In Proc. 8th Workshop on Compilers for Parallel Computing (CPC 98), Sweden, June 1998

# Advanced Compilation Techniques for HPF

J. Ramanujam* S. Dutta A. Venkatachar A. Thirumalai
Department of Electrical and Computer Engineering
Louisiana State University, Baton Rouge, LA 70803, USA
http://www.ee.lsu.edu/jxr/index.html

**Abstract**

Data-parallel languages like High Performance Fortran allow users to specify mappings of array by first aligning elements to an abstract Cartesian grid called templates and then distributing the templates across processors. Code generation then includes the generation of the sequence of local addresses accessed on a processor. Address sequence generation for non-unit alignment strides—referred to as the two-level mapping problem—is difficult. This paper presents efficient solutions to the problem of address generation for two-level mapping for general CYCLIC($k$) distribution. Our approach involves the construction of pattern tables which incurs negligible runtime overhead compared to other existing solutions for this problem. We use two applications of the integer-lattice-based method developed by Thirumalai and Ramanujam [33] to generate the pattern of accesses using a variety of techniques. Extensive experiments demonstrate that the techniques presented in this paper significantly outperform current solutions to the two-level mapping problem. In addition, we present a brief overview of our work on related problems in code generation for HPF such as communication generation, code generation for complex subscripts and array redistribution.

## 1 Introduction

Distributed memory multiprocessors are attractive for high performance computing in that they offer potentially high levels of flexibility, scalability and performance. However, programming these machines to realize their promised performance (which requires a full orchestration of the execution through careful partitioning of computation and data, and placement of message passing) remains a difficult task. The extreme difficulty of writing correct and efficient programs is a major obstacle to the widespread use of parallel high-performance computing. The main objective of efforts such as High Performance Fortran (HPF) [11, 17], Fortran D [8], and Vienna Fortran [4] (which grew out of the earlier SUPERB effort) is to raise the level of programming on distributed memory machines while retaining the object code efficiency derived, for example, from message passing.

Languages such as Fortran D [8], Vienna Fortran [4] and High Performance Fortran (HPF) [11] include directives such as ALIGN and DISTRIBUTE that describe how data is distributed among processors in a distributed-memory multiprocessor. For example in HPF, arrays are mapped to processors in two steps. In the first step, arrays are *aligned* to an abstract discrete Cartesian grid called a *template;* the template is then distributed onto the processors. The effect of this two-level mapping onto $p$ processors is to create $p$ disjoint pieces of the array, with each processor being able to address only data items in its locally allocated piece. Thus, an HPF compiler must generate code that executes on each processor (referred to as node code) that accesses only locally owned pieces directly, and inserts communication for non-local accesses. In order to generate node code we need to know the sequence of local memory addresses accessed by each processor and the sequence of sends and receives for a given processor to access non-local data. A regular access pattern in terms of the global data structure can appear to be irregular within each locally allocated piece. For example, an array section $A(\ell : h : s)$ exhibits a regular access sequence of stride $s$; but with an HPF-style data mapping, the access sequence can become irregular. The array section $A(\ell : h : s)$ is equivalent to the set of elements $A(\ell + js)$ where $j$ is an integer and $0 \leq j \leq \lfloor \frac{h-l}{s} \rfloor$.

The problem of address generation for one-level mapping has received much attention. Address sequence generation for non-unit alignment strides, i.e., two-level mapping, can be difficult. This paper presents a variety of solutions to the problem of address generation for such a case when the data objects are mapped to processor memories using CYCLIC($k$) distribution. We present several methods of generating the address sequence. Our approach involves construction of pattern tables which incur negligible runtime overhead compared to other existing solutions for this problem. We view the sequence of addresses as the composition of two integer lattices [9] and use two applications of the method developed by Thirumalai and Ramanujam [33] to generate the pattern of accesses. After presenting our solution to the two-level mapping problem, we discuss the work carried out by our research group on several related problems in HPF code generation.

This paper is organized as follows. In Section 2, we discuss the address sequence generation problem for two-level mapping and Section 3 discusses related work. Section 4 presents several algorithms. In Section 5, we present experimental results that demonstrate the superiority of our solution to other solutions. In Section 6, we present a brief overview of our work on related problems in code generation for HPF such as communication generation, code generation for complex subscripts, array redistribution and operations on regular sections. Section 7 concludes with a summary.

## 2 Two-level mapping problem

Consider the following HPF code.

```
        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
```

*Supported in part by NSF Young Investigator Award CCR-9457768 and NSF grant CCR-9210422. jxr@ee.lsu.edu

A compiler that generates the node code for the above HPF program has to generate the set of local elements of array A *accessed* by processor $m$. When the alignment stride $a > 1$ the data mapping is called a *two-level* mapping. This mapping results in having many template cells which do not have any array elements aligned to them. These empty template cells are termed as *holes*. We should not allocate memory for these *holes* in the local address space during mapping. The challenge now is to generate the set of accessed elements in this local address space that incurs zero memory wastage.

Figure 1(a) shows the distribution of the template cells onto a processor arrangement. For this example, the alignment stride $a$ and access stride $s$ are both equal to 3. The number of processors $p$ is 4 and we assume a CYCLIC(4) distribution. This example is from Chatterjee *et al.* [5]. The set of global indices of array elements that are aligned to some template cell on this processor is called the set of *allocated* elements. The set of global indices of accessed array elements that are aligned to some template cell on this processor is called the set of *accessed* elements. These *accessed* elements are however a subset of *allocated* elements. For the given example 0, 1, 6, 11, 16, $\cdots$ is the set of *allocated* elements and 0, 6, 27, 33 ,48, $\cdots$ is the set of *accessed* elements for processor 0. Figure 1(b) shows the local address space of template cells on all the processors. The problem of generating *accessed* elements for this template space is similar to a one-level mapping problem where the stride $s$ is replaced with $a * s$. However using this method we incur huge memory wastage and suffer from data locality resulting in higher execution times.

However if one eliminates holes in this layout, we can have a significant savings in memory usage. This can be achieved by viewing the local address space as shown in Figure 1(c). This local address space does not have any memory wastage. However additional work at address generation time has to be done to switch from the template space to the local space. Due to the absence of these *holes* we can expect improved data locality and thus leading to faster execution times. The address generation problem now is to generate the set of elements *accessed* in this local address space. The algorithms discussed in this paper aims at solving this problem using efficient runtime techniques.

| 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 1: Two-level mapping of array A when $a = 3$, $s = 3$, $p = 4$, $k = 4$ and $l = 0$

## 3 Related work

### 3.1 One-level mapping

Several papers have addressed this code generation problem [1, 3, 5, 10, 13, 14, 16, 18, 27, 28, 2, 31]. The first author Ramanujam [20] presents a detailed survey of integer-lattice based address generation techniques. Koelbel [16] derived techniques for compile-time address and communication generation for BLOCK and CYCLIC distribution for non-unit stride accesses. Chatterjee *et al.* [5] derived a technique that identifies a repeating access pattern characterized as a finite-state machine. Their $O(k \log\ k)$ algorithm involves a solution of $k$ linear Diophantine equations to determine the pattern of addresses accessed, followed by sorting these addresses to derive the accesses in linear order.

In an exhaustive study of this problem Stichnoth [28, 29] presented a framework to enumerate local addresses and generation of communication sets. Gupta *et al.* [10] derived the *virtual-block* and the *virtual-cyclic* schemes. The virtual block (cyclic) scheme views the global array as a union of several cyclically (block) distributed arrays. The virtual cyclic scheme does not preserve the access order in the case of DO loops (this is not a problem for array assignments). Ancourt *et al.* [1] use a linear algebra framework to generate code for fully parallel loops in HPF, which does not exploit repeating sequence of accesses. Recently, building on the finite-state machine approach of Chatterjee *et al.,* Kennedy *et al.* [14] derived a runtime solution that determines the basis vectors by solving $2k - 1$ linear Diophantine equations. The improvement over [5] comes from avoiding the sorting step at the expense of solving an additional set of $k-1$ linear Diophantine equations. Thirumalai and Ramanujam [33] showed that solving $2k-1$ equations is not required. The algorithm which is of complexity $O(\log \min(s, pk) + min(s, k)$ requires the solution of only 2 linear Diophantine equations. The same authors also showed how to derive closed forms for basis vectors of the array section lattice in several cases [32, 35]. Recently, we have derived an $O(\log \min(s, pk))$ algorithm [19] for basis vector generation.

### 3.2 Two-level mapping

A few methods have been proposed to address the code generation problem for two-level mapping. The solution by

| INDEX | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ALLOCATED | 0 | 3 | 6 | 9 | 12 | 15 | 18 | 21 | 24 | 27 | 30 | 33 |

(a)

| ACCESSED | 0 | 6 | 21 | 27 |
|---|---|---|---|---|

(b)

| PATTERN | 0 | 2 | 7 | 9 |
|---|---|---|---|---|

(c)

Figure 2: Local addresses of *accessed* and *allocated* elements along with *two-level* access pattern when $a = 3$, $s = 3$, $p = 4$, $k = 4$ and $l = 0$

| Y1 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|
| Y2 | 0 | 3 | 2 | 1 |

(a)

| X1 | 0 | 1 | 5 | 6 |
|---|---|---|---|---|
| X2 | 0 | 2 | 1 | 3 |

(b)

| ITABLE | 0 | 3 | 2 | 1 |
|---|---|---|---|---|

(c)

| PATTERN | 0 | 2 | 7 | 9 |
|---|---|---|---|---|

(d)

Figure 3: Two dimensional coordinates of *allocated* and *accessed* elements along with ITABLE and *two-level* access pattern when $a = 3$, $s = 3$, $p = 4$, $k = 4$ and $l = 0$

Chatterjee *et al.* [5] involves two applications of the one-level algorithm, where the input strides are $a$ for the first and $a*s$ for the second. They build two finite state machines which will generate the access sequence for both *allocated* and *accessed* elements. The next step involves using the finite state machines for the template space to rebuild a new finite state machine for the local address space. This fix-up involves computing expensive division and mod operations to determine the addresses of *accessed* elements in the local memory space. One other drawback of their technique when compared to our technique is the fact that their one-level pattern tables contain local memory gaps and not actual addresses. However their execution preserves lexicographic ordering and they do not incur any memory wastage.

Ancourt *et al.* [1] presented a linear algebraic framework to solve the problem of address generation for distributed memory machines. Their framework to solve two-level mapping involves a change of basis which leads to compression of holes but still incur some memory wastage. The node-code generated by this framework has complicated loops and incurs high execution overhead. However their execution order does not preserve lexicographic ordering.

Kaushik [13] uses a processor virtualization scheme to handle block-cyclic distributions. His method involves generation of addresses for with and without both hole compression. He first determines the amount of memory to be allocated using regular section characterization for *block* and *cyclic* distributions. He then extends this technique to virtual processor approach for handling *block-cyclic* distributions. However he cannot ensure lexicographic execution in the case of virtual cyclic approach. He also cannot guarantee a 100% compression and hence incur small amount of memory wastage.

## 4 Algorithms for two-level mapping

The algorithms proposed in this section solve the problem of generating addresses for a compressed space for two-level mapping. These algorithms exploit 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 [33].

The main idea behind these algorithms is to construct tables which store indices to switch from the template space to the local space. Since we do not allocate memory for *holes*, we have no memory wastage. We also do not incur high costs for generating access function to switch from the template space to local address space, this leads to faster execution times. 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 algorithms discussed in this paper can be broadly classified into two different methods. These algorithms differ mainly in the manner in which the tables are constructed in order to switch from the template space to the local address space. The first algorithm constructs a table of offsets whereas the algorithms in the second method uses different search techniques to locate the *accessed* elements in the set of *allocated* elements in the compressed space.

All algorithms first view 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 [33]. Two application of the one-level algorithm [33] with input strides being $a$ and $a * s$ in each case, generates the set of accesses for both *allocated* and *accessed* elements. Figure 2(a) shows the first set of repetition pattern of local addresses of the set of accesses for *allocated* elements. The numbers in boxes are the set of elements *accessed*, and the pattern of repetition of these elements is shown in Figure 2(b). In a compressed space we need to locate the position of these *accessed* elements, in the a set of *allocated* elements. So we record the positions of these entries in a separate table as shown in Figure 2(c). The main objective of these algorithms is to generate this switching table which helps in switching from the template space to the local compressed space. The construction of these switching tables is discussed in the following sections.

### 4.1 The ITABLE method

The main idea behind this algorithm is to construct a table of offsets, which is used to help switch from the non-compressed space to the compressed space. The algorithm exploits the repetition of accesses of both the *allocated* and the *accessed* elements. This algorithm first generates a two dimensional view of the set of accesses of the *allocated* and the *accessed* elements for the non-compressed space. This is done by the application of the one-level algorithm with input strides being $a$ and $a * s$ respectively. Recording the two-dimensional view of these sets does not incur any extra overhead such as expensive division or mod operations due

```
Input: Layout parameters (p,k), loop limits (ℓ,h), access
       stride s, alignment stride a for array A, processor m
Output: Two_level
Method:
 1  d_1 ← gcd(a, pk)
 2  d_2 ← gcd(a * s, pk)
 3  (length_a, Y_1, Y_2) ← one_level(p, k, a, m, d_1)
 4  (length_as, X_1, X_2) ← one_level(p, k, a * s, m, d_2)
 5  for i = 0, length_a − 1 do
 6      ITABLE[Y_2[i]] ← i
 7  enddo
 8  a_cou ← a / d_1
 9  for i = 0, length_as − 1 do
10      Two_Level[i] ← (X_1[i] / a_cou) * length_a + ITABLE[X_2[i]]
11  enddo
12  return Two_level
```

Figure 4: Algorithm that constructs the ITABLE for determining the two-level access pattern table

to the way we generate the set of accesses using the one-level algorithm.

Figures 3(a) and (b) lists the two-dimensional coordinates of the *accessed* and the *allocated* elements for the first set of pattern repetition for the example discussed previously. The second coordinates of both these sets indicate the offsets of the elements from the beginning of each course. A quick glance clearly indicates that the *accessed* elements are a subset of *allocated* elements. Using this information the algorithm first builds a table of offsets called the ITABLE for the first repetition pattern of allocated elements. The allocated access pattern repeats itself after every $\frac{a}{\gcd(a,pk)}$ courses. This table records the order in which the offsets of the *allocated* elements are accessed in lexicographic order.

The next stage involves using this table to determine the location of the *accessed* element in the compressed space. The problem now is to find two things. Firstly we need to determine the repetition block in which the *accessed* element is located. Secondly we need to find its position among a set of *allocated* elements in that particular repetition block. Finding the repetition block in which the element is located is straight forward, as we know the number of courses after which the set of *allocated* elements repeat and the length of this set. In order to find the position of the element in a list of *allocated* elements in a particular repetition block we need to index into the ITABLE which gives the position of the *accessed* element based on its offset from the beginning of the course. Hence by finding the repetition block in which the element exists and the position of the element in that block we can determine the local address of the element.

A detailed listing of the algorithm is as shown in the Figure 4. Lines 1–4 generate the pattern tables for the case when stride is $a$ and $a * s$. These tables record the two dimensional indices of elements accessed. $Y_1$ and $Y_2$ hold the two dimensional coordinates of the *allocated* elements while $X_1$ and $X_2$ hold the two dimensional coordinates for the *accessed* elements. Lines 5–7 construct the ITABLE which records the positions of offsets of *allocated* elements accessed in lexicographic order. The length of this table is always of size $k$. Lines 9–11 involves generation of the two-level pattern table. For each element in the *accessed* element set, a corresponding entry in the ITABLE will help determine the location of this element in the *allocated* set. The complexity of the algorithm is $O(\log \min(a, pk) + \log \min(as, pk) + \min(as, k))$.

Let us consider the example in Figure 1. We see that elements 0, 1, 6, 11 have offsets 0, 3, 2, 1 respectively from the beginning of the course. The two-dimensional coordinates for both *allocated* and *accessed* elements are as shown in Figures 3(a) and (b). Based on the entries in the $Y_2$ table, the ITABLE is constructed and is as shown in Figure 3(c). In this case the second coordinates of the *allocated* elements are same as that of the ITABLE, but in general the entries in the ITABLE depends on the value of $\gcd(a, pk)$. The ITABLE is always of size $k$, as there could be a maximum of $k$ elements for each pattern of repetition. In order to construct the *Two-Level* pattern table, let us consider the third entry $(5, 1)$ from the table of *accessed elements* as shown in Figure 3(b). This means that the accessed element lies in course number 5 and hence falls in the second repetition block of *allocated* elements. The value 1 in the second coordinate corresponds to the offset of the *accessed* element from the beginning of the course. This serves as an index into the ITABLE. Hence the value at position 1 of the ITABLE will yield 3 as shown in Figure 3(c). This value gives the position of the *accessed* element in that particular repetition block. Since we know the number of elements present in a single block ( which corresponds to 4 in this example), we can simply evaluate $4 * 1 + 3 = 7$, which gives us the position of the third element accessed in the compressed local space. Figure 3(d) shows the positions of *accessed* elements among a set of *allocated* elements for the first set of pattern repetition. The next sections deal with generating the set of *accessed* elements using search techniques.

### 4.2 Optimization of the ITABLE method

As can be seen from the algorithm in Figure 4, line 10 computes the $Two\_Level$ pattern table. However, line 10 of the algorithm is a big bottleneck since it includes one expensive integer division and an integer multiplication. Here, we explore the possibility of eliminating the expensive *div* and *multiplication* operations that hinder the performance of the ITABLE algorithm. The key point to note is that all the divisions performed are integer divisions. The division under question is $\frac{X_1[i]}{a_{cou}} * length_a$, where both $a_{cou}$ and $length_a$ are invariants. 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, which is a time versus space tradeoff. 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 5. The basic structure of the algorithm remains the same as that of the ITABLE algorithm presented in the previous subsection.

A detailed explanation of the algorithm follows. It may be noted that the main aim of the algorithm in lines 5–23 is to form a lookup table that will be used in place of the integer division and multiplication operations; this table is called the $lookup\_acc_1$ table 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 $\frac{X}{Y}$ is to be performed and the value of $X$ ranges from $a \cdots b$ for a fixed integer $Y$. So the minimum value of $X$ is $a$ and the maximum value is $b$. Let is consider specific values for $X$, $Y$, $a$ and $b$ for the purpose of illustration. 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 6.

Input: Layout parameters $(p, k)$, loop limits $(\ell, h)$, access stride $s$, alignment stride $a$ for array $A$, processor $m$
Output: $Two\_level$
Method:

1 $d_1 \leftarrow \gcd(a, pk)$
2 $d_2 \leftarrow \gcd(a * s, pk)$
3 $(length_a, Y_1, Y_2) \leftarrow one\_level(p, k, a, m, d_1)$
4 $(length_{as}, X_1, X_2) \leftarrow one\_level(p, k, a * s, m, d_2)$
5 $a_{cou} \leftarrow \frac{a}{d_1}$
6 $first \leftarrow X_1[0]$
7 $tmp_1 \leftarrow length_a * \frac{first}{a_{cou}}$
8 $tmp_2 \leftarrow first \bmod a_{cou}$
9 **for** $i = tmp_2$ **to** $a_{cou} - 1$ **do**
10 $\quad lookup\_acc_1[first] \leftarrow tmp_1$
11 $\quad first \leftarrow first + 1$
12 **enddo**
13 $tmp_1 \leftarrow tmp_1 + length_a$
14 $last \leftarrow X_1[length_{as} - 1]$
15 **while** $(first \leq last)$ **do**
16 $\quad i \leftarrow 0$
17 $\quad$ **while** $(i < a_{cou}$ and $first \leq last)$ **do**
18 $\quad\quad lookup\_acc_1[first] \leftarrow tmp_1$
19 $\quad\quad i \leftarrow i + 1$
20 $\quad\quad first \leftarrow first + 1$
21 $\quad$ **enddo**
22 $\quad tmp_1 \leftarrow tmp_1 + length_a$
23 **enddo**
24 **for** $i = 0$ **to** $length_a - 1$ **do**
25 $\quad \text{ITABLE}[Y_2[i]] \leftarrow i$
26 **enddo**
27 **for** $i = 0$ **to** $length_{as} - 1$ **do**
28 $\quad Two\_Level[i]$
$\quad\quad \leftarrow lookup\_acc_1[X_1[i]] + \text{ITABLE}[X_2[i]]$
29 **enddo**
30 **return** $Two\_level$

Figure 5: Faster algorithm to compute the ITABLE for determining the two-level access pattern by substituting integer divides with table lookups

As can be seen from the figure, if a series of divisions have to be performed 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 $first$. $tmp_1$ 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; this is space versus time tradeoff. 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 experimental results (Section 5). This method is not only useful for two-level mapping address generation problem but also useful as a generic technique whenever integer divisions are required.

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

(a)

| | | | | |
|---|---|---|---|---|
| ALLOCATED | 0 | 3 | 6 | 9 |

(b)

| | | | | |
|---|---|---|---|---|
| ACCESSED | 0 | 6 | 21 | 27 |

(c)

| INDEX | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ALLOCATED | 0 | 3 | 6 | 9 | 12 | 15 | 18 | 21 | 24 | 27 | 30 |

(d)

| | | | | |
|---|---|---|---|---|
| PATTERN | 0 | 2 | 7 | 9 |

Figure 7: Local addresses of *allocated* and *accessed* elements, the replicated *allocated* table, along with two-level access pattern when $a = 3$, $s = 3$, $p = 4$, $k = 4$ and $l = 0$

## 4.3 Search-based algorithms

The main idea behind determining the local addresses of *accessed elements*, is to find the location of *accessed* elements in a list of *allocated* elements (expanded to accommodate the longest element in the accessed set), since *accessed* elements are a subset of *allocated* elements. This can be achieved by using a naive approach by searching for the index of the *accessed* element in the list of *allocated* elements. In this section we propose new search methods which exploit the property that the list of elements are in sorted order.

The first step in performing these methods is to run the one-level algorithm to obtain the local addresses of the set of accesses for the first pattern of repetition for both *allocated* and *accessed* elements. These entries are in lexicographic order and are assumed to be stored in $pattern_a$ and $pattern_{as}$ tables respectively. These tables are as shown in Figures 7(a) and (b) for the example in Figure 1. The entries in these table correspond to the local addresses in a non-compressed template space. The table size of the former depends on $\frac{k}{\gcd(a,pk)}$, whereas that of the latter table depends on $\frac{k}{\gcd(as,pk)}$.

Here we see that not all elements in the *accessed* set are present in the *allocated* set for the first pattern of repetition. This is due to the fact that the *accessed* elements lie in different repetition blocks of the local address space. Hence we need to expand the set of *allocated* elements in order to represent all the elements in the *accessed* table. The total number of elements in the first repetition block of *allocated* table in the uncompressed space is $\frac{ak}{\gcd(a,pk)}$ whereas the total number of elements for the *accessed* table is $\frac{ask}{\gcd(as,pk)}$. Hence the factor needed to expand the *allocated* elements table is $s\frac{\gcd(a,pk)}{\gcd(as,pk)}$. Performing the required expansion is straight forward. It involves replicating the first set of *allocated* elements as many times depending on the factor of replication. This is accomplished by copying elements one at a time from the first pattern of *allocated* elements to the extended memory space with a suitable increment. One another technique can be implemented where replication is performed using a demand driven approach.

A search now has to be performed to locate the position of an *accessed* element in this new replicated table. Figure 7(c) shows the replicated table after expansion for the example discussed previously. The factor for replication in this case was found to be 3. Since the length of the table which holds the addresses of the *accessed* elements is never greater than the length of the table which holds the *allocated* elements, the algorithm needs to find the locations of common elements from two tables of different size. Several search algorithms can be implemented for finding the locations of *accessed* elements. We discuss several such algorithms in the following sections. These algorithms differ mainly in their complexities and the speed of execution. We also propose a search technique to find the set of *accessed* elements by not having to perform replication.

### 4.3.1 Linear search

The algorithm for linear search builds the two-level pattern table needed to switch from the template space to the local address space. Figure 8 lists the complete algorithm. Lines 1–4 discusses the build up of the *accessed* and *allocated* table. The next step is to find the factor for replication and is as shown in Line 5. This factor is used to replicate the *allocated* table. Once replication is performed, we now need to perform a simple search in order to locate the positions of each *accessed* element in the *allocated* table. Lines 8–14 shows the search algorithm. For each entry in the *accessed* table, it determines the location of this element in the replicated *allocated* set. As and when the location is determined, the position is recorded into a *Two-Level* pattern table. The entries in this table reflects the local address of the *accessed* element in the compressed space.

Figure 7 can be used to explain the functioning of this

Input: Layout parameters $(p,k)$, loop limits $(\ell,h)$, access stride $s$, alignment stride $a$ for array $A$, processor $m$
Output: $Two\_level$
Method:

1 $d_1 \leftarrow \gcd(a,pk)$
2 $d_2 \leftarrow \gcd(as,pk)$
3 $(length_a, pattern_a) \leftarrow one\_level(p,k,a,m,d_1)$
4 $(length_{as}, pattern_{as}) \leftarrow one\_level(p,k,a*s,m,d_2)$
5 Factor of replication $f \leftarrow \frac{\frac{ask}{d_2}}{\frac{ak}{d_1}}$
6 Replicate $pattern_a$ by factor $f$
7 $i \leftarrow 0; j \leftarrow 0$
8 **while** $(j < length_{as})$ **do**
9     **if** $(pattern_{as}[j] = pattern_a[i])$ **then**
10         $Two\_level[j] \leftarrow i$
11         $j \leftarrow j+1$
12     **endif**
13     $i \leftarrow i+1$
14 **enddo**
15 **return** $Two\_level$

Figure 8: Algorithm which constructs a two-level access pattern table using linear search

algorithm. Let us consider the element 21 from the *accessed* set as shown in Figure 7(b). The search involves finding the position of this element in the replicated set as shown in Figure 7(c). This element can be found at location 7. This entry is then stored in the *Two-level* pattern table. The complete pattern table for the example is as shown in Figure 7(d). Since the search is performed on a sorted table of length $f * len_a$ and no element of this table is accessed more than once, the complexity of the algorithm is $O(f * len_a)$.

### 4.3.2 Binary search

If the factor of replication becomes very large, then the previous algorithm suffers from the time taken to perform the search. In this case it is better to resort to faster techniques for searching such as the binary search after replication. Since the table is sorted, the search for the next entry begins at the location where the last entry was found. This algorithm has a complexity of $O(len_{as} * \log(f * len_a))$ which includes the complexity involved in the binary search.

The straight forward binary search technique can be improved by reducing the amount of search, each time we encounter a common element. Exploiting the fact that the elements in the replicated table are in sorted order and are stored in different repetition blocks of length $length_a$ (length of the $pattern_a$ table), we first compute the repetition block in which the element we are searching exists. Secondly knowing the repetition block, we know that the element will surely lie within the next $len_a$ elements. We then perform a binary search within this block in order to find the location of the element. This leads to faster search techniques and reduced complexity. Since the search is limited only to the length of the $pattern_a$ table, the complexity of this algorithm is $O(len_{as} * \log len_a)$.

The complete algorithm is shown in Figure 9. Line 11 in the algorithm shows the process of obtaining the repetition block in which the element exists. If $val$ is the value we are searching, the repetition block in which this element exists is

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

```
d1 ← gcd(a, pk)
d2 ← gcd(as, pk)
(length_a, pattern_a) ← one_level(p, k, a, m, d1)
(length_as, pattern_as) ← one_level(p, k, a * s, m, d2)
Factor of replication f ← (s / d2) * d1
Replicate pattern_a by a factor f
acou ← a / d1
low ← 0
for j = 0, length_as − 1 do
    val ← pattern_as[j]
    low ← ((val / k) / acou) × len_a
    high ← low + len_a − 1
    while (low ≤ high) do
        mid ← (high + low) / 2
        midval ← pattern_a[mid]
        if (val < midval) then
            high ← mid − 1
        else if (val > midval) then
            low ← mid + 1
        else
            Two_level[j] ← mid
        endif
    enddo
enddo
return Two_level
```

Figure 9: Algorithm to construct a pattern table using the modified binary search method

given by $\frac{\frac{val}{k}}{acou}$ where *acou* is $\frac{a}{\gcd(a,pk)}$. The remaining parts of the algorithm is exactly similar to the one discussed in the previous section.

### 4.3.3 Search algorithms with no replication

As discussed before, all the above mentioned search algorithms require the set of *allocated* elements to be replicated in order to determine the position of the *accessed* element. Replication in general depends on the individual values of $\gcd(a, pk)$ and $\gcd(a * s, pk)$. If the factor of replication is large, than the time for replication can be more than the time for search rendering the search algorithms inefficient.

Given an element from the *accessed* set we can determine the repetition block in which it exists. We can also find the offset (memory gap) of this element from the beginning of this repetition block. The offset we obtain will definitely match an entry in the pattern table for the first block of repetition of *allocated* elements. Hence knowing the address of an *accessed* element we need to just search for the location of the offset of this element in the pattern table of *allocated* elements. This completely avoids replication of the *allocated* set and hence saves a lot on runtime overheads. The search within this pattern table can be accomplished using a simple or a binary search.

Figure 10 lists the complete algorithm. Line 9 shows the factor to determine the repetition block in which the element exists. Line 10 provides the offset of the *accessed* element within the repetition block. Line 11 computes the memory gap between the *accessed* element and the corresponding entry in the first pattern of repetition of the *allocated* set. Consider element 27 in Figure 7(b). If we need to find its location in the local template space, we first need to map this element into the first block of repetition of the *allocated* set as shown in Figure 7(a). This is done by first finding the repetition block in which this element exists. In this case the number of the repetition block is equal to 2. The offset of this element within this block is 3 which matches the second entry in the *allocated* set. We now need to search for the position of this element in the *allocated* table. we use binary search within the first block of repetition to search for element 2. Once the location of 3 which is 1 is recorded, we now need to determine the actual location of the element. Since we know the repetition block (2) and the number of elements in each block we obtain the actual location by performing $4 * 2 + 1 = 9$ which gives the local address of the *accessed element*.

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

```
1   d1 ← gcd(a, pk)
2   d2 ← gcd(as, pk)
3   (length_a, pattern_a) ← one_level(p, k, a, m, d1)
4   (length_as, pattern_as) ← one_level(p, k, a * s, m, d2)
5   acou ← a / d1
6   incr ← ak / d1
7   for j = 0, length_as − 1 do
8       val ← pattern_as[j]
9       factor ← (val / k) / acou
10      val ← val − factor * incr
11      offset ← factor * length_a
12      low ← 0; high ← len_a − 1
13      while (low ≤ high) do
14          mid ← (high + low) / 2
15          midval ← pattern_a[mid]
16          if (val < midval) then
17              high ← mid − 1
18          else if (val > midval) then
19              low ← mid + 1
20          else
21              Two_level[j] ← mid + offset
22          endif
23      enddo
24  enddo
25  return Two_level
```

Figure 10: Algorithm which constructs a two-level access pattern table using binary search without performing replication

## 5 Experimental results for two-level mapping

In order to compare all the above mentioned methods, we ran our experiments on a varying number of problem parameters. These experiments were done 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. We fixed the number of processors to $p = 32$ in all our experiments. For each value

Table 1: Table generation times (in $\mu s$) for two-level mapping for $p = 32$, $a = 2$

| $k$ | s=3 | | | | | s=5 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | ITABLE* | ITABLE | SEARCH | NOREP | RIACS | ITABLE* | ITABLE | SEARCH | NOREP | RIACS |
| 4 | 2 | 2 | 1 | 3 | 18 | 2 | 2 | 1 | 3 | 18 |
| 8 | 2 | 3 | 2 | 6 | 26 | 2 | 3 | 2 | 6 | 26 |
| 16 | 2 | 6 | 2 | 12 | 43 | 2 | 6 | 4 | 12 | 43 |
| 32 | 4 | 11 | 4 | 24 | 81 | 4 | 11 | 6 | 25 | 79 |
| 64 | 6 | 21 | 7 | 51 | 153 | 6 | 21 | 12 | 51 | 150 |
| 128 | 10 | 42 | 14 | 107 | 297 | 10 | 42 | 23 | 107 | 299 |
| 256 | 21 | 85 | 28 | 222 | 589 | 21 | 83 | 46 | 219 | 586 |

| $k$ | s=7 | | | | | s=11 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | ITABLE* | ITABLE | SEARCH | NOREP | RIACS | ITABLE* | ITABLE | SEARCH | NOREP | RIACS |
| 4 | 2 | 2 | 2 | 3 | 18 | 2 | 2 | 2 | 3 | 18 |
| 8 | 2 | 3 | 3 | 6 | 26 | 2 | 3 | 3 | 6 | 26 |
| 16 | 2 | 6 | 4 | 12 | 43 | 2 | 6 | 6 | 12 | 44 |
| 32 | 4 | 11 | 8 | 25 | 79 | 4 | 11 | 12 | 25 | 79 |
| 64 | 6 | 21 | 16 | 51 | 150 | 6 | 21 | 23 | 52 | 153 |
| 128 | 10 | 42 | 32 | 107 | 299 | 11 | 42 | 49 | 107 | 299 |
| 256 | 21 | 85 | 64 | 223 | 588 | 21 | 85 | 99 | 223 | 589 |

| $k$ | s=23 | | | | | s=99 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | ITABLE* | ITABLE | SEARCH | NOREP | RIACS | ITABLE* | ITABLE | SEARCH | NOREP | RIACS |
| 4 | 2 | 2 | 2 | 3 | 18 | 2 | 2 | 6 | 3 | 18 |
| 8 | 2 | 3 | 5 | 6 | 26 | 2 | 3 | 12 | 6 | 26 |
| 16 | 2 | 6 | 9 | 12 | 44 | 2 | 6 | 23 | 12 | 45 |
| 32 | 4 | 11 | 19 | 25 | 81 | 4 | 11 | 47 | 25 | 80 |
| 64 | 6 | 22 | 40 | 51 | 154 | 6 | 22 | 97 | 52 | 154 |
| 128 | 10 | 43 | 85 | 108 | 299 | 10 | 43 | 210 | 108 | 300 |
| 256 | 21 | 85 | 183 | 224 | 590 | 22 | 86 | 446 | 226 | 594 |

Table 2: Table generation times (in $\mu s$) for two-level mapping for $p = 32$, $a = 3$

| $k$ | s=3 | | | | | s=5 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | ITABLE* | ITABLE | SEARCH | NOREP | RIACS | ITABLE* | ITABLE | SEARCH | NOREP | RIACS |
| 4 | 2 | 3 | 2 | 6 | 28 | 2 | 3 | 2 | 6 | 28 |
| 8 | 3 | 6 | 3 | 12 | 44 | 3 | 6 | 4 | 12 | 43 |
| 16 | 4 | 11 | 4 | 24 | 81 | 4 | 11 | 6 | 24 | 80 |
| 32 | 7 | 21 | 8 | 50 | 152 | 7 | 21 | 12 | 51 | 150 |
| 64 | 11 | 42 | 14 | 107 | 297 | 10 | 42 | 23 | 104 | 298 |
| 128 | 21 | 85 | 28 | 223 | 588 | 21 | 83 | 46 | 222 | 591 |
| 256 | 42 | 169 | 54 | 463 | 1167 | 43 | 169 | 90 | 458 | 1171 |

| $k$ | s=7 | | | | | s=11 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | ITABLE* | ITABLE | SEARCH | NOREP | RIACS | ITABLE* | ITABLE | SEARCH | NOREP | RIACS |
| 4 | 2 | 3 | 3 | 6 | 28 | 2 | 3 | 3 | 6 | 26 |
| 8 | 3 | 6 | 5 | 12 | 44 | 3 | 6 | 6 | 12 | 44 |
| 16 | 4 | 11 | 8 | 25 | 79 | 4 | 11 | 12 | 25 | 80 |
| 32 | 7 | 21 | 16 | 51 | 151 | 7 | 21 | 23 | 51 | 154 |
| 64 | 11 | 42 | 32 | 108 | 298 | 11 | 42 | 50 | 108 | 300 |
| 128 | 21 | 83 | 64 | 221 | 588 | 22 | 85 | 99 | 223 | 589 |
| 256 | 42 | 169 | 128 | 463 | 1173 | 42 | 169 | 203 | 464 | 1171 |

| $k$ | s=23 | | | | | s=99 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | ITABLE* | ITABLE | SEARCH | NOREP | RIACS | ITABLE* | ITABLE | SEARCH | NOREP | RIACS |
| 4 | 2 | 3 | 5 | 6 | 26 | 2 | 3 | 12 | 6 | 26 |
| 8 | 3 | 6 | 9 | 12 | 44 | 3 | 6 | 23 | 12 | 44 |
| 16 | 4 | 11 | 19 | 25 | 81 | 4 | 11 | 47 | 25 | 82 |
| 32 | 7 | 21 | 40 | 51 | 152 | 7 | 22 | 98 | 52 | 151 |
| 64 | 11 | 43 | 84 | 109 | 299 | 11 | 43 | 208 | 108 | 302 |
| 128 | 21 | 85 | 179 | 225 | 591 | 22 | 86 | 447 | 226 | 595 |
| 256 | 44 | 170 | 411 | 463 | 1173 | 44 | 172 | 932 | 469 | 1181 |

Table 3: Table generation times (in $\mu s$) for two-level mapping for $p = 32$, $a = 5$

| $k$ | s=3 | | | | | s=5 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | ITABLE* | ITABLE | SEARCH | NOREP | RIACS | ITABLE* | ITABLE | SEARCH | NOREP | RIACS |
| 4 | 2 | 3 | 1 | 6 | 26 | 2 | 3 | 2 | 6 | 27 |
| 8 | 3 | 6 | 2 | 12 | 44 | 3 | 6 | 4 | 12 | 44 |
| 16 | 4 | 11 | 4 | 24 | 80 | 4 | 11 | 7 | 25 | 81 |
| 32 | 7 | 21 | 8 | 51 | 153 | 7 | 21 | 12 | 51 | 151 |
| 64 | 11 | 42 | 15 | 107 | 298 | 11 | 42 | 23 | 107 | 299 |
| 128 | 21 | 85 | 28 | 220 | 589 | 21 | 85 | 46 | 220 | 589 |
| 256 | 42 | 169 | 54 | 462 | 1169 | 42 | 169 | 90 | 462 | 1171 |

| $k$ | s=7 | | | | | s=11 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | ITABLE* | ITABLE | SEARCH | NOREP | RIACS | ITABLE* | ITABLE | SEARCH | NOREP | RIACS |
| 4 | 2 | 3 | 3 | 6 | 26 | 2 | 3 | 3 | 6 | 26 |
| 8 | 3 | 6 | 4 | 12 | 44 | 3 | 6 | 6 | 12 | 44 |
| 16 | 4 | 11 | 8 | 25 | 79 | 4 | 11 | 12 | 25 | 81 |
| 32 | 7 | 21 | 17 | 52 | 154 | 7 | 21 | 23 | 51 | 152 |
| 64 | 11 | 42 | 32 | 107 | 298 | 11 | 42 | 49 | 107 | 300 |
| 128 | 21 | 85 | 64 | 222 | 589 | 21 | 85 | 99 | 223 | 592 |
| 256 | 42 | 169 | 129 | 463 | 1174 | 42 | 169 | 203 | 460 | 1173 |

| $k$ | s=23 | | | | | s=99 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | ITABLE* | ITABLE | SEARCH | NOREP | RIACS | ITABLE* | ITABLE | SEARCH | NOREP | RIACS |
| 4 | 2 | 3 | 5 | 6 | 26 | 2 | 3 | 12 | 6 | 26 |
| 8 | 3 | 6 | 9 | 12 | 44 | 3 | 6 | 23 | 12 | 44 |
| 16 | 4 | 11 | 19 | 25 | 79 | 4 | 11 | 47 | 25 | 81 |
| 32 | 7 | 22 | 39 | 52 | 154 | 7 | 22 | 97 | 52 | 152 |
| 64 | 12 | 43 | 84 | 108 | 300 | 12 | 43 | 210 | 109 | 302 |
| 128 | 21 | 85 | 179 | 225 | 593 | 21 | 84 | 448 | 227 | 596 |
| 256 | 43 | 170 | 409 | 462 | 1176 | 44 | 172 | 932 | 470 | 1178 |

Table 4: Table generation times (in $\mu s$) for two-level mapping for $p = 32$, $a = 9$

| $k$ | s=3 | | | | | s=5 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | ITABLE* | ITABLE | SEARCH | NOREP | RIACS | ITABLE* | ITABLE | SEARCH | NOREP | RIACS |
| 4 | 2 | 3 | 1 | 6 | 26 | 2 | 3 | 2 | 6 | 26 |
| 8 | 3 | 6 | 2 | 12 | 45 | 3 | 6 | 4 | 12 | 45 |
| 16 | 4 | 11 | 4 | 24 | 81 | 4 | 11 | 6 | 24 | 80 |
| 32 | 7 | 21 | 8 | 51 | 150 | 7 | 21 | 12 | 51 | 153 |
| 64 | 11 | 42 | 15 | 107 | 299 | 11 | 42 | 23 | 106 | 299 |
| 128 | 21 | 85 | 28 | 222 | 587 | 21 | 85 | 46 | 220 | 591 |
| 256 | 42 | 169 | 54 | 461 | 1167 | 42 | 169 | 90 | 464 | 1167 |

| $k$ | s=7 | | | | | s=11 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | ITABLE* | ITABLE | SEARCH | NOREP | RIACS | ITABLE* | ITABLE | SEARCH | NOREP | RIACS |
| 4 | 2 | 3 | 2 | 6 | 26 | 2 | 3 | 3 | 6 | 26 |
| 8 | 3 | 6 | 5 | 12 | 44 | 3 | 6 | 6 | 12 | 44 |
| 16 | 4 | 11 | 9 | 25 | 79 | 4 | 11 | 12 | 25 | 81 |
| 32 | 7 | 21 | 17 | 52 | 153 | 7 | 21 | 23 | 51 | 151 |
| 64 | 11 | 42 | 32 | 105 | 299 | 11 | 42 | 50 | 108 | 299 |
| 128 | 21 | 85 | 64 | 224 | 592 | 21 | 85 | 98 | 223 | 594 |
| 256 | 42 | 169 | 128 | 462 | 1172 | 42 | 169 | 203 | 461 | 1173 |

| $k$ | s=23 | | | | | s=99 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | ITABLE* | ITABLE | SEARCH | NOREP | RIACS | ITABLE* | ITABLE | SEARCH | NOREP | RIACS |
| 4 | 2 | 3 | 5 | 6 | 26 | 2 | 3 | 12 | 6 | 26 |
| 8 | 3 | 6 | 9 | 12 | 45 | 3 | 6 | 24 | 12 | 44 |
| 16 | 4 | 11 | 19 | 25 | 80 | 4 | 11 | 47 | 25 | 82 |
| 32 | 7 | 21 | 40 | 52 | 154 | 7 | 22 | 99 | 52 | 152 |
| 64 | 11 | 43 | 87 | 108 | 300 | 11 | 43 | 209 | 108 | 303 |
| 128 | 21 | 85 | 183 | 224 | 592 | 22 | 86 | 449 | 222 | 598 |
| 256 | 42 | 170 | 408 | 463 | 1181 | 43 | 172 | 928 | 468 | 1185 |

of alignment stride $a$, we varied both the block-size $k$ and the access stride $s$. The method which builds a table of offset is termed as ITABLE. The optimized version of ITABLE, i.e., the version that replaces extensive divisions by table lookups described in Section 4.2 is referred to as ITABLE*. The best of the search algorithms which performs replication was chosen for the results. It is termed as SEARCH in the tables. The search algorithm which does not perform replication is termed as NOREP in the results. The method due to Chatterjee *et al.* [5] is termed as RIACS. Tables 1–4 show the time it takes to build the two-level pattern tables.

The results indicate that the times taken by all the above mentioned methods depend on the value of $k$, $s$ and $a$. If the access and alignment strides are small, the ITABLE* and the SEARCH techniques are competitive; this is because the time taken for replication and the overhead in performing a search is very minimal. But as $s$ and $k$ increases we notice that the SEARCH starts performing worse. This is because as $s$ and $k$ start increasing the time for replication in the SEARCH dominates over search and renders this method inefficient. The construction of the ITABLE forms the major part of time taken for two-level pattern build up. This construction is purely a function of $a$ and $k$ and not of $s$. Hence as $s$ increases the times for ITABLE does not vary widely. The method ITABLE* performs the best over a wide range of parameters.

The method by RIACS suffers with large block sizes due to the expensive runtime overheads. The NOREP method performs better than the SEARCH method as $s$ starts increasing. This is due to the fact that we do not pay the overhead due to replication. But for large $k$ we see that the times for search increases rendering NOREP inefficient.

## 6 Other Problems in Code Generation

In this section, we provide an overview of other work from our group on several problems in code generation and runtime support for data-parallel languages. These include our work on communication generation, code generation for complex subscripts, runtime data structures, support for operations on regular sections and array redistribution.

### 6.1 Communication generation

In addition to problems in address generation, we have explored techniques for communication generation and optimization [34, 36, 37, 38]. A compiler for languages such as HPF that generates node code (for each processor) has also to compute the sequence of sends and receives for a given processor to access non-local data. While the address generation problem has received much attention, issues in communication have not been dealt with extensively. A novel approach for the management of communication sets and strategies for local storage of remote references is presented in [37, 36]. In addition to algorithms for deriving communication patterns [34, 36, 38], two schemes that extend the notion of a local array by providing storage for non-local elements (called overlap regions) interspersed throughout the storage for the local portion are presented [37, 36]. The two schemes, namely course padding and column padding enhance locality of reference significantly at the cost of a small overhead due to unpacking of messages. The performance of these schemes are compared to the traditional buffer-based approach and improvements of up to 30% in total time are demonstrated. Several message optimizations such as offset communication, message aggregation and coalescing are also discussed.

### 6.2 Union and difference of regular sections

Operations on regular sections are very common in code generation. The intersection operation on regular sections is easy (since regular sections are closed under intersection). Union and difference of regular sections are needed for efficient generation of communications sets; unfortunately, regular sections are not closed under union and difference operations. We [7, 22] present an efficient runtime technique for supporting support for union and other operations on regular sections. These deal with both the generation of the pattern under these operations as well as with the efficient code that enumerates the resulting sets using the patterns.

### 6.3 Code generation for complex subscripts

The techniques presented in this chapter assumed simple subscript functions. Array references with arbitrary affine subscripts can make the task of compilers for such languages highly involved. Work from our group [7, 21, 24, 25, 36] deals with the efficient address generation in programs with array references having two types of commonly encountered affine references, namely coupled subscripts and subscripts containing multiple induction variables (MIVs). These methods utilize the repetitive pattern of the memory accesses. In the case of MIV, we address this issue by presenting runtime techniques which enumerate the set of addresses in lexicographic order. Our approach to the problem incorporates a general approach of computing in $O(k)$ time, the start element on a processor for a given global start element. Several methods are proposed and evaluated here for generating the access sequences for MIV based on problem parameters. With coupled subscripts, we present two construction techniques, namely searching and hashing which minimize the time needed to construct the tables. Extensive experiments were conducted and the results were then compared with other approaches to demonstrate the efficiency of our approach.

### 6.4 Data structures for runtime efficiency

In addition to algorithms for address sequence generation, we addressed the problem of how best to use the address sequences in [6, 7]. 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 [15] so far addresses the effect of the data structures used in representing the access sequence on the execution time. In [6, 7], we present several new data structures along with node code that is suitable for both DO loops and FORALL constructs. The methods, namely *strip-mining* and *table compression* facilitate the generation of time-efficient code for execution on each processor. While strip-mining views the problem as a double nested loop, table compression proves to be a worthwhile data structure for faster execution. The underlying theory behind the data structures introduced is explained and their effects on all possible set of problem parameters is observed. Extensive experimental results show the efficacy of our approach. The results compare very favorably with the results of the earlier methods proposed by Kennedy et al. [14] and Chatterjee et al. [5].

### 6.5 Array redistribution

Array redistribution is used in languages such as High Performance Fortran to dynamically change the distribution of arrays across processors. Performing array redistribution incurs two overheads: (1) an *indexing overhead* for determining the set of processors to communicate with and the array elements to be communicated, and (2) a *communication overhead* for performing the necessary irregular all-to-many personalized communication. We have presented efficient runtime methods for performing array redistribution [12, 30]. In order to reduce the indexing overhead, precise closed forms for enumerating the processors to communicate with and the array elements to be communicated are developed for two special cases of array redistribution involving block-cyclically distributed arrays. The general array redistribution problem for block-cyclically distributed arrays can be expressed in terms of these special cases. Using the developed closed forms, a distributed algorithm for scheduling the irregular communication for redistribution is developed. The generated schedule eliminates node contention and incurs the least communication overhead. The scheduling algorithm has an asymptotically lower scheduling overhead than techniques presented in the literature. Following this, we have developed efficient table-based runtime techniques (based on integer lattices) that incur negligible cost [7, 23].

## 7 Summary

The success of data parallel languages such as High Performance Fortran and Fortran D critically depends on efficient compiler and runtime support. In this paper, we first presented efficient compiler algorithms for generating local memory access patterns for the various processors (node code) given a general (e.g., non-unit-stride) alignment of arrays to a template and a `CYCLIC(k)` distribution of the template onto the processors. Having the advantage of saving on memory wastage and faster construction of table we have shown in this paper that the method which constructs a table of offsets performs very efficiently for a wide range of values of block size and access strides. In addition to these algorithms, we provided an overview of other work from our group on several problems such as

- communication generation [34, 36, 37, 38];
- code generation for complex subscripts [7, 21, 24, 25, 36];
- effect of data structures for table lookup at runtime [6, 7];
- runtime array redistribution [7, 12, 23, 30]; (and)
- efficient support for union and other operations on regular sections [7, 22].

Work is in progress on the problem of code generation and optimization for general affine access functions in whole programs.

## References

[1] A. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution. *Scientific Programming,* 6(1):3–28, Spring 1997.

[2] P. Banerjee, J. Chandy, M. Gupta, E. W. Hodges, J. Holm, A. Lain, D. Palermo, S. Ramaswamy, and E. Su. The PARADIGM compiler for distributed-memory multicomputers. *IEEE Computer,* 28(10):37–47, October 1995.

[3] 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.

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

[5] 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.

[6] 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, December 1996.

[7] S. Dutta. Compilation and run-time techniques for data-parallel programs. M.S. Thesis, Department of Electrical and Computer Engineering, Louisiana State University, December 1997.

[8] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D Language Specification. Technical Report CRPC-TR90079, Center for Research on Parallel Computation, Rice University, December 1990.

[9] P. Gruber and C. 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,* 32(2):155–172, February 1996.

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

[12] S. Kaushik, C. Huang, J. Ramanujam, and P. Sadayappan. Multiphase array redistribution: Modeling and Evaluation. Technical Report OSU-CISRC-9/94-TR52, Department of Computer and Information Science, The Ohio State University., September 1994. A short version appears in *Proc. 9th International Parallel Processing Symposium,* Santa Barbara, CA, pages 441–445, April 1995.

[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] 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, pages 102–111, July 1995.

[15] K. Kennedy, N. Nedeljkovic, and A. Sethi. Efficient address generation for block-cyclic distributions. In *Proc. ACM International Conference on Supercomputing,* Madrid, Spain, pages 180–184, July 1995.

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

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

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

[19] J. Ramanujam. Efficient computation of basis vectors of the address sequence lattice. Submitted for publication, 1997.

[20] J. Ramanujam. Integer lattice based methods for local address generation for block-cyclic distributions. To appear in *Languages, Compilation Techniques and Run Time Systems for Scalable Parallel Systems,* D. Agrawal and S. Pande (Eds.), Lecture Notes in Computer Science, Springer-Verlag, 1998.

[21] J. Ramanujam and S. Dutta. Code generation for coupled subscripts with block-cyclic distributions. Technical Report TR-96-07-01, Dept. of Elec. & Comp. Engineering, Louisiana State University, July 1996.

[22] J. Ramanujam and S. Dutta. Runtime solutions to operations on regular sections. Technical Report TR-96-12-03, Dept. of Elec. & Comp. Engineering, Louisiana State University, December 1996.

[23] J. Ramanujam and S. Dutta. Efficient runtime array redistribution. Technical Report TR-97-01-01, Dept. of Elec. & Comp. Engineering, Louisiana State University, January 1997.

[24] J. Ramanujam, S. Dutta, and A. Venkatachar. Code generation for complex subscripts in data-parallel programs. In *Languages and Compilers for Parallel Computing,* Z. Li et al. (Editors), Lecture Notes in Computer Science, Vol. 1366, pages 49–63, Springer-Verlag, 1998.

[25] J. Ramanujam and A. Venkatachar. Code generation for complex subscripts with multiple induction variables in the presence of block-cyclic distributions. Technical Report TR-96-03-01, Dept. of Elec. & Comp. Engineering, Louisiana State University, March 1996.

[26] J. Ramanujam, A. Venkatachar, and S. Dutta. Efficient address sequence generation for two-level mappings in High Performance Fortran. Submitted for publication, 1998.

[27] C. van Reeuwijk, H. Sips, W. Denissen, and E. Paalvast. An implementation framework for HPF distributed arrays on message-passing parallel computer systems. *IEEE Transactions on Parallel and Distributed Systems,* 7(9):897–914, September 1996.

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

[29] J. Stichnoth, D. O'Hallaron, and T. Gross. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing,* 21(1):150–159, April 1994.

[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. 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.

[33] 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.

[34] 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, pages 311–316, Norwell, MA, 1996.

[35] 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.

[36] 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.

[37] 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.

[38] A. Venkatachar, J. Ramanujam, and A. Thirumalai. Communication generation for block-cyclic distributions. *Parallel Processing Letters,* 7(2):195–202, June 1997.