Generating Local Addresses and Communication Sets for Data-Parallel Programs*

Siddhartha Chatterjee [†] John R. Gilbert [‡] Fred J. E. Long [§]

Robert Schreiber [†] Shang-Hua Teng [¶]

* *An early version of this paper appears in the Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, CA, May 1993, pages 149–158.*

† Research Institute for Advanced Computer Science, Mail Stop T27A-1, NASA Ames Research Center, Moffett Field, CA 94035-1000 (sc@riacs.edu, schreibr@riacs.edu). The work of these authors was supported by the NAS Systems Division via Cooperative Agreement NCC 2-387 between NASA and the Universities Space Research Association (USRA).

‡ Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304-1314 (gilbert@parc.xerox.com).

§ Systems Engineering Associates, 2204 Garnet Avenue, Suite 303, San Diego, CA 92109 (flong@sea.com). This work was performed while the author was a summer student at RIACS.

¶ Department of Mathematics, Massachusetts Institute of Technology, Cambridge, MA 02139 (steng@theory.lcs.mit.edu). This work was performed while the author was a postdoctoral scientist at Xerox PARC.

Proposed running head: Local Addresses and Communication Sets
Direct all correspondence to:

Dr. John R. Gilbert
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94394-1314
Phone: (415) 812-4487
Email: gilbert@parc.xerox.com

Abstract

Generating local addresses and communication sets is an important issue in distributed-memory implementations of data-parallel languages such as High Performance Fortran. We demonstrate a storage scheme for an array A affinely aligned to a *template* that is distributed across p processors with a *cyclic*(k) distribution that does not waste any storage, and show that, under this storage scheme, the local memory access sequence of any processor for a computation involving the regular section $A(\ell : h : s)$ is characterized by a finite state machine of at most k states. We present fast algorithms for computing the essential information about these state machines, and extend the framework to handle multidimensional arrays. We also show how to generate communication sets using the state machine approach. Performance results show that this solution requires very little runtime overhead and acceptable preprocessing time.

List of symbols

$\lceil, \rceil, \lfloor, \rfloor$	ceiling and floor symbols
$\mathcal{P}, \mathcal{C}, \mathcal{O}, \mathcal{M}, \mathcal{T}, \mathcal{L}$	uppercase letters in calligraphic font
ℓ	ell
$\alpha, \beta, \lambda, \mu, \tau$	lowercase Greek letters
Δ, Σ, Ω	uppercase Greek letters
$\mathbb{Z}$	set of integers
$\perp$	bottom

A	a distributed array
n	size of array A
a	stride of alignment of A to the template
b	offset of alignment of A to the template
p	number of processors to which the template is distributed
k	block size of distribution of the template
ℓ	lower bound of regular section of A
h	upper bound of regular section of A
s	stride of regular section of A
m	processor number
$\mathcal{P}$	processor holding array element
$\mathcal{C}$	course of blocks within processor holding array element
$\mathcal{O}$	offset of array element within course
$\mathcal{M}$	local memory location where array element is stored

Table 1: Symbols used in this paper.

1 Introduction

Languages such as Fortran D [5] and High Performance Fortran [7] allow the programmer to annotate programs with *alignment* and *distribution* statements that describe how to map arrays to processors in a distributed-memory environment. Such languages typically introduce an auxiliary Cartesian grid called a *template*; a template is “an abstract index space that can be distributed and with which arrays may be aligned” [11, page 101]. Arrays are aligned to templates, and templates are distributed across the processors. The alignment of an array to a template and the distribution of the template determine the final *mapping* of the array. The compiler must partition the arrays and produce “node code” for execution on each processor of a distributed-memory parallel computer.

Consider a (one-dimensional) array A of size n aligned to a template such that array element $A(i)$ is aligned to template cell $ai + b$. The template is distributed across p processors using a *cyclic(k)* distribution, in which template cell i is held by processor $(i \text{ div } k) \bmod p$. (We number array elements, template cells, and processors starting from zero.) Thus the array A is split into p local arrays residing in the local memories of the processors. Consider a computation involving the *regular section* $A(\ell : h : s)$, *i.e.*, the set of array elements $\{A(\ell + js) : 0 \leq j \leq (h - \ell)/s\}$, where $s > 0$. (Table 1 summarizes the notation.) In this paper we consider the following problems:

- What is the sequence of local memory addresses that a given processor m must access while performing its share of the computation?
- What is the set of messages that a given processor m must send while performing its share of the computation?

The *cyclic*(k) distributions generalize the familiar *block* and *cyclic* distributions (which are *cyclic*$(\lceil n/p \rceil)$ and *cyclic*(1) respectively). Dongarra *et al.* [4] have shown these distributions to be essential for efficient dense matrix algorithms on distributed-memory machines. High Performance Fortran includes directives for multidimensional *cyclic*(k) distribution of templates.

The paper is organized as follows. Section 2 solves the problem for a one-level mapping $(a = 1, b = 0)$. Section 3 reduces the problem for two-level mappings to combining the solutions from two subproblems involving one-level mappings. Section 4 extends the framework to handle multidimensional arrays, and Section 5 extends it to handle generation of communication sets. Section 6 discusses performance results. Section 7 discusses related work, and Section 8 presents conclusions.

(a)

Course	Processor 0			Processor 1			$\dots$	Processor $p-1$		
0	0	$\dots$	$k-1$	k	$\dots$	$2k-1$	$\dots$	$(p-1)k$	$\dots$	$pk-1$
1	pk	$\dots$	$(p+1)k-1$	$(p+1)k$	$\dots$	$(p+2)k-1$	$\dots$	$(2p-1)k$	$\dots$	$2pk-1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Offset	0	$\dots$	$k-1$	0	$\dots$	$k-1$	$\dots$	0	$\dots$	$k-1$

(b)

Local memory location	0	$\dots$	$k-1$	k	$\dots$	$2k-1$	$\dots$
Array index (processor 0)	0	$\dots$	$k-1$	pk	$\dots$	$(p+1)k-1$	$\dots$
Array index (processor 1)	k	$\dots$	$2k-1$	$(p+1)k$	$\dots$	$(p+2)k-1$	$\dots$
Array index (processor $p-1$)	$(p-1)k$	$\dots$	$pk-1$	$(2p-1)k$	$\dots$	$2pk-1$	$\dots$

Figure 1: A one-level mapping. (a) Layout parameters. Processors run across the page, while memory is organized as a two-dimensional array of courses and offsets. Each box represents a memory cell, and the number in the cell indicates the array element it holds. (b) Layout of array elements in local processor memories. Each processor's memory is one row of the diagram.

2 One-level mappings

We first consider the problem where a one-dimensional array A is aligned to a template with $a = 1$. This is called a *one-level mapping*. In this case, the array and the template are in one-to-one correspondence, and we specify the problem by the pair (p, k) describing the distribution of the template across the processors. As shown in Figure 1(a), we visualize the layout of the array in the processor memories as courses of blocks. Three functions $\mathcal{P}$, $\mathcal{C}$, and $\mathcal{O}$ describe the location of array element $A(i)$. $\mathcal{P}(i; p, k)$ is the processor holding $A(i)$, $\mathcal{C}(i; p, k)$ is the course of blocks within this processor holding $A(i)$, and $\mathcal{O}(i; p, k)$ is the offset of $A(i)$ within the course. The layout of array elements in the local processor memories linearizes the courses of blocks, as shown in Figure 1(b).

The mapping functions are defined as follows.

$$\mathcal{P}(i; p, k) \quad = \quad (i \bmod pk) \text{ div } k = (i \text{ div } k) \bmod p \tag{1}$$

$$\mathcal{C}(i;p,k) = i \text{ div } pk = (i \text{ div } k) \text{ div } p \quad (2)$$

$$\mathcal{O}(i;p,k) = i \bmod k \quad (3)$$

Therefore, we have

$$i = pk \cdot \mathcal{C}(i;p,k) + k \cdot \mathcal{P}(i;p,k) + \mathcal{O}(i;p,k). \quad (4)$$

For a one-level mapping ($a = 1$), the following function gives the local memory location (with respect to the base of the local array) where element $A(i)$ is stored within processor $\mathcal{P}(i;p,k)$:

$$\mathcal{M}(i;p,k) = k \cdot \mathcal{C}(i;p,k) + \mathcal{O}(i;p,k). \quad (5)$$

We are interested in the sequence of local memory locations that an individual processor, numbered m, accesses while performing its share of the computation on the regular section $A(\ell : h : s)$. We specify the sequence of accesses by its starting location and by the differences $\Delta\mathcal{M}(i_1, i_2) = \mathcal{M}(i_2;p,k) - \mathcal{M}(i_1;p,k)$, where i_1 and i_2 are two successive indices of the part of the regular section mapped to processor m. As the examples in Figure 2 illustrate, the spacing $\Delta\mathcal{M}$ need not be constant.

The key insight is that (for a particular p, k, and access stride s) the offset $\mathcal{O}(i_1;p,k)$ of one element of the regular section determines the offset $\mathcal{O}(i_2;p,k)$ of the next element of the regular section in the same processor, and also determines the spacing $\Delta\mathcal{M}(i_1, i_2)$. Thus we can represent the offset sequence and the $\Delta\mathcal{M}$ sequence as a simple table indexed by offset. We visualize this table as the transition diagram of a finite state machine (FSM) whose k states are the offsets. This FSM is particularly simple because it has no input, and each state has only one outgoing transition; in fact, it is just a union of state-disjoint cycles. Thus the sequence of $\Delta\mathcal{M}$'s is periodic, with period at most k.

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

	Proc. 0				Proc. 1				Proc. 2			
(b)	$\boxed{0}$	1	2	$\boxed{3}$	4	5	$\boxed{6}$	7	8	$\boxed{9}$	10	11
	$\boxed{12}$	13	14	$\boxed{15}$	16	17	$\boxed{18}$	19	20	$\boxed{21}$	22	23

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

Figure 2: One-level array mappings. The corresponding state tables are shown in Examples 1–3. (a) $k = 4, p = 4, s = 3$. (b) $k = 4, p = 3, s = 3$. (c) $k = 4, p = 3, s = 5$.

The contents of the table, or the transitions of the FSM, depend on p, k, and s, but not on the section bounds ℓ and h or the processor number m. The same FSM describes the offsets for every processor, though each processor sees at most one cycle of the FSM. The first memory location used by the section in processor m (and its offset, which is the "start state" for that processor's view of the FSM) depends on ℓ and m as well as p, k, and s.

Following equation (5), we can write $\Delta\mathcal{M}$ in terms of the differences in course and offset:

$$\Delta\mathcal{M}(i_1, i_2) = k \cdot \Delta\mathcal{C}(i_1, i_2) + \Delta\mathcal{O}(i_1, i_2), \tag{6}$$

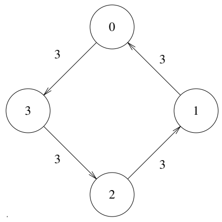
where the definitions of $\Delta\mathcal{C}$ and $\Delta\mathcal{O}$ are analogous to that of $\Delta\mathcal{M}$.

Here are some examples. We assume that $\ell = 0$ and that h is large.

Example 1 $k = 4, p = 4, s = 3$. The distribution is shown in Figure 2(a). The transition table T

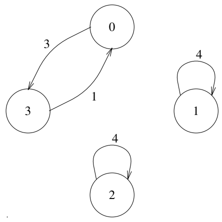
is as follows. For convenience, we also tabulate $\Delta\mathcal{C}$ and $\Delta\mathcal{O}$. This shows a simple case where all processors access local memory with constant stride.

STATE	NEXT	$\Delta\mathcal{M}$	$\Delta\mathcal{C}$	$\Delta\mathcal{O}$
0	3	3	0	3
1	0	3	1	-1
2	1	3	1	-1
3	2	3	1	-1



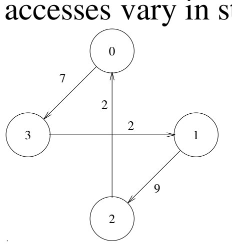
Example 2 $k = 4, p = 3, s = 3$. The distribution is shown in Figure 2(b). The transition table *T* is as follows. This demonstrates that the FSM for a problem can consist of disjoint cycles. Each processor sees at most one cycle of the FSM, depending on its start state.

STATE	NEXT	$\Delta\mathcal{M}$	$\Delta\mathcal{C}$	$\Delta\mathcal{O}$
0	3	3	0	3
1	1	4	1	0
2	2	4	1	0
3	0	1	1	-3



Example 3 $k = 4, p = 3, s = 5$. The distribution is shown in Figure 2(c). The transition table *T* is as follows. The $\Delta\mathcal{M}$ sequence has period k, and local memory accesses vary in stride.

STATE	NEXT	$\Delta\mathcal{M}$	$\Delta\mathcal{C}$	$\Delta\mathcal{O}$
0	3	7	1	3
1	2	9	2	1
2	0	2	1	-2
3	1	2	1	-2



2.1 The algorithm

Preparatory to finding a processor's $\Delta\mathcal{M}$ sequence, we solve the problem of finding its starting memory location. Given a processor number m, regular section $A(\ell : h : s)$, and layout parameters p and k, we wish to find the first element (if any) of the regular section that resides on that processor.

This is equivalent to finding the smallest nonnegative integer j such that $\mathcal{P}(\ell + sj; p, k) = m$.

Substituting the definition of $\mathcal{P}$ from equation (1), we get

$$((\ell + sj) \text{ mod } pk) \text{ div } k = m.$$

This reduces to

$$km \leq (\ell + sj) \text{ mod } pk \leq k(m+1) - 1$$

which is equivalent to finding an integer q such that

$$km - \ell \leq sj - pkq \leq km - \ell + k - 1. \tag{7}$$

We rewrite inequality (7) as a set of k linear Diophantine equations in the variables j and q,

$$\{sj - pkq = \lambda : km - \ell \leq \lambda \leq km - \ell + k - 1\}. \tag{8}$$

The equations in this set are to be solved independently (rather than simultaneously). Solutions exist for an individual equation if and only if λ is divisible by GCD (s, pk) [14, page 95]

The general solution to the linear Diophantine equation $sj - pkq = \lambda$ is found as follows. Let $d = \text{GCD}(s, pk) = \alpha s + \beta pk$, with $\alpha, \beta \in \mathbb{Z}$. The quantities α, β, and d are determined by the extended Euclid algorithm [8, page 325]. Then the general solution of the equation is the one-parameter family $j = (\lambda\alpha + pk\gamma)/d$ and $q = (-\lambda\beta + s\gamma)/d$, for $\gamma \in \mathbb{Z}$.

For each solvable equation in set (8), we find the solution having the smallest nonnegative j. The minimum of these solutions gives us the starting index value $\ell + js$, which we then convert to the starting memory location using equation (5). Note that successive index values in the family of solutions differ by LCM (s, pk). We compute the ending memory location similarly, by finding the largest solutions no greater than h, and taking the maximum among these solutions.

Now we extend this idea to build the $\Delta\mathcal{M}$ sequence for the processor. A sorted version of the set of smallest positive solutions determined above gives us the sequence of indices that will be successively accessed by the processor. A linear scan through this sequence is sufficient to generate the $\Delta\mathcal{M}$ sequence for the cycle of the FSM visible to processor m. If the FSM has multiple cycles, the local $\Delta\mathcal{M}$ sequences may be different for different processors; otherwise, the sequences are cyclic shifts of one another. To avoid following the NEXT pointers at runtime, the $\Delta\mathcal{M}$ table we compute contains the memory spacings in the order seen by the processor.

All of these elements are combined in Algorithm 1, which provides the solution for the memory address problem for a one-level mapping. The running time of Algorithm 1 is $O(\log \min(s, pk)) + O(k \log k)$, which reduces to $O(\min(k \log k + \log s, k \log k + \log p))$. As the output can be of size k in the worst case, any algorithm for this problem takes $\Omega(k)$ time. Finding an algorithm with $O(k)$ running time remains an open problem.

Algorithm 1 can be simplified by noting that the right hand sides of successive solvable equations differ by d, and that the identity of the first solvable equation is easily determined from the value of $(km - \ell) \bmod d$. This removes the conditional from the loop in lines 5–12. Similar incremental computations can also be used to simplify the floor and ceiling computations in that loop.

The node code for each processor consists of the following loop.

```
base ← startmem; i ← 0
while (base ≤ lastmem) do
   compute with address base
   base ← base + ΔM[i]
   i ← (i + 1) mod length
enddo
```

Some codes may require the global index as well as an input to the computation; *e.g.*, consider the FORALL statement `forall(i=1:n) a(i) = i-1`. It is simple to modify the tables described above to carry this information. Essentially, following equation (4), we can write the difference $\Delta\mathcal{I}$ between successive global indices on the same processor in terms of the differences in course and offset:

$$\Delta\mathcal{I}(i_1, i_2) = pk \cdot \Delta\mathcal{C}(i_1, i_2) + \Delta\mathcal{O}(i_1, i_2). \tag{9}$$

Algorithm 1 (Memory access sequence for a one-level mapping.)
Input: Layout parameters (p, k), regular section $A(\ell : h : s)$, processor number m.
Output: The $\Delta\mathcal{M}$ table, the length of the table, starting memory location, ending memory location, and $\sum \Delta\mathcal{M}$ for processor m.
Method:

integer $\Delta\mathcal{M}$*[k], length*
integer λ*, i, start, end, d,* α, β*, lcourse, loffset, course, offset*
integer *loc[k+1], sorted[k+1]*

(d, α, β*)* $\leftarrow$ EXTENDED-EUCLID*(s, pk); length* $\leftarrow$ *0; start* $\leftarrow$ $h+1$*; end* $\leftarrow$ $\ell - 1$
for $\lambda = km - \ell$ **to** $km - \ell + k - 1$ **do**
  if $\lambda \bmod d = 0$ **then** /* Solvable equation */
    loc[length] $\leftarrow \ell + \frac{s}{d}(\lambda\alpha + pk\lceil\frac{-\lambda\alpha}{pk}\rceil)$
    start $\leftarrow$ min*(start, loc[length])*
    end $\leftarrow$ max*(end,* $\ell + \frac{s}{d}(\lambda\alpha + pk\lfloor\frac{(h-l)d-\lambda\alpha s}{pks}\rfloor)$*)*
    length $\leftarrow$ *length + 1*
  endif
enddo
if *length = 0* **or** *start* $>$ *h* **then**
  return $\Delta\mathcal{M}$, $\perp$, $\perp$, $\perp$, $\perp$ /* No work for processor */
endif
sorted $\leftarrow$ SORT *(loc, length)*
sorted[length] $\leftarrow$ *sorted[0]* $+pks/d$*; lcourse* $\leftarrow$ $\mathcal{C}$*(sorted[0]; p, k); loffset* $\leftarrow$ $\mathcal{O}$*(sorted[0]; p, k)*
for *i = 0* **to** *length* $-$ *1* **do**
  course $\leftarrow$ $\mathcal{C}$*(sorted[i + 1]; p, k); offset* $\leftarrow$ $\mathcal{O}$*(sorted[i + 1]; p, k)*
  $\Delta\mathcal{M}$*[i]* $\leftarrow$ *k·(course – lcourse) + (offset – loffset)*
  lcourse $\leftarrow$ *course; loffset* $\leftarrow$ *offset*
enddo
return $\Delta\mathcal{M}$*, length,* $\mathcal{M}$*(start; p, k),* $\mathcal{M}$*(end; p, k),* $sk/\text{GCD}(s, pk)$

We illustrate the actions of Algorithm 1 on Example 3 for processor 0. Here, $p = 3$, $k = 4$, and $s = 5$. The extended Euclid algorithm returns $d = 1$, $\alpha = 5$, and $\beta = -2$. Since each linear Diophantine equation is solvable, the loop in lines 5–12 is executed four times. At the end of this loop, *loc* = [0, 25, 50, 15], *length* = 4, *start* = 0, and *end* = 50. After line 17, *sorted* = [0, 15, 25, 50, 60]. At the end of the loop in lines 18–22, $\Delta\mathcal{M}$ = [7, 2, 9, 2].

2.2 Special cases

There are several special cases where the solutions have a particularly simple form and can be determined in constant time. Our implementation recognizes the following special cases.

1. $p = 1$: The template is "distributed" in the local memory of a single processor. Local memory accesses have constant stride s.

2. $s = 1$: Once again, local memory accesses have constant stirde $s = 1$.

3. s divides k: If the access stride is a divisor of the block size, it is easy to see that the access pattern repeats at every level of the course. Hence local memory is accessed with constant stride s.

4. k divides s: If the access stride s is a multiple of the block size k, local memory access is still constant stride. The stride is given by $ks/\text{GCD}(s, pk)$.

5. pk divides s: In this case, local memory is accessed with constant stride s/p.

6. A single course of blocks: In this case, local memory is accessed with stride s.

(a)

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

(b)

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

(c)

Local template cell	0	1	2	3	4	5	6	7	8	9	10	11	$\cdots$
Processor 0	0			1			6			11			$\cdots$
Processor 1			2			7			12			13	$\cdots$
Processor 2		3			8			9			14		$\cdots$
Processor 3	4			5			10			15			$\cdots$

Figure 3: A two-level mapping of an array A aligned to the template with stride 3. (a) Layout of template cells. Numbers are array elements; boxed numbers are in the regular section $A(0:42:3)$. (b) Local memory storage for array. (c) Local template space for array.

3 Two-level mappings

Two-level mappings (where $a > 1$) are solved by two applications of Algorithm 1. As an example, let A be an array with element $A(i)$ aligned with template cell $3i$, let the template be mapped with $p = k = 4$, and let the regular section be $A(0:42:3)$, which we abbreviate to B. The picture is shown in Figure 3(a). The boxes represent template cells, the number j indicates the location of $A(j)$, and numbers in squares are part of the regular section $A(0:42:3)$. Two-level mappings encompass all mappings possible with the linguistic constructs of nested sections, affine alignment, and *cyclic*(k) distribution.

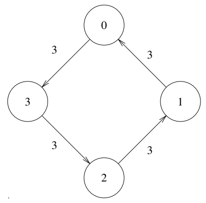
Unlike the one-level case, all template cells do not contain array elements. We therefore have a choice of memory allocation strategies. We could choose to allocate local storage for every template cell, irrespective of whether it is occupied or not. This makes the problem equivalent to the one-level case, but has the undesirable property of wasting a potentially unbounded amount of storage (the waste is proportional to the access stride s). Given that distributed-memory parallel machines usually have a limited amount of local memory per processor, we deem such waste unacceptable. We therefore allocate local storage only for the *occupied* cells of the template. Thus, the local memory storage for A in the example above is as shown in Figure 3(b). We handle this complication by first doing all our computations with respect to the template (the *local template space*), and then switching back to the local memory space in the final step. The picture in local template space is shown in Figure 3(c).

Note that A and B are both aligned to "regular sections" of the template: $A(i)$ is aligned to template cell $3i$, and $B(i)$ is aligned to template cell $9i$. We therefore create state tables for A and

B with respect to the local template space using Algorithm 1. The "$\Delta\mathcal{M}$" entries returned by the algorithm are actually spacings in local template space; we therefore rename this column $\Delta\mathcal{T}$. This gives the tables for A and B shown below.

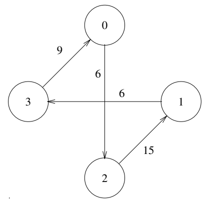
$T_A =$

STATE	NEXT	$\Delta\mathcal{T}$	$\Delta\mathcal{M}$
0	3	3	?
1	0	3	?
2	1	3	?
3	2	3	?



$T_B =$

STATE	NEXT	$\Delta\mathcal{T}$	$\Delta\mathcal{M}$
0	2	6	?
1	3	6	?
2	1	15	?
3	0	9	?



Now all we need is to switch back to local memory space and fill in the $\Delta\mathcal{M}$ entries.

As we traverse the state table T_A for any processor m, we access each element of A stored on that processor, in order. But since storage is allocated based on A, this translates to accessing consecutive locations in local memory. Thus the $\Delta\mathcal{M}$ entries for T_A are all unity. Now it is a simple matter to find the $\Delta\mathcal{M}$ entries for the desired section B. For instance, the first line of T_B says that if we are at an element of the section that has offset 0, the next element has offset 2 and is 6 template cells away. To find the corresponding $\Delta\mathcal{M}$, we start in state 0 of T_A, and traverse the transitions of the state table, accumulating $\Delta\mathcal{T}$ and $\Delta\mathcal{M}$ counts. We stop when we have accumulated the desired $\Delta\mathcal{T}$ count. (It is easy to show that at this point we will have arrived at the correct offset as well.) The accumulated $\Delta\mathcal{M}$ value is the desired entry for that line of T_B. This procedure results in the following table for B.[1]

[1] In this case, $\Delta\mathcal{M}$ turns out to be $\Delta\mathcal{T}$ divided by the alignment stride $a = 3$, but this is not true in general. Try the following case: $a = 3$, $b = 0$, $p = 3$, $k = 8$, $s = 3$.

STATE	NEXT	$\Delta\mathcal{T}$	$\Delta\mathcal{M}$
0	2	6	2
1	3	6	2
2	1	15	5
3	0	9	3

The algorithm as presented above requires $O(k)$ time to compute each $\Delta\mathcal{M}$ entry of T_B. Two optimizations reduce this complexity to $O(1)$, allowing the final fixup phase to run in $O(k)$ time. First, we compute the sum of the $\Delta\mathcal{T}$ and the $\Delta\mathcal{M}$ around the state machine cycle, so that we can cast out any full cycles that must be traversed. Second, we tabulate the distance of each offset from a fixed origin (the *numarcs* table in Algorithm 2), allowing the computation of the distance between any two offset positions by two table lookups and some simple arithmetic. Consider the third line of T_B, which has a $\Delta\mathcal{T}$ of 15. Using the precomputed sums, we determine in constant time that this corresponds to one full cycle plus 3 more template cells, and that the full cycle contributes a memory offset of 4. Now all we need is to determine the number of memory cells encountered in going from offset position 2 to offset position 1 in T_A. We find this distance in constant time using the *numarcs* table. Algorithm 2 shows the method with these optimizations incorporated.

Algorithm 2 (Memory access sequence for a two-level mapping.)

Input: Alignment parameters (a, b), layout parameters (p, k), array length n, regular section $A(\ell : h : s)$, processor number m.

Output: The $\Delta\mathcal{M}$ table, length of the table, starting memory location, ending memory location, and the length of A in processor m.

Method:

integer *i, mm, numarcs[k]*
integer $\Delta\mathcal{M}$*[k], length, startmem, lastmem*
integer $\Delta\mathcal{M}$*1[k], length1, startmem1, lastmem1, memcycle1*
integer $\Delta\mathcal{M}$*2[k], length2, startmem2, lastmem2, memcycle2*

function *residual(x) = length1* $\cdot \lfloor$ *(x–startmem1)/memcycle1* $\rfloor$
function *major(x) = (numarcs[x* mod *k] – numarcs[startmem1* mod *k])* mod *length1*
function *realmem(x) = major(x) + residual(x)*

($\Delta\mathcal{M}$*1, length1, startmem1, lastmem1, memcycle1)* $\longleftarrow$ *Algorithm 1(*b, $a(n-1)+b$, a, p, k, m*)*
($\Delta\mathcal{M}$*2, length2, startmem2, lastmem2, memcycle2)* $\longleftarrow$ *Algorithm 1(*$a\ell+b$, $ah+b$, as, p, k, m*)*
if *length1* = $\perp$ **or** *length2* = $\perp$ **then**
  return $\Delta\mathcal{M}, \perp, \perp, \perp,$ *0*   /* No work for processor */
endif
mm $\longleftarrow$ *startmem1*
for *i = 0* **to** *length1* $-$ *1* **do**
  numarcs[mm mod *k]* $\longleftarrow$ *i*
  mm $\longleftarrow$ *mm* $+ \Delta\mathcal{M}$*1[i]*
enddo
mm $\longleftarrow$ *startmem2*
for *i = 0* **to** *length2* $-$ *1* **do**
  $\Delta\mathcal{M}$*[i]* $\longleftarrow$ *realmem(mm* $+ \Delta\mathcal{M}$*2[i])* $-$ *realmem(mm)*
  mm $\longleftarrow$ *mm* $+ \Delta\mathcal{M}$*2[i]*
enddo
return $\Delta\mathcal{M}$*, length2, realmem(startmem2), realmem(lastmem2), realmem(lastmem1)* $-$ *realmem(startmem1)+1*

4 Multidimensional arrays

So far, we have characterized the access sequence of a single processor by a start address and a

Processor $(0,0)$				Processor $(0,1)$			
$\boxed{(0,0)}$	$(0,1)$	$(0,4)$	$(0,5)$	$(0,2)$	$\boxed{(0,3)}$	$\boxed{(0,6)}$	$(0,7)$
$(1,0)$	$(1,1)$	$(1,4)$	$(1,5)$	$(1,2)$	$(1,3)$	$(1,6)$	$(1,7)$
$\boxed{(2,0)}$	$(2,1)$	$(2,4)$	$(2,5)$	$(2,2)$	$\boxed{(2,3)}$	$\boxed{(2,6)}$	$(2,7)$
$(9,0)$	$(9,1)$	$(9,4)$	$(9,5)$	$(9,2)$	$(9,3)$	$(9,6)$	$(9,7)$
$\boxed{(10,0)}$	$(10,1)$	$(10,4)$	$(10,5)$	$(10,2)$	$\boxed{(10,3)}$	$\boxed{(10,6)}$	$(10,7)$
$(11,0)$	$(11,1)$	$(11,4)$	$(11,5)$	$(11,2)$	$(11,3)$	$(11,6)$	$(11,7)$
Processor $(1,0)$				Processor $(1,1)$			
$(3,0)$	$(3,1)$	$(3,4)$	$(3,5)$	$(3,2)$	$(3,3)$	$(3,6)$	$(3,7)$
$\boxed{(4,0)}$	$(4,1)$	$(4,4)$	$(4,5)$	$(4,2)$	$\boxed{(4,3)}$	$\boxed{(4,6)}$	$(4,7)$
$(5,0)$	$(5,1)$	$(5,4)$	$(5,5)$	$(5,2)$	$(5,3)$	$(5,6)$	$(5,7)$
$\boxed{(12,0)}$	$(12,1)$	$(12,4)$	$(12,5)$	$(12,2)$	$\boxed{(12,3)}$	$\boxed{(12,6)}$	$(12,7)$
$(13,0)$	$(13,1)$	$(13,4)$	$(13,5)$	$(13,2)$	$(13,3)$	$(13,6)$	$(13,7)$
$\boxed{(14,0)}$	$(14,1)$	$(14,4)$	$(14,5)$	$(14,2)$	$\boxed{(14,3)}$	$\boxed{(14,6)}$	$(14,7)$
Processor $(2,0)$				Processor $(2,1)$			
$\boxed{(6,0)}$	$(6,1)$	$(6,4)$	$(6,5)$	$(6,2)$	$\boxed{(6,3)}$	$\boxed{(6,6)}$	$(6,7)$
$(7,0)$	$(7,1)$	$(7,4)$	$(7,5)$	$(7,2)$	$(7,3)$	$(7,6)$	$(7,7)$
$\boxed{(8,0)}$	$(8,1)$	$(8,4)$	$(8,5)$	$(8,2)$	$\boxed{(8,3)}$	$\boxed{(8,6)}$	$(8,7)$
$(15,0)$	$(15,1)$	$(15,4)$	$(15,5)$	$(15,2)$	$(15,3)$	$(15,6)$	$(15,7)$
$\boxed{(16,0)}$	$(16,1)$	$(16,4)$	$(16,5)$	$(16,2)$	$\boxed{(16,3)}$	$\boxed{(16,6)}$	$(16,7)$
$(17,0)$	$(17,1)$	$(17,4)$	$(17,5)$	$(17,2)$	$(17,3)$	$(17,6)$	$(17,7)$

Figure 4: Mapping a two-dimensional array. Layout of array elements on processor arrangement with $p = k = 3$ in the vertical dimension and $p = k = 2$ in the horizontal dimension. The boxed pairs are $A(0:17:2, 0:7:3)$.

memory stride pattern, the latter being generated by the transitions of a FSM. For a multidimensional array, each dimension will be characterized by a (p, k) pair and have a FSM associated with it, and each processor will execute a set of nested loops. Instead of the start address being fixed as it was before, it too will be generated by the transitions of a FSM. For simplicity, we will deal with a two-dimensional array. The extension to higher dimensions is straightforward. An array dimension with $p = 1$ is "mapped to memory".

Consider a one-level mapping of an array $A(0:17, 0:7)$ with parameters $(p_1, k_1) = (3, 3)$ in the first dimension and $(p_2, k_2) = (2, 2)$ in the second dimension, and let the section of interest be $A(0:17:2, 0:7:3)$. The picture is shown in Figure 4.

Let each block be laid out in column-major order in local memory. Thus, the sequence of index values corresponding to successive memory elements of processor (0,0) is {(0,0), (1,0), (2,0), (9,0), (10,0), (11,0), (0,1), ...}. Such a decomposition is used by Dongarra *et al.* for dense linear algebra codes on distributed-memory machines [4]. Then the tables corresponding to the two dimensions of A are

$T_1 =$

STATE	NEXT	$\Delta\mathcal{M}$
0	2	2
1	0	2
2	1	2

and

$T_2 =$

STATE	NEXT	$\Delta\mathcal{M}$
0	1	30
1	0	6

.

(Had the section been larger in the second dimension, the column accessed by processor $(0, 0)$ immediately after column 0 would be column 9. The elements $(0, 0)$ and $(0, 9)$ would be 30 locations apart in local memory. This explains the $\Delta\mathcal{M}$ entry in the first row of T_2.)

The FSM T_2 is used to generate memory spacings for successive base addresses, while the FSM T_1 is used to generate memory spacings between successive elements with the same base address. Algorithm 3 computes these FSMs.

Algorithm 3 (Memory access sequence for a multidimensional array with a two-level mapping.)
Input: Number of dimensions d; alignment parameters (a_j, b_j), layout parameters (p_j, k_j), dimension length n_j, regular section $A(l_j : h_j : s_j)$, processor number m_j, for each dimension j $(1 \le j \le d)$.
Output: $\Delta\mathcal{M}_j$, *length*$_j$, *startmem*$_j$, *lastmem*$_j$ $(1 \le j \le d)$.
Method:

integer *j, mem, m, k*

mem $\leftarrow$ *1*
for *j = 1* **to** *d* **do**
    $(\Delta\mathcal{M}_j$, *length*$_j$, *startmem*$_j$, *lastmem*$_j$, *m*$)$ $\leftarrow$ *Algorithm 2*$(a_j, b_j, p_j, k_j, n_j, l_j, h_j, s_j, m_j)$
    startmem$_j$ $\leftarrow$ *startmem*$_j \cdot$ *mem*
    lastmem$_j$ $\leftarrow$ *lastmem*$_j \cdot$ *mem*
    for *k = 0* **to** *length*$_j - 1$ **do**
        $\Delta\mathcal{M}_j[k] \leftarrow \Delta\mathcal{M}_j[k] \cdot$ *mem*
    enddo
    mem $\leftarrow$ *mem* $\cdot$ *m*
enddo
return $\Delta\mathcal{M}_j$, *length*$_j$, *startmem*$_j$, *lastmem*$_j$ $(1 \le j \le d)$

For the two-dimensional case, each processor executes the following doubly-nested loop. Higher-dimensional cases follow the same pattern, with one loop for each dimension.

base$_2$ $\leftarrow$ *startmem*$_2$
i$_2$ $\leftarrow$ *0*
while *base*$_2 \le$ *lastmem*$_2$ **do**
    base$_1$ $\leftarrow$ *startmem*$_1$
    i$_1$ $\leftarrow$ *0*
    while *base*$_1 \le$ *lastmem*$_1$ **do**
        compute with address *base*$_2$ + *base*$_1$

$base_1 \leftarrow base_1 + \Delta\mathcal{M}_1[i_1]$
$i_1 \leftarrow (i_1 + 1) \bmod length_1$
enddo
$base_2 \leftarrow base_2 + \Delta\mathcal{M}_2[i_2]$
$i_2 \leftarrow (i_2 + 1) \bmod length_2$
enddo

5 Generating communication sets

The set of messages that a given processor m must send while performing its share of the computation is called its *communication set*. We now extend the FSM approach to generate communication sets for regular communication patterns such as shifts and stride changes. In this section, we assume a one-dimensional array with a single-level mapping. Extensions to multilevel and multidimensional cases are as in the address generation problem.

We adopt Koelbel's execution model for parallel loops [9], in which a processor goes through four steps:

1. Send those data items it holds that are required by other processors.
2. Perform local iterations, *i.e.*, iterations for which it holds all data.
3. Receive data items from other processors.
4. Perform nonlocal iterations.

We combine the final two steps into a receive-execute loop.

Our approach puts all the intelligence on the sending side, keeping the receive-execute loop simple. We wish to minimize the number of messages sent and maximize cache benefits. To this end, each processor makes a single pass over the data in its local memory using the FSM technique described above, figuring out the destination of each data element and building messages. The messages contain (address, value) pairs which the receiving processor uses to perform its computations. All the data that must go from processor i to processor j are communicated in a single message. The data items corresponding to the local iterations are conceptually sent by the processor to itself. Of course, this message is never sent; rather, the local iterations are performed out of the message buffer. The idea is similar to the inspector-executor model [13] for irregular loops.

5.1 Shifts

Consider the computation

$$A(0\!:\!h\!:\!s) = A(0\!:\!h\!:\!s) + A(\ell\!:\!\ell + h\!:\!s),$$

where the regular section $A(\ell\!:\!\ell + h\!:\!s)$ is to be moved to align with $A(0\!:\!h\!:\!s)$. (For simplicity, assume $\ell > 0$.) In this case, a processor communicates with at most two processors. We show how to augment the FSM with a $\Delta\mathcal{P}$ and a $\Delta\mathcal{L}$ column, such that adding these quantities to the processor and memory location of the current element gives the remote processor number and the memory location in the remote processor of the element with which it interacts.

Suppose we know that element $A(i)$ is located on processor m at memory location $\mathcal{M}(i;p,k)$ with offset $O_i = \mathcal{O}(i;p,k)$. Then the problem is to find the processor and local memory location of the element $A(i-\ell)$. Let $\ell = q \cdot pk + r$, and $\Delta\mathcal{P} = \lceil (r - O_i)/k \rceil$. Then $\mathcal{P}(i-\ell;p,k) = (m - \Delta\mathcal{P} + p) \bmod p$. Since $0 \le O_i < k$, $\Delta\mathcal{P}$ can assume only two values as the offset O_i varies.

We also define the quantity $\Delta\mathcal{L}$ such that $\mathcal{M}(i-\ell;p,k) = \mathcal{M}(i;p,k) + \Delta\mathcal{L}(O_i)$.

$$\begin{aligned}\Delta\mathcal{L}(O_i) &= ((O_i - r) \text{ mod } k) - O_i - kq - \eta, \\ \eta &= \begin{cases} k & \text{if } (mk - r + O_i) < 0, \\ 0 & \text{otherwise.} \end{cases}\end{aligned}$$

5.2 Stride changes

Now consider the communication required to move the regular section $A(0\!:\!s_2n\!:\!s_2)$ to align with the regular section $A(0\!:\!s_1n\!:\!s_1)$. (Change of distribution of an array is also equivalent to change of stride.) This communication action is harder than a shift; there is no simple lookup technique for generating the communication sets. The problem is that the pattern of destination processors can have period longer than k. The only fact that appears to be true in this case is that the sum of the periods over all processors divides pk. We therefore resort to a simple technique.

We generate the FSM to reflect the storage properties of the s_2-section. Now, the j^{th} element of this section interacts with index $i = j \cdot s_1$ of A. Let $q_1 = i$ div pk, $r_1 = i$ mod pk, $q_2 = r_1$ div k, and $r_2 = r_1$ mod k. Then the desired element is stored in memory location $q_1 \cdot k + r_2$ on processor q_2.

Many modern RISC microprocessors perform integer division and remainder operations in software, which are rather slow. The equations above can be rewritten to require only two integer divisions, which substantially speeds up the implementation. If either p or k is a power of two, the computation can be sped up further by using shifts and bit masks instead of the division operations. Our implementation incorporates these optimizations.

5.3 Termination

Each processor must determine when it has completed all its operations. As we do not send messages between every pair of processors, we cannot use the number of messages to determine termination. However, it is easy to figure out the total number of operations that processor m will execute (by a variant of Algorithm 1). A simple termination test initializes a counter to this value, decrements it by the number of array elements received in each message, and exits when the counter reaches zero.

6 Experimental results

In this section we present experimental results of implementing the algorithms above. We measure both the preprocessing cost of table generation and the runtime loop overhead.

Our computational kernel is the DAXPY operation

```
y(1:n:s) = y(1:n:s) + a*x(1:n:s).
```

The ratio of memory operations to computation is high in this kernel, making it unfavorable for our algorithms. The overhead due to irregular memory access patterns would decrease with more computation in the loop. All experiments reported in this section were done on a Sparcstation 2, for which Dongarra [3] reports a 100×100 double-precision LINPACK performance of 4.0 Mflops. (We measured 3.3 Mflops.) A repetition of the experiments on a single processor of an iPSC/860 showed similar characteristics. We used the GNU C compiler, with optimization at the `-O2` level. A higher optimization level increased compilation time without improving the quality of the generated

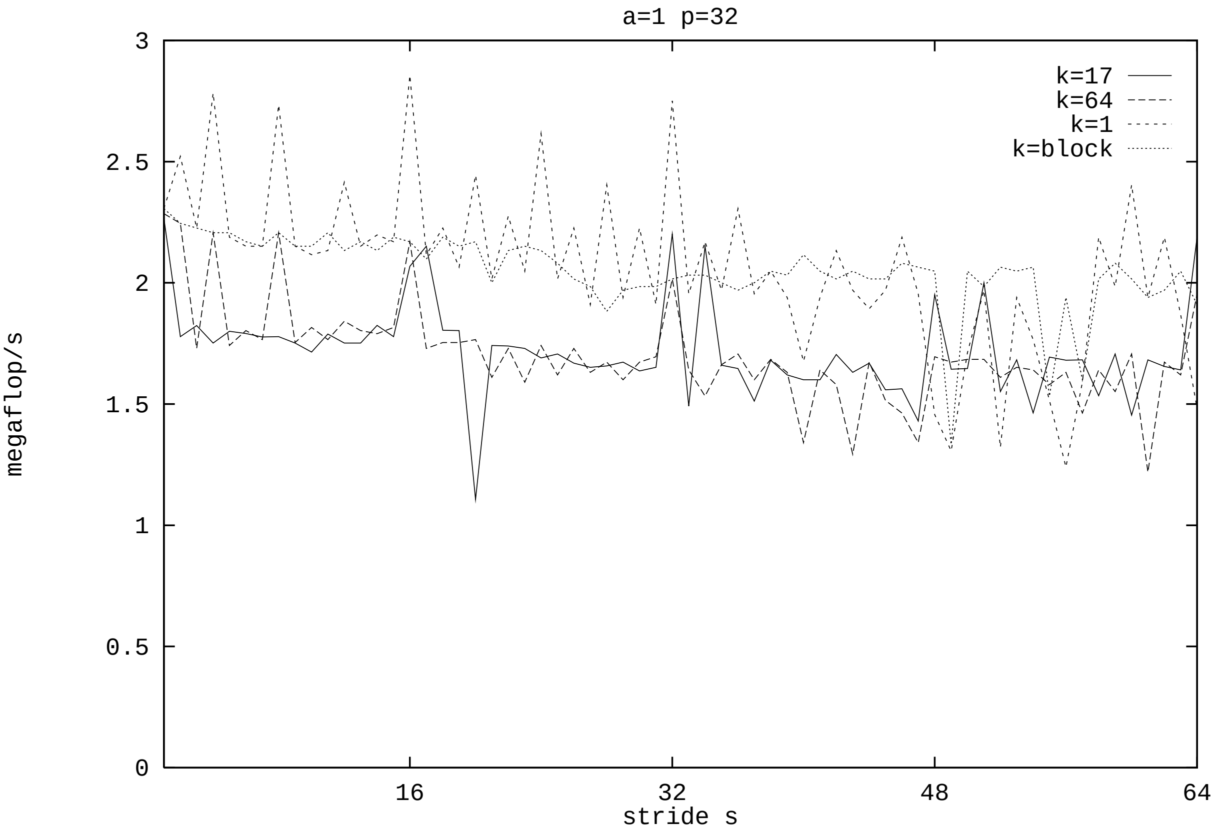


Figure 5: Performance of DAXPY for various block sizes k.

code. With $s = 1$ the loop compiled into 9 instructions, while the general case with table lookup compiled into 17 instructions. Our measured performance for the $s = 1$ case was 2.0 Mflops for long vectors.

6.1 Local address generation

We have seven parameters for a one-dimensional array: two alignment parameters a and b, two layout parameters p and k, and three section parameters ℓ, h, and s. This is too large a space over which to present results. We now explain how we reduce the number of variables.

- **Alignment stride a:** For any constant c, the memory layout for parameters ca and ck is exactly the same as that for a and k. Thus we experiment only with $a = 1$; the effect of changing a can be simulated by changing k instead.

- **Alignment offset b:** This is irrelevant for large sections. For small sections where end effects could be important, we observed that changes in b affected results by less than 5%. We therefore take $b = 0$.

- **Section lower bound ℓ:** Again, this is irrelevant for large sections and has negligible effect even for small sections. We therefore take $\ell = 0$.

- **Section upper bound h:** This affects the number of elements in the DAXPY, which in turn affects the tradeoff between loop startup and iteration time in a single processor. We vary h along with s, keeping h/s fixed, so that all our experiments do the same amount of work regardless of stride.

- **Section stride s:** This is the x-axis of our plots, as one of the two independent variables.
- **Number of processors p:** We run our experiments with a fixed number of processors. Varying the number of processors has the effect of scaling the plots.
- **Block size k:** This varies from plot to plot, as the second of the two independent variables.

We use the execution time per element of the DAXPY loop as our measure of performance. For a given set of parameters, we compute the tables for each processor and time the execution of the distributed loop. Let τ_m be the time taken by processor m to execute the n_m iterations it receives. Then the execution time per element is

$$t_e = \max_m \frac{\tau_m}{n_m}.$$

This factors out the effect of load imbalance.

We plot $2/t_e$ (the megaflops per second per processor) against stride s for various block sizes k. The plots are shown in Figure 5. The curves are the sum of two effects.

1. **Baseline:** The lower envelope is a smooth, relatively flat function of s. The value of the envelope is approximately the asymptotic performance for the single-processor DAXPY.
2. **Jiggles:** The plot jiggles up and down some, presumably due to cache stride effects [2].

Although the general loop has almost twice as many instructions as the unit-stride loop, the difference in performance is much less. This is because the execution time is dominated by the floating-point operations and references to operands in memory. The references to the $\Delta\mathcal{M}$ array

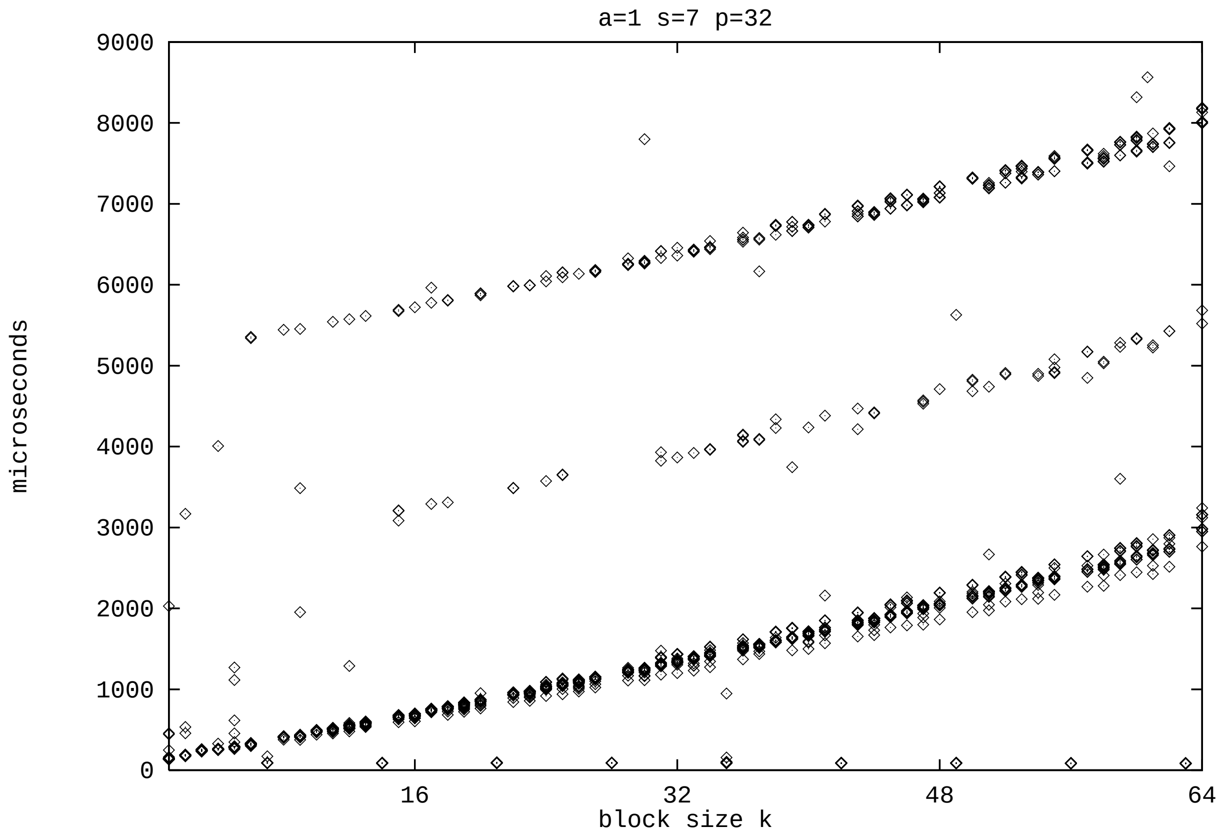


Figure 6: Preprocessing overhead for memory stride computations. The graph corresponds to an alignment stride of 1. The multiple points for each block size show the processing times for each of 32 processors.

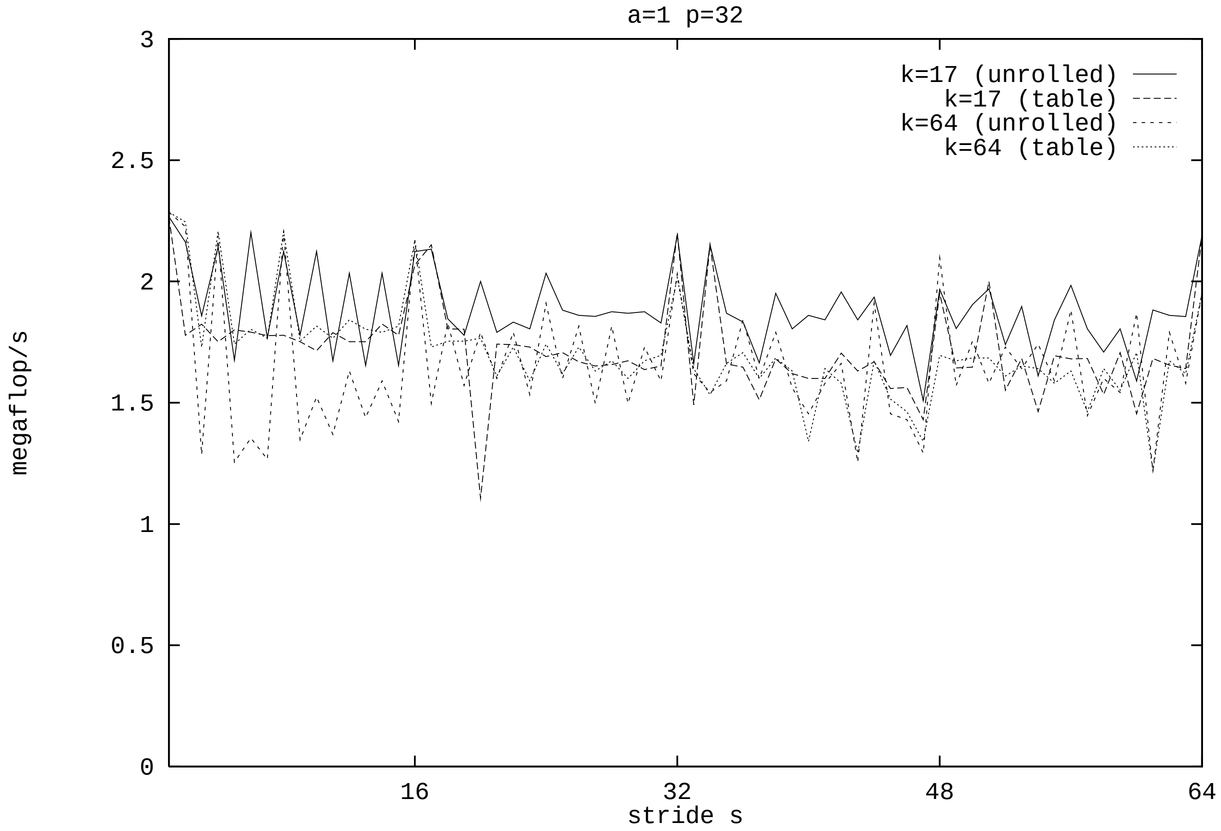


Figure 7: Effect of loop unrolling on performance of DAXPY for various block sizes k.

usually hit in cache, and the additional loop control instructions probably execute in overlapped fashion.

The preprocessing time required to compute the $\Delta\mathcal{M}$ tables is shown in Figure 6, where we plot the running time of Algorithm 1 against block size k. The running time is approximately $O(k)$, with the special cases discussed in Section 2.2 being handled much faster. Experimental observation confirmed that the running times for Algorithm 2 were no more than twice those of Algorithm 1.

If the $\Delta\mathcal{M}$ sequence is known at compile time, unrolling the loop bodies avoids the indirection into the $\Delta\mathcal{M}$ array. Figure 7 shows that the benefits of this are mixed. While loop unrolling removes the indirection, it also increases the size of the loop body; thus the benefits of a simpler loop body may be undone by misses in the instruction cache caused by the larger loop body. This is a factor for larger block sizes, since the amount of unrolling may be as large as k. We recommend unrolling only if the $\Delta\mathcal{M}$ sequences are short, or can be compacted, *e.g.*, by run-length encoding.

6.2 Communication set generation

For the communication set generation algorithms, we measured the time required to generate the FSM tables and the time required to marshal the array elements into messages. We did not measure the actual communication time, since that varies widely among machines. The results are presented in Table 2.

7 Related work

Solutions to many special cases of these problems have been known for a long time, but the general solution did not appear in the literature or in any implementation until very recently. In this section, we present relevant work by other researchers and discuss how these relate to the work reported in this paper.

Koelbel and Mehrotra [10] treat special cases where the array is mapped using a *block* or a *cyclic* mapping, and the regular section has unit stride. Koelbel [9] handles non-unit-stride regular sections with *block* or *cyclic* mappings. Their algorithms are essentially reproduced in the special

(a)

k	s	ℓ	FSM gen. time (μs)	Run time/elt (μs)
1	3	10	360	3.3
17	3	10	1831	3.1
17	3	40	1870	3.1
64	3	10	6632	3.2
64	3	100	6508	3.2
blk	3	10	408	3.2

(b)

k	s_1	s_2	FSM gen. time (μs)	Run time/elt (μs)
1	3	5	360	5.3
17	3	5	1891	14.3
64	3	5	6542	5.4
blk	3	5	400	11.3

Table 2: Performance of communication set generation algorithms. All numbers are for 32 processors. (a) Shift communication: the shift distance is ℓ. (b) Change of stride: the stride is changed from s_2 to s_1.

cases that we recognize (Section 2.2).

MacDonald *et al.* [12] discuss the problem in the context of Cray Research's MPP Fortran, and conclude that a general solution requires unacceptable address computation overhead in inner loops. They therefore restrict block sizes and number of processors to powers of two, allowing the use of bit manipulation in address computation. Experimental implementation of our solution shows that address computation and interprocessor communication can be performed efficiently without these restrictions.

Gupta *et al.* [6] and Stichnoth [15] have independently come up with a different general solution based on the observation that the active elements within a processor can be described as the union of at most k regular sections. Under this scheme, each dimension of an array is processed by a doubly-nested loop: one over offsets, the other over courses corresponding to a fixed offset. While these loops have constant stride, it is possible for both schemes to have bad cache performance. The schemes can be extended to two-level mappings, but neither paper gives explicit formulas for handling this case. Stichnoth's communication set generation algorithm trades memory locality for buffer space: it reuses a single buffer but makes multiple passes over the array data.

Ancourt *et al.* [1] use a linear algebraic approach based on the Hermite Normal Form to generate code for such programs in the general case. They provide a scheme that requires a small amount of additional storage but still results in constant-stride loops. Once again, their scheme uses a doubly-nested loop for each dimension.

With at least three different general solutions to this problem available in the literature, it would be useful to understand the tradeoffs among the solutions and perform extensive tests of when each performs best. We hope to undertake such a performance evaluation study in the near future.

8 Conclusions

Our table-driven approach to the generation of local memory addresses and communication sets unifies and subsumes previous solutions to these problems. The tables required in our method are easy to compute, and the runtime overheads due to indirect addressing of data are small. Our data access patterns exploit temporal locality and make effective use of cache. We also support blocking of messages to reduce message startup overheads.

Our prototype implementation has demonstrated our performance claims. An optimized implementation of our techniques should deliver close to maximum performance for general $cyclic(k)$ distributions of arrays.

The full-fledged table-driven approach is only required in the general case of *cyclic*(k) distributions. In many of the special cases, such as *block* and *cyclic* distributions, local memory accesses are constant-stride, and the algorithms can be simplified considerably. In Section 2.2, we identified several important special cases that benefit from this optimization.

The table-driven algorithm is useful when we need to find local memory mappings of a sequence of array elements. It is not efficient for computing the local memory mapping for a single array element. Equations (1)–(5) should be used in this case.

Generating the lookup tables and using them to generate local memory addresses are analogous to the roles of the inspector and executor loops in Saltz's techniques for parallelizing irregular loops [13]. Ideally, we would evaluate the lookup tables at compile time and use their values at runtime. This may not be possible because if we do not have numerical values for all the necessary parameters at compile time. As the structure of the FSM does not depend on the starting and ending section indices (ℓ and h), we could construct this FSM at compile time and have each processor extract its cycle and start state information at runtime. This would also be useful in situations where the starting offset of the section varied, *e.g.*, within a do-loop. In general, we expect that there will be significant opportunities to reuse a single FSM table for multiple computations, thus amortizing the overhead of generating the tables.

Our algorithms are fully parallel in that each processor of a parallel machine can generate its tables independently. Finding a fully parallel algorithm running in $O(k)$ parallel time or with $O(p + k)$ total work remains an open problem.

Acknowledgments

J. Ramanujam proofread an early draft and suggested the simplifications to Algorithm 1. Fabien Coelho raised several questions concerning the original conference version of the paper, which we have tried to address here.

References

[1] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution. In *Workshop on Compilers for Parallel Computers, Delft*, Dec. 1993.

[2] D. H. Bailey. Unfavorable strides in cache memory systems. RNR Technical Report RNR-92-015, Numerical Aerodynamic Simulation Systems Division, NASA Ames Research Center, Moffett Field, CA, May 1992.

[3] J. J. Dongarra. Performance of various computers using standard linear equations software. *Computer Architecture News*, 20(3):22–44, June 1992.

[4] J. J. Dongarra, R. van de Geijn, and D. W. Walker. A look at scalable dense linear algebra libraries. In *Proceedings of the Scalable High Performance Computing Conference*, pages 372–379, Williamsburg, VA, Apr. 1992. IEEE Computer Society Press. Also available as technical report ORNL/TM-12126 from Oak Ridge National Laboratory.

[5] G. C. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu. Fortran D language specification. Technical Report Rice COMP TR90-141, Department of Computer Science, Rice University, Houston, TX, Dec. 1990.

[6] S. K. S. Gupta, S. D. Kaushik, S. Mufti, S. Sharma, C.-H. Huang, and P. Sadayappan. On compiling array exprressions for efficient execution on distributed-memory machines. In A. N. Choudhary and P. B. Berra, editors, *Proceedings of the 1993 International Conference on Parallel Processing*, volume II, pages 301–305. CRC Press, Inc., Aug. 1993.

[7] High Performance Fortran Forum. High Performance Fortran language specification version 0.4. Draft, Nov. 1992.

[8] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley Publishing Company, Reading, MA, second edition, 1981.

[9] C. Koelbel. Compile-time generation of regular communication patterns. In *Proceedings of Supercomputing'91*, pages 101–110, Albuquerque, NM, Nov. 1991.

[10] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, Oct. 1991.

[11] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel. *The High Performance Fortran Handbook*. Scientific and Engineering Computation. The MIT Press, Cambridge, MA, 1994.

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

[13] R. Mirchandaney, J. H. Saltz, R. M. Smith, K. Crowley, and D. M. Nicol. Principles of runtime support for parallel processors. In *Third International Conference on Supercomputing*, pages 140–152. ACM Press, July 1988.

[14] I. Niven and H. S. Zuckerman. *An Introduction to the Theory of Numbers*. John Wiley & Sons, New York, NY, 1964.

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

Biographies

Siddhartha Chatterjee received his B.Tech. degree in electronics and electrical communications engineering from the Indian Institute of Technology (Kharagpur) in 1985, and his Ph.D. in computer science from Carnegie Mellon University in 1991. He is currently a Postdoctoral Scientist at the Research Institute for Advanced Computer Science (RIACS) at NASA Ames Research Center in Moffett Field, California. His research interests include the design and implementation of data-parallel languages for irregular problems, and compiler techniques for automatic data placement on distributed-memory parallel machines.

John R. Gilbert received his bachelor's degree in mathematics at the University of New Mexico in 1973 and his PhD in computer science at Stanford in 1981. From 1981 to 1988 he was Assistant and Associate Professor of Computer Science at Cornell. Since 1988 he has been a member of the Computer Science Laboratory at the Xerox Palo Alto Research Center. He has held visiting appointments at the University of Iceland and at the University of Bergen, Norway. His research is in parallel computing, geometric and combinatorial algorithms, and sparse matrix computation.

Fred J. E. Long received his B.A. degree in computer science from the University of California at San Diego in 1990. He is currently a Software Engineer with Systems Engineering Associates in San Diego. His research interests include optimizing and parallelizing compilers.

Robert Schreiber is a Senior Scientist at the Research Institute for Advanced Computer Science, NASA Ames Research Center. He received a PhD in Computer Science from Yale University in 1977. He is known for important basic research in sequential and parallel algorithms for sparse matrix computation, is a contributor to the Matlab scientific computing environment, and is a codeveloper of the High Performance Fortran programming language. He has written over seventy scientific papers. He is on the editorial boards of three journals in the scientific and supercomputing fields. Schreiber has held faculty positions at Stanford University and Rensselaer Polytechnic Institute and was Chief Scientist of Saxpy Computer Corporation. He has broad experience as a consultant in the scientific and technical computing industry.

Shang-Hua Teng is a faculty member in the Department of Applied Mathematics and Laboratory for Computer Science at Massachusetts Institute of Technology. He received his Ph.D. degree in computer science from Carnegie Mellon University in 1991, the M.S. degree in computer science from the University of Southern California in 1987, and the B.S. degree in computer science and electrical engineering from Jiao Tong University, Shanghai, in 1985. His research interests include parallel scientific computing, the theory of parallel and distributed computing, compiler optimization, computational geometry, and cryptography.