedy et al. [16] approach this problem by solving $k_1 k_2$ equations to determine the common iterations and by sorting the common iterations. This is an $O(k_1 k_2 log(k_1 k_2))$ algorithm. On a closer look the problem is once again that of searching and finding the common iterations in the two dimensions. This can therefore be again solved using our search algorithms proposed in [38] or the hashing method mentioned earlier. The problem has been approached using both these methods and it was observed that the results were highly favorable. Once again the method of hashing provides an innovative approach into the problem.

1.2.5 REDISTRIBUTION

The problem of redistribution refers to dynamically changing the distribution of arrays across processors. When the array distribution has to change dynamically a lot of issues come into play. Now any processor m, after the redistribution may require some data from other processors. Similarly the same processor may need to send data to other processors. So the processor m has to perform message packing and unpacking in the most efficient manner. In the message packing phase, the processor packs all the data it needs to send into the *sendbuf* and also stores information as to which processor it has to send the data. In the message unpacking phase, the processor has to unpack the data from the *recvbuf* and assign it to the local memory elements. The previous work on redistribution [14, 30] performed the redistribution in a multi-phase manner. In Thakur et al. [30] the first step involved the redistribution from a CYCLIC(k_1) distribution to CYCLIC(lcm (k_1, k_2)) (CYCLIC(gcd (k_1, k_2))) distribution. The second step involved the redistribution from CYCLIC(lcm (k_1, k_2)) (CYCLIC(gcd (k_1, k_2))) to CYCLIC(k_2) distribution. However in such a multi-phase redistribution the communication overhead due to communication startups, etc. is very high. In all the previous work the enumeration required the traversing of the whole iteration space. This required a lot of guard elimination and conditional testing. But in the method proposed in this work, a repeating pattern is recognized. That is, it is known that the send and the receive buffers exhibit a repeating pattern of sequence. This is used to construct a pattern table for the send and the receive buffers. This eliminates costly guards when a large number of array elements are involved. The message packing and unpacking involve simple table lookups which are less expensive for moderately sized tables. The pattern is found to repeat after lcm (k_1, k_2) elements and therefore the length of the table is lcm (k_1, k_2). The node code execution for message packing and unpacking becomes extremely efficient due to the method proposed by us.

1.3 ORGANIZATION OF THIS THESIS

This thesis is organized as follows. In Chapter 2 we discuss issues relating to the union and difference of regular sections. As the regular sections are a very general concept and can be used in many instances of problems in HPF, this concept is therefore explained at the beginning. Here table construction techniques are described following which node code using these tables are explained. Extensive results show the efficacy of our approach. In Chapter 3 we present address generation techniques for two-level mapping. Important concepts relating to the optimization of the ITABLE method and hashing are explained. In Chapter 4 the various new data structures proposed are discussed. The concept of the *reset* table mentioned earlier has been explained here. Comparison of the use of one data structure against the another is shown using actual node code. Chapter 5 describes address generation for coupled subscripts. Methods such as hashing and search techniques are discussed and extensive results are presented. Chapter 6 of the thesis discusses problems related to redistribution of arrays and efficient ways to solve the problem. A pattern table is essentially constructed to facilitate the efficient enumeration of data to be packed and unpacked. Chapter 7 of the report concludes by summarizing all the above discussed methods and also points to future work.

CHAPTER 2

REGULAR SECTIONS

A regular section is a set of elements of an array A denoted usually by $A(l : h : s)$. The parameter l is the lower bound on the section, h the upper bound and s the access stride; the section starts at the array bound l and accesses elements in steps of s, and no elements outside the range $[l \ldots h]$ are accessed. This set constitutes the set of accessed elements. In numerous problems concerning HPF regular sections play an important part. Consider two regular sections $A_1(l_1 : u_1 : s_1)$ and $A_2(l_2 : u_2 : s_2)$. Enumerating the accessed elements for a union or difference of the two regular sections in the most efficient manner is important. One can view an array section with access stride s as a finite-state machine (FSM) with a single state with transitions marked by s. The intersection of two regular sections with strides s_1 and s_2 forms an FSM with one state with transitions marked by lcm (s_1, s_2). However the union or difference form FSMs with more than one state. Recognizing and deriving FSMs that represent the union and difference operations is the topic of this chapter. We present results on the execution (or access) of the elements of a regular section.

This chapter is organized as follows. The first section discusses the problem statement. The second section discusses the related work done in this area. The third section explains our solution to the enumeration of all accessed elements in the case of the union of two regular sections. The fourth section deals with the difference of two regular sections and the final section presents the results and conclusions drawn from the chapter and the work done.

2.1 PROBLEM STATEMENT

Consider a regular section $A_1(l_1 : u_1 : s_1)$, where l_1 is the lower bound of the section, u_1 the upper bound and s_1 the access stride.

The regular section with the accessed elements for $l_1 = 5$, $u_1 = 30$, and $s_1 = 3$ is shown in Figure 2.1; the accessed elements are the one which are circled in the figure. The corresponding FSM is shown in Figure 2.2; the FSM consists of only one state with the access stride 3. Consider the next regular section shown in Figure 2.3 with the lower bound $l_2 = 5$ and the upper bound $u_2 = 30$ and the access stride for this regular section is given by $s_2 = 4$.

When a union of the two regular sections is taken the combined set of elements is shown in Figure 2.4. The union of the two sections consists of elements from the first set

5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30

Figure 2.1: Regular section $A_1(5 : 30 : 3)$

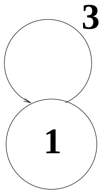

Figure 2.2: Finite state machine(FSM) for a regular section with one state

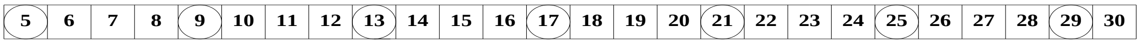

Figure 2.3: Regular section $A_2(5:30:4)$

and the elements from the second set. However the problem becomes complex as the set has to be in the lexicographic ordering of the elements. If the union of two regular sections was not lexicographically ordered then the union of $A_1(5:30:3)$ and $A_2(5:30:4)$ is the concatenation of the two arrays but care must be taken to prevent duplication of elements in the union. It is the particular lexicographic ordering that makes the problem more difficult. As can be seen, now the accessed elements appear to be in no simple pattern unlike the case for the individual array sections.

Most of the work in this area so far failed to recognize the pattern in the union of the two regular sections. One of the brute force methods of generating the accessed elements is combining the two sections and then sorting them. However sorting can be really expensive even for moderately sized arrays. New methods have to be devised which do not involve any sorting.

The second operation of considerable importance in the regular section is the difference. The difference of two regular sections consists of all the accessed elements in the first regular section which are not accessed in the second regular section. In a naive enumeration this may involve the searching of each accessed element of the first regular section in the second regular section. As stated above any regular section is an FSM with a single state. The next question that arises is whether the union of two regular sections is an FSM. As the following sections show, the union and difference of two regular sections is an FSM.

The number of states in the FSM is

$$\operatorname{lcm}(s_1, s_2)\left(\frac{1}{s_1} + \frac{1}{s_2}\right) - 1. \tag{2.1}$$

The notation that is going to be followed in this chapter is as follows.

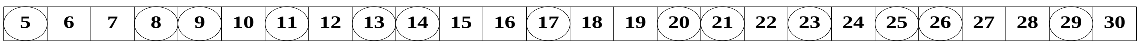

Figure 2.4: Union of $A_1(5:30:3)$ and $A_2(5:30:4)$

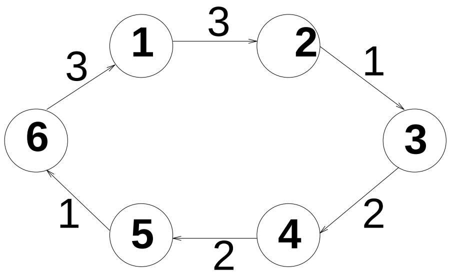

Figure 2.5: FSM for the union of $A_1(5:30:3)$ and $A_2(5:30:4)$; start state $= 1$

- A_i: any array that is considered
- l: the lower bound of the array that is considered
- u: the upper bound of the array
- s: the access stride of the elements

It can be noted that any of the parameters l, u and s can be negative. Our algorithm and subsequent implementation takes into account such cases too. The particular cases when the stride is negative needs some description of how the access elements are generated. When the stride is negative in the particular case of $A(-4:-75:-7)$, the accessed elements are $\{-4, -11, -18, \ldots, -74\}$. In formal notation this can be stated in the following manner. For positive strides the array section $A(l:u:s)$ enumerates the set of elements = (l, $l+s$, $l+2s$, ..., $l+s\lfloor\frac{u-l}{s}\rfloor$). For negative strides the set of elements accessed is (l, $l-s$, $l-2s$, ..., $l+s\lfloor\frac{|u-l|}{|s|}\rfloor$). The other type of notation that is commonly followed is the following. The array section is represented by four parameters instead of the usual three. The fourth parameter is r and is known as the representative element. The array section is shown like this $A(l:u:s;r)$. The accessed elements satisfy the following constraint: $u \geq (r+is) \geq l$. The representative element is a useful method of representation of any regular section. This representation is interpreted in the following manner: regardless of the bounds on the array the representative element has to be accessed. Consider an example, let $A(5:30:4)$ be an array section. The representative element may be any of the elements $(\ldots, -3, -1, 5, 9, 13, 17, \ldots)$ in order to define the same regular section; This representation is particularly useful for negative strides.

2.2 Union of two regular sections

The union of two regular sections is an important problem consideration in the development of compilers for parallel languages like HPF. Let us consider the same example that was shown in the previous section for which the FSM is shown in Figure 2.5.

As can be seen from the figure the FSM consists of six states. So all the accessed elements of the array section can be derived by starting at an element and a particular state in the FSM. The next state of the FSM is reached by adding the FSM value like the one shown above the arrows. This is actually the difference between the two states and hence the gap (difference) between the two accessed elements. This FSM can also be represented by a pattern table. The elements in the pattern table for the particular example that was shown in the previous section is (3, 1, 2, 2, 1, 3). The elements that were accessed were (5, 8, 9, 11, 13, 14, 17, ...). It can be seen that the pattern starts repeating with a period lcm (s_1, s_2), i.e., the sum of the gaps around a cycle = lcm (s_1, s_2). So in our example the period is 12. The first block consists of the elements (5, 8, 9, 11, 13, 14) and the second block of repeating elements consists of (17, 20, 21, 23, 25, 26). Also it can be clearly observed that it is only in the overlapping region of the two array sections, i.e., in the range $[\max(l_1, l_2) \ldots \min(u_1, u_2)]$ that the FSM pattern is in effect. Outside this range, the access sequence is either s_1 or s_2. In our strategy we generate the FSM or the pattern table and exploit it for future accesses. The generation of the pattern table is done by comparison of the two array section elements. The pattern table can also be generated through hashing which may be effective for small values of s_1 and s_2. Our algorithm is shown in Figure 2.6.

A step by step explanation of the algorithm follows. $lcmval$ stores the of s_1 and s_2. It is implemented by making a call to the lcm function which is a trivial piece of code. Steps 2 and 3 of the algorithm compute the first element accessed on the section. For positive values of s_1 this does not look important but for negative strides this plays an important part. Steps 5-6 are initializations for the algorithm. Typically l_1 is assigned the largest accessed element in A_1 less than l_2. In step 7 the algorithm iterates over the first accessed element in the array A_2 to the length of $lcmval$. $lcmval$ is the length of the pattern table ΔE. Steps 7-19 form the main routine for the algorithm and need special consideration. If ptr_1 is smaller than ptr_2 then the difference between ptr_1 and the previous value is stored in the ΔE table and the last value accessed is set to this ptr_1 in Step 10 so that it can be used for the next iteration. If ptr_1 is equal to ptr_2 then the differential value will be entered twice in the table. Therefore this condition is checked for and one of the pointers to the table is incremented in the Steps 11-13. In step 14 the pointer ptr_1 is incremented by s_1 to point to the next iteration to get prepared for the next iteration of the **for** loop. Steps 16-18 are the same analogous steps if ptr_1 is greater than ptr_2. The ΔE is returned as the output of the algorithm. Steps 21-35 shows code symmetrical to the code presented in Steps 5-19. Steps 5-19 show code for the case when l'_1 is less than l'_2 and Steps 21-35 show code for the case when l'_2 is less than l'_1. Therefore the second half of the algorithm need not be explained and the same explanation that goes for Steps 5-19 as above holds. This table is then used for the union and the difference of regular sections. The difference of regular sections is dealt in the next section while the union is dealt in this section itself. Once the pattern table has been determined the code for the enumeration is pretty simple. It just involves determining the start element and then having a table lookup into the pattern table each time. The value acquired from the pattern table is added to the current element that has been accessed to determine the next element. This is obviously because

Input: Array sections $A_1(l_1 : u_1 : s_1)$ and $A_2(l_2 : u_2 : s_2)$
Output: $PatternTable \Delta E$ for the union of array sections
Method:

$lcmval \leftarrow lcm(s_1, s_2)$
$l'_1 \leftarrow$ Compute the first element accessed in A_1
$l'_2 \leftarrow$ Compute the first element accessed in A_2
if $(l'_1 < l'_2)$ **then**
  $ptr_1 \leftarrow l'_1 + \lceil \frac{l'_2 - l'_1}{s_1} \rceil * s_1$
  $prev\_val \leftarrow l'_2;\quad index \leftarrow 0$
  for $ptr_2 = l'_2 + s_2$ **to** $l'_2 + lcmval - 1$ **do**
    if $(ptr_1 \leq ptr_2)$ **then**
      $\Delta E[index] \leftarrow ptr_1 - prev\_val$
      $index \leftarrow index + 1;\quad prev\_val \leftarrow ptr_1$
      if $(ptr_1 = ptr_2)$ **then**
        $ptr_2 \leftarrow ptr_2 + s_2$
      endif
      $ptr_1 \leftarrow ptr_1 + s_1$
    else
      $\Delta E[index] \leftarrow ptr_2 - prev\_val$
      $index \leftarrow index + 1;\quad prev\_val \leftarrow ptr_2;\quad ptr_2 \leftarrow ptr_2 + s_2$
    endif
  enddo
else
  $ptr_2 \leftarrow l'_2 + \lceil \frac{l'_1 - l'_2}{s_2} \rceil * s_2$
  $prev\_val \leftarrow l'_1;\quad index \leftarrow 0$
  for $ptr_1 = l'_1 + s_1$ **to** $l'_1 + lcmval - 1$ **do**
    if $(ptr_1 \leq ptr_2)$ **then**
      $\Delta E[index] \leftarrow ptr_1 - prev\_val$
      $index \leftarrow index + 1; prev\_val \leftarrow ptr_1$
      if $(ptr_1 = ptr_2)$ **then**
        $ptr_2 \leftarrow ptr_2 + s_2$
      endif
      $ptr_1 \leftarrow ptr_1 + s_1$
    else
      $\Delta E[index] \leftarrow ptr_2 - prev\_val$
      $index \leftarrow index + 1;\quad prev\_val \leftarrow ptr_2;\quad ptr_2 \leftarrow ptr_2 + s_2$
    endif
  enddo
endif
return ΔE

Figure 2.6: Algorithm which constructs the pattern table ΔE for determining the union of two regular sections.

the pattern table consists of the difference in the addresses of the consecutive elements accessed. The code for the access enumeration is presented in Figure 2.7.

A step by step detailed explanation of the algorithm follows. Step 1 of the algorithm computes the lcm of s_1 and s_2. Step 2-3 compute the first element as was done in the algorithm for finding the pattern table. When l_1 is smaller than l_2 then the Steps 5-9 state that the elements in the union will be the elements of A_1 accessed in steps of s_1. Next step 12 makes a call to rsd function. This returns the pattern table as ΔE and also the length of the table is returned. The next few steps from 13–20 use this table since the two array sections overlap over this region. Since the section where the two arrays overlap, the union of the two sections form an FSM as given by the ΔE table. The outer loop runs from l_2 to u_2 since this is the region over which the two arrays are supposed to overlap though not necessarily since there may be a case when u_1 is less than u_2. Then the overlap region is from l_2 to u_2. This will taken care of shortly. The loop is iterated and the elements are accessed from the pattern table and added to the current accessed element as this is a differential table storing the difference in consecutive rather than the absolute addresses. In step 14 the case when u_1 is less than u_2 is taken care of because the algorithm exits out of the loop if this condition is satisfied. Once the algorithm comes out of the loop it means that the overlap region has been traversed and now only one of the array sections has to be accessed using its particular stride. This is what the steps from 23-29 do. If the end element has not been accessed in any of these sections then it is done now. Step 31 states that the exact same code should have been executed with the parameters switched if the case l'_2 is less than l'_1. Many cases had to be borne in mind for the enumeration. These were the cases that were considered while forming the union.

Figures 2.8–2.10 clearly show how the union can be done in the most efficient manner. In the Figure 2.8 the union is generated by taking the FSM over the entire regular section as the arrays overlap throughout from the start to the beginning. However for the cases 2 and 3 the initial part of the array A_1 does not overlap the second array. Therefore in the union it is just $A_{union}(0:6:s_1)$. No specific stride has been considered. For the next part of the union the two arrays overlap and therefore the union for that part is an FSM and the values depend on the pattern table values. The third part of the array consists only of the second array and can be written as $A_2(15:21:s_2)$. In this way the union of the two arrays is generated for the 5 cases that have been discussed. The methods described above are theoretically very efficient and the simulation results prove the efficacy of our approach.

Lastly it may be concluded that the complexity of the algorithm is the number of states in the FSM.

2.3 DIFFERENCE OF TWO REGULAR SECTIONS

This section discusses the computation of the difference set between two regular sections. In set theory the difference between two sets S_1 and S_2 is given by all the elements in set S_1 that are not in S_2. This means recognizing all the common elements between the two

Input: Array section parameters $A_1(l_1 : u_1 : s_1)$ and $A_2(l_2 : u_2 : s_2)$, ΔE table and its $length$
Output: Accessed elements in the union
Method:

```
lcmval ← lcm(s_1, s_2)
l'_1 ← First accessed element of A_1
l'_2 ← First accessed element of A_2
if (l'_1 ≤ l'_2) then
   for ptr_1 = l'_1 to l'_2 − 1 do
      if (ptr_1 ≤ u_1) then
         element ← ptr_1
         element is the accessed element
         ptr_1 ← ptr_1 + s_1
      endif
   enddo
   (ΔE, length) ← rsd(l'_1, l'_2, u_1, u_2, s_1, s_2)
   for ptr_2 = l'_2 to u_2 − 1 do
      if (ptr_2 ≤ u_1) then
         for i = 0 to length − 1 do
            if (ptr_2 ≤ u_2 and ptr_2 ≤ u_1) then
               i ← i + 1; element ← ptr_2
               ptr_2 ← ptr_2 + ΔE[i]
            endif
         enddo
      endif
   enddo
   ptr_1 ← l'_1 + ⌈(element−l'_1)/s_1⌉ ∗ s_1;ptr_2 ← l'_2 + ⌈(element−l'_2)/s_2⌉ ∗ s_2
   while (ptr_1 ≤ u_1 and element ≠ u_2) do
      element ← ptr_1; ptr_1 ← ptr_1 + s_1
   enddo
   while (ptr_2 ≤ u_2 and element ≠ u_2) do
      element ← ptr_2; ptr_2 ← ptr_2 + s_2
   enddo
else
   The exact code as above with the parameters switched
endif
return all accessed elements in the union of the regular sections
```

Figure 2.7: Algorithm for the efficient execution of all the accessed elements in the union of the two input regular sections using the pattern table ΔE

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Figure 2.8: Case 1: $l_1 = l_2$ and $u_1 = u_2$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14							
							7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

Figure 2.9: Cases 2 and 3: $l_1 < l_2$ and $u_1 < u_2$ or $l_2 < l_1$ and $u_2 < u_1$ with $s_1 \cap s_2 \neq \phi$

sets and eliminating them from the first set. An example of the difference between the two regular sections A_1 and A_2 is shown in the Figure 2.13.

As can be seen from the figure the only elements missing from the above figure as compared to the array A_1 are the elements (14,26,38). These elements are also present in the second array A_2. The elements which are common to both the arrays are in steps of $lcm(s_1, s_2)$. The lcm in the above case is 12; so the common elements occur in steps of 12 as can be seen. This is a pattern which can be exploited for efficient enumeration. The difference between two sets can be expressed as

$$A_1 - A_2 = A_1 - (A_1 \cap A_2) \quad (2.2)$$

$$A_1 - A_2 = (A_1 \cup A_2) - A_2 \quad (2.3)$$

The second equation is the one which is used for our method of enumeration. This is particularly useful because the earlier section described how to enumerate the elements in a union of two sections. So the second method involves finding the union of the two sets and then removing the elements of A_2 from it. Removal of the elements of the set A_2 is easy because the elements of the set A_2 occur in a regular access stride of s_2. It may be noted that the difference between two sets or two regular sections is once again a regular section. This can be clearly explained from the second equation above. The union of two sets is a regular section as was shown in the previous section. Removal of elements of A_2 whose access elements occur in steps of s_2 till preserves the regularity and therefore the difference of the two sections is also an FSM. The complete algorithm is shown in Figure 2.14. The pattern table for the union is the same as the ΔE table that was constructed in the first algorithm in this chapter in the previous section.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
								8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27						

Figure 2.10: Cases 4 and 5: $l_1 < l_2$ and $u_1 > u_2$ or $l_2 < l_1$ and $u_2 > u_1$

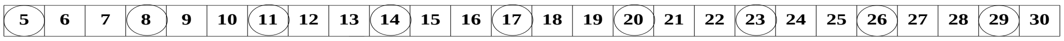

Figure 2.11: Array section $A_1(5:48:3)$

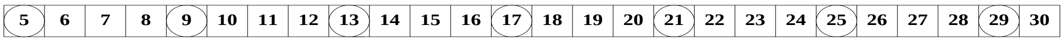

Figure 2.12: Array section $A_2(10:55:4)$

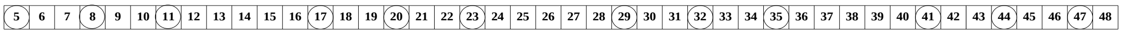

Figure 2.13: Difference between A_1 and A_2

Input: Array section parameters $A_1(l_1:u_1:s_1)$ and $A_2(l_2:u_2:s_2)$
Output: Accessed elements in the difference of the two sections
Method:

$lcmval \leftarrow lcm(s_1, s_2)$
$l'_1 \leftarrow$ First element accessed in A_1
$l'_2 \leftarrow$ First element accessed in A_2
$(union, length) \leftarrow rsdexec(l'_1, l'_2, u_1, u_2, s_1, s_2)$
/* Array $union$ is the union of A_1 and A_2 */
$index \leftarrow 0$; $ptr_2 \leftarrow l'_2$
while ($ptr_2 \leq u_2$ and $ptr_2 \leq u_1$)
  while ($union[index] \leq ptr_2$ and $index < length$)
    if ($union[index] \neq ptr_2$) **then**
      $union[index]$ is the access element
    endif
    $index \leftarrow index + 1$
  endwhile
  $ptr_2 \leftarrow ptr_2 + s_2$
endwhile
return all accessed elements computing the difference of the regular sections

Figure 2.14: Algorithm for the efficient execution of all the accessed elements in the difference of the two input regular sections using the pattern table ΔE

The algorithm for finding the difference of two regular section is explained step by step in a detailed manner. The first step finds the lcm of the two strides s_1 and s_2. Step 2 of the algorithm is a step which has been used in all the previously discussed algorithms in this chapter. This step states that the algorithm that follows defines steps for the case when l_1 is smaller than l_2. If l_2 is smaller than l_1 then the same algorithm can be used with the variables switched. Step 3 makes a call to $rsdexec$ function. This function returns the union of the two regular sections in the $enum\_arr$ array and also returns the length in the variable $length$. Initializations for the algorithm are done in the step 4. The loop in Step 5 runs l_2 to u_2. These are the bounds for the second regular section. This is because the union has been computed. Now the work of the algorithm is to eliminate all the elements that are present in A_2. This will produce as result, the difference between A_1 and A_2. The step 6 will terminate the loop if the ptr_2 is smaller than u_2. The reason for this is that ptr_2 means that the overlapping section between the two sections A_1 and A_2 is between the bounds l_2 and u_1. The ptr_2 always points to the element being pointed to in the second regular section A_2. Step 8 states that in the union $enum\_arr$ all the elements till the next ptr_2 are to be included in the list of accessed elements. Once the ptr_2 is exceeded by the union array then the ptr_2 is incremented by s_2 and the loop executes again. So the main work of the algorithm in steps 8-12 is to access the union array and remove all the A_2 elements from it.

2.4 RESULTS AND DISCUSSION

Extensive experiments were performed on a Sun Ultra SPARC workstation running Solaris 2. The compiler optimizations used were the -O option with the cc compiler. The experiments were conducted with an aim of finding the table construction overhead as compared to the node code execution time. If the table construction time is very high compared to the execution times then the validity of our method does not hold true. The first two columns in the table given in Table 2.1 represent the problem parameters. These are the access strides for the two regular sections given by s_1 and s_2. The next two columns show the result of the simulation results. These are the times for the table construction and the node code execution measured in microseconds of CPU time.

The next table given in Table 2.2 shows the timings for the difference of the two tables. The same pattern table that is ΔE is used for computing the difference of the two regular sections also. The same columns and their interpretations apply here.

It can be concluded from the results that the table construction is not very expensive as compared to the execution time for moderately sized tables. As the table sizes increases it may be observed that the ratio between the execution times and the table construction time decreases. Especially for such large strides as 25 and 99 it is seen that the ratios is particularly low. This is because of the fact that the table size is $lcm(25, 99) = 2475$ and since the array sizes are approximately 10000 or less the number of blocks is equal to approximately 4. In this case the table size itself is comparable to the array size and

Table 2.1: Timings for table construction and enumeration for the union of two regular sections $A_1(15:10500:s_1)$ and $A_2(145:13500:s_2)$

s_1	s_2	Table (μsec)	Exec (μsec)
2	6	4	347
2	12	4	286
3	5	5	266
3	7	4	240
3	11	5	209
4	7	5	195
4	10	4	183
5	7	5	176
6	9	4	179
6	15	4	139
7	2	4	331
7	5	5	176
7	8	5	138
7	12	6	115
7	14	4	152
7	25	6	93
7	99	7	80
8	10	4	124
8	12	5	130
8	14	5	102
9	2	5	320
9	5	5	160
9	6	4	175
9	15	5	104
10	5	4	215
10	13	6	93
12	4	4	230
12	5	5	151
12	10	5	98
13	3	5	209
13	9	6	99
13	10	6	93
14	6	4	125
14	12	5	87
14	99	8	53
25	7	6	98
25	9	6	82
99	25	11	37

Table 2.2: Timings for table construction and enumeration for the difference of two regular sections $A_1(15:10500:s_1)$ and $A_2(145:13500:s_2)$

s_1	s_2	Table (μsec)	Exec (μsec)
2	6	4	1000
2	12	4	840
3	5	5	815
3	7	4	729
3	11	5	658
4	7	5	637
4	10	4	586
5	7	5	578
6	9	4	575
6	15	4	482
7	2	4	1000
7	5	5	594
7	8	5	491
7	12	6	438
7	14	4	519
7	25	6	380
7	99	7	358
8	10	4	452
8	12	5	477
8	14	5	408
9	2	5	961
9	5	5	559
9	6	4	592
9	15	5	406
10	5	4	555
10	13	6	382
12	4	4	711
12	5	5	527
12	10	5	394
13	3	5	691
13	9	6	403
13	10	6	389
14	6	4	477
14	12	5	360
14	99	8	327
25	7	6	409
25	9	6	364
99	25	11	375

therefore not much gain is noticed. Also for such large tables possible cache effects gain prominence.

2.5 CHAPTER SUMMARY

This chapter discusses some simple but very important concepts with regard to union and difference of regular sections. In this section the following points were discussed. First of all the problem with regard to union and difference were explained. It was stated that the operators like union and the difference occur in almost every problem of HPF and therefore should be given close consideration. It was then stated how the regular sections can be represented by a single state and how the union changes the FSM into one with several states. A precise formula was given as to how many states will there be in the FSM representing the union of two regular sections. The union of two regular sections can be formed by recognizing the pattern table called the ΔE. The ΔE table represents the FSM of the union in a tabular form where consecutive elements of the table represent the edge weights of the FSM or the transition weights of the FSM. Table construction techniques for the ΔE table are presented. Once the table is constructed the table is used for the node code execution for the union of two regular sections. Cases such as the strides being negative, lower and upper bounds being negative are all considered. Various cases are presented and it is shown that the table construction technique may actually be very less expensive if the tables are of moderate size. Next part of the chapter discusses how to find the difference between two regular sections. The difference between two regular sections is computed using the pattern table computed for the union of two regular sections. Efficient node code for determining the difference is shown. As can be seen the node code is very simple and does not involve costly guards or conditionals. Finally the experimental results are presented for the simulations run on selected platforms. It was seen that the table construction time occupies a small portion of the execution time and therefore proves to be an effective method for manipulating regular sections and since all the conditionals or guards are replaced by table lookups this method proves very useful.

CHAPTER 3

ADDRESS SEQUENCE GENERATION FOR TWO-LEVEL MAPPING

Generating the sequence of elements accessed on a processor when a two-level mapping of data, i.e., non-unit stride alignment followed by general block-cyclic distribution, poses a challenge for HPF compilers. This chapter presents two techniques, one based on hashing, and the other an improvement (through the replacement of expensive integer operations with table lookups) over the ITABLE technique presented by Venkatachar [38].

3.1 PROBLEM STATEMENT

Consider the following HPF code, for which an HPF compiler must generate the sequence of elements of array A on processor m.

```
        REAL A(N)
!HPF$   TEMPLATE T(a*N + b)
!HPF$   PROCESSORS PROCS(p)
!HPF$   ALIGN A(j) WITH T(a*j + b)
!HPF$   DISTRIBUTE T(CYCLIC(k)) ONTO PROCS
        do i = 0, ⌊(h-ℓ)/s⌋
             A(ℓ + is) = ···
        enddo
```

When the alignment stride a is greater than 1, the data mapping specified by the user is called a two-level mapping; with such a mapping, some template cells may not have any array element aligned to them. A compiler should not waste memory in this case. The sequence of local addresses in such a situation is more irregular than for unit-stride alignment [38]. In general, the sequence of addresses accessed on a processor is no longer an integer lattice and the FSM offsets can no longer be described as a linearization of two basis vectors.

Figure 3.1(a) shows a CYCLIC(4) $(k = 4)$ distribution of template cells onto three $(p = 3)$ processors, in which the global addresses starting from zero are accessed with a stride $s = 3$; this is an example from Chatterjee et al. [4]. We refer to the set of global array elements that are aligned to some template cell mapped to a processor m as the set of *allocated* elements on m; similarly, the set of global elements of the array accessed (due to the array section with stride s) by processor m is called the the set of *accessed* elements. Since a processor can access only a subset of elements it owns, the set of *accessed* elements is a subset of the set of *allocated* elements. For the example

Processor 0				Processor 1				Processor 2				Processor 3			
0			1			2			3			4			5
		6			7			8			9			10	
	11			12			13			14			15		
16			17			18			19			20			21
		22			23			24			25			26	
	27			28			29			30			31		
32			33			34			35			36			37
		38			39			40			41			42	
	43			44			45			46			47		

(a): Global layout of template cells on $p = 4$ processors.

Processor 0	0			1			6			11		
Processor 1			2			7			12			13
Processor 2		3			8			9			14	
Processor 3	4			5			10			15		

(b): Local memory layout for template cells

Processor 0	0	1	6	11	16	17	22	27	32	33	38	43
Processor 1	2	7	12	13	18	23	28	29	34	39	44	45
Processor 2	3	8	9	14	19	24	25	30	35	40	41	46
Processor 3	4	5	10	15	20	21	26	31	36	37	42	47

(c): Local memory layout for array cells.

Figure 3.1: Two-level mapping of array A when $a = 3$, $s = 3$, $p = 4$, $k = 4$ and $l = 0$

in Figure 3.1(a), the set $\{0, 1, 6, 11, 16, 17, 22, \cdots\}$ is the set of *allocated* elements and the set $\{0, 6, 27, 33, \cdots\}$ is the set of *accessed* elements for processor 0. The set of local template cells on the processors is shown in Figure 3.1(b). The problem of generating the accessed set for the template space is the same as the address generation problem for one-level mapping, where the stride s is replaced with $a * s$. Such an approach incurs huge memory wastage and suffers from poorer locality. For example, on processor 1 in Figure 3.1(b), the first twelve memory locations would hold only the elements $\{0, 1, 6, 11\}$ resulting in eight "holes."

We can save significant amount of memory by eliminating "holes" in this as shown in layout, we can have a Figure 3.1(c), where the local address space does not have any memory wastage. However, address generation is complicated and requires translating between the local template space and the local address space. The address generation problem for two-level mapping is to generate the set of elements *accessed* in this local address space.

3.2 RELATED WORK

Chatterjee et al. [4] presented a solution that uses two applications of the one-level algorithm. The first application with stride a builds the FSM for determining allocated elements, and the second with a stride $a * s$ builds the FSM of accessed cells. Then, using a fix-up phase which involves expensive integer operations such as division and mod, they build an FSM of accessed elements in the local array address space. Our experiments show that their technique incurs huge run-time overhead compared to other techniques. Their technique incurs no memory wastage and accesses elements in lexicographic order.

Ancourt et al. [1] presented a linear algebraic solution for the two-level address generation problem. Their solution applies a linear transformation that compresses holes but does not fully eliminate holes. For a CYCLIC(k) mapping, the amount of memory wastage is at most $2k - 1$ and is independent of the array size. In addition, the generated node-code has complicated loops, does not access elements in lexicographic order and incurs high execution overhead.

Kaushik [13] uses a method which generates addresses for both hole-compression and for the one without hole-compression. He first determines the amount of memory needed using a regular section characterization for *block* and *cyclic* distributions. He then extends this technique using a virtual processor approach for handling *block-cyclic* distributions. In this case, the amount of memory wastage can be high and is increases with the size of the array, unlike the technique of Ancourt et al. [1].

Venkatachar [38] developed a variety of techniques for the problem of address sequence generation for two-level mapping. The schemes involve no memory wastage and are efficient. The techniques developed by Venkatachar [38] include the ITABLE method—which we will discuss in detail shortly— in addition to methods based on search (linear and binary) with without replication. In this thesis, we improve the already efficient ITABLE technique.

0	**3**	**6**	**9**	**12**	**15**	**18**	**21**	**24**	**27**	**30**	**33**

Figure 3.2: Addresses accessed on processor 0 for parameters $p = 4$, $k = 4$, $s = 3$

0	**6**	**21**	**27**

Figure 3.3: Addresses accessed on processor 0 for parameters $p = 4$, $k = 4$, $s = 3$, $a = 3$

3.3 HASHING METHOD FOR TWO-LEVEL ADDRESS GENERATION

The earlier work on two-level address generation had been done by Venkatachar [38] . This work exploits the repetitive pattern of accesses by constructing pattern tables for the local address space. These pattern tables are then used to generate the complete set of accesses for the array layout just like in the case of one level mapping. The previous work that was done will be clearly explained in the section on the optimization of the ITABLE method. This section describes a new method which further enhances the efficiency and the timing for the table construction for the two-level problem. The method proposed is that of hashing. The name hashing is very similar to the general concept of hashing and the following section will explain how the concepts are remarkably similar. The previous section explained the problem regarding the two level mapping. The concept of hashing is also used in the chapter on coupled subscripts and therefore is not confined only to two-level mapping. Just as the concept of general hashing is used in a variety of areas involving searches the same concept is used in a lot of areas in HPF. Here is a detailed description of the concept and the methodology. The same example of the previous section is considered here. Recall that the parameters that were considered for the example were $p = 4$, $k = 4$, $s = 3$, $a = 3$. In the first level of mapping the strides to be considered is 3. In the second level of mapping the stride to be considered is $a * s = 9$. The algorithms by [4, 38] take this approach. Further improvements to this basic algorithm were suggested by [38]. The approach taken by us involves the consideration of two tables corresponding to the two strides. Finding the common elements between the two tables forms the crux of the table. Exhaustive search methods can be employed as in the case of [38]. In our case the table does not consist of the difference between consecutive elements but the actual elements themselves are stored in the table.

The elements that are accessed are the ones that are same for both the arrays. It may be noted that the tables just represent one repeating block of the elements. Since the pattern repeats and the nature of the pattern is known it suffices to find the tables for just one repeating pattern. Therefore now the problem boils down to recognizing the common elements between the two tables. This is where the concept of hashing comes into play. The two tables are further hashed into a table whose size is the size before

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
0	0	0	1	0	0	2	0	0	3	0	0	4	0	0	5	0	0	6	0	0	7	0	0	8	0	0	9	0	0	10	0	0	11

1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0

Figure 3.4: The two tables hashed into the hash table

which the pattern repeats. The size of the table after which the pattern repeats is $\frac{pkas}{\text{gcd}\,(as,pk)}$. The two tables hashed into the hash table is shown in Figure 3.4. One could view the concept of hashing as another although space-inefficient search method. Also the hashing method that has been used is the most naive hashing method that has been used. More sophisticated hashing methods may be used, although at the expense of increased running times. This scheme was used just to illustrate the basic efficiency of hashing and how the concept widely used for searching in systems like operating systems and compilers can also be used here. Work is going on to see the effects of other hashing schemes like the one mentioned above.

As can be seen from Figure 3.4 the two hash tables are constructed by hashing the index into the hash table. So the concept is to hash into the hash table with the index of the allocated element. This is important because when the accessed elements hash into the table second time around then the element found at the particular location of the hash table can be directly used without any conversions. This provides an optimization for the code. Two tables are provided to illustrate the hashing of the allocated and the accessed elements into the same hash table. A brief description of the figure follows below. At the corresponding positions of $\{0, 3, 6, 9, 12, \ldots\}$ the values are 1 while at the other locations they are all 0. A similar rule applies to the second table. This method makes the code for finding out the accessed elements very simple and therefore highly efficient. The code for the implementation of this scheme is shown in Figure 3.5.

The algorithm is very simple and easy to implement. The table $new\_pattern\_a$ hashes into a table $hash$. These are the allocated elements. Observe that the place where the allocated elements are hashed into the hash table the value of the array is equal to **i** that is the present index number. This is an interesting point because this shows how the allocated elements are compressed from the template space onto the local memory space. This basically compresses the so called holes in the local memory gap. We therefore do not incur high costs for generating the access function to switch from the template space to the local memory space and this leads to faster execution time. This coupled with the fact that no memory is wasted proves that these methods are superior to any other existing methods which executes elements lexicographically. The value input at the place of hash is the index of the $new\_pattern\_a$ table. The length of the pattern table is new_len_a since the table has been replicated. This explains the basic concept of hashing as applied to the two-level mapping case. One of the obvious disadvantages is that the hashing method consumes a lot of space. However the space and time tradeoff is very effectively achieved. With the advent of newer and newer machines with much lesser

Input: Layout parameters (p, k), loop limits (ℓ, h), access stride s, alignment stride a for array A, processor m
Output: Two-Level pattern table $two\_pattern$
Method:

```
d_1 ← gcd(a, pk)
d_2 ← gcd(a * s, pk)
(length_a, pattern_a) ← one_level(p, k, a, m, d_1)
(length_as, pattern_as) ← one_level(p, k, a * s, m, d_2)
fac ← d_1 s/d_2
new_len_a ← fac * length_a
a_cou ← a/d_1
inc ← 0
len ← 0
for j = 1 to fac do
   for i = 0 to length_a − 1 do
      new_pattern_a[len] ← pattern_a[i] + inc
      len ← len + 1
   enddo
   inc ← inc + k * a_cou
enddo
for i = 0 to new_len_a − 1 do
   hash[new_pattern_a[i]] ← i
enddo
for i = 0 to length_as − 1 do
   two_pattern[i] ← hash[pattern_as[i]]
enddo
return two_pattern
```

Figure 3.5: Hashing based algorithm to construct the two level pattern table

memory restrictions compared to the earlier machines hashing may prove to be overall a very efficient technique in general.

3.4 OPTIMIZATION OF THE ITABLE METHOD FOR TWO-LEVEL ADDRESS GENERATION

As discussed in the previous section another efficient technique that provides for efficient two-level address generation is the ITABLE method. This method is described in the following paragraph. This will help the reader to grasp a better understanding of the optimization done by us. This optimization is explained after the brief overview of the above named technique. A detailed description may be found in [38]. As stated earlier the main idea behind the algorithms is to to construct tables which store indices to switch from the template space to the local space. The algorithm views the address space as an integer lattice and use basis vectors to generate the access sequence of both allocated and accessed elements. The basis vectors are generated using the one-level address generation algorithm [15, 34]. Two application of the one-level algorithm with the input strides a and $a * s$ in each case, generates the set of accesses for both the allocated and accessed elements. In a compressed space we need to locate the position of these accessed elements in the set of allocated elements. The main objective of these algorithms is to generate the switching table which helps in switching from the template space to the local compressed space. The algorithm will be presented here from [38] to illustrate better the modifications done to the algorithm. A full explanation of the algorithm may be had from [38].

As can be seen from the algorithm, line 10 which is an assignment statement forms the Two_Level table. This table is then returned by the algorithm. However, line 10 of the algorithm is a big bottleneck for the efficiency of the algorithm as there is one division which is involved and there is one multiplication. There are three options to this problem of eliminating the expensive div and the $multiplication$ algorithm. The first is to modify the algorithm sufficiently such that the costly div and the mod operations are removed. This possibility has not been explored in our work. The second option is to come up with new algorithms with totally different concepts. This has already been dealt with in the case of [38] and it is observed that all the other methods used for search have not been able to match the performance of the ITABLE concept. Therefore coming up with an algorithm much efficient than the ITABLE remains an open problem. The third option is to eliminate the div and $multiplication$ operations that hinder the performance of the ITABLE algorithm. This is the approach that has been taken in the work done by us. Many techniques already exist for the substitution of the multiplication algorithm by other simple operations like the add done repeatedly. Here a major characteristic of the div operation is taken into consideration while performing the optimization. The point to be noted is that all the divisions in this particular example are integer divides. The division under question is $\frac{X_1[i]}{a_{cou}} * length_a$. A clear optimization has been done for the division and the multiplication using table lookups. That is the division and the multiplication are replaced by table lookups. As stated by Knies et al. [18] there is a time versus space tradeoff.

Input: Layout parameters (p, k), loop limits (ℓ, h), access stride s, alignment stride a for array A, processor m
Output: Two_level
Method:

$d_1 \longleftarrow \gcd(a, pk)$
$d_2 \longleftarrow \gcd(a * s, pk)$
$(length_a, Y_1, Y_2) \longleftarrow one\_level(p, k, a, m, d_1)$
$(length_{as}, X_1, X_2) \longleftarrow one\_level(p, k, a * s, m, d_2)$
for $i = 0$ **to** $length_a - 1$ **do**
    ITABLE$[Y_2[i]] \longleftarrow i$
enddo
$a_{cou} \longleftarrow \frac{a}{d_1}$
for $i = 0$ **to** $length_{as} - 1$ **do**
    $Two\_Level[i] \longleftarrow \frac{X_1[i]}{a_{cou}} * length_a +$ ITABLE$[X_2[i]]$
enddo
return Two_level

Figure 3.6: Algorithm that constructs the ITABLE for determining the two-level access pattern from Venkatachar [38]

The optimization on the time is done by expending more space due to table lookups. To understand the optimization the optimized algorithm is presented in Figure 3.7. The basic structure of the algorithm remains the same as does the concept.

A detailed explanation of the algorithm follows. It may be noted that the main aim of the algorithms in lines 5-23 is to form a lookup table that will be used in place of the divisions and the multiplication. The table is known as $lookup\_acc_1$ in the algorithm. The division is an integer division and this property or characteristic is used in forming the lookup table. The basic concept of the table is as follows. Let us assume that an integer division of $\frac{X}{Y}$ is to be performed and the value of X ranges from $a \cdots b$. So the minimum value of X is a and the maximum value is b. Considering particular values for X, Y, a and b a good example can be set up. Let us assume that the values are $Y = 4$, $a = 0$ and $b = 25$. The lookup table which is formed is shown in the figure Figure 3.8.

As can be seen from the figure if a division has to be made and the dividend is in the range 6 to 25 and the divisor is 4 then any integer division can be performed by a simple table lookup. For example if we need to know the integer division of 17 by 4. Then the table lookup of $lookup\_access_1[17]$ yields 4 which is obviously the correct answer. In this manner the table lookup proves useful. The manner in which the table is formed is what is achieved in the lines from 5 to 23. These lines are now explained. In the algorithm for this integer division to be effective the range of $X_1[i]$ is to be known. Since X_1 is in a sorted order and more precisely in an increasing order the range over which X_1 spreads is simply $X_1[0]$ and $X_1[length_a - 1]$. So in the algorithm the lower value is assigned to

Input: Layout parameters (p, k), loop limits (ℓ, h), access stride s, alignment stride a for array A, processor m
Output: Two_level
Method:

$d_1 \leftarrow \text{gcd}\,(a, pk)$
$d_2 \leftarrow \text{gcd}\,(a * s, pk)$
$(length_a, Y_1, Y_2) \leftarrow one\_level(p, k, a, m, d_1)$
$(length_{as}, X_1, X_2) \leftarrow one\_level(p, k, a * s, m, d_2)$
$a_{cou} \leftarrow \frac{a}{d_1}$
$first \leftarrow X_1[0]$
$tmp_1 \leftarrow length_a * \frac{first}{a_{cou}}$
$tmp_2 \leftarrow first \bmod a_{cou}$
for $i = tmp_2$ **to** $a_{cou} - 1$ **do**
    $lookup\_acc_1[first] \leftarrow tmp\_1$
    $first \leftarrow first + 1$
enddo
$tmp_1 \leftarrow tmp_1 + length_a$
$last \leftarrow X_1[length_{as} - 1]$
while $(first \leq last)$ **do**
    $i \leftarrow 0$
    while ($i < a_{cou}$ and $first \leq last$) **do**
        $lookup\_acc_1[first] \leftarrow tmp_1$
        $i \leftarrow i + 1$
        $first \leftarrow first + 1$
    enddo
    $tmp_1 \leftarrow tmp_1 + length_a$
enddo
for $i = 0$ **to** $length_a - 1$ **do**
    ITABLE$[Y_2[i]] \leftarrow i$
enddo
for $i = 0$ **to** $length_{as} - 1$ **do**
    $Two\_Level[i] \leftarrow lookup\_acc_1[X_1[i]] +$ ITABLE$[X_2[i]]$
enddo
return Two_level

Figure 3.7: Faster algorithm to compute the ITABLE for determining the two-level access pattern by substituting integer divides with table lookups

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
						4	4	8	8	8	8	12	12	12	12	16	16	16	16	20	20	20	20	24	24	24	24	28	28	28

Figure 3.8: $lookup$ table for integer division with parameters Y, divisor=4 and the range is $6 \ldots 25$

$first$. tmp_1 is assigned the result of integer division of tmp_1 with a_{cou}. tmp_2 is assigned the result of the mod of tmp_1 with a_{cou}. In the example considered above the value of $first$ is 6 since it is the first element that is considered or it is $X_1[0]$. The value of tmp_1 is equal to 1 as the integer division of 6 with 4 is 1. Similarly the value of tmp_2 is 2 since the mod of tmp_1 and a_{cou} is 2. The first block may be incomplete and therefore these values need to be computed as they have been done.It can be noted that in this case the first block from 6 to 7 is incomplete and has only two elements. For this incomplete block the $lookup$ table is given the value of tmp_1 which in this case is 1. So lines 9-12 of the algorithm presented fill in entries for the first block of the lookup table. Next to fill up entries for the next block the value of tmp_1 has to be incremented by $length\_a$. The value of tmp_1 in our case becomes 8 as $length_a$ is assumed to be 4. The last value for which the integer division has to be performed is $acc\_1[length_{as}]$. Therefore in line number 14 the value is set to last. From line 15 to 23 the $lookup\_acc_1$ table is filled with the appropriate entries. The loop has to be iterate till the value of $first$ reaches $last$. Since this has to be done in blocks, the value of i in Line 17 ranges from 0 to a_{cou}. $first$ is incremented each time and it forms the index for the lookup table. The corresponding entries for the lookup table are entered. The loop exits when the last element is reached. The inner loop at Line 17 exits when a complete block is done. The lookup table is now ready to be used by the ITABLE algorithm. In Line 28 of the algorithm the lookup table is used as can be seen. The value $X_1[i]$ forms the index for the lookup table. It may be seem surprising that the lookup technique should give better results than the one using raw division. But as stated by [18] there is always a time versus space tradeoff involved. Here to gain on performance with regard to time the space overhead was increased because a lookup table was required for the division. A considerable gain in time was achieved which will be shown in the section on results. This method is not only useful for two-level mapping address generation but is a generic technique for wherever the integer divisions are needed. The same technique has been shown in the chapter on redistribution. Also techniques such as these will be applied in the work on communication where a great amount of performance boost is expected. As with the hashing technique memory overhead in the case of long ranges that is the range $a \ldots b$ will be very high.

3.5 RESULTS AND DISCUSSION

Experiments were run on a Sun UltraSparc 1 Workstation with Solaris 2. The compiler used was the Sun C compiler `cc` with the `-xO2` flag. Though the experiments were run for a huge set of input values only a limited number of results and times are shown here. The number of processors is kept constant at 8 for all the runs. The block sizes considered are

4, 16 and 128. The values of the access stride s and the alignment stride a are shown in the second and the third columns of the table. As can be seen from the three tables shown that FAST-ITABLE method performs well for larger parameters.

3.6 CHAPTER SUMMARY

This chapter discusses extensions to the work done by Venkatachar [38]. In [38] the problem of address generation was solved by using a lot of search techniques. A second important method that was developed in his work was the ITABLE method. Both these methods are dealt with extensively in their work. In this chapter we attempt to provide optimizations for the address generation problem for two level. The first is an innovative approach to the problem based on hashing. It uses the popular concept of hashing used in many computers and other areas such as compilers etc. The hashing scheme used is the simple hashing scheme where elements map onto a unique place in the table. This scheme is seen to be considerably efficient for certain problem parameters and is seen to perform extremely well for small problem parameters. The overhead is the construction of the hash table and space overhead due to the hash table. However the space and table construction overheads are overshadowed by the results demonstrated by this technique.

The second method described is an optimization of the existing ITABLE technique. This means that the ITABLE method is still retained and no basic changes to the algorithm of [38] are made. However in the ITABLE method there are expensive integer division operations involved. This method replaces the expensive integer divisions with table lookups. Each table lookup gives the precise integer division value that is required. So the concern is the table construction time for such a method and the overhead involved with a lookup. As can be seen from the section on results the timings for both the methods proposed, hashing and ITABLE optimization is extremely good and compares closely and most of the times better than the ITABLE method proposed.

Table 3.1: Comparison of Riacs, Hashing, ITABLE and the FAST-ITABLE methods. All times in microseconds. $p = 8$, $a = 2$

s	k	Riacs	Hashing	ITABLE	FAST-ITABLE
3	4	14.80	0.50	1.55	1.55
3	8	22.45	0.80	2.75	2.00
3	16	36.85	1.30	5.20	2.75
3	32	69.65	2.95	9.80	3.90
3	64	130.50	5.80	19.55	5.80
3	128	256.65	9.50	37.95	9.55
5	4	14.40	0.60	1.60	1.65
5	8	22.45	0.95	2.75	2.10
5	16	38.55	2.25	5.25	2.90
5	32	69.40	4.50	9.95	4.05
5	64	132.50	7.30	19.05	5.95
5	128	255.05	12.75	38.05	9.90
7	4	14.90	0.70	1.55	1.65
7	8	22.60	1.15	2.75	2.10
7	16	38.00	3.40	5.30	3.05
7	32	69.25	5.35	9.80	4.00
7	64	131.35	8.85	19.30	6.00
7	128	255.40	15.65	39.65	9.80
11	4	14.95	0.90	1.60	1.85
11	8	22.90	1.50	2.75	2.30
11	16	37.90	4.25	5.25	3.10
11	32	69.35	6.85	9.75	4.35
11	64	131.90	11.90	19.50	6.35
11	128	255.65	21.90	38.10	10.05
23	4	15.05	1.40	1.60	2.00
23	8	22.70	2.65	2.80	2.60
23	16	38.55	6.55	5.25	3.50
23	32	68.70	11.40	9.75	4.90
23	64	131.60	21.25	19.05	7.05
23	128	250.95	42.55	37.50	10.65
25	4	15.00	1.50	1.60	2.05
25	8	22.90	4.25	2.80	2.75
25	16	38.40	6.95	5.25	3.55
25	32	70.10	12.20	9.85	4.80
25	64	132.20	23.55	19.35	7.15
25	128	253.35	45.30	37.95	10.70

Table 3.2: Comparison of Riacs, Hashing, ITABLE and the FAST-ITABLE methods. All times in microseconds. $p = 8$, $a = 3$

s	k	Riacs	Hashing	ITABLE	FAST-ITABLE
3	4	22.85	0.75	2.85	2.30
3	8	37.95	1.30	5.30	3.30
3	16	69.05	2.95	9.85	4.50
3	32	131.55	5.75	19.40	6.40
3	64	251.05	9.50	37.40	10.25
3	128	507.10	17.75	73.40	18.00
5	4	22.95	0.95	2.80	2.50
5	8	38.60	2.25	5.25	3.35
5	16	69.10	4.50	10.05	5.05
5	32	130.55	7.25	19.20	6.65
5	64	256.20	12.55	37.25	10.70
5	128	508.00	23.75	74.30	18.50
7	4	22.70	1.15	2.80	2.80
7	8	38.15	3.40	5.20	3.65
7	16	69.75	5.25	10.00	5.30
7	32	132.35	9.15	19.35	7.45
7	64	259.00	16.15	37.70	10.95
7	128	510.90	31.00	74.05	18.60
11	4	22.50	1.50	2.80	2.05
11	8	38.35	4.25	5.15	3.70
11	16	69.85	6.80	10.15	5.30
11	32	131.00	11.90	19.35	8.50
11	64	252.75	21.90	38.10	11.70
11	128	510.10	48.05	75.85	19.25
23	4	22.70	2.65	2.90	4.55
23	8	38.15	6.55	5.20	5.70
23	16	70.60	11.40	9.90	7.00
23	32	131.75	21.25	19.30	9.25
23	64	260.25	43.80	38.85	15.05
23	128	513.15	92.50	77.55	22.25
25	4	22.75	2.85	2.85	5.95
25	8	38.50	6.95	5.20	5.90
25	16	70.10	12.45	10.20	7.50
25	32	133.00	22.95	19.15	10.50
25	64	257.50	46.50	38.00	15.05
25	128	508.15	99.05	75.05	22.60

Table 3.3: Comparison of Riacs, Hashing, ITABLE and the FAST-ITABLE methods. All times in microseconds. $p = 8$, $a = 5$

s	k	Riacs	Hashing	ITABLE	FAST-ITABLE
3	4	22.55	0.75	2.80	2.30
3	8	38.25	1.30	5.30	3.15
3	16	69.50	2.95	9.80	4.65
3	32	132.70	5.80	19.00	6.55
3	64	250.30	9.85	38.50	10.75
3	128	498.25	17.95	74.65	18.20
5	4	22.45	0.95	2.80	2.65
5	8	38.75	2.25	5.25	3.45
5	16	69.60	4.55	9.90	5.05
5	32	132.65	7.55	19.35	7.40
5	64	249.70	12.55	37.60	11.00
5	128	486.20	23.80	75.80	18.55
7	4	22.50	1.15	2.80	2.80
7	8	38.25	3.40	5.25	3.95
7	16	69.75	5.30	9.95	5.50
7	32	131.85	8.80	19.50	7.85
7	64	256.85	15.65	37.35	11.15
7	128	505.05	33.55	76.40	19.10
11	4	22.85	1.50	2.75	3.30
11	8	37.90	4.30	5.35	4.50
11	16	69.90	6.80	10.20	6.00
11	32	132.75	11.90	19.95	8.75
11	64	256.70	23.20	38.75	13.20
11	128	501.50	49.45	75.70	20.60
23	4	23.05	2.70	2.90	6.05
23	8	38.65	6.60	5.30	7.05
23	16	69.90	11.45	10.10	7.50
23	32	129.05	21.60	19.20	9.90
23	64	259.55	42.75	38.55	14.40
23	128	500.40	89.10	74.70	24.00
25	4	22.75	2.85	2.85	6.95
25	8	38.75	7.10	5.30	6.65
25	16	69.15	12.15	10.00	7.80
25	32	132.05	23.40	19.10	10.45
25	64	256.90	47.45	38.40	15.60
25	128	509.25	95.90	75.20	25.70

CHAPTER 4

NODE CODE OPTIMIZATION USING EFFICIENT DATA STRUCTURES

One of the tasks of an HPF compiler is to generate node code for accessing the local elements mapped onto each processor. While the accesses due to an array section $A(\ell : h : s)$ are regular with a stride of s, the template alignment and distribution make the array access on each individual processor appear to be irregular. First, a compiler has to determine the local access sequence. Then, exploiting the repetitive nature of the local access pattern, node code can be generated for accessing all the array elements assigned to the individual processor.

Several techniques exist for determining the access sequence. Chatterjee et al. [4] view the address sequence as a finite-state machine (FSM) and present an $O(k \log k)$ algorithm for generating the address sequence. Thirumalai and Ramanujam [34] and Kennedy et al. [15] view the problem as one of enumerating the elements of an integer lattice. They present $O(k)$ algorithms for finding suitable basis vectors of the lattice and for enumerating the address sequence using the basis vectors. Thirumalai and Ramanujam [35] present closed forms for basis vectors that are used to enumerate the elements in lexicographic order. These four techniques [4, 15, 34, 35] are thus suitable for both DO loops and the FORALL construct; with FORALL this allows exploiting locality [4]. Several others have addressed code generation for FORALL where lexicographic enumeration is not required. Once the access pattern is determined, it is important to examine how best to represent the sequence as a data structure and how best to generate node code that exploits each of data structure resulting in improved memory and time efficiency; thus, efficient techniques for generating node code on distributed-memory machines is important. For array sections, node code generation must exploit the repetitive access pattern exhibited by the accesses to distributed arrays. Several techniques for the efficient enumeration of the access pattern already exist. But only one paper so far addresses the effect of the data structures used on the run-time performance of the generated node code. This chapter presents several new data structures and associated node codes which show a significant performance gain over existing techniques on several platforms.

This chapter is organized as follows. Section 4.1 deals with the related work in this area and also briefly describes the problem. Section 4.2 of describes the various data structures that have been used for the node code generation and the underlying theory behind their use. Section 4.3 provides the actual node codes using the data structures described in the previous section. Six forms of node code have been provided and their time and space efficiency is discussed and explained for. Two of the node codes have been devised keeping into account the problem parameters where they may be beneficial. However the rest of the node code perform efficiently for all problem parameters. Section 4.4 provides the detailed experimental results along with a comparison of these results with the results of techniques provided in earlier papers. Section 4.5 concludes the chapter.

4.1 BACKGROUND AND RELATED WORK

We consider an array A identically aligned to the template T; this means that if, $A(i)$ is aligned with $T(ai+b)$, then the alignment stride a = 1 and the alignment offset b = 0. Further we assume that the template is distributed using a CYCLIC(k) distribution onto p processors. Given an array assignment $A(\ell : h : s)$ (where ℓ and h are the bounds and s is the access stride) with HPF-style data mappings, it is our aim to generate SPMD node code for each processor.

Thirumalai and Ramanujam [34] presented an address generation algorithm based on integer lattices. They showed that the set of array elements referenced can be generated by integer linear combinations of the basis vectors of the lattice. Let B be the basis matrix of the two dimensional space, where, $B = \begin{bmatrix} L_1 & R_1 \\ L_2 & R_2 \end{bmatrix}$. It was shown [34] that the determinant of B is the access stride s itself. The access pattern for the array A repeats after every $\frac{s}{\mathrm{gcd}\,(s,pk)}$ courses of elements. Thus, the pattern of accesses in the first $\frac{s}{\mathrm{gcd}\,(s,pk)}$ courses can be used to determine the rest of the local addresses accessed. Using a transformation $T = B^{-1}$ and Fourier-Motzkin elimination [25, 41], the code shown below for address pattern generation was derived. The node code for processor m executing $A(l : h : s)$ is easily obtained by making minor modifications to the loop bounds of the nested loop before applying the transformation T [34].

$len \leftarrow 0$
for $u = \left\lceil \frac{(-mk-k+1)R_1}{s_1} \right\rceil$ **to** $\left\lfloor \frac{(s_1-1)R_2-mkR_1}{s_1} \right\rfloor$ **do**
  if $R_1 = 0$
    $lower \leftarrow \left\lceil \left(\frac{mk-uL_2}{R_2} \right) \right\rceil$; $upper \leftarrow \left\lfloor \left(\frac{mk+k-1-uL_2}{R_2} \right) \right\rfloor$
  else
    $lower \leftarrow \left\lceil \max \left(\frac{mk-uL_2}{R_2}, \frac{-uL_1}{R_1} \right) \right\rceil$; $upper \leftarrow \left\lfloor \min \left(\frac{mk+k-1-uL_2}{R_2}, \frac{s_1-1-uL_1}{R_1} \right) \right\rfloor$
  endif
  for $v = lower$ **to** $upper$ **do**
    $\mathit{loc\_pattern}_m[len] \leftarrow (L_1u + R_1v)k + L_2u + R_2v - mk$
    $len \leftarrow len + 1$
  enddo
enddo
for $u = 0$ **to** $len - 2$ **do**
  $\Delta M_m[u] \leftarrow \mathit{loc\_pattern}_m[u+1] - \mathit{loc\_pattern}_m[u]$
enddo
$\Delta M_m[len-1] \leftarrow \mathit{loc\_pattern}_m[0] + k\frac{s}{\mathrm{gcd}\,(s,pk)} - \mathit{loc\_pattern}_m[len-1]$

Processor 0				Processor 1				Processor 2			
0	1	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59

Figure 4.1: Memory layout for an array A aligned to a template where $p = 3, k = 4, s = 5$

Several papers have dealt with the code generation techniques. Chatterjee et al. [4] in their paper describe a technique to identify the repeating access sequence which is represented in the form of a finite-state machine. Their node code involves expensive mod and conditional statements. Gupta et al. [10] derived the virtual block and virtual cyclic scheme. The virtual block or the virtual cyclic scheme involves the union of several cyclically or block distributed arrays. However the lexicographic ordering of elements is not preserved. A recent paper by Kennedy et al. [15] describe three new methods of node code generation. In fact it is the only known paper having actually timed the code generation part. While the other papers also describe the code-generation it is mostly towards finding the access pattern and the time required for the construction of a suitable data structure for the access pattern [4, 15].

Kennedy et al. [15] describe three methods of code generation [15]. The first one is based on conditionals while the second one is entirely compiler-specific for languages with C-like GOTO statements. The third method is a representation of the finite-state machine in the form of a data structure involving a table or an array of offsets. This method though time efficient may not be memory or space efficient. Let us consider an example to show how the pattern table is constructed. On a processor, an access to an array section with stride 5 starting from 0 will result in accesses to $\{0, 5, 10, 15, \ldots\}$. Now consider a system of 3 processors with an array distributed CYCLIC(4) and accessed with stride 5; the local memory layout on each processor and the accessed elements are shown in Figure 4.1. As can be seen the uniform access stride no longer remains true when the array is distributed onto an imaginary template cell. This is the major problem of address generation. The successive local gaps between the elements form the ΔM table and is shown for this example in Figure 4.2.

Memory Gap or $\triangle$ *M* TABLE	**7**	**2**	**9**	**2**

Figure 4.2: The ΔM table for the array access on processor number $m = 0$

7	32	40	65	73	98	106	131	139	164	172	197	205	230	238	271	304	337	370	403	436	469	502	535	568	601	634	667	700	733	766	799

(a) Sequence of local address in repeating access pattern

25	8	25	8	25	8	25	8	25	8	25	8	25	8	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	8

(b) Memory gap table for the access pattern

Figure 4.3: Local pattern and memory gap (ΔM) tables

4.2 DATA STRUCTURES

This section deals with six new data structures that will be used in the next section for the code generation. An obvious assumption that has been made is that the pattern table whose repetitive nature has to be exploited is already built and is available for node-code execution.

4.2.1 MEMORY GAP (ΔM) TABLE

The first data structure explained is the most commonly used one [4, 15]. It consists of storing the sequence of local memory gaps on a processor in the consecutive locations of an array called the *memory gap* table. Therefore the local addresses accessed on a processor is as shown in Figure 4.3(a) before the pattern repeats itself. The table called the memory gap or (ΔM) table for this enumeration is shown in Figure 4.3(b).

The length of the ΔM table is 32. It is a simple data structure and many techniques have been proposed for its efficient enumeration [4, 15, 34]. Some of the other advantages of this data structure are that it is easy to build and does not contain any redundant information. The disadvantages lie in the fact that in the actual node-code generation phase it involves expensive mods and conditionals. Our technique presented in this chapter uses this data structure without the above mentioned disadvantages.

4.2.2 COMPRESSED MEMORY GAP TABLE

The second data structure makes use of the fact that the gaps between consecutive accesses corresponds to the use of one of three vectors: $\vec{R}$, $\vec{L}$, $R \vec{+} L$ where $\vec{R} = (R_1, R_2)$, $\vec{L} = (L_1, L_2)$ and $\vec{R} + \vec{L} = (L_1 + R_1, L_2 + R_2)$ [15, 34]. Consider the memory gap table shown in Figure 4.3. The compressed memory gap of *cgap* table and the corresponding *count* tables are shown in Figure 4.4.

1	1	1	1	1	1	1	1	1	1	1	1	1	1	17	1

25	8	25	8	25	8	25	8	25	8	25	8	25	8	33	8

Figure 4.4: Compressed tables for the ΔM table of Figure 4.3

9	2	7	2

Figure 4.5: The original memory gap table

The compressed table can be interpreted as follows. The access gap *cgap_table*[0] occurs *count_table*[0] times followed by *cgap_table*[1] occurs *count_table*[1] times followed by *cgap_table*[2] occurs *count_table*[2] times, etc. Hence the data structures count_table and cgap_table encode the original table in a compressed manner. As, a result, we incur less table lookups at run-time as each table entry provides for multiple information; this helps in decreased conditional evaluations. There are two tables but there may be possible space savings because of its compressed nature. The main disadvantage is that it increases the depth of loop nests with possibly very short innermost loops.

4.2.3 Expanded memory gap table

The third data structure to be discussed in this chapter comprises of the following features. Let us assume the length of the pattern table derived through the various techniques [15, 34, 4] is *len*. The new table which arises out of this is a replicated table, with a replication factor of *fac*. Consider the memory gap table shown in Figure 4.5. An example of replication for this is shown in Figure 4.6; the replication factor here is three. The choice of the replication factor is primarily based on the need to overcome the negative performance effects of short innermost loops.

So while the method in the previous subsection dealt with compressing the table this is actually an expansion of the table. Though this table seems to be space-inefficient

9	2	7	2	9	2	7	2	9	2	7	2

Figure 4.6: Replicated (expanded) memory gap table $fac = 3$

25	8	25	8	25	8	25	8	25	8	25	8	25	8	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	8

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0

Figure 4.7: ΔM table and the corresponding RESET table

with data redundancy it is very helpful with improved cache performance and reduces the number of conditionals. This is exactly the time versus space tradeoff mentioned by Knies et al. [18]. The node code derived using this representation is compact and the data structure formation does not involve much coding. Also this data representation can be efficiently combined with the structures mentioned above to provide for efficient execution. The table size cannot be extended beyond a certain limit because of cache effects that may finally outweigh the performance enhancement achieved.

4.2.4 MEMORY GAP + RESET TABLE

The next data structure comprises of having a regular memory gap (ΔM) table along with a second table called the RESET TABLE, whose length is the same as that of the memory gap table. Figure 4.7 shows an example of memory gap and RESET tables. The last entry in the RESET table is zero, i.e., RESET$[last-1] = 0$ and for any i except $last-1$ RESET$[i] = i+1$. The advantages of using this form of a representation are that even though it uses space it is much more time-efficient. This observation is based on the paper by [18] where the time versus space tradeoff is explained. So using a table instead of expensive conditionals makes the code execution faster. Also the maximum size of the reset table is equal to the size of the ΔM table which is $\left\lceil \frac{k}{gcd(s,pk)} \right\rceil$. In comparison, the *Nextoffset* table used by Kennedy et al. [15] to encode state transitions in the finite-state machine is always of size k, the block length. Thus, the RESET table data structure introduced here strikes a better compromise between faster execution and space utilization compared to the *Nextoffset* table.

4.3 CODE GENERATION

This section deals with the node code generation using the various data structures presented in the previous section. Basically six types of node code are presented. Code generation presents a problem as various factors have to be taken care of during run-time. The six node codes presented below is a tradeoff between these various factors.

4.3.1 PREVIOUS METHODS FOR NODE CODE GENERATION

There are two known methods for node code generation described by Chatterjee et al. [4] and Kennedy et al. [15]. There codes are described below as they will be referred to quite often throughout this chapter. The first node code, called the RIACS method, in Figure 4.8 employs the memory gap table traversed through its length. Once the end

Input: ΔM table, len: the length of the ΔM table, $startmem$
Method:

$add \leftarrow startmem$
$i \leftarrow 0$
while $(add \leq lastmem)$ **do**
  $A[add] \leftarrow \ldots$
  $add \leftarrow add + \Delta M[i]$
  $i \leftarrow i + 1$
  if $(i = len)$ **then**
    $i \leftarrow 0$
  endif
endwhile
return

Figure 4.8: RIACS method using conditionals

Input: ΔM table, $startmem$, *Nextoffset* table
Method:

$add \leftarrow startmem$
$i \leftarrow startmem \bmod k$
while $(add \leq lastmem)$ **do**
  $A[add] \leftarrow \ldots$
  $add \leftarrow add + \Delta M[i]$
  $i \leftarrow$ *Nextoffset*$[i]$
endwhile
return

Figure 4.9: Rice method using *Nextoffset* table

of the table is reached, the counter, namely *i*, is reset to 0 and the traversal starts again. The addresses that are generated are then used to index into the computational array. The second node code shown in Figure 4.9 requires two tables. One is the *Nextoffset* table and the other is the ΔM table. The *Nextoffset table* is used to compute the next offset or the index in the ΔM table. Hence the *Nextoffset* table is used to index into the ΔM table. Therefore if the *Nextoffset* entry is *Nextoffset*[2] = 5 it is interpreted as follows: After indexing second entry of the ΔM table, index the fifth entry in the same table. The local start offset is equal to *start* mod k.

4.3.2 STRIP-MINED NODE CODE USING THE MEMORY GAP TABLE

Input: $\Delta M[i]$, $startmem$
Method:
$add \leftarrow start\_loc$
$blocklen \leftarrow k * \frac{s}{\text{gcd}\,(s,p*k)}$
$num\_block \leftarrow (last\_mem - start\_loc)\,/blocklen$
for $i = 0$ **to** $num\_block - 1$ **do**
  for $j = 0$ **to** $length - 1$ **do**
    $A[add] = \ldots$
    $add \leftarrow add + \Delta M[j]$
  enddo
enddo
$j \leftarrow 0$
while $(add \leq lastmem)$ **do**
  $A[add] = \ldots$
  $add \leftarrow add + \Delta M[j]$
  $j \leftarrow j + 1$
enddo
return

Figure 4.10: Strip-mined node code using the memory gap table

This node code will be referred to as the *strip-mining* node-code. The data structure used in this node code is the memory gap or ΔM table which stores the successive memory gaps. Knowing the length of the table and given the table itself, generating node code becomes an easy task. The pattern is known to repeat after $k * \frac{s}{gcd(s,pk)}$ courses. Also the local address of the last memory location accessed, *lastmem* if any, is known. With these two quantities the number of blocks on each processor can be calculated and is called *num_block* . For each *num_block* the length of the table has to be traversed and this is precisely what is illustrated by the nested *for* loop in the first part of the code. However, depending on the *lastmem* the last block need not be traversed for the entire length. So the last block has to be treated specially as shown by the *while* loop in the last part of the code. The advantages of enumerating the elements in the method described above are that no costly conditionals need to be included in the body of the main loop which may account for the better performance. Also it uses the well known ΔM table.

The second node combines the structures described in Section 4.2.1 and Section 4.2.3. These structures are the ΔM table and the *replicated table*. A detailed description of the node code follows. First of all, a replicated ΔM table is formed. The factor by which it is replicated is pre-computed and depends on certain factors. However to keep, the table comparatively small and to avoid possible cache effects the table is replicated 32/k times. Therefore if the size or the length of the ΔM table is 4 then the factor of replication is 32/4 = 8. After this replication, and with the availability of the new data structure the

Input: $start\_loc$, $replicated\_\Delta M$ table, $rfactor$, num_block
Method:

```
num_block ← num_block / rfactor
length ← length * rfactor
add ← start_loc
for i = 0 to num_block − 1 do
    for j = 0 to length − 1 do
        A[add] ← ...
        add ← add + replicated_ΔM[j]
    enddo
enddo
j ← 0
while (add ≤ lastmem) do
    A[add] ← ...
    add ← add + replicated_ΔM[j]
    j ← j + 1
enddo
return
```

Figure 4.11: Node code using the expanded memory gap table

node code proceeds in the same manner as described for the previous node code. So our main goal has been to reduce the number of blocks from *num_block* to *num_block/fac* and to increase the size of the ΔM table to *rfactor*length*. The advantages of this technique for code-execution involve checking for less conditionals. However the replication factor cannot be arbitrarily large and should be computed to achieve optimality.

4.3.3 STRIP MINED NODE CODE USING THE COMPRESSED MEMORY GAP TABLE

This section presents the node code using the *compressed table* data structure presented in the Section 4.2.2. There are two tables: the cgap_table and the count_table. The cgap_table contains the local memory gap value while the count_table consists of the number of times the address has to be incremented by this memory gap value. The node code consists of three nested loops. The first loop iterates through the number of blocks, *num_block*. The second loop iterates through the length of the cgap_table or the count_table both of which are the same. The third loop is for iterating through the number of times an address has to be incremented by the particular memory gap indexed or by *val*. The second part of the code takes into account the last block which may not be fully complete and may therefore have to be traversed for only a part of it. The advantages of this node code are, if the problem parameters permit compression to a great extent then the table lookups

Input: $cgap\_table$, $count\_table$, $start\_loc$
Method:

```
add ← start_loc
for b = 0 to num_block − 1 do
   for i = 0 to length − 1 do
      count_val ← count_table[i]
      val ← cgap_table[i]
      for j = 1 to count_val do
         add ← add + val
      enddo
   enddo
enddo
while (add ≤ lastmem) do
   A[add] ← ...
   count_val ← count_table[i]
   val ← cgap_table[i]
   for j = 1 to count_val do
      add ← ...
      if (add ≤ lastmem) then
         return
      endif
   enddo
enddo
return
```

Figure 4.12: Node code using compressed memory gap table

are considerably lower. This is expected to give considerable performance in execution. However the efficiency of this node code is strongly influenced by the problem parameters.

4.3.4 STRIP-MINED CODE USING THE EXPANDED COMPRESSED MEMORY GAP TABLE

This node code combines the compressed and the expanded form. The node code is the same as the node code presented for the *compressed* table except for the changes mentioned below. This node code incorporates the *replicated_count_table* and the *replicated_cgap_table*. The tables are the same as the unreplicated *pass* or *count* table presented above except for being replicated by a suitable factor. Therefore the new length of the replicated tables is original length times that factor. So this node code involves the same three nested loops but the length of the loops differ. The number of blocks, *num_block* decreases by the expansion factor while the length of the table increases by the factor

Input: $replicated\_cgap\_table$, $replicated\_count\_table$

Method:

```
num_block ← num_block / rfactor
length ← length * rfactor
add ← start_loc
for b = 0 to num_block − 1 do
    for j = 0 to length − 1 do
        count_val ← replicated_count_table[i]
        val ← replicated_cgap_table[i]
        for j = 1 to count_val do
            add ← add + val
            A[add] ← ...
        enddo
    enddo
enddo
while (add ≤ lastmem) do
    A[add] ← ...
    count_val ← replicated_count_table[i]
    val ← replicated_cgap_table[i]
    for j = 1 to count_val do
        add ← add + val
        if (add > lastmem) then
            return
        endif
        A[add] ← ...
    enddo
enddo
return
```

Figure 4.13: Strip mined code using the expanded and then-compressed memory gap table

Input: ΔM, $reset$, $start\_loc$
Method:
$add \leftarrow start\_loc$
$i \leftarrow 0$
while $(add \leq lastmem)$ **do**
  $A[add] \leftarrow \ldots$
  $add \leftarrow add + \Delta M[i]$
  $i \leftarrow reset[i]$
enddo
return

Figure 4.14: Node code using the memory gap and the reset table

times. So the outer two loops differ in their lengths. However it is obvious that their product still remains the same. The advantages of using this node code are that it utilizes the benefits of both expansion and compression and is therefore expected to provide for better execution times.

4.3.5 NODE CODE USING THE MEMORY GAP AND RESET TABLE

The RESET is the data structure used by the node code presented in Figure 4.14. The concept of the RESET table was introduced in an earlier section. The simple node code for the RESET table involves two table lookups. The first table lookup involves the lookup into the ΔM table and the second table lookup is into the RESET table. The memory gaps stored in consecutive locations of the ΔM table are added to the starting location till the end of the ΔM table is reached. The indexing into the ΔM table is done by the RESET table. When the end of the *reset* table is reached it enables the index to be reset to the start of the ΔM table, hence the name. The advantages of this node code are the code is compact and simple. Also the tables are of size $\lceil k/\text{gcd}\,(s, pk)\rceil$. This is a considerably space efficient compared to the method employing *Nextoffset* table [15] where the sizes of the table are always of size *k*. It may be observed that the construction of the RESET is much simpler than the *Nextoffset* table in [15].

4.3.6 NODE CODE USING THE MEMORY GAP, RESET AND THE EXPANDED TABLE

This subsection deals with node code generation using the ΔM table, the RESET table and the expansion concept. The node code is similar to the node code presented in the previous subsection. The *replicated_reset* table is not exactly a replicated reset table. The length replicated_reset table is *rfactor* times the length of the original reset table. The *replicated_reset* table is of the form replicated_reset[i] = i + 1 and replicated_reset[rfactor * length - 1] = 0. This node code inherits the benefits of replication mentioned earlier and

Table 4.1: Execution times (in μsec) for all node codes on an UltraSparc for the case $p = 32$, number of iterations per processor $= 4096$, and strides = $3, 5, 7$.

LEGEND: riacs: ΔM table; **rice**: NextOffset and ΔM table; **lsu-a**: Strip-mined code; **lsu-b**: Strip-mined code with expansion; **lsu-c**: Compressed ΔM table; **lsu-d**: Compressed ΔM table with expansion; **lsu-e**: Reset with ΔM tables; **lsu-f**: Reset with ΔM tables with expansion.

s	k	**riacs**	**rice**	**lsu-a**	**lsu-b**	**lsu-c**	**lsu-d**	**lsu-e**	**lsu-f**
3	4	208	147	128	80	205	150	140	142
3	8	202	142	101	80	187	166	143	140
3	16	199	141	83	80	187	171	147	147
3	32	197	140	78	80	118	118	147	147
3	64	197	147	72	77	83	83	140	143
3	128	196	147	72	75	66	66	144	142
3	256	196	142	74	74	57	57	147	141
3	512	196	147	74	70	52	53	147	142
5	4	208	147	128	80	202	148	142	142
5	8	202	140	101	80	181	181	147	147
5	16	199	147	84	80	184	176	147	142
5	32	191	144	80	80	155	158	141	147
5	64	190	147	77	77	103	103	142	147
5	128	196	142	75	71	76	119	147	145
5	256	196	147	74	70	62	62	147	140
5	512	196	147	69	74	56	56	140	140
7	4	208	140	125	77	208	155	147	147
7	8	196	147	101	80	178	155	141	147
7	16	194	142	87	80	162	156	147	142
7	32	192	141	80	80	197	191	147	147
7	64	190	147	77	77	123	123	147	147
7	128	196	142	75	75	81	81	147	147
7	256	196	147	74	74	67	67	142	147
7	512	190	147	74	70	58	58	147	140

Table 4.2: Execution times (in μsec) for all node codes on an UltraSparc for the case $p = 32$, number of iterations per processor = 4096, and strides = $9, 15, 99$.

LEGEND: riacs: ΔM table; **rice**: NextOffset and ΔM table; **lsu-a**: Strip-mined code; **lsu-b**: Strip-mined code with expansion; **lsu-c**: Compressed ΔM table; **lsu-d**: Compressed ΔM table with expansion; **lsu-e**: Reset with ΔM tables; **lsu-f**: Reset with ΔM tables with expansion.

s	k	**riacs**	**rice**	**lsu-a**	**lsu-b**	**lsu-c**	**lsu-d**	**lsu-e**	**lsu-f**
9	4	208	147	128	80	208	155	140	144
9	8	196	147	101	80	178	149	147	143
9	16	192	142	87	80	158	155	143	147
9	32	197	147	75	79	197	198	142	141
9	64	197	147	74	74	143	139	147	140
9	128	196	147	75	71	100	101	147	147
9	256	196	142	74	74	72	72	142	147
9	512	196	141	74	74	61	58	145	147
15	4	208	147	123	80	214	209	147	140
15	8	195	147	101	80	147	132	147	140
15	16	196	147	87	80	145	136	147	147
15	32	197	147	80	80	155	157	142	140
15	64	190	147	77	77	203	203	147	147
15	128	193	147	75	72	126	126	147	143
15	256	196	140	74	74	87	82	147	147
15	512	196	141	74	74	68	68	147	147
99	4	202	146	128	80	202	155	143	140
99	8	196	140	101	80	180	163	145	145
99	16	199	140	87	78	179	171	147	147
99	32	197	147	76	80	117	112	147	142
99	64	192	141	77	72	152	152	143	147
99	128	190	147	75	75	181	187	147	147
99	256	190	143	74	74	152	159	147	140
99	512	196	147	74	69	173	167	143	147

Table 4.3: Execution times (in μsec) for all node codes on an UltraSparc for the case $p = 32$, number of iterations per processor $= 8192$, and strides = $3, 5, 7$.

LEGEND: riacs: ΔM table; **rice**: NextOffset and ΔM table; **lsu-a**: Strip-mined code; **lsu-b**: Strip-mined code with expansion; **lsu-c**: Compressed ΔM table; **lsu-d**: Compressed ΔM table with expansion; **lsu-e**: Reset with ΔM tables; **lsu-f**: Reset with ΔM tables with expansion.

s	k	**riacs**	**rice**	**lsu-a**	**lsu-b**	**lsu-c**	**lsu-d**	**lsu-e**	**lsu-f**
3	4	417	294	257	161	417	309	294	289
3	8	399	288	197	161	368	327	289	289
3	16	398	294	170	155	374	355	288	294
3	32	388	294	161	161	231	233	287	294
3	64	385	287	151	145	167	159	294	285
3	128	386	289	145	150	132	129	287	287
3	256	392	286	142	149	108	115	294	289
3	512	385	287	148	148	106	106	294	294
5	4	417	294	257	161	417	304	294	294
5	8	399	289	200	158	362	362	294	289
5	16	394	290	174	161	373	359	290	294
5	32	395	290	152	151	316	306	288	287
5	64	394	294	147	154	202	198	287	294
5	128	393	292	146	150	150	152	294	288
5	256	388	286	149	149	119	125	286	294
5	512	392	291	148	148	111	108	294	287
7	4	411	289	252	155	417	309	294	294
7	8	399	289	202	161	350	308	290	294
7	16	394	294	168	161	333	317	288	294
7	32	395	294	153	161	395	384	284	294
7	64	394	294	147	147	238	247	294	285
7	128	393	289	150	150	166	167	288	285
7	256	392	294	144	143	135	135	294	290
7	512	392	294	148	141	114	116	294	287

Table 4.4: Execution times (in μsec) for all node codes on an UltraSparc for the case $p = 32$, number of iterations per processor $= 8192$, and strides = $9, 15, 99$.

LEGEND: riacs: ΔM table; **rice**: NextOffset and ΔM table; **lsu-a**: Strip-mined code; **lsu-b**: Strip-mined code with expansion; **lsu-c**: Compressed ΔM table; **lsu-d**: Compressed ΔM table with expansion; **lsu-e**: Reset with ΔM tables; **lsu-f**: Reset with ΔM tables with expansion.

s	k	**riacs**	**rice**	**lsu-a**	**lsu-b**	**lsu-c**	**lsu-d**	**lsu-e**	**lsu-f**
9	4	417	293	257	155	414	304	289	294
9	8	399	289	202	161	350	309	292	289
9	16	398	288	174	161	319	304	289	288
9	32	395	285	161	161	397	397	294	294
9	64	394	286	148	154	285	279	292	285
9	128	393	285	142	150	192	192	294	285
9	256	392	294	142	149	145	145	288	294
9	512	386	287	148	143	117	121	294	289
15	4	417	292	251	161	429	418	294	294
15	8	399	294	200	161	294	274	291	289
15	16	398	288	169	161	291	274	288	288
15	32	387	294	153	161	314	307	294	286
15	64	390	287	147	154	399	399	290	290
15	128	385	294	145	142	252	252	293	291
15	256	392	288	144	149	175	175	294	288
15	512	392	294	148	148	132	136	289	287
99	4	413	288	252	157	411	309	290	288
99	8	398	294	195	155	368	324	294	287
99	16	398	286	174	155	358	347	286	294
99	32	395	283	160	151	235	225	287	283
99	64	394	287	149	154	304	301	287	287
99	128	393	294	145	150	367	375	286	287
99	256	386	294	142	149	311	318	287	294
99	512	392	294	148	148	346	341	294	294

Table 4.5: Execution times (in μsec) for all node codes on an UltraSparc for the case $p = 32$, number of iterations per processor = 16384, and strides = $3, 5, 7$.

LEGEND: riacs: ΔM table; **rice**: NextOffset and ΔM table; **lsu-a**: Strip-mined code; **lsu-b**: Strip-mined code with expansion; **lsu-c**: Compressed ΔM table; **lsu-d**: Compressed ΔM table with expansion; **lsu-e**: Reset with ΔM tables; **lsu-f**: Reset with ΔM tables with expansion.

s	k	**riacs**	**rice**	**lsu-a**	**lsu-b**	**lsu-c**	**lsu-d**	**lsu-e**	**lsu-f**
3	4	1030	584	508	315	827	613	582	582
3	8	1103	584	398	319	741	658	588	581
3	16	1136	582	343	315	741	711	582	582
3	32	1072	589	318	317	467	467	583	583
3	64	1163	582	301	301	330	331	582	582
3	128	1170	587	299	301	265	263	586	586
3	256	1167	581	290	290	224	230	588	588
3	512	1176	588	296	291	213	209	583	583
5	4	1030	582	515	322	827	613	582	588
5	8	1103	581	397	315	723	716	588	581
5	16	1035	582	343	317	748	720	582	582
5	32	1155	583	317	320	632	631	583	583
5	64	1161	582	308	301	407	414	582	582
5	128	1170	586	301	301	305	305	586	588
5	256	1168	582	290	291	244	246	581	584
5	512	1169	582	291	289	223	219	582	583
7	4	1030	588	515	322	834	613	588	588
7	8	1103	587	402	317	708	612	581	584
7	16	1134	582	343	316	668	637	582	587
7	32	1000	584	322	318	788	787	584	584
7	64	1161	582	301	301	493	493	582	588
7	128	1172	588	301	301	345	345	588	588
7	256	1005	584	294	297	266	270	584	585
7	512	1170	582	291	292	228	233	582	583

Table 4.6: Execution times (in μsec) for all node codes on an UltraSparc for the case $p = 32$, number of iterations per processor = 16384, and strides = $9, 15, 99$.

LEGEND: riacs: ΔM table; **rice**: NextOffset and ΔM table; **lsu-a**: Strip-mined code; **lsu-b**: Strip-mined code with expansion; **lsu-c**: Compressed ΔM table; **lsu-d**: Compressed ΔM table with expansion; **lsu-e**: Reset with ΔM tables; **lsu-f**: Reset with ΔM tables with expansion.

s	k	**riacs**	**rice**	**lsu-a**	**lsu-b**	**lsu-c**	**lsu-d**	**lsu-e**	**lsu-f**
9	4	1030	582	508	322	834	619	582	582
9	8	1103	584	398	322	705	612	582	584
9	16	1134	582	346	320	643	619	582	582
9	32	1159	582	315	315	787	790	582	582
9	64	1165	582	301	301	569	568	582	585
9	128	1172	588	301	301	385	385	588	588
9	256	1170	585	296	294	288	287	585	586
9	512	1170	582	290	290	243	238	583	583
15	4	1030	582	508	319	1001	830	582	582
15	8	1012	587	400	322	583	550	586	584
15	16	1134	582	343	322	576	542	582	582
15	32	1156	582	322	317	626	625	582	582
15	64	1161	582	301	302	806	806	583	582
15	128	1051	588	301	301	504	504	588	588
15	256	1175	582	297	291	344	350	582	582
15	512	1170	583	296	290	268	269	583	583
99	4	1030	582	508	315	827	613	582	582
99	8	1103	582	398	322	736	656	583	582
99	16	1033	583	345	322	715	694	584	583
99	32	1152	582	316	318	469	469	582	582
99	64	1003	588	308	308	608	608	588	588
99	128	1172	588	301	301	750	750	588	588
99	256	1169	582	292	292	630	630	583	582
99	512	1171	583	292	291	687	686	588	583

Input: $rfactor$,$replicated\Delta M$, $replicated\_reset$
Method:
$add \leftarrow start\_loc$
$i \leftarrow 0$
while $(add \leq lastmem)$ **do**
  $A[add] \leftarrow \ldots$
  $add \leftarrow add + replicated\_\Delta M[i]$
  $i \leftarrow replicated\_reset[i]$
enddo
return

Figure 4.15: Node code using the memory gap, reset and the expanded table

from the RESET table. The node code description is exactly the same as the node code for the unreplicated RESET table mentioned in the previous subsection but with the modified lengths of the ΔM table and the replicated RESET tables.

4.4 IMPLEMENTATION, RESULTS AND DISCUSSION

This section discusses the implementation of the node code and the results that were obtained after performing the experiments. The experiments were performed on a Sun UltraSparc 1 using the `cc` compiler was used with the `-xO2` optimization turned on. The number of iterations executed by each processor was made to be the same by adjusting the local upper bound on each processor. The number of iterations considered were 4096, 8192, 16384.

We make the following observations regarding the performance of the several node-code execution strategies:

- For all block sizes, codes using RESET and *Nextoffset* tables performed similarly.
- For small block sizes, $k \leq 32$, the expansion of the tables helped significantly.
- The expansion factor was a function of the machine. We used a factor $\frac{32}{\text{length}}$.
- Strip-mined code with table expansion performed better than the RIACS and Rice methods for small and large values of k (about 20% better).
- The compressed memory gap tables performed extremely well for cases with large k (block-size) and small s (stride).
- The larger the value of
$$\max\left(\frac{|L_2|}{R_2}, \frac{R_2}{|L_2|}\right),$$
the better the performance of the compressed table technique.

- Compressed table technique was sometimes 2.5 times better than other methods.

4.5 CHAPTER SUMMARY

This chapter evaluates the existing data structures like the ΔM and notes its shortcomings. It has been stated that the particular choice of the data structure makes a lot of difference in the efficiency of the code [18]. The ΔM is a simple and a useful data structure. It represents the difference in the memory locations between two consecutive memory accesses. So given the starting element the subsequent memory accesses can all be determined. This was one of the earliest data structures presented in the research. Using this data structure the node code was generated. But it was observed that the node code for this type of data structure involved a lot of guards or conditionals as for each iteration of the loop it need to be tested whether the entire table had been traversed and the index reset to 0. To overcome this problem [16] proposed the *Nextoffset* table. This table though highly efficient proved to be expensive in terms of space tradeoff.

This chapter derived six new methods of generating new code which promise to be time and space efficient. The new methods described utilize the properties of the structure to represent data. Three new concepts have been introduced namely, the expanded, the compressed and the reset tables. The six node code methods provided employ a combination of these tables for best performance. This chapter presents six new data structures for the efficient enumeration of all the elements in the local memory space. The six new data structures are a combination of one another. The first technique of strip mining basically divides the entire local memory space into blocks which can then be traversed using nested **for** loops which prove to be highly efficient. The second data structure consists of the compressed gap table which consisted of compressing the ΔM table by representing repeating entries of gaps by a single quantity.The third method consists of the $reset$ table. the $reset$ table is the best data structure possible because it is an exact compromise between the space and time as noted by [18]. It attempts to reset the index to which the ΔM table to the starting of the table. This does not involve conditionals as the *Nextoffset* table and also need not be of length k as the *Nextoffset* table. Extensive results have demonstrated that all the six techniques used for the node code execution perform extremely well and provide the user and the compiler writer with a varied and large amount of choice to choose his method of representation of the data. The compressed tables performed extremely well in several cases.

CHAPTER 5

COUPLED SUBSCRIPTS

This chapter address the problem of code generation for coupled subscripts. Two new table construction techniques for references to arrays with CYCLIC(k_1, k_2) distribution that contain coupled subscripts are proposed and evaluated. Techniques useful for multidimensional arrays with independent subscripts—such as the application of one-dimensional address generation algorithm for each dimension separately—do not apply in the case of coupled subscripts [1, 16, 23]. Fast table construction is essential for the efficient execution of programs.

5.1 PROBLEM STATEMENT AND RELATED WORK

Consider the loop with coupled subscripts shown below.

```
!HPF$   PROCESSORS PROCS(p1, p2)
!HPF$   DISTRIBUTE A(CYCLIC(k1),CYCLIC(k2)) ONTO PROCS
```

do $j = 0, n_1$

$$A(l_1 + js_1, l_2 + js_2) = \cdots$$

enddo

Here, array A is distributed onto $p_1 \times p_2$ processors using CYCLIC(k_1) distribution along processor dimension one, and using CYCLIC(k_2) distribution along processor dimension two. The access strides for the loops are s_1 and s_2 respectively. As the above example demonstrates, the subscript expressions $l_1 + js_1$ and $l_2 + js_2$ both contain the loop index j; we say that these two subscripts expressions are coupled by the loop index j. Let us compute the set of iterations for each dimension independently. Then, the memory addresses referenced on a processor correspond to iterations common to both the dimensions. This cannot be deduced simply by two applications of the algorithms in [4, 15, 34]; we need to expand the tables up to the least common multiple of the periods (in terms of iterations) of the two dimensions. This forms the basis of our table generation techniques.

Kennedy et al. [16] solve this problem by computing the starting location on each processor; then, the general solutions for the iteration on each dimension is computed by solving the set of linear diophantine equations. For an array with CYCLIC(k_1, k_2) distribution accessed with strides s_1 and s_2, the method requires solving $k_1 \times k_2$ equations, solutions to which give common iterations. The common iterations are not in increasing order in general. Thus, one needs to use a sorting algorithm; this results in an overall complexity of $O(k_1 k_2 \log(k_1 k_2))$. While the theoretical complexity of the algorithm is low, implementations require expensive integer arithmetic. The table of iterations that is obtained is used to compute the set of global addresses referenced on the processor followed by global-to-local address conversion. The sequence of addresses is to used in the SPMD node code which uses either one two-dimensional table or two one-dimensional tables.

The solutions—namely, *search* and *hashing*—presented in this chapter attempt to minimize the table generation overhead; there is no need to solve for $k_1 \times k_2$ equations and the costly sorting step is avoided. Both these are practically efficient methods to enumerate the common iterations. In addition, our methods do not incur global to local conversion in the address computation.

5.2 SEARCH METHOD FOR COMMON ITERATION COMPUTATION

The algorithm for computing common iterations using the *search* method is shown in Figure 5.1. The first phase of the algorithm is the computation of the ΔM_1, ΔM_2, $\Delta iter_1$, $\Delta iter_2$ tables. The ΔM_1 and ΔM_2 are the same tables constructed using algorithms in [15] and [34]. The iteration gap tables are two additional tables that are constructed. These tables are the iteration gaps between two successive address references on the same processor. There construction is simple and needs minor modification of the code when constructing the ΔM_1 and ΔM_2 tables. In fact similar to the $\vec{r} = (r_1, r_2)$ and $\vec{\ell} = (\ell_1, \ell_2)$ vectors that are formed for the memory gaps, three values may be formed for the iteration gaps; we will call these the *right_iteration_gap, left_iteration_gap* and the *right_left_iteration_gap*. The modifications that are needed in the code that computes the ΔM_1 table are as follows:

- right_iteration_gap $= \frac{r_1 * pk + r_2}{s}$
- left_iteration_gap $= \frac{l_1 * pk + l_2}{s}$
- right_left_iteration_gap $=$ right_iteration_gap $+$ left_iteration_gap

Recall that these gaps are simply the global basis vectors divided by the stride. Phase 2 (Lines 3 and 4) of the algorithm computes the replication length of the table. This length defines the iteration number after which the iteration gap start repeating and therefore defines the point after which the local access pattern starts repeating. Since in the one dimensional case the iterations repeat after $\frac{p_1 k_1}{d_1}$ iterations it explains the expression stated in the algorithm. Phase 3 of the algorithm uses this replication length to find the replication factor of each table. This table provides the factor by which each of the tables has to be replicated. So the new lengths are $rlen_1$ and $rlen_2$. Lines 5–14 accomplish replication of the tables. Next, the algorithm chooses the smaller of the two tables and scans its elements one by one. Each iteration of the smaller table is compared with entries in the bigger table; if the two iterations match then the value of the iteration is stored in a new table called the $repcomits$ table. This table will then be used for generation of SPMD code and its subsequent execution. The search method employed in the above algorithm is a simple search in which the sorted sequence of the table is taken advantage of. The point of last

Input: Compute $\Delta M_1, \Delta M_2, \Delta iter_1, \Delta iter_2, length_1, length_2$
Output: $reppat$ table (consisting of the set of all local addresses), and its length len
Method:

```
d_1 ← gcd(s_1, p_1 k_1)
d_2 ← gcd(s_2, p_2 k_2)
rep ← lcm(p_1 k_1 / d_1, p_2 k_2 / d_2)
rlen_1 ← length_1 × (rep*d_1)/(p_1 k_1);  rlen_2 ← length_2 × (rep*d_2)/(p_2 k_2)
Compute start_it_1 and start_it_2 on processor
its_rep_1 ← start_it_1;  its_rep_2 ← start_it_2;  i ← 0;
while (i < rlen_1) do
   for k ← 0 to length_1 − 1 do
      repits_1[i] ← Δiter_1[k] + its_rep_1
   enddo
   its_rep_1 ← its_rep_1 + (p_1 k_1)/d_1
   i ← i + length_1
enddo
Compute the replicated table repits_2 in the same manner as above
Choose the smaller of the table. Assume repits_1 is smaller. If repits_2 is smaller,
  the code is symmetric
j ← 0;  len ← 0
for i ← 0 to rlen_1 − 1 do
   ptr ← j;  temp ← repits_1[i]
   for j ← ptr to rlen_2 − 1 do
      if (temp ≤ repits_2[j]) then
         if (temp = repits_2[j]) then
            repcomits[len] ← temp
            len ← len + 1
            Continue with next iteration i
         endif
      endif
   enddo
enddo
```

Figure 5.1: Algorithm to compute the common iterations table using the searching technique

search in the bigger of the two tables is saved and it is from this point that the search is continued when iterating the next time over the smaller table. A lot of other efficient searching methods can be applied to this algorithm but the basic idea remains the same. The use of varied search methods and their combination thereof for table construction for another related problem of MIV can be viewed in the previous section on MIV in this chapter and [38]. The same search methods can be used here also. The two tables which are computed are analogous in the sense one consists of the iterations while the other consists of the memory references. The advantages of using such a method for table construction is that there is no need to solve k_1k_2 diophantine equations and sorting is avoided. The space complexity is $O(min(rlen_1, rlen_2))$; theoretically the search method is inferior to the Rice method, it performs extremely well in practice.

5.3 HASHING METHOD FOR COMMON ITERATION ENUMERATION

The hashing algorithm is shown in Figure 5.2; this is a simple and effective algorithm. Lines 1–6 of the algorithm are the same as the first six lines of the algorithm for searching. Line 7 in the algorithm creates and initializes a hashtable of size rep. This is the size of the table before the pattern starts repeating. Lines 8–17 show that the iteration values in the first dimension are read one by one from the iteration gap table. The iteration values are used to hash into the hashtable. So the temporary array consists of a series of zeros and ones as its entries. Next the iterations in the second dimension are considered. The iterations are again used to index into the temporary array. If at the indices in the temporary array the value is one then the index value is the common iteration in the two dimension and is stored in the $repcomits$ table. The algorithm returns the repcomits table which consists of all the common iteration in the two dimensions. The algorithm also returns the $reppat$ table which consists of the set of local addresses accessed on that processor. The length of these table are also returned. The method is called hashing because of the fact that the iteration values are the values which are used to index into the temporary array. Any hashing function could be used but this is the simplest function that can be used.

The advantage of this code is its simplicity. The code in comparison with the *search* code involves less conditionals and hence should be more time-effective. This fact is borne out from the experimental results that are presented at the end of the chapter. The space complexity is $O(lcm(\frac{p_1k_1}{d_1}, \frac{p_2k_2}{d_2}))$. This section describes the two new methods of table construction. This table is used during the SPMD node code execution.

5.4 EXECUTION STRATEGIES

The table of common iterations obtained from the above table construction mechanisms can be used to produce efficient SPMD node code for execution on each processor. The SPMD node code provided in this section will use extensively the ideas presented in

Input: Compute ΔM_1, ΔM_2, $\Delta iter_1$, $\Delta iter_2$, $length1, length2$
Output: $reppat$ table (consisting of the set of all 2-D local addresses), and its length, len
Method:

$d_1 \leftarrow$ gcd (s_1, p_1k_1)
$d_2 \leftarrow$ gcd (s_2, p_2k_2)
$rep \leftarrow$ lcm $(\frac{p_1k_1}{d_1}, \frac{p_2k_2}{d_2})$
$rlen_1 \leftarrow length_1 \times \frac{rep*d_1}{p_1k_1}$; $rlen_2 \leftarrow length_2 \times \frac{rep*d_2}{p_2k_2}$
Compute $start\_it_1$ and $start\_it_2$ on processor
$its\_rep_1 \leftarrow start\_it_1$; $its\_rep_2 \leftarrow start\_it_2$
Initialize hashtable[] to 0; $i \leftarrow 0$
while $(i < rlen_1)$ **do**
  $j \leftarrow i$
  for $k \leftarrow 0$ **to** $length_1 - 1$ **do**
    $temp \leftarrow \Delta iter_1[k] + its\_rep_1$
    hashtable$[temp] \leftarrow 1$
    $j \leftarrow j + 1$
  enddo
  $its\_rep_1 \leftarrow its\_rep_1 + \frac{p_1k_1}{d_1}$
  $i \leftarrow i + length_1$
enddo
Compute the replicated table $repits_2$ in the same manner as above
$i \leftarrow 0$; $len \leftarrow 0$
while $(i < rlen_2)$ **do**
  for $k \leftarrow 0$ **to** $length_2 - 1$ **do**
    temp $\leftarrow its\_rep_2 + \Delta iter_2[k]$
    if (hashtable[temp] $=$ 1) **then**
      repcomits[len] $\leftarrow$ temp
      len $\leftarrow$ len $+1$
    endif
    $its\_rep_2 \leftarrow its\_rep_2 + \frac{p_2k_2}{d_2}$
    $i \leftarrow i + length_2$
  enddo
enddo

Figure 5.2: Algorithm to compute the common iterations table using the hashing technique

an earlier chapter [5]. The two methods discussed here are the ones using conditional statements, the next one uses the concept of a *reset* table. Both these as well as the technique from [16] are described next.

The SPMD node code using the *Nextoffset* table (Rice) is as follows.
Input: $\Delta_2 M$

Compute $start_1, start_2, end_1, end_2$
Compute a two-dimensional *Nextoffset* table
$a_1 \longleftarrow start_1$
$a_2 \longleftarrow start_2$
$o_1 \longleftarrow start_1$ mod k_1
$o_2 \longleftarrow start_2$ mod k_2
while $(a_1 \leq end_1) and (a_2 \leq end_2)$ **do**
    $array[a_1, a_2] \longleftarrow \cdots$
    $(a_1, a_2) \longleftarrow (a_1, a_2) + \Delta_2 M[o_1, o_2]$
    $(a_1, a_2) \longleftarrow$ *Nextoffset*$[o_1, o_2]$
enddo

The SPMD node code using conditionals is as follows.
Input: ΔM

len $\longleftarrow length(\Delta M)$
Compute $start_1, start_2, end_1, end_2$
$a_1 \longleftarrow start_1$
$a_2 \longleftarrow start_2$
$k \longleftarrow 0$
while $(a_1 \leq end_1) and (a_2 \leq end_2)$ **do**
    $array[a_1, a_2] \longleftarrow \cdots$
    $(a_1, a_2) \longleftarrow (a_1, a_2) + \Delta M[k]$
    $k \longleftarrow k + 1$
    if $(k = len)$ **then** $k \longleftarrow 0$
    endif
enddo

The SPMD node code using the RESET technique is as follows.
Input: ΔM

len $\longleftarrow length(\Delta M)$
for $i \longleftarrow 0$ **to** $len - 2$ **do**
    reset$[i] \longleftarrow i + 1$
enddo
reset$[len - 1] \longleftarrow 0$
Compute $start_1, start_2, end_1, end_2$
$a_1 \longleftarrow start_1$

$a_2 \leftarrow start_2$
$k \leftarrow 0$
while $(a_1 \leq end_1) and (a_2 \leq end_2)$ **do**
  $array[a_1, a_2] \leftarrow \cdots$
  $(a_1, a_2) \leftarrow (a_1, a_2) + \Delta M[k]$
  $k \leftarrow \mathit{reset}[k]$
enddo

5.5 EXPERIMENTAL RESULTS

All experiments were done on a Sun UltraSparc. For table generation, we used a fixed processor geometry of 8×4. Tables 5.1–5.3 show table generation times for the Rice method and the two methods presented in this chapter, for several values of k_1 and k_2 for three different combinations of strides. It is clear that for small values of k_1 and k_2, the three methods are comparable. But, as the block sizes increase, the hash and search methods significantly outperform the Rice method by as much as a factor of 100.

Tables 5.4–5.6 show the execution times using the *Nextoffset* method (called **Rice** in the tables) [16], the method using increment and test (called **MOD** in the tables), and the RESET method. As observed by earlier experiments in [5, 16, 18], the Rice method is usually better than the conditional method; but, the RESET table outperforms the other methods consistently, sometimes by as much as a factor of two in execution time. In addition, it is noticeable that Rice method of address generation uses up to 25% of the execution overhead for 10,000 iterations.

5.6 CLOSED FORMS FOR SPECIAL CASES

The above sections presented the execution strategies for coupled subscripts. While dealing with the strategies it may have been clearly realized that a lot of special cases existed for more efficient execution. This means that for some special cases the particular characteristic of the problem enabled the enumeration of elements much more efficiently than the enumeration using the algorithm. These special cases are the one to be discussed here. The special cases will be mentioned and then the techniques for their enumeration will be listed. The various special cases follows

5.6.1 (BLOCK, BLOCK) DISTRIBUTION

This special case mentioned here considers the case when the distribution of the array is (BLOCK, BLOCK) on processor (m_1, m_2). This means that in both the dimensions of distribution the distribution is (BLOCK, BLOCK). Let the number of elements in the first dimension be equal to N_1 and in the second dimension be equal to N_2. Then the

Table 5.1: Table construction times (in μsec). Processor geometry $= 8 \times 4$; $s_1 = 1$; $s_2 = 1$

k_1	k_2	**Rice**	**Search**	**Hashing**
4	4	46	28	26
4	8	55	29	27
8	4	59	30	28
8	8	98	31	29
8	16	135	33	31
8	32	210	39	37
8	64	354	52	48
8	128	701	78	69
8	256	1426	131	111
16	4	84	35	32
16	8	136	37	34
16	16	266	38	35
32	4	141	45	40
32	8	211	45	40
32	32	800	52	46
64	4	275	78	53
64	8	415	62	53
64	64	2767	78	68
128	4	536	100	82
128	8	823	102	96
128	128	10014	132	110
256	4	1073	181	162
256	8	1671	174	159
256	256	38412	241	234

Table 5.2: Table construction times (in μsec). Processor geometry $= 8 \times 4$; $s_1 = 5$; $s_2 = 3$

k_1	k_2	**Rice**	**Search**	**Hashing**
4	4	36	33	32
4	8	44	36	35
8	4	52	37	36
8	8	70	39	37
8	16	104	39	37
8	32	191	47	44
8	64	373	59	54
8	128	742	87	81
8	256	1485	140	126
16	4	83	43	40
16	8	124	46	43
16	16	193	46	43
32	4	154	48	43
32	8	224	52	46
32	32	660	58	53
64	4	300	73	62
64	8	443	72	67
64	64	2553	87	75
128	4	585	112	94
128	8	877	109	89
128	128	9564	139	120
256	4	1235	198	177
256	8	1809	194	173
256	256	38671	241	245

Table 5.3: Table construction times (in μsec). Processor geometry $= 8 \times 4$; $s_1 = 2$; $s_2 = 7$

k_1	k_2	**Rice**	**Search**	**Hashing**
4	4	21	31	29
4	8	36	34	34
8	4	35	32	31
8	8	44	35	34
8	16	68	35	33
8	32	113	45	42
8	64	219	59	53
8	128	431	80	70
8	256	873	147	126
16	4	51	34	33
16	8	70	38	36
16	16	104	35	34
32	4	78	39	38
32	8	118	53	49
32	32	339	47	44
64	4	143	48	44
64	8	216	52	47
64	64	1246	65	61
128	4	274	67	59
128	8	423	69	60
128	128	4763	92	82
256	4	548	109	87
256	8	833	107	89
256	256	19337	172	157

Table 5.4: Execution overhead (in μsec) for 10,000 iterations per processor; Processor geometry $= 8 \times 4$; $s_1 = 1; s_2 = 1$

k_1	k_2	**Rice**	**Reset**	**MOD**
4	4	658	479	478
4	8	687	507	507
8	4	708	507	508
8	8	771	588	558
8	16	891	702	671
8	32	1103	925	895
8	64	1557	1382	1348
8	128	2498	2335	2285
8	256	4378	4252	4208
16	4	761	564	565
16	8	876	701	669
16	16	1161	978	931
32	4	1605	701	671
32	8	1105	927	897
32	32	2579	2429	2366
64	4	1879	978	929
64	8	2320	1425	1385
64	64	8118	8070	7944
128	4	2425	1942	1479
128	8	3333	2805	2383
128	128	31718	30365	30157
256	4	3899	2653	2595
256	8	5381	4483	4399
256	256	119990	118839	119108

Table 5.5: Execution overhead (in μsec) for 10,000 iterations per processor; Processor geometry = 8×4; $s_1 = 5; s_2 = 3$

k_1	k_2	**Rice**	**Reset**	**MOD**
4	4	650	469	569
4	8	678	497	597
8	4	685	504	598
8	8	742	562	653
8	16	869	666	765
8	32	1090	905	909
8	64	1565	1390	1353
8	128	2547	2345	2293
8	256	4503	4282	4240
16	4	762	571	571
16	8	865	686	695
16	16	1097	909	916
32	4	1608	707	677
32	8	1115	931	898
32	32	2475	2273	2282
64	4	1890	986	946
64	8	2356	1444	1402
64	64	7953	7809	7756
128	4	2460	1964	1496
128	8	3351	2879	2410
128	128	30311	29764	29670
256	4	6322	2683	2630
256	8	7968	4535	4576
256	256	122394	119238	121116

Table 5.6: Execution overhead (in μsec) for 10,000 iterations per processor; Processor geometry $= 8 \times 4$; $s_1 = 2$; $s_2 = 7$

k_1	k_2	**Rice**	**Reset**	**MOD**
4	4	638	457	397
4	8	678	486	587
8	4	666	484	584
8	8	715	528	627
8	16	819	621	717
8	32	992	810	804
8	64	1371	1181	1144
8	128	2108	1958	1916
8	256	3667	3477	3382
16	4	1160	540	630
16	8	806	622	718
16	16	971	790	893
32	4	1041	630	630
32	8	1178	812	818
32	32	2037	1851	1898
64	4	1748	832	805
64	8	1666	1185	1158
64	64	6324	6048	6141
128	4	2166	1241	1213
128	8	2879	1943	1899
128	128	23144	23655	23576
256	4	2979	2184	2014
256	8	4743	3593	3496
256	256	93276	92114	91950

inequalities that can be derived from this data is the following:

$$\frac{N_1 m_1}{p_1} \le l_1 + i s_1 \quad \le \quad \frac{N_1(m_1+1)}{p_1} - 1 \tag{5.1}$$
$$\frac{N_2 m_2}{p_2} \le l_2 + i s_2 \quad \le \quad \frac{N_2(m_2+1)}{p_2} - 1. \tag{5.2}$$

These two equations are the basis for the derivation of the equations for the minimum and the maximum iterations on the processor. Once the maximum and the minimum iterations on that particular processor known it is very simple to enumerate all the iterations as they follow a pattern. To find the common iterations on each processor the above two equations will be considered. The values of i_1 and i_2 from the above two equations is as follows:

$$\left\lceil \frac{\frac{N_1 m_1}{p_1} - l_1}{s_1} \right\rceil \le \quad i \quad \le \left\lfloor \frac{N_1(m_1+1)}{p_1} - 1 \right\rfloor \tag{5.3}$$
$$\left\lceil \frac{\frac{N_2 m_2}{p_2} - l_2}{s_2} \right\rceil \le \quad i \quad \le \left\lfloor \frac{N_2(m_2+1)}{p_2} - 1 \right\rfloor. \tag{5.4}$$

These equation beginning from the first can be easily explained. When the arrays are block distributed on both the dimensions then the first element on processor m_1 lies between $\frac{N_1 m_1}{p_1}$ and $\frac{N_1(m_1+1)}{p_1} - 1$. This is expected because each block on the processor is made up of $\frac{N_1}{p_1}$ elements. The same analogous rule applies to the second dimension also. The next two equations just reiterate the previous two equations and have been put in a slightly modified form to illustrate the importance of the fist two equations. From the four inequalities presented above the values for the maximum and the minimum iterations can be found out.

$$i_{min} \quad = \quad \left\lceil \text{max}\left(\frac{N_1 m_1 - l_1 p_1}{p_1 s_1}, \frac{N_2 m_2 - l_2 p_2}{p_2 s_2} \right) \right\rceil \tag{5.5}$$
$$i_{max} \quad = \quad \left\lfloor \text{min}\left(\frac{N_1 m_1 + N_1 - p_1 - p_1 l_1}{p_1 s_1}, \frac{N_2 m_2 + n_2 - p_2 - p_2 l_2}{p_2 s_2} \right) \right\rfloor. \tag{5.6}$$

As the equations show that the i_max and the i_min have been found. These can then be used to enumerate the array elements in a more efficient form then the algorithmic approach for this particular case. Since closed forms have been formed for this special case the compiler can just plug in the appropriate values at run time for the unknowns like s_1 and s_2 to compute the values. This section therefore considers one of the special cases for enumerating the accessed elements in the case of coupled subscripts. The local addresses it may be mentioned is the following for a (BLOCK, BLOCK) distribution: $(l_1 + j s_1 - \frac{N_1 m_1}{p_1}, l_2 + j s_2 - \frac{N_2 m_2}{p_2})$ and j lies between the range $i_{min} \le j \le i_{max}$.

5.6.2 (CYCLIC, CYCLIC) DISTRIBUTION

This section deals with the second special case. This is the case when the array is distributed cyclically in the two dimensions. The cyclic distribution in the two dimensions may be represented by two useful equations given below.

$$\begin{aligned} l_1 + is_1 &= c_1p_1 + m_1 & 0 \le m_1 \le p_1 - 1 \quad (5.7)\\ l_2 + is_2 &= c_2p_2 + m_2 & 0 \le m_2 \le p_2 - 1 \quad (5.8)\\ is_1 - c_1p_1 &= m_1 - l_1 = \lambda_1 & (5.9)\\ is_2 - c_2p_2 &= m_2 - l_2 = \lambda_2. & (5.10) \end{aligned}$$

The left hand side values are the general referencing equations used everywhere while the right hand side values are particular to a cyclic distribution. The next two equations are just the first two equations stated in a different manner. It may be noted that $m_1 - l_1$ and $m_1 - l_2$ are constants. Therefore the variables are i, c_1 and c_2. The solution has to be found for therefore for equation in two variables and is found in the same manner as in [4]. The same solution will be obtained if the right hand side is substituted with the gcd (s_1, p_1) and gcd (s_2, p_2) values. This is exactly what is done in the next two equations which are shown here for convenience.

$$\begin{aligned} \alpha_1 s_1 - \beta_1 p_1 &= \gcd(s_1, p_1) = d_1 & (5.11)\\ \alpha_2 s_2 - \beta_2 p_2 &= \gcd(s_2, p_2) = d_2 & (5.12) \end{aligned}$$

In matrix notation the same can be expressed as:

$$\begin{aligned} \begin{pmatrix} i \\ c_1 \end{pmatrix} &= \begin{pmatrix} \frac{\lambda_1\alpha_1}{d_1} \\ \frac{\lambda_1\alpha_1}{d_2} \end{pmatrix} + t_1 \begin{pmatrix} \frac{p_1}{d_1} \\ \frac{s_1}{d_1} \end{pmatrix} & (5.13)\\ \begin{pmatrix} i \\ c_2 \end{pmatrix} &= \begin{pmatrix} \frac{\lambda_2\alpha_2}{d_2} \\ \frac{\lambda_2\alpha_2}{d_2} \end{pmatrix} + t_2 \begin{pmatrix} \frac{p_2}{d_2} \\ \frac{s_2}{d_2} \end{pmatrix}. & (5.14) \end{aligned}$$

The two equations may be combined by the elimination of the variable i. The consolidated equation that results because of the elimination is given below.

$$\begin{aligned} \frac{\lambda_1\alpha_1}{d_1} + \frac{t_1p_1}{d_1} &= \frac{\lambda_2\alpha_2}{d_2} + \frac{t_2p_2}{d_2} & (5.15)\\ t_1p_1d_2 - t_2p_2d_1 &= \lambda_2\alpha_2d_1 - \lambda_1\alpha_1d_2. & (5.16) \end{aligned}$$

The equation has solution if there is a solution for the equation with the right hand side substituted by gcd (p_1d_2, p_2d_1). Rewriting p_1d_2 as α, p_2d_1 as β and $\lambda_2\alpha_2d_1 - \lambda_1\alpha_1d_2$ as λ, the resulting equation is

$$t_1\alpha - t_2\beta = d \quad (5.17)$$

$$\begin{pmatrix} t_1 \\ t_2 \end{pmatrix} = \begin{pmatrix} \frac{\lambda\alpha}{d} \\ \frac{\lambda\beta}{d} \end{pmatrix} + t \begin{pmatrix} \frac{p_2 d_1}{d} \\ \frac{p_1 d_2}{d} \end{pmatrix}. \quad (5.18)$$

Applying the conditions that the value of the variable i has to be greater than 0 and less than n the minimum and the maximum values of t_1 and t_2 can be evaluated. The corresponding values of t_1 and t_2 are shown below.

$$t_1{}^{min} = \left\lceil \frac{-\lambda_1 \alpha_1}{p_1} \right\rceil \quad (5.19)$$

$$t_1{}^{max} = \left\lfloor \frac{n d_1 - \lambda_1 \alpha_1}{p_1} \right\rfloor \quad (5.20)$$

$$t_2{}^{min} = \left\lceil \frac{-\lambda_2 \alpha_2}{p_2} \right\rceil \quad (5.21)$$

$$t_2{}^{max} = \left\lfloor \frac{n d_2 - \lambda_2 \alpha_2}{p_2} \right\rfloor . \quad (5.22)$$

The equivalent expressions for t^{min} and t^{max} are given below.

$$t^{min} = \left\lceil \max\left(\frac{t_1{}^{min} - \lambda\alpha}{p_2 d_1}, \frac{d(t_2{}^{min} - \lambda\alpha)}{p_1 d_2} \right) \right\rceil \quad (5.23)$$

$$t^{max} = \left\lfloor \max\left(\frac{t_1{}^{max} - \lambda\alpha}{\beta_2 d_1}, \frac{d(t_2{}^{max} - \lambda\alpha)}{\beta_1 d_2} \right) \right\rfloor . \quad (5.24)$$

The same set of equations can be used to derive the values for c_1 and c_2; we omit the details of the derivation.

$$(c_1, c_2) = \left(\frac{\lambda_1 \beta_1}{d_1} + \frac{\lambda\alpha s_1}{d d_1} + \frac{t p_2 s_1}{d} t^{min} : \frac{\lambda_1 \beta_1}{d_1} + \frac{\lambda\alpha s_1}{d d_1} + \frac{t p_2 s_1}{d} t^{max} : \frac{p_2 d_1}{d} \right) \quad (5.25)$$

This completes the discussion on the special case for cyclic distribution in both the dimensions. Solutions to equations in two variables were found and therefore closed forms for this special case were determined.

5.6.3 (BLOCK, CYCLIC) DISTRIBUTION

The next special case to be considered is when the array is distributed in a BLOCK fashion in the first dimension and in the CYCLIC fashion in the second dimension. The equations to begin with are the following equations.

$$\frac{m_1 N_1}{p_1} \leq l_1 + i s_1 \leq \frac{(m_1 + 1) N_1}{p_1} - 1 \quad (5.26)$$

$$l_2 + i s_2 = c_2 p_2 + m_2 \quad (5.27)$$

There exists a solution to this equation if there is a solution to the following equation.

$$\begin{aligned} \alpha_2 s_2 - \beta_2 p_2 &= \gcd(s_2, p_2) = d_2 && (5.28) \\ \begin{pmatrix} i \\ c_2 \end{pmatrix} &= \begin{pmatrix} \frac{\lambda_2 \alpha_2}{d_2} \\ \frac{\lambda_2 \beta_2}{d_2} \end{pmatrix} + t \begin{pmatrix} \frac{\lambda_2}{d_2} \\ \frac{s_2}{d_2} \end{pmatrix} && (5.29) \end{aligned}$$

The value of i ranges from 0 to n_2. The values of i_{max} and i_{min} can be determined from these ranges just mentioned above. The values of i_{max} and i_{min} derived are shown below.

$$\begin{aligned} i_{min} &= \max\left(0, \left\lceil \frac{m_1 N_1 - l_1 p_1}{p_1 s_1} \right\rceil\right) && (5.30) \\ i_{max} &= \min\left(n, \left\lfloor \frac{(m_1+1)N_1 - p_1(l_1+1)}{p_1 s_1} \right\rfloor\right) && (5.31) \end{aligned}$$

Having got the values for i_{min} and i_{max} the values for t_{min} and t_{max} can be derived. The values are shown below.

$$\begin{aligned} t_{min} &= \left\lceil \frac{d_2 i_{min} - \lambda_2 \alpha_2}{p_2} \right\rceil && (5.32) \\ t_{max} &= \left\lfloor \frac{d_2 i_{max} - \lambda_2 \alpha_2}{p_2} \right\rfloor && (5.33) \end{aligned}$$

The closed form expression for i which can be derived from this is given as a regular section shown below

$$i = \left[\frac{\lambda_2 \alpha_2}{d_2} + t_{min} \frac{p_2}{d_2} : \frac{\lambda_2 \alpha_2}{d_2} + t_{max} \frac{p_2}{d_2} : \frac{p_2}{d_2} \right]. \qquad (5.34)$$

This section therefore successfully derives closed form expressions for the correct enumeration of iterations on processor m.

5.6.4 Other distributions

This section discusses the special case when the distribution is block fashion in the first dimension and not distributed, i.e., $*$ in the other. For such a type of distribution the expressions for i_{min} and i_{max} are derived as follows:

$$\begin{aligned} i_{min} &= \left\lceil \frac{\frac{m_1 N - 1}{p_1} - l_1}{s_1} \right\rceil && (5.35) \\ i_{max} &= \left\lfloor \frac{\frac{(m_1+1)N_1}{p_1} - 1}{s_1} \right\rfloor. && (5.36) \end{aligned}$$

It may be noted that similar derivations are needed if the distribution is of the form (*, BLOCK) or (CYCLIC, *) or (*, CYCLIC) and therefore will not be shown here.

5.7 CHAPTER SUMMARY

In this chapter, we have shown several techniques to generate accessed elements in an array, when the array subscripts contain coupled subscripts. Our techniques involve building tables for exploiting the pattern of accesses. In the case of coupled subscripts, we presented two methods of address generation which outperform an existing technique [16] by as much as a factor of 100 for large block sizes, while demonstrating comparable performance for small block sizes. We also show that the performance of the node code using the *reset* table introduced by us in [5] is consistently better than other methods. The two methods that were introduced were the hashing and the search method. The earlier methods were based on the more expensive sorting and therefore the improvement in performance. The results clearly demonstrate the efficacy of our approach. Work is in progress for communication generation for complex subscripts, and in address and communication generation for general affine subscripts in areas related to this work.

CHAPTER 6

EFFICIENT REDISTRIBUTION OF ARRAYS

Redistribution of arrays in distributed memory machines is used in many applications and therefore requires close attention. It refers to dynamically (i.e., at run-time) changing the distribution of arrays onto the processors. Languages such as HPF [11] require adequate compiler support to efficiently execute applications that require redistribution, which is an executable statement. It is the job of the compiler to perform the redistribution of the arrays in the most efficient manner. In this chapter, we present table construction techniques for overcoming the indexing overhead. This indexing overhead determines the set of processors to communicate with and the data elements to communicate. This contrasts with the earlier methods [14, 30] which used precise closed form expressions to determine the send and receive data/processor sets but required a two-step process. The first step involved the redistribution from CYCLIC(k_1) to either CYCLIC$(\text{lcm}\,(k_1, k_2))$ or CYCLIC$(\text{gcd}\,(k_1, k_2))$; the second step involved the redistribution from the lcm or the gcd value to k_2. Our method does not involve two steps and therefore is expected to provide more communication efficient code as it attempts to minimize on the communication startup overhead and node contention. Experimental results demonstrate the efficacy of our approach.

6.1 PROBLEM STATEMENT

Consider an array A distributed in a CYCLIC(k_1) fashion onto p processors. The block-cyclic distribution such as CYCLIC(k_1) is the most general distribution in which an array with block sizes of k_1 is distributed in a round robin manner onto p processors. Let and HPF program demand that the array A be redistributed among the p processors in a CYCLIC(k_2) fashion. This requires that some of the array elements be sent from each processor to the other processors. This gives rise to the concept of processor and data sets. The send (resp. receive) processor set is defined as the set of processors to (resp. from) which the data is to be sent (resp. received). Similarly, for a given processor in the send (resp. receive) processor set, the send (resp. receive) data set refers to the data elements to be sent to (resp. received from) that processor. Let m_2 be the receiving processor. So the send set for a processor m_1 can be defined as $Send(m_1, m_2)$ = Data set for (m_1, m_2). The complete send set refers to all the elements that have to be sent to all the processors that m_1 is going to communicate. Computing such sets constitutes the indexing overhead for the processor. Let us consider an example to illustrate the following. Consider an array A aligned with a template T. This template is then distributed onto 3 processors with a block size of $k = 4$. Next the array is redistributed with a block size of $k = 5$. The figure illustrating this alignment and distribution is shown in Figure 6.1.

Processor 0				Processor 1				Processor 2			
0	1	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59

(a) array distribution with block size $k = 4$ on 3 processors.

Processor 0					Processor 1					Processor 2				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53	54	55	56	57	58	59

(b) array distribution with block size $k = 5$ on 3 processors.

Figure 6.1: Redistribution of an array with block size k_1 = 4 to an array with block size k_2 = 5

As can be seen from the figure when the array distribution is changed from CYCLIC(4) to CYCLIC(5) each processor has to send and receive some elements. Consider the processor $m = 0$. This processor has to send the following set to processor ($m = 1$) = $\{$24, 36, 37, 38, 39, 50, 51$\}$. Similarly the send sets for different processors can be calculated in this manner. Also the receive set for processor $m = 0$ from $m = 1$ is the set $\{$4, 16, 17, 18, 19, 30, 31$\}$. The issue of computing these send and receive data sets and the processor sets forms the core for the redistribution problem.

This chapter discusses the above problem of redistribution. All the previous work done on this problem [14, 30] derived closed form expressions for the send and the receive data and processor sets. Moreover the approach for the general case of redistribution from CYCLIC(k_1) to CYCLIC(k_2) involved a multi-step process where the distribution was done from CYCLIC(k_1) to CYCLIC(lcm (k_1, k_2)). The second step involved the distribution from CYCLIC(lcm (k_1, k_2)) to CYCLIC(k_2). An alternative approach suggested in the same paper was again a two step process. In this case the first step involved the redistribution from k_1 to gcd (k_1, k_2) and the second step involved the redistribution from gcd (k_1, k_2) to k_2. Our method does not use closed-form expressions but detects a pattern for the send and receive data and processor sets. This pattern gives rise to a table which is used for redistribution.

This chapter is organized as follows. The next section discusses our method of approach to the problem. It explains the one step process of redistribution and how the send and the receive sets are constructed. The node code execution using the tables

Input: Problem parameters m, p, k_1, k_2

Output: Table $recvproc$ that stores the receiving processor for the element and the $sendvals$ table for the value to be sent. $sendsize$ is the sizes of data sets to be sent to the individual processors

Method:

$lcmval \leftarrow \text{lcm}\,(k_1, k_2)$
$scou \leftarrow \frac{lcmval}{k_1}$
$rcou \leftarrow \frac{lcmval}{k_2}$
$pk_1 \leftarrow p * k_1$
$pk_2 \leftarrow p * k_2$
$startcou \leftarrow 0; ind \leftarrow 0$
for $i = 0$ **to** $scou - 1$ **do**
    for $j = startcou + m * k_1$ **to** $startcou + (m + 1) * k_1 - 1$ **do**
        $owner \leftarrow (j \bmod pk_2) div k_2$
        $recvproc[ind] \leftarrow owner$
        $sendvals[ind] \leftarrow i * k_1 + j - (startcou + m * k_1)$
        $ind \leftarrow ind + 1$
        $sendsize[owner] \leftarrow sendsize[owner] + 1$
    enddo
    $startcou \leftarrow startcou + pk_1$
enddo
return table $recvproc$, table $sendvals$, table $sendsize$

Figure 6.2: Algorithm to compute the send-processor and the send-data sets.

derived from the send and receive table construction is explained. The next section presents experimental results.

6.2 BLOCK-CYCLIC REDISTRIBUTION: GENERAL CASE

6.2.1 SEND TABLE CONSTRUCTION

A detailed description of the algorithm presented in Figure 6.2 follows. The length of the pattern for the sending and the receiving of the data elements is lcm (k_1, k_2). The above statement means the following: if an array should be redistributed from a CYCLIC(k_1) distribution onto a CYCLIC(k_2) distribution, then it suffices to compute the send data and processor sets for the first lcm (k_1, k_2) values. The pattern starts repeating after this first block. This is precisely what is achieved through the algorithm that is presented in this chapter. The algorithm is simple in that it examines the first lcm (k_1, k_2) elements and determines the processor to which they are destined. An array section is distributed among the various processors. Each processor views this section using an imaginary template space. The idea is to recognize a pattern of repeating Send/Receive processor/data sets. Once this pattern is known the compiler generates a table of size lcm (k_1, k_2) using expensive mod and div operations. This table can then be used to compute the corresponding sets for the whole array section using simple computations. This avoids expensive computation operations on the whole array distributed on a processor. This leads to reduced execution times. The algorithm is given in its entirety in Figure 6.2. A step by step analysis of the algorithm will help in a good understanding of the algorithm. Lines 1–4 of the algorithm involve initializations in the algorithm. The $lastadd$ stores the last address on each processor. The last address on each processor can be calculated very easily since all the problem parameters like p the number of processors, n the number of elements in the array, and k_1, are all known. The variables $scou$, $rcou$ indicate the number of courses before the pattern starts repeating. $startcou$ points to the first element on each course.Since this algorithm involves the "send" of the data most of the factors pertain to k_1. The first **for** loop in line 6 iterates over all the courses of the original array. The second and inner **for** loop iterates over all the elements on each course. Nested **for** loop iterates over all the elements in a course. So the two for loops serve the purpose of iterating over all the courses and the inner loop iterating over all the elements in each course. The element j points to the individual element. From this the owner of the redistributed array is computed. The owner for this data is stored in a table called the $recvproc$. The particular data that has to be sent has to first undergo a global to local conversion before it can be stored in the $sendvals$ table. This conversion from the local to the global is done by the expression $i * k_1 + j - startcou + m * k_1$. Line 12 increments the number of elements that are to be sent to the processor $owner$. The algorithm returns as output the table $recvproc$, $sendvals$, $sendsize$. These tables can then be used for code execution. In the execution node code to be described later these tables are used very efficiently to enumerate all the elements to be communicated to the different processors.

6.2.2 RECEIVE TABLE CONSTRUCTION

The algorithm for constructing the receive-table is shown in Figure 6.3 and is analogous to the send-table algorithm shown in Figure 6.2. The details of this algorithm are therefore similar to the send table construction algorithm. The algorithm is described briefly. The initializations for the algorithm are similar to the send-table algorithm. The outer loop in line 6 now iterates through the courses in the redistributed array. The views that are considered in determining the send and the receive sets is important. For the send sets the view was from the original distributed array but in the case of the receive set the view is from the redistributed array and not the original array. The $startcou$ gives the first global element on the course on processor $m = 0$. This helps in determining the first element on the processor on each course. Note that the $startcou$ is incremented by $p * k_1$ each time a course is completed. The inner loop iterates from the start element on each course to the last element on each course. The owner is determined simply as in line 10 by a simple mod and div operation. The owner is stored in the table $sendproc$ and the value is done a global to a local conversion and is stored in the table $recvvals$. The number of elements to be sent to each processor is stored in the table $recvsize$. In this manner two nested

Input: Problem parameters m, p, n, k_1, k_2

Output: Table $sendproc$ to indicate the sending processor for the element and the $recvvals$ table for the value to be received. $sendsize$ consists of sizes of data sets to be received from the other processors

Method:

$lcmval \leftarrow lcm(k_1, k_2)$
$pk_1 \leftarrow p * k_1$
$pk_2 \leftarrow p * k_2$
$scou \leftarrow lcmval$ div k_1
$rcou \leftarrow lcmval$ div k_2
$pk_1 \leftarrow p * k_1$
$pk_2 \leftarrow p * k_2$
$startcou \leftarrow 0; ind \leftarrow 0$
for $i = 0$ **to** $rcou - 1$ **do**
    for $j = startcou + m * k_2$ **to** $startcou + (m + 1) * k_2 - 1$ **do**
        $owner \leftarrow (j$ mod $pk_1)$ div k_1
        $sendproc[ind] \leftarrow owner$
        $recvvals[ind] \leftarrow i * k_2 + j - (startcou + m * k_2)$
        $ind \leftarrow ind + 1$
        $recvsize[owner] \leftarrow recvsize[owner] + 1$
    enddo
    $startcou \leftarrow startcou + pk_2$
enddo
return table $sendproc$, table $recvvals$, table $recvsize$

Figure 6.3: Algorithm to compute the receive-processor and the receive-data sets.

loops iterate over each and every element of the pattern so that the table formation can take place. It may be noted that the div and the mod operation on each iteration of the loop looks expensive. However clever optimizations were done on the code to make the code less expensive then it actually is. This involved taking into account some of the program characteristics into account. The following paragraph illustrates the optimization done to the code which is of particular interest.

The code presented above in Figure 6.4 is very simple but very effective. The div operation basically is an integer operation and provides the floor of the division of two integers. Also the division is always by a constant pk_2 and the dividend is always between 0 and $pk_2 - 1$ since a mod by pk_2 is performed on it. These characteristics convert the division into just one table lookup which is extremely inexpensive. As the above code obviously shows, based on the value of the dividend it will be assigned to a processor. This is what is illustrated by line 4. The mod operations in the program are optimized by replacing all the mods with conditionals. It is known that for all the cases in redistribution the mods are incremental. This means that if a mod expression is of the form i mod j, then in the next iteration the mod expression will be $(i + 1)$ mod j. Such type of mods can easily be substituted by conditionals such as that presented in the Figure 6.4. This technique has been very clearly described in Chapter 3 on two-level address generation. The same technique is shown to to perform very well there too.

6.2.3 Execution of node code for redistribution of arrays

Having obtained the tables for the send and the receive, for the data and the processor sets the next step is to effectively use this for enumerating all the elements which will be communicated to/from each processors. We present here two codes for the enumeration of all the elements that will be sent using the send and receive table. The codes are very simple and self explanatory.

As can be seen from the code for the message packing of the send and the receive sets the tables $sendproc$, $recvproc$, $sendvals$, $recvvals$ make the code simple and efficient. In the code for the message packing of the receive set, the variable tmp is assigned the value of the processor which is to receive the values that are to be sent. At each consecutive location the value to be sent is stored. A step by step explanation of the algorithm follows. Lines 1 and 2 of the algorithm are initializations. The outer **for** loop iterates over the whole pattern table. This is the table that gives the communication pattern. Since the length of the table is $lcmval = \text{lcm}\,(k_1, k_2)$ therefore this loop runs throughout the length of the table whose size is $lcmval$. As the table is traversed tmp is assigned the value of the sending processor. cnt sets the counter for the particular processor. It keeps a count of the number of values packed till the present iteration of the loop. The message is packed into sendbuf or recvbuf. Since the pattern repeats after $lcmval$ and all the messages are packed till the last address is exceeded. Therefore the tables provide an easy and efficient means for message packing for send and receive. The input for the send execution code algorithm is the $recvproc$, $sendvals$ and the $sendsize$ tables. The input similarly for

```
pk1 ← p * k1; pk2 ← p * k2; mk1 ← m * k1
lcmval ← lcm(k1, k2)
lastadd ← last address on the processor
processor ← 0
for i = 0 to p − 1 do
   for j = i * k2 to i * k2 + k2 − 1 do
      divarray[j] ← processor
   enddo
   processor ← processor + 1
enddo
scou ← lcmval div k1; rcou ← lcmval div k2
ind ← 0; startcou ← 0
for i = 0 to scou − 1 do
   tmp ← startcou + mk1
   tmp2 ← i * k1
   tmp3 ← tmp mod pk2
   for j = tmp to tmp + k1 − 1 do
      if (tmp3 = pk2) then
         tmp3 ← 0
      endif
      owner ← divarray[tmp3]
      recvproc[ind] ← owner
      sendvals[ind] ← tmp2
      sendsize[owner] ← sendsize[owner] + 1
      ind ← ind + 1
      tmp2 ← tmp2 + 1
      tmp3 ← tmp3 + 1
   enddo
   startcou ← startcou + pk1
enddo
return
```

Figure 6.4: Code optimizations in the program

Input: Problem parameters m, p, n, k_1, k_2, tables $recvproc$, $sendvals$, $sendsize$
Output: $sendbuf$ array-consisting of the processor number and all the elements that are to be sent to it from this processor
Method:

```
lastadd ← last address on processor m
lcmval ← lcm (k1, k2)
for i = 0 to lcmval − 1 do
   tmp ← recvproc[i]; cnt ← count[tmp]; j ← sendvals[i]
   while (j ≤ lastadd) do
      sendbuf[tmp][cnt] ← A[j]
      cnt ← cnt + 1
      j ← j + lcmval
   enddo
   count[tmp] ← cnt
enddo
return table sendbuf
```

Figure 6.5: Algorithm to perform message packing for send sets

Input: Problem parameters m, p, n, k_1, k_2, tables $sendproc$, $recvvals$, $recvbuf$
Method:

```
lastadd ← last address on processor m
lcmval ← lcm (k1, k2)
for i = 0 to lcmval − 1 do
   tmp ← sendproc[i]; cnt ← count[tmp]; j ← recvvals[i]
   while (j ≤ lastadd)
      A[j] ← recvbuf[tmp][cnt]
      cnt ← cnt + 1
      j ← j + lcmval
   endwhile
   count[tmp] ← cnt
enddo
return
```

Figure 6.6: Algorithm to perform message unpacking for receive sets

the receive execution code algorithm is the $sendproc$, $recvvals$ and the $recvsize$ tables. These tables are derived in the algorithms in figures 5.5 and 5.6.

6.3 RESULTS AND DISCUSSION

The experiments were performed on a Sun UltraSparc workstation. To measure the actual amount of node contention the experiments are planned to be conducted on an actual parallel machine. The node contention is expected to be less as the message packing occurs only once and not in steps of two or three as in the case of Thakur et al. [30]. The experiments were run for a large number of parameter values. But due to constraints in space only a small subset of the results are shown here in Tables 6.1 and 6.2. The first three columns in the table indicate the input parameters for the problem. The first column represents the number of processors on which the redistribution takes place. The second column indicates the block size on the processors before the redistribution and the third column represents the block size after redistribution. The next two columns show the time taken by the method described by Thakur et al [30]. As explained in the previous sections their method basically involves a two step process in which the redistribution occurs from k_1 to lcm (k_1, k_2) and then from lcm (k_1, k_2) to the final distribution of k_2. In the second method, also known as the GCD method in the column, the redistribution occurs from k_1 to gcd (k_1, k_2) and then from gcd (k_1, k_2) to k_2. The choice of the methods and their performance depending on the various input parameters have been discussed in detail in their paper. Here the times for the two methods have been shown in comparison to the times for our method. It can be clearly seen that the times for the LSU method are much better than either of the LSU or the GCD method. This result may have significant impact as it shows that the results conducted for the node execution on a single processor clearly has very good performance. However it may be noted that when such optimizations are done on a parallel system with communication overheads like startup time etc. a lot of factors may come into effect. On the outset it seems that a redistribution involving two step process will be much costlier than the one involving a single step method like the LSU. But as also stated in the paper by [30] node contention plays an important part when such type of communication takes place. In the case of methods like GCD or the LCM the node contention will be less as the communication is "sparse". However in communication due to methods like LSU the communication is expected to be "dense" because all the elements have to finally reach the same destination no matter which way they take. To study the effects of node contention and other related factors such as communication startup times, message length optimizations experiments will be run on parallel machines. However the results presented above provide a promising methodology for redistribution of arrays on parallel machines.

In this work we stated the previous methods of redistribution and sought to improve on that work. A new approach was presented which is a deviation from the traditional two step approach to redistribution. A repeating sequence of data and processor Send/Receive sets were identified. Table construction techniques for the send and the receive sets were discussed. These tables were then used for efficient node code generation. Extensive results were performed to prove the efficacy of our approach.

6.4 CHAPTER SUMMARY

This chapter focuses on the efficient computation of the send and receive data and processor sets when arrays are required to change their distribution dynamically. When arrays need to be redistributed, data needs to be sent and received from different processors. This requires each processor to compute its send data set and its send processor set. Also the processor needs to compute its receive data set and its receive processor set. The previous work [30, 14] had concentrated on multi-phase redistribution where the arrays are redistributed in steps from CYCLIC(x) to CYCLIC(y) and from CYCLIC(y) to CYCLIC(z). Here y may the lcm (x, z) or it may be the gcd (x, z). This is assumed to provide for better node code efficiency since it involves lesser guards.

However in the method proposed in this chapter a pattern table for the send and the receive sets for the data and the processor is determined. This pattern table as the name suggests recognizes a pattern for the send and the receive. This pattern table has served two purposes. Firstly, like all the previous address generation problems it has improved node code efficiency as the table construction times for such tables is very small. Secondly, the redistribution takes place in a one-step manner which may provide avoid node contentions better than the previous working algorithms. The time for the table construction techniques have been presented and it has shown to be not a very big overhead as compared to the message packing and message unpacking times. Also it has been shown that message packing and unpacking with tables proves to be very comparable to the earlier methods proposed by Thakur et al. Simulations were not run on an actual parallel machine but it is expected that our method of redistribution will suit more favorably to the node contention problem as compared to the previous work.

Table 6.1: Comparison of techniques for a redistribution from a CYCLIC(k_1) distribution to a CYCLIC(k_2) distribution for $n = 32000$.
Note: The GCD and LCM columns show the timing values for the method by Thakur et al. LSU column shows timings for our method.

p	k_1	k_2	Table(μsec)	LSU Exec(μsec)	LCM(μsec)	GCD(μsec)
2	2	7	1448	4589	16097	33993
2	5	4	1084	3965	11769	38983
2	7	15	1023	3514	7578	38281
2	9	17	845	3321	6768	38269
2	15	25	765	3098	5436	11316
2	25	99	1071	3911	4451	37835
4	2	7	681	2816	8121	17046
4	5	4	638	2573	5554	16698
4	7	15	552	2410	3918	19230
4	9	17	482	2359	3481	19259
4	15	25	434	2171	2807	5802
4	25	99	781	3126	2467	19037
8	2	7	452	2190	4227	8743
8	5	4	327	1989	2962	8522
8	7	15	293	1902	2124	9814
8	9	17	281	1901	1916	9787
8	15	25	255	1770	1576	3052
8	25	99	630	2682	1398	9607
16	2	7	265	1843	2377	4653
16	5	4	209	1738	1773	4577
16	7	15	201	1648	1396	5236
16	9	17	204	1651	1276	5207
16	15	25	180	1654	1080	1804
16	25	99	589	2558	1033	5182
32	2	7	167	1813	2055	3225
32	5	4	166	1879	1716	3130
32	7	15	169	1783	1260	3196
32	9	17	166	1752	1213	3215
32	15	25	152	1723	1104	1472
32	25	99	613	2665	1256	3223

Table 6.2: Comparison of techniques for a redistribution from a CYCLIC(k_1) distribution to a CYCLIC(k_2) distribution for $n = 64000$.
Note: The GCD and LCM columns show the timing values for the method by Thakur et al. LSU column shows timings for our method.

p	k_1	k_2	Table(μsec)	LSU Exec(μsec)	LCM(μsec)	GCD(μsec)
2	2	7	2684	9694	31931	67747
2	5	4	2323	8415	24510	78008
2	7	15	1637	6619	14901	76930
2	9	17	1685	6683	13344	75665
2	15	25	1553	6175	10840	22696
2	25	99	1718	6752	8545	75299
4	2	7	1362	5703	16330	34087
4	5	4	1085	5162	10902	33759
4	7	15	862	4495	7657	38479
4	9	17	829	4482	6734	38304
4	15	25	777	4187	5528	11202
4	25	99	1099	5135	4553	37958
8	2	7	700	4094	8187	17167
8	5	4	627	3670	5669	16949
8	7	15	471	3578	3984	19343
8	9	17	543	3624	3687	19482
8	15	25	445	3310	2916	5828
8	25	99	781	4322	2536	19167
16	2	7	404	3400	4398	8933
16	5	4	364	3244	3069	8764
16	7	15	349	3072	2528	10224
16	9	17	313	2988	2097	10042
16	15	25	271	2989	1756	3220
16	25	99	701	4035	1607	9904
32	2	7	280	3201	2785	5091
32	5	4	251	3194	2387	5238
32	7	15	217	3019	1761	7420
32	9	17	221	3016	1966	5903
32	15	25	203	2999	1429	2201
32	25	99	656	3894	1432	5545

CHAPTER 7

SUMMARY AND CONCLUSIONS

This thesis work covers a broad spectrum of areas related to code generation and optimization of High Performance Fortran (HPF). It deals with operations on regular sections, address generation for two-level mapping, coupled subscripts,and data redistribution, and data structures for efficient node code execution.

Chapter 2 of this thesis addresses the problem of efficiently enumerating the union and difference of regular sections. Such operations on regular sections are ubiquitous in code generation for HPF. It was shown that the union (and difference) of two regular sections forms a finite-state machine. Most other work on this problem requires the use of expensive run-time solutions for this. A technique to construct a pattern table was derived, and this is useful both the union and difference operations. Simulation results were presented to highlight the efficiency of our solution.

In Chapter 3, we discussed address generation for two-level mapping. Venkatachar [38] presented several efficient solutions to this problem. We take two approaches to the problem. The first is the concept of hashing. It is used effectively in compilers, operating systems and many other areas of application. Hashing as used here incurs high memory overhead but is fast. The second method presents an optimization of the ITABLE technique presented in [38], that eliminates expensive integer operations such as division and mod. This optimization substitutes integer divisions with table lookups, where the tables themselves are computed using just a single division followed by less expensive integer operations. The extensive results presented at the end of the chapter show that the hash table technique is useful when parameters are small and that the optimized ITABLE performs well over a wide range of parameters.

In Chapter 4 the effect of using different data structures has been examined. The existing data structures are discussed and their possible shortcomings are noted. To overcome the possible shortcomings and to provide better data structures suited for faster execution but at the same time providing for less space overhead six new data structures have been presented. Node code using all these six data structures are shown. Each one is analyzed and compared against the use of each. This presents the user with a wide variety of data structures to choose from given the problem parameters. The RESET table which is presented is an important concept that is general enough to be used in many problems. The strip mined code performs very well for several cases.

In Chapter 5 address generation for coupled subscripts is presented. Coupled subscripts are an important class of complex affine subscripts. Coupled subscripts involve solving for iterations in the two dimensions independently and then searching for the common iterations in the two dimensions. Two methods based on hashing and search were presented. They outperform the solution due to Kennedy et al. [16]. In addition, the hashing based methods is better than search by about 10%.

In Chapter 6 we discuss issues related to efficient redistribution of arrays. We consider the most general case of redistribution from CYCLIC(x) to CYCLIC(y). Current solutions to this problem require two all-to-all communication steps and in each step closed forms for communication sets are used; these solutions exploit the fact that closed forms exits for CYCLIC(x) to CYCLIC$(k * x)$ and for CYCLIC$(k * x)$ to CYCLIC(x) redistribution, where k is an integer. The saving in using closed forms is significantly outweighed by the need to perform two all-to-all communication steps. We showed that by constructing a pattern table (by exploiting the fact that the communication pattern repeats after $\mathrm{lcm}\,(k_1, k_2)$ elements) and using it for communication generation, our solution requires a single all-to-all communication step and the run-time overhead for communication set generation is comparable to that of current solution. This results in a clear advantage.

7.1 FURTHER RESEARCH

Several problems need further study. Some of those problems (to name a few) are listed below.

- efficient data structures and execution strategies with two-level mapping.
- more extensive study of efficient data structures and execution strategies for coupled subscripts.
- address generation for coupled subscripts and multiple induction variables with two-level mapping.
- efficient communication set generation for two-level mapping for array sections as well as for redistribution.
- address and communication generation for general affine subscripts.

BIBLIOGRAPHY

[1] A. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution. In *Proc. of the 4th Workshop on Compilers for Parallel Computers,* Delft, The Netherlands, December 1993.

[2] S. Benkner. Handling block-cyclic distributed arrays in Vienna Fortran 90. In *Proc. International Conference on Parallel Architectures and Compilation Techniques,* Limassol, Cyprus, June 1995.

[3] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming,* 1(1):31–50, Fall 1992.

[4] S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S. Teng. Generating local addresses and communication sets for data parallel programs. *Journal of Parallel and Distributed Computing*, 26(1):72–84, 1995.

[5] S. Dutta and J. Ramanujam. Data structures for efficient execution of programs with block-cyclic distributions. Technical Report TR-96-11-01, Dept. of Elec. & Comp. Engineering, Louisiana State University, Nov. 1996. Preliminary version presented at the *6th Workshop on Compilers for Parallel Computers,* Germany, Dec. 1996.

[6] F. Coelho, C. Germain, J.L. Pazat. State of the Art in Compiling HPF. PRS Spring School, Les Menuires, France, March 1996.

[7] G. Fox, S. Hiranandani, K. Kennedy, C Koelbel, U. Kremer, C.-W. Tseng, and M. Wu. Fortran D language specification. Technical Report TR-91-170, Dept. of Computer Science, Rice University, Dec. 1991.

[8] M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience*, 2(3):171–193, September 1990.

[9] P. M. Gruber and C. G. Lekkerkerker. *Geometry of numbers.* North-Holland Mathematical Library Volume 37, North-Holland, Amsterdam, 1987 (Second edition).

[10] S. Gupta, S. Kaushik, C. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. *Journal of Parallel and Distributed Computing*, 1996.

[11] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming,* 2(1-2):1–170, 1993.

[12] S. Hiranandani, K. Kennedy, J. Mellor-Crummey, and A. Sethi. Compilation techniques for block-cyclic distributions. In *Proc. ACM International Conference on Supercomputing*. July 1994

[13] S. Kaushik. Compile-time and run-time strategies for array statement execution on distributed-memory machines. Ph.D. Thesis, Department of Computer and Information Science, The Ohio State University, 1995.

[14] S. D. Kaushik, C.-H. Huang, J. Ramanujam, and P. Sadayappan. Multi-Phase Array Redistribution: Modeling and Evaluation *In Proceedings of the 9th International Parallel Processing Symposium,* pages 441–445, April 1995.

[15] K. Kennedy, N. Nedeljkovic, and A. Sethi. A linear-time algorithm for computing the memory access sequence in data-parallel programs. In *Proc. of Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[16] K. Kennedy, N. Nedeljkovic, and A. Sethi. Efficient address generation for block-cyclic distributions In *Proc. ACM International Conference on Supercomputing,* Madrid, Spain, July 1995.

[17] K. Kennedy, N. Nedeljkovic, and A. Sethi. Communication generation for cyclic(k) distributions. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, B. Szymanski and B. Sinharoy (Eds.), Kluwer Academic Publishers, 1995.

[18] A. Knies, M. O'Keefe, and T. MacDonald. High Performance Fortran: A practical analysis. *Scientific Programming,* 3(3):187–199, Fall 1994.

[19] C. Koelbel. Compile-time generation of communication for scientific programs. In *Supercomputing '91*, pages 101–110, Nov. 1991.

[20] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *High Performance Fortran Handbook.* The MIT Press, 1994.

[21] T. MacDonald, D. Pase, and A. Meltzer. Addressing in Cray Research's MPP Fortran. In *Proceedings of the Third Workshop on Compilers for Parallel Computers* (Vienna, Austria, July 1992), Austrian Center for Parallel Computation, pages 161–172.

[22] S. Midkiff. Local iteration set computation for block-cyclic distributions. In *Proc. International Conference on Parallel Processing*. Vol. II, pages 77–84, August 1995.

[23] S. Midkiff. Comuting the local iteration set of a block-cyclically distributed reference with affine subscripts. In *Proc. 6th Workshop on Compilers for Parallel Computers,* Aachen, Germany, December 1996.

[24] J. Ramanujam. Non-unimodular transformations of nested loop. In *Proc. Supercomputing 92,* pages 214–223, Nov. 1992.

[25] J. Ramanujam. Beyond unimodular transformations. *The Journal of Supercomputing,* 9(4):365-389, Dec. 1995.

[26] J. Ramanujam, S. Dutta, and A. Venkatachar. Code generation for complex subscripts in data-parallel programs. To appear in *Languages and Compilers for Parallel Computing,* Zhiyuan Li *et al.,* (Eds.), Lecture Notes in Computer Science, Springer-Verlag, 1997.

[27] C. van Reeuwijk, H.J. Sips, W. Denissen, and E. M. Paalvast. Implementing HPF distributed arrays on a message-passing parallel computer system. CP Technical Report series, TR9506, Delft University of Technology, 1995.

[28] J. M. Stichnoth. Efficient compilation of array statements for private memory multicomputers. Technical Report CMU-CS-93-109, School of Computer Science, Carnegie-Mellon University, Feb. 1993.

[29] E. Su, A. Lain, S. Ramaswamy, D.J. Palermo, E.W. Hodges IV, and P. Banerjee. Advanced compilation techniques in the PARADIGM compiler for distributed-memory multicomputers. In *Proc. 1995 ACM International Conference on Supercomputing,* Barcelona, Spain, July 1995.

[30] R. Thakur, A. Choudhary and J. Ramanujam. Efficient algorithms for array redistribution. *IEEE Transactions on Parallel and Distributed Systems,* 7(6):587–594, June 1996.

[31] A. Thirumalai. Code generation and optimization for High Performance Fortran. M.S. Thesis, Department of Electrical and Computer Engineering, Louisiana State University, August 1995.

[32] A. Thirumalai and J. Ramanujam. Code generation and optimization for array statements in HPF. Technical Report TR-94-11-02, Dept. of Electrical and Computer Engineering, Louisiana State University, Nov. 1994. revised August 1995

[33] A. Thirumalai and J. Ramanujam. An efficient compile-time approach to compute address sequences in data parallel programs. In *Proc. 5th International Workshop on Compilers for Parallel Computers,* Malaga, Spain, pages 581–605, June 1995.

[34] A. Thirumalai and J. Ramanujam. Fast address sequence generation for data-parallel programs Using Integer Lattices. In *Languages and Compilers for Parallel Computing,* C.-H. Huang *et al.* (Editors), Lecture Notes in Computer Science, Vol. 1033, pages 191–208, Springer-Verlag, 1996.

[35] A. Thirumalai and J. Ramanujam. Address sequence determination using closed forms for basis vectors of the array section lattice. Technical Report, Dept. of Electrical and Computer Engineering, Louisiana State University, Apr. 1995.

[36] A. Thirumalai, J. Ramanujam, and A. Venkatachar. Communication generation and optimization for HPF. In *Languages, Compilers, and Run-Time Systems for Scalable Computers,* B. Szymanski and B. Sinharoy (editors), Kluwer Academic Publishers, Chap. 29, pages 311–316, Norwell, MA, 1995

[37] A. Thirumalai and J. Ramanujam. Efficient computation of address sequences in data-parallel programs using closed forms for basis vectors. *Journal of Parallel and Distributed Computing,* 38(2):188–203, November 1996.

[38] A. Venkatachar. Efficient address and communication generation for data-parallel programs. M.S. Thesis, Department of Electrical and Computer Engineering, Louisiana State University, December 1996.

[39] A. Venkatachar, J. Ramanujam and A. Thirumalai. Generalized overlap regions for communication optimization in data parallel programs. In *Languages and Compilers for Parallel Computing,* D. Sehr *et al.* (Editors), Lecture Notes in Computer Science, Vol. 1239, pages 404–419, Springer-Verlag, 1997.

[40] L. Wang, J. Stichnoth, and S. Chatterjee. Runtime performance of parallel array assignment: An empiricial study. In *Proc. Supercomputing 96,* November 1996.

[41] Michael Wolfe. *High performance compilers for parallel computing.* Benjamin/Cummings Publishing Co., 1996.

VITA

Swaroop Dutta was born in Davangere, Karnataka, India, on November 15, 1972. After completing his high school, he joined the Bachelor of Technology program in Regional Engineering College, Warangal, India in August 1990. He received his bachelor's of technology degree in Electronics and Communication Engineering in Spring 1994. He then went on to work as a Graduate Engineer Trainee at Kirloskar Electric Company Limited, Bangalore, India, from September 1994 to May 1995. He was involved in the design of Automation Software for a Hot Strip Mill project.

Swaroop joined the master's program in the Louisiana State University in Fall 1995. He was working as a Research Assistant at the Department of Electrical Engineering where he worked in the area of Parallelizing Compilers until Summer of 1997. He has written book chapters and conference papers in this research area. His next goal is to obtain practical experience in the design and development of compilers in the industry.

He is a candidate for a master of science degree in Electrical Engineering from Louisiana State University at the Fall 1997 commencement.

Dutta, Swaroop Vasudeva, B.Tech., Regional Engineering College, Warangal, India, 1994
Master of Science in Electrical Engineering, Fall Commencement, 1997
Major: Electrical Engineering
Compilation and Run-Time Techniques for Data-Parallel Programs
Thesis directed by Associate Professor Jagannathan Ramanujam
Pages in thesis, 101. Words in abstract, 284.

ABSTRACT

With the advent of data-parallel languages such as High Performance Fortran and the enormous compiler support for these languages, the difficult task of parallel programming is becoming easier. This thesis addresses several important problems in compilation of programs in such languages. Union and difference of regular sections (routinely required in code generation) are expensive operations. The results of these operations on regular sections exhibit a pattern that can be expressed as a finite-state machine; this thesis presents efficient techniques for constructing the pattern. Two methods are presented for generating the access sequence with two-level mapping of data onto processors supported by HPF; the first is based on hashing, and the second is an improvement over an existing technique as a result of replacing expensive integer operations. A large part of the efficiency of the node code derived by a compiler for HPF depends on the type of data structure used for node code generation. Techniques such as strip mining, compression and expansion of tables etc. using new efficient data structures are introduced. A key part in the address generation of complex subscripts is the address generation for coupled subscripts. Coupled subscripts have the deepest loop index common among the two dimensions, which makes the address generation difficult. Methods using hashing and search are employed once again to enumerate the common iterations in the various array dimensions. Dynamic redistribution plays an important part in HPF and determining the send and receive processor and data sets form the crux of the redistribution problem. Current approaches rely on multi-phase redistribution which incurs high communication overhead. This thesis presents an efficient single-phase table-based solution. Experimental results show the clear superiority of the solutions presented in this thesis for each of the five problems addressed.