Efficient Generation of Local Index Sets for Distributed Arrays

Deborah K. Weisser *

Abstract

An important component of parallel programs with distributed data structures is local address generation for various index patterns. In this paper we present a general approach as well as two specific algorithms for common problems to perform this task quickly on distributed arrays. Our algorithms usually run in $O(|output|)$ time, which is optimal. We have implemented our algorithms. When the problem size is small, they perform as well as or better than naive algorithms. When the problem size is large, they outperform naive algorithms by several orders of magnitude.

1 Introduction

Many parallel programs are built upon some type of parallel loop over distributed data structures, oftentimes multi-dimensional arrays. Because of resource allocation issues, such as load balancing, the distribution of data is often complex. A combination of these complex distributions and various access patterns make calculating indices a non-trivial overhead in parallel programs, particularly since, as remote access time decreases, it tends less to dominate running time. This paper presents a technique which reduces this overhead to its theoretical lower bound and can be implemented efficiently.

We describe a framework for generating regular index patterns for any number of problems, e.g. regular section, lower triangular, wavefront, and tri-diagonal. We present the specifics for algorithms to generate local addresses for regular section and lower triangular computations.

The first problem we discuss is that of computing the *regular section* of an array. The regular section $A(\mathit{offset}, \mathit{max\_index}, \mathit{stride})$ is the set of elements $A(\mathit{offset} + k * \mathit{stride})$ where $\mathit{stride} > 0$, $\mathit{offset} \geq 0$, and $0 \leq k \leq \frac{\mathrm{max\_index} - \mathrm{offset}}{\mathrm{stride}}$.

The second problem is that of outputting the array indices of a lower triangular matrix given a slope. For example, if A is an n x m array, the lower triangular matrix is the elements $A(i, j)$, where $i \geq \frac{m}{n} j$.

Our algorithms achieve the theoretical lower bound when $\mathit{stride} \leq b$, where b is the block size, and are close to the lower bound when $\mathit{stride} > b$. In addition, they can be implemented efficiently and perform well in practice, achieving speedups from several times to several orders of magnitude.

At the heart of any index computation is the data layout. We examine array mapping strategies for HPF [3] and Split-C [2]. We chose HPF because it is prevalent. In HPF, the user specifies one level of array alignment, and the compiler specifies another. The algorithms we present are thus intended to be implemented at the compiler level in HPF. We discuss array mapping in Split-C as well because the user

*Computer Science Division, University of California, Berkeley (dweisser@cs.berkeley.edu)

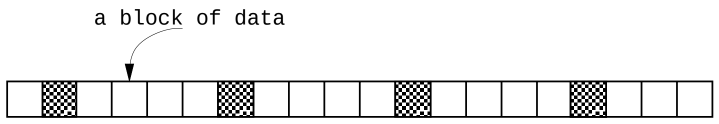


Figure 1: 5 processors, 20 blocks of data. Blocks owned by processor 1 are highlighted.

explicitly controls the entire array layout. Split-C can be thought of as an intermediate language for a compiler. Since it is executable we can test our algorithms in Split-C, and then our algorithms can be implemented inside a compiler for HPF with the same performance and functionality.

The remainder of this paper is organized as follows. Section 2 describes the general framework of our algorithm. Section 3 describes the one- and two-dimensional array layouts in Split-C and HPF. In Section 4, we describe the regular section problem and present optimal algorithms to generate the appropriate index sequences for one- and two-dimensional arrays. Section 5 demonstrates how to generate sequences for the lower triangle of a two-dimensional matrix. All of the algorithms presented in this paper can be extended to more than two dimensions. Empirical results are presented in Section 6. A sample implementation is shown in the Appendix.

2 Algorithmic Overview

We present a simple framework which can be used to generate local indices for various regular access patterns, such as regular section, lower diagonal, wavefront, and tri-diagonal. Computations for regular section and lower diagonal are described in detail in Sections 4 and 5.

The technique is quite straightforward but deserves attention for its efficiency and usefulness in parallel programs.

The general method is as follows. We create an expression which includes the index pattern and layout features. We solve the expression for some variable such as the row number. We then compute local starting and ending offsets based on the layout. Finally, we compute local indices based on the access pattern.

We build state tables of row and column offsets, one table for each dimension's block size. Each entry of the table takes is computed in constant time. The tables can be built in advance or concurrently with the index generations. The size of each table is at most the block size. Once the table is created, subsequent indices are generated by a table lookup or an addition.

A key to creating the initial expression is understanding the array layout. In the next section we describe distributed array layouts in some detail.

3 Data Layout

In this section, we discuss layouts and declarations of arrays in Split-C and HPF. For generality, we assume that the arrays are arranged in a (possibly skewed) block/cyclic fashion.

There are two differences in array layouts between Split-C and HPF. First, Split-C is row-major, while HPF is column-major, i.e. the third element in a 3 x 4 array in Split-C is $(2, 0)$, and in HPF is $(0, 2)$.

Row Number	Processor 0			Processor 1			...	Processor p−1		
0	0	...	b−1	b	...	2b−1	...	(p−1)b	...	pb−1
1	pb	...	(p+1)b−1	(p+1)b−1	...	(p+2)b−1	...	(2p−1)b	...	2pb−1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
r−1										

Figure 2: A one-dimensional array

(Assume rows and columns are numbered from 0.) The other difference is that in HPF the array mapping is determined by both user and compiler specifications, whereas in Split-C it is entirely specified by the user.

3.1 One-Dimensional Arrays

In a block/cyclic layout, if there are more blocks of data than there are processors, multiple blocks of data are stored contiguously on a processor (see Figure 1).

We use the following parameters to describe a one-dimensional array layout:

m = array size, a positive integer

$procs$ = number of processors, a positive integer

b = block size, a positive integer

$myproc\_blocks = \lceil \frac{m}{b*procs} \rceil$ or $\lfloor \frac{m}{b*procs} \rfloor$, the number of blocks owned by processor $myproc$.

Each processor $myproc$ owns data elements:

$$(i*procs*b + b*myproc), \ldots, (i*procs*b + b(myproc+1) \Leftrightarrow 1) \text{ for all } 0 \leq i < myproc\_blocks \qquad (1)$$

Figure 2 shows an example of a one-dimensional array blocked into b-element one-dimensional blocks.

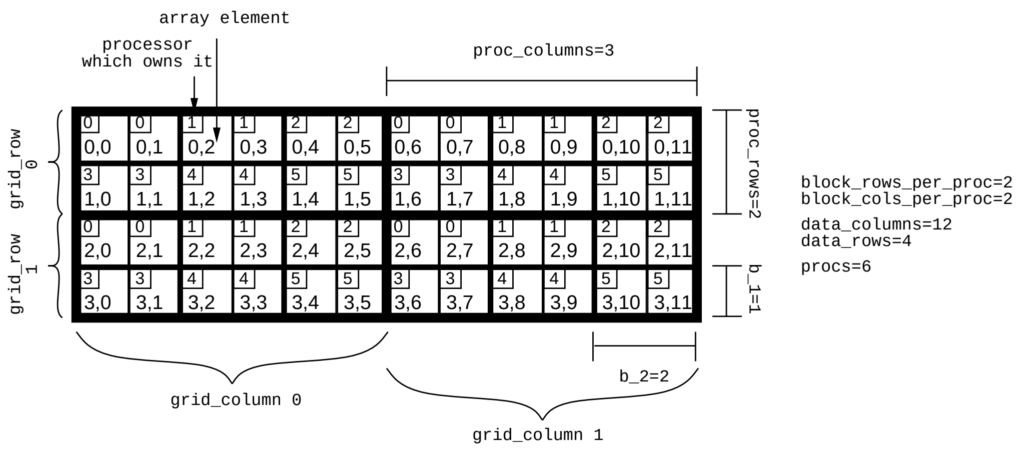
HPF

In HPF, an array declaration for the array in Figure 2 may be of the form $A(m)$. The size of b is determined by the compiler. In addition to the type and size, the user may specify a Cartesian grid, or *template*, such that element $A(i)$ is aligned to template cell $qi + r$. The template is distributed across the processors using a $cyclic(b)$ distribution so template cell j resides on processor $(j$ **div** $b)$ **mod** $procs$.

Split-C

In Split-C, an array declaration is of the form <left indices>::<right indices>, where <left indices> specify the distribution of the data, i.e. the number and size of dimensions, and <right indices> specify the units of data distribution. So a one-dimensional array declaration for the array in Figure 2 may be of the form $A[m] :: [b]$, where m is the number of elements and b is the block size.

In Split-C, a single-dimensional array can use blocks of multiple dimensions. So there may be more than one dimension of blocking factors b_i.



Split-C:
```
A[block_rows_per_proc][block_cols_per_proc][proc_rows][proc_cols]::[b_1][b_2]
                    on proc_rows*proc_cols processors
```

HPF:
```
A(4,12)
```

Figure 3: A six-dimensional array in Split-C, or a two-dimensional array in HPF.

3.2 Two-Dimensional Arrays

First we discuss two-dimensional array layouts in general. We then discuss how two-dimensional arrays are declared and accessed in HPF and Split-C. Figure 3 shows a two-dimensional array describing most of the parameters.

In the two-dimensional case, the data can be subdivided into blocks of size b_1 x b_2. Multiple ($proc\_rows$ x $proc\_columns$) grids of processors are superimposed on the blocks (see Figure 3). We assume that $procs = proc\_rows * proc\_columns$.

We use the following parameters to describe the two dimensional layout

$data\_rows$ = total number of rows of data in the array

$data\_columns$ = total number of columns data in the array

$proc\_rows$ = vertical processor dimension

$proc\_columns$ = horizontal processor dimension

b_1 = vertical block dimension (rows of data per block)

b_2 = horizontal block dimension (columns of data per block)

Three additional parameters are defined using the ones above

$procs$ = number of processors = $proc\_rows * proc\_columns$

$block\_rows\_per\_proc$ = number of rows of data blocks per processor = $\frac{data\_rows}{proc\_rows * b_1}$

$block\_columns\_per\_proc$ = number of columns of data blocks per processor = $\frac{data\_columns}{proc\_columns * b_2}$

Split-C

A declaration in Split-C may be of the form

$A[block\_rows\_per\_proc][block\_columns\_per\_proc][proc\_rows][proc\_columns] :: [b_1][b_2]$ (see Figure 3.

In Split-C, an element in this type of block/cyclic array is specified as

Processor 0				Processor 1				Processor 2				Processor 3			
0	$\boxed{1}$	2	3	4	5	$\boxed{6}$	7	8	9	10	$\boxed{11}$	12	13	14	15
$\boxed{16}$	17	18	19	20	$\boxed{21}$	22	23	24	25	$\boxed{26}$	27	28	29	30	$\boxed{31}$
32	33	34	35	$\boxed{36}$	37	38	39	40	$\boxed{41}$	42	43	44	45	$\boxed{46}$	47
48	49	50	$\boxed{51}$	52	53	54	55	$\boxed{56}$	57	58	59	60	$\boxed{61}$	62	63
64	65	$\boxed{66}$	67	68	69	70	$\boxed{71}$	72	73	74	75	$\boxed{76}$	77	78	79

Figure 4: A regular section on a one-dimensional array where $procs = 4, b = 4, \mathit{offset} = 1, \mathit{stride} = 5, \mathit{max\_index} \geq 80$. Elements in the sequence are boxed.

$$[grid\_row][grid\_column][myproc\_row][myproc\_column][local\_row][local\_column]$$

where:

- $[grid\_row][grid\_column]$ denotes location of the block of processors, which we will refer to as the *grid location* (see Figure 3),
- $[myproc\_row][myproc\_column]$ denotes the processor within the grid, and
- $[local\_row][local\_column]$ specifies the position within the block.

Thus block $[grid\_row][grid\_column][myproc\_row][myproc\_column]$ is owned by processor $((myproc\_row * proc\_rows) + proc\_column)$.

Array element $A(x, y)$ is owned by processor $(proc\_columns*((x \textbf{ div } b_1) \textbf{ mod } proc\_rows)) + ((y \textbf{ div } b_2) \textbf{ mod } proc\_colum$

HPF

In HPF a declaration may be of the form $A(data\_rows, data\_columns)$, and the compiler decides what values to assign to $proc\_rows, proc\_columns, b_1$, and b_2.

Array element $A(x, y)$ is owned by processor $((x \textbf{ div } b_1) \textbf{ mod } proc\_rows, (y \textbf{ div } b_2) \textbf{ mod } proc\_columns)$.

4 Regular Section

Now that we have described the layout details, we present the algorithm to generate the sequence of local memory addresses that a processor accesses while performing its share of the computation on a regular section. The regular section $A(\mathit{offset}, max\_index, stride)$ is the set of elements $A(\mathit{offset} + k * stride)$ where $stride > 0$, $\mathit{offset}(theoriginaloffset) \geq 0$, max_index is the maximum index, and $0 \leq k \leq \frac{max\_index - offset}{stride}$. (See Figure 4.) When the array is of more than one dimension, each dimension has its own offset, stride, and maximum index. We describe algorithms for one- and two-dimensional arrays. The techniques can be extended to arrays of higher dimensions.

Some of the **div**s and **mod**s in the algorithm descriptions can be replaced by simpler operations, but we include them for clarity. A **mod** can often be replaced by a multiplication and a subtraction. A **div** can be replaced by a shift if the divisor is a power of two, as will often be the case in these algorithms when the number of processors is a power of two. In addition, max_index is frequently the size of the array, which also simplifies the computation.

The implementations can be made much more efficient in practice without compromising the theoretical running time by building state tables, which we describe at the end of the following section. The state table can be built in $O(b)$ steps (while generating indices at every step), where b is the block size. Thus after the first b rows' indices are computed, the algorithm generates indices by a table lookup or an addition.

4.1 One-Dimensional Arrays

We now present an algorithm to produce a list of addresses for a given processor.

$O(|output|)$ is clearly a lower bound on the running time for any algorithm to generate adresses for this problem, where $|output|$ is the number of indices generated.

The algorithm presented here runs in time $O(|output|)$ when $stride \leq b$, where b is the block size. When $stride > b$, the running time for this algorithm is $O(|output| + procs)$. The best previously known algorithm has time complexity $O(|output| + b \log b + \log(min(stride, procs * b)))$ [1].

The following parameters vary over a single instance of the regular section problem

$myproc$ = current processor number $(0 \ldots procs \Leftrightarrow 1)$

row_number = current row number

$local\_row\_offset$ = local offset within a row for a particular processor $(0 \ldots b \Leftrightarrow 1)$

For a particular processor $myproc$, we want to output the sequence of indices such that

$$i * stride + offset = myproc * b + row\_number * procs * b + local\_row\_offset,$$

for some choice of row_number and $local\_row\_offset$, where i is a some non-negative integer (see Equation 1).

$$\Leftrightarrow \quad \frac{myproc * b + row\_number * procs * b + local\_row\_offset \Leftrightarrow offset}{stride} = i$$

$$\Leftrightarrow \quad (myproc * b + row\_number * procs * b + local\_row\_offset \Leftrightarrow offset) \textbf{ mod } stride \equiv 0 \qquad (2)$$

since i can be any non-negative integer.

$myproc, b, procs, offset$, and $stride$ are fixed, so we have to find values for row_number and $local\_row\_offset$. The algorithm first fixes a value for row_number and then checks to see if a value for $local\_row\_offset$ exists that satisfies (2).

Finding rows when $stride < b$

When $stride$ is small, i.e. less than $procs*b$, the algorithm looks in every row starting at $offset$ **div** $(p*b)$ for a solution to equation (2).

Finding rows when $b \leq stride \leq procs * b$

Note that in the special cases where $stride = b$ and $stride = procs * b$, only one processor will own elements hit by the stride. It is trivial to determine whether $myproc$ will always or never be hit by the stride.

When $b < stride < procs * b$, we want to find the first row number, row, in which $myproc$ is hit by the stride, i.e. we want to find a value for row such that:

$$
\begin{aligned}
& (i * stride + \mathit{offset} \Leftrightarrow row * ((procs * b) \textbf{ mod } stride)) \textbf{ div } b = myproc \\
\Leftrightarrow \quad & i * stride + \mathit{offset} \Leftrightarrow row * ((procs * b) \textbf{ mod } stride) \geq myproc * b, \text{ and} \\
& i * stride + \mathit{offset} \Leftrightarrow row * ((procs * b) \textbf{ mod } stride) < (myproc + 1) * b \\
\Leftrightarrow \quad & row \leq \frac{i * stride + \mathit{offset} \Leftrightarrow myproc * b}{(procs * b) \textbf{ mod } stride}, \text{ and} \qquad (3) \\
& row > \frac{i * stride + \mathit{offset} \Leftrightarrow (myproc + 1) * b}{(procs * b) \textbf{ mod } stride} \qquad (4)
\end{aligned}
$$

To find a value for row which satisfies Equations 3 and 4, we cycle throught values for i from 0 to $\lfloor \frac{procs*b}{stride} \rfloor$. At each iteration we set

$$row = \left\lceil \frac{i * stride + \mathit{offset} \Leftrightarrow (myproc + 1) * b}{(procs * b) \textbf{ mod } stride} \right\rceil$$

and check to see if Equation 3 is satisfied. If it is not satisfied for any i, then there is no row for which $myproc$ owns an element that is hit by the stride. Note that there are $\lfloor \frac{proc*b}{stride} \rfloor + 1 = O(procs)$ maximum iterations.

Given the row number, to find the element's local column, we set

$$column = (\mathit{offset} + i * stride \Leftrightarrow row * ((procs * b) \textbf{ mod } stride)) \textbf{ mod } b.$$

Subsequent elements can be found in the same way, using $column$ as the offset.

This calculation is included to prove the theoretical bound, but in practice it may be faster to look in every row and avoid the computation of $(procs * b)$ **mod** $stride$. If the processors are running bulk-synchronously, for example, it may pay off to look at every row.

Finding rows when $stride > procs * b$

To find the first element "hit" by the stride, we want to find i, the number of iterations of the stride, such that

$$
\begin{aligned}
& (procs * b + \mathit{offset} \Leftrightarrow (procs * b \Leftrightarrow (stride \textbf{ mod } (procs * b))) * i) \textbf{ div } b = myproc \\
\Leftrightarrow \quad & procs * b + \mathit{offset} \Leftrightarrow (procs * b \Leftrightarrow (stride \textbf{ mod } (procs * b))) * i \geq myproc * b], \text{ and} \\
& procs * b + \mathit{offset} \Leftrightarrow (procs * b \Leftrightarrow (stride \textbf{ mod } (procs * b))) * i < (myproc + 1) * b] \\
\Leftrightarrow \quad & i \leq \frac{procs * b + \mathit{offset} \Leftrightarrow myproc * b}{procs * b \Leftrightarrow stride \textbf{ mod } (procs * b)}, \text{ and} \qquad (5) \\
& i \geq \frac{procs * b + \mathit{offset} \Leftrightarrow (myproc + 1) * b}{procs * b \Leftrightarrow stride \textbf{ mod } (procs * b)} \qquad (6)
\end{aligned}
$$

We can find an i which satisfies Equations 5 and 6 by setting

$$i = \left\lceil \frac{procs * b + \mathit{offset} \Leftrightarrow (myproc + 1) * b}{procs * b \Leftrightarrow stride \ \mathbf{mod}\ (procs * b)} \right\rceil$$

and checking to make sure that Equation 5 still holds. Now that we have solved for i, we know which stride iteration we are interested in. We still need to compute the row and column numbers of this element:

$$row = (\mathit{offset} + (stride * i)) \ \mathbf{div}\ (procs * b) \text{ and}$$
$$column = (\mathit{offset} + (stride * i)) \ \mathbf{mod}\ b \tag{7}$$

Consecutive elements can be calculated similarly, using the current column as the offset.

This calculation is included to prove the theoretical bound, but in practice it may be faster to look in every row and avoid the **div** s and **mod** s. If the processors are running bulk-synchronously, for example, it may pay off to look at every row.

State Transition Table

The algorithm can be implemented more efficiently by creating a state transition table for each processor with at most b states. $Table[i]$ contains the number of rows to skip the the next column in the output sequence, given that the first column to be output in the current row is i. For example, in the example in Figure 4, Processor 1's table would be as follows:

$Table[0] = (1, 3)$
$Table[1] = (0, 0)$
$Table[2] = (0, 1)$
$Table[3] = (0, 2)$

The first index indicates the number of rows to skip, and the second index indicates the first column of the new row that that contains an element in the output sequence.

4.2 Two-Dimensional Arrays

In this section we show how to extend the method of Section 4.1 to two-dimensional arrays. The algorithm generates a sequence of local addresses for a particular processor.

Our algorithm runs in time $O(|output|)$, which is of course the lower bound.

We require the following parameters:

$stride\_1$ = vertical stride
$stride\_2$ = horizontal stride
offset_1 = vertical initial offset
offset_2 = horizontal initial offset
max_index_1 = vertical maximum index
max_index_2 = horizontal maximum index

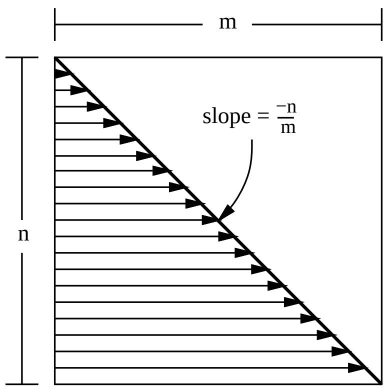


Figure 5: A calculation involving the lower triangle of a matrix

4.2.1 Split-C

Block $[grid\_row][grid\_column][myproc\_row][myproc\_column]$ contains an element hit by the stride if and only if

$$((grid\_row * proc\_rows * b_1 + myproc\_row * b_1 + offset_1 + local\_row\_offset) \textbf{ mod } stride_1 \equiv 0)$$

for some integer $local\_row\_offset$ such that $0 \leq local\_row\_offset < b_1$, and

$$((grid\_column * proc\_rows * b_2 + myproc\_column * b_2 + offset_2 + local\_column\_offset) \textbf{ mod } stride_2 \equiv 0)$$

for some integer $local\_column\_offset$ such that $0 \leq local\_column\_offset < b_2$.

We use the technique described in Section 4.1 to find $local\_row\_offset$ and $local\_column\_offset$. As in Section 4.1, the algorithm looks only in rows and columns that contain elements hit by the stride. Specifically, it looks only in rows where $grid\_row = (i{*}stride_1 + offset_1) \textbf{ div } (proc\_rows{*}b_1)$ and $myproc\_row = (i{*}stride_1 + offset_1) \textbf{ mod } (proc\_rows{*}b_1)$ for integers $i \geq 0$. The algorithm considers columns where $grid\_column = (j * stride_2 + offset_2) \textbf{ div } (proc\_columns{*}b_2)$ and $myproc\_column = (j * stride_2 + offset_2) \textbf{ mod } (proc\_columns * b_2)$ for integers $j \geq 0$.

We build tables as described in Section 4.1, one table for each processor in each dimension to make the implementation efficient.

5 Lower Triangular Matrices

In this section we describe an optimal algorithm to generate local addresses for each processor for an operation on the lower triangle of a matrix $A(data\_rows, data\_columns)$. In this case, we use the diagonal from element $(0,0)$ to element $(data\_rows \Leftrightarrow 1, data\_columns \Leftrightarrow 1)$, although with minor modifications any diagonal could be used. The running time for this algorithm is $O(|output|)$, which is the lower bound.

Suppose we have a two-dimensional array arranged as described in Section 3.2. We want to output all elements (i,j) where $i \leq \frac{data\_columns}{data\_rows} j$.

An important part of the algorithm is finding starting rows and ending columns. To do so, we examine the position of a block with respect to the diagonal. There are three types of data blocks, those which lie entirely above the diagonal, those on or below the diagonal, and those which have some elements above and some below the diagonal.

Let $[row][column]$ denote the position of a block within the $\frac{data\_rows}{b_1}$ x $\frac{data\_columns}{b_2}$ array of blocks. Thus for a block $[grid\_row][grid\_column][myproc\_row][myproc\_column]$ in Split-C, $row = grid\_row * proc\_rows + myproc\_row$, and $column = grid\_column * proc\_columns + myproc\_column$. An $A(i, j)$ is in block $[i \textbf{ div } b_1][j \textbf{ div } b_2]$.

A block $[row][column]$ lies entirely above the diagonal if its lower left corner is above the diagonal, i.e.

$$(row + 1)b_1 < \frac{data\_rows}{data\_cols} column * b_2 \tag{8}$$

Block $[row][column]$ lies entirely within the lower triangle if its upper right corner is on or below the diagonal, i.e.

$$(row * b_1) \geq \frac{data\_rows}{data\_cols}(column + 1)b_2 \tag{9}$$

Finally, block $[row][column]$ has elements above and below the diagonal if Equations (8) and (9) are both not satisfied.

Let $[u][column]$ be the first data block belonging to processor $[row][column]$ that is not entirely above the diagonal.

$$\begin{aligned}
(u + 1)b_1 &\geq \frac{data\_rows}{data\_cols}(column * b_2) \text{ (from Equation (8))} \\
\Rightarrow \qquad u &\geq \frac{data\_rows * column * b_2}{data\_cols * b_1} \Leftrightarrow 1
\end{aligned}$$

Substituting $u = row + i * proc\_rows$ for some integer i:

$$i = \left\lceil \frac{1}{proc\_rows} \left(\frac{data\_rows * column * b_2}{data\_cols * b_1} \Leftrightarrow 1 \Leftrightarrow row \right) \right\rceil \tag{10}$$

Thus we can compute i and set $u = row + i * row$.

6 Computational Results

6.1 Stride

We have implemented the one-dimensional regular section problem in Split-C. [1] (see Section A at end), which we will refer to as *table_lookup*. We compare the running time *table_lookup* to that of a program running the conventional algorithm, henceforth referred to as *naive*, which looks at every element hit by the stride and checks to see whether the desired processor owns that element. Once the state transition table is built, *table_lookup* generates each index only by a table lookup or an addition. *Table_lookup* runs from several times to several orders of magnitude faster, depending upon the number of index calculations and, less importantly, the table size.

Both programs are compiled with optimization, and *naive* uses loop hoisting to remove unnecessary **div**s and **mod**s.

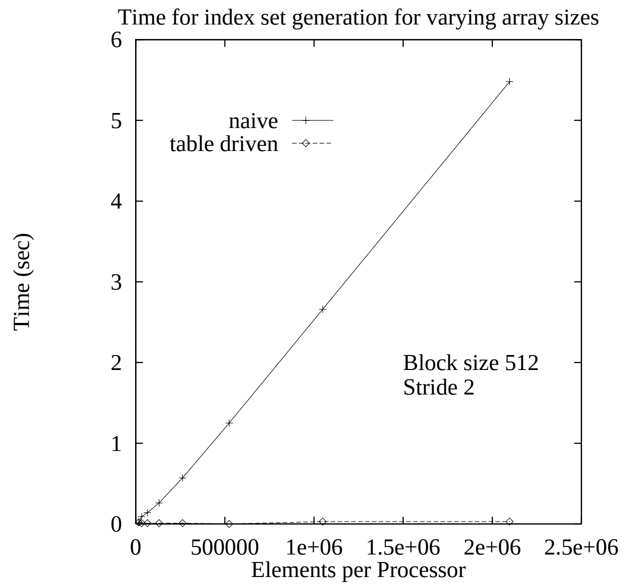


Figure 6: Vary size of problem

We assume that we are making only one pass over the data. If we were making more than one pass, we could store the table and eliminate the (minimal) overhead of its construction.

In Figure 6, we vary the number of elements per processor from 1000 to 256,000 while keeping all other parameters constant. Even when the problem size is very small, *table_lookup* outperforms *naive*. Thus the overhead of table construction is negligible. Even if the table is always "under construction" and we never get a chance to do table lookups, *table_lookup* outperforms *naive*.

In Figure 7, we vary the block size from 16 to 16,000. In the first picture, the programs are run on problems where each processor has 20 blocks. In the second picture, each processor has 200 blocks. As the block size increases, the number of elements each processor accesses, or *output set size*, increases. We see that the running time of *naive* increases linearly with block size, while the running time of *table_lookup* increases negligibly, even when each processor has only 20 blocks.

In Figures 8 and 9, we vary the stride. In Figure 8, we don't adjust the problem size, so the output set size decreases as the stride increases. As the output set size decreases, the relative performance of *table_lookup* becomes less important. In this case, the output set size varies from 512 down to 1. In Figure 9, the problem size is adjusted as the stride increases so that the output set size remains constant. As expected, the running time of *table_lookup* remains relatively constant.

7 Conclusions and Future Work

In this paper we have presented efficient algorithms to generate sequences of local memory addresses for several common functions on parallel arrays. The theoretical running times are optimal or close to optimal. The algorithms can be implemented efficiently so that the running times of the kernel problems yield extremely

[1] In the final version, we will show results for the two-dimensional regular section and lower triangular problems.

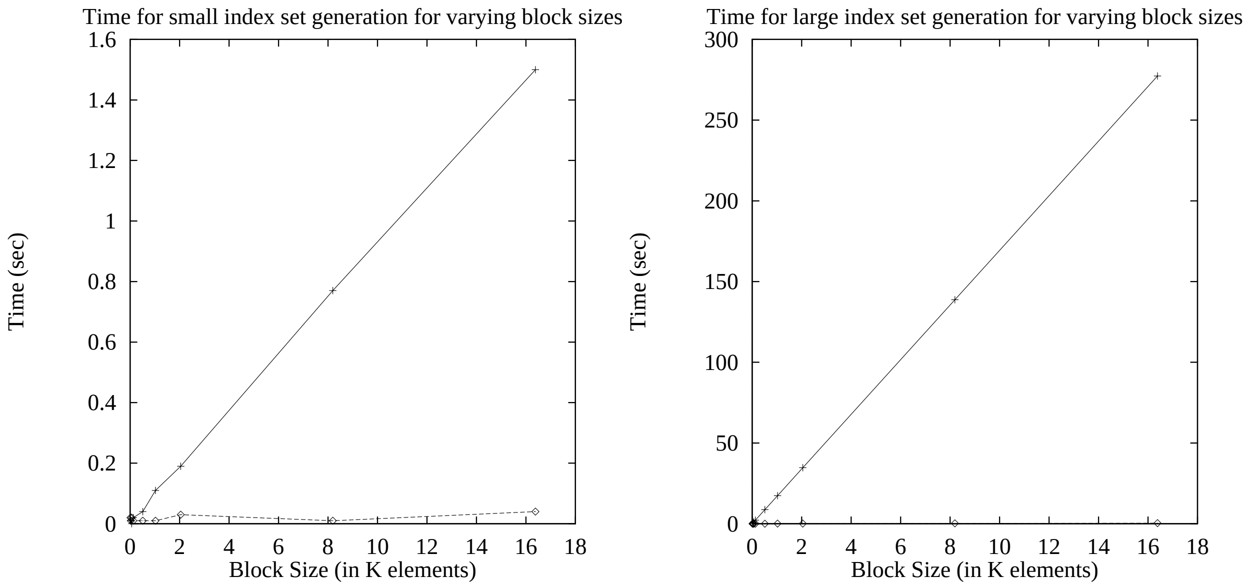


Figure 7: Vary block size. Each proc keeps same number of blocks.

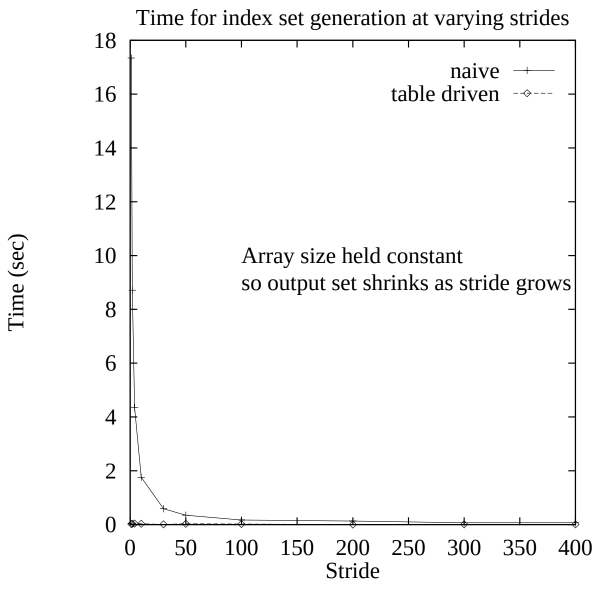


Figure 8: Change stride, causing output set size to decrease

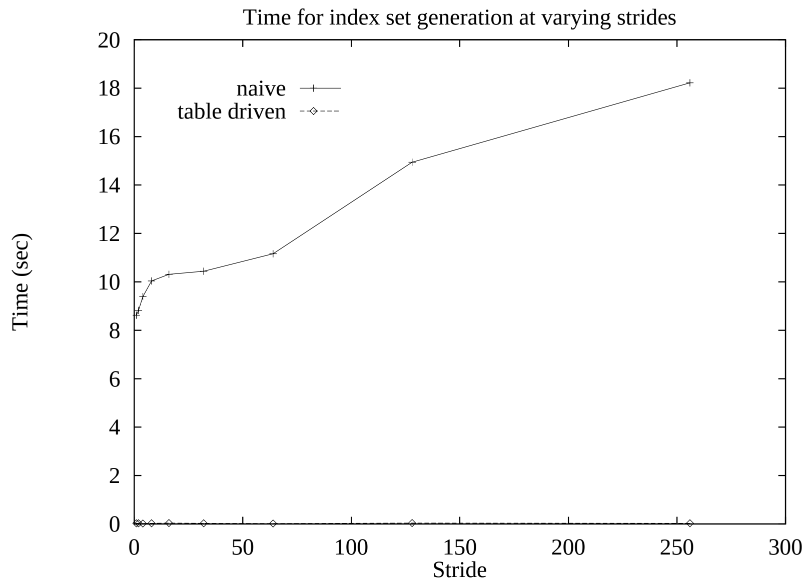


Figure 9: Change stride, but increase problem size to keep output set size constant.

good results. Even when the number of indices each processor generates is low, our algorithms outperform the naive algorithms. When the number in indices to generate is high, they outperform the naive algorithms by several orders of magnitude.

The techniques used to generate indices can be extended to higher dimensions and to other problems as well. For example, they could be used at the compiler level to generate local indices for remote accesses (see [4].)

References

[1] S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S. Teng. Generating local addresses and communication sets for data-parallel programs. In *ACM PPOPP*, pages 149–158, 1993.

[2] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in split-c. In *Proceedings of the Supercomputing '93 Conference*, Nov. 1993.

[3] High Performance Fortran Forum. *High Performance Fortran Language Specification Version 0.4*, 1992.

[4] Charles Koelbel. Compile-time generation of regular communications patterns. In *Proceedings of Supercomputing '91*, Nov. 1991.

Variable Name	Meaning
`max_index`	upper bound of the loop
`my_proc`	processor that loop is running on
`nprocs`	number of processors
`b2`	block size
`table`	table to store next column offset
`delta_row`	table to store next row offset
`num_els`	size of the array
`offset`	lower bound of the loop
`stride`	stride

Table 1: Parameters used in code fragments.

Variable Name	Meaning
`cur_block`	current element's block
`poff`	previous offset
`coff`	current element's offset
`max_row`	maximum row in the block

Table 2: Temporary variables introduced by the transformation.

A Code for the One-Dimensional Regular Section

Here we present the code used for handling the one-dimensional regular section problem. There are two possible transformations for the one-dimensional stride case. If the stride and block size (*b2*) can be determined at compile time and $stride < b2$, then the streamlined transformation in Section A.1 is used. Otherwise, the transformation in Section A.2 is used. In both cases, they share an essential routine: `get_offset` presented in Section A.3.

A.1 Stride known and less than block size

The transformation of the for loop depends on the layout parameters, defined in Table 1. The temporary variables used by the transformed code are defined in Table 2.

Using these parameters a loop is transformed as follows (for $stride < b2$):

```
for (i = offset; i<max_index; i += stride) {  ⟹
   ... A[i] ...
}

                 if (table[b2] == -1)        /* is table precalculated? */
                   table_init0(stride,offset,max_index,my_proc,nprocs,b2,table,num_els);
                 prev_offset=b2;
                 max_row=table[b2+2];

                 for (cur_block = table[b2+1]; cur_block <= max_row; cur_block ++) {
                   for (poff=coff=table[poff]; coff<b2; coff+=stride) {
                     ... A[cur_block][coff] ...
                   }
                 }
```

`table_init0` initializes the finite state machine used by the transformed loop statements.

```
void table_init0(int stride, int offset, int max_index, int my_proc,
                 int nprocs, int b2, int *table, int num_els)
{
  int start_row, max_index_row;
  int cur_block;
  int cur_offset, prev_offset;
  int global_row;       /* global row size */

  global_row = nprocs*b2;

  if (offset>global_row)
    start_row = offset/global_row;
  else
    start_row=0;

  if (num_els<=max_index)
    max_index_row = (num_els-1)/global_row;
  else
    max_index_row = (max_index-1)/global_row;

  table[b2+1] = start_row;
  table[b2+2] = max_index_row;

  prev_offset = b2;

  for(cur_block = start_row; cur_block<= max_index_row; cur_block++) {
    cur_offset=get_offset(my_proc,b2,global_row,offset,stride,cur_block);
    if (cur_offset != -1) {
      table[prev_offset] = cur_offset;
      if (table[cur_offset] != -1) break;
      prev_offset = cur_offset;
    }
  }
}
```

A.2 One-dimensional stride when stride unknown

When the stride is greater than the block size, or if the stride is unknown, a slightly less efficient mechanism is used to generate the indices.

```
for (i = offset; i<max_index; i += stride) {  ⟹
   ...  A[i] ...
}
```

```
if (table[b2] == -1)        /* is table precalculated? */
  table_init(stride,offset,max,my_proc,nprocs,b2,table,delta_row,num_els);
max_row=table[b2+2];
poff=b2;

for (cur_block=table[b2+1]; cur_block <= max_row; cur_block += delta_row[prev_offset]) {
  for (poff=coff=table[poff]; coff<b2; coff+=stride) {
    ...  A[cur_block][coff] ...
  }
}
```

The table initialization routine is `table_init`.

```
void table_init(int stride, int offset, int max, int my_proc, int nprocs,
                int b2, int *table, int *delta_row, int num_els)
{
  int global_row;       /* global row size */
  int start_row, max_row;
  int cur_row;
  int cur_delta_row;
  int cur_offset, prev_offset;

  global_row=nprocs*b2;

  if (offset>global_row)
    start_row = offset/global_row;
  else
    start_row=0;

  if (num_els<=max)
    max_row = (num_els-1)/global_row;
  else
    max_row = (max-1)/global_row;

  table[b2+1] = start_row;
  table[b2+2] = max_row;

  prev_offset = b2;
  cur_delta_row=1;

  for(cur_row = start_row; cur_row<= max_row; cur_row++)
  {
    cur_offset=get_offset(my_proc,b2,global_row,offset,stride,cur_row);

    if (cur_offset != -1) {
      table[prev_offset] = cur_offset;
      delta_row[cur_offset]=cur_delta_row;
      cur_delta_row=1;

      if (table[cur_offset] != -1) break;

      prev_offset = cur_offset;
    } else {
      cur_delta_row++;
    }
  }
}
```

A.3 Code for get_offset

.

The most important calculation is performed by the `get_offset` routine.

```
int get_offset(int my_proc, int b2, int global_row, int offset, int stride,
               int cur_row)
{
  int temp;
  int cur_offset;

  temp = (my_proc*b2 + cur_row*global_row - offset) % stride;

  if (temp > 0) {
    cur_offset = stride-temp;
    if (cur_offset >= b2) cur_offset = -1;
  } else {
    cur_offset=0;
  }

  return(cur_offset);
}
```