1988

A Formalism for Describing Data Distribution

Charles Koelbel

Report Number:
88-803

A FORMALISM FOR DESCRIBING DATA DISTRIBUTION

Charles Koelbel

CSD TR-803
August 1988

A Formalism for Describing Data Distribution

Charles Koelbel
Department of Computer Sciences
Purdue University

Abstract

In many existing and planned parallel machines, memory cannot be considered as a single homogeneous resource. Instead, each processor has a "local" section of memory which is more accessible than others. Because of this ease of access, it is necessary to distribute the data across the system so that most references are made to local data. In this paper, we give a mathematical description of data distribution in parallel machines. We then show its application to strip mining, a common transformation for converting sequential programs to run on parallel hardware. Strip mining using data distribution information enhances the locality of reference in the resulting program, thus speeding performance.

1 Introduction

An important recent development in computer science has been the advent of parallel computer architectures. Because sequential computers are approaching fundamental physical limits on their performance, parallel machines are seen as the next major advance in high-performance computing machinery. This has led to work both in constructing parallel hardware and writing parallel software.

Many parallel machines have been designed and built. One fundamental design decision in these machines has been the memory structure used

in each. Current parallel machines can be roughly grouped into three categories:

- (Pure) shared memory architectures
- Non-shared memory architectures
- Hybrid architectures

Pure shared memory architectures are conceptually the simplest class. They have a single region of memory which can be accessed equally quickly from all processors. Generally, this is implemented by a bus connecting the processors to the shared memory. Examples of this architecture include machines from Encore [1] and Sequent [11] corporations. Non-shared memory machines represent the opposite extreme of the design space. In these machines, each processor is connected to a local memory which no other processor can access. Processors coordinate activity by explicit messages sent via a communications network. Examples of this class of architecture include the Intel Hypercube [7] and Cosmic Cube [10] machines. Finally, hybrid architectures bridge the gap between these two extremes. At least some memory is shared by all processors, but it is not homogeneous. Each section of memory is local to one processor, which enjoys faster access to that section than to other sections. Two examples of this organization are the University of Illinois Cedar [3] and IBM RP3 [8] architectures.

A major consideration in programming both the non-shared memory and hybrid classes of machines is the placement of data. Because of the difference in access times (or even the possibility of access), it is vital that data be spread among the sections of memory so that most accesses are made to local data. The problem of arranging this is called the *data distribution problem*. The usual approach to data distribution is a static distribution of the data. A *data distribution pattern* is chosen for each array to be distributed which describes where each element will be stored. The computation is then structured so that each processor does as much computation with its local data as possible. In the next section, we give a mathematical description of data distribution and apply our description to several common distribution patterns. The following section shows an application of this formalism to transforming sequential programs for parallel execution. There, we try to restructure the computation to use local data as much as possible.

2 The Model

We describe a data distribuion by giving the set of elements stored on each processor. Mathematically, this can be modeled as a function from processors to sets of array elements. For *Proc* the set of processors and *Elem* the set of elements of an array A, we define the function

$$local : Procs \rightarrow 2^{Elem}$$

by

$$local(p) = \{a \in Elem \mid a \text{ is stored locally on } p\}$$

In the examples that follow, we will represent *Proc* and *Elem* by their index sets, which will be tuples of small integers. Also, if there is an ambiguity as to the identity of the array, we will use the array name as a subscript.

Other approaches to data distribution [9,4] have taken a different path toward formalizing the distribution. Generally, these approaches define a function

$$proc : Elem \rightarrow Proc$$

which gives the processor storing each element. If every element is stored on exactly one processor, then the two methods are equivalent. (In this case, *local* is simply $proc^{-1}$.) If an element can be stored on several processors, however, the two methods lead to different results. Some theoretical methods for converting shared-memory programs to nonshared-memory programs lead to several copies of shared data. In practice, it is also common to have some "overlap" between the regions stored on neighboring processors to reduce communication. It is not obvious how such a distribution scheme could be modeled using a single-valued *proc* function. Using the above *local* function, no problem arises from such distribution schemes; the only consequence is that the *local* sets of distinct processors are not disjoint. Because of this added generality, we prefer the *local* function approach given here.

2.1 One-Dimensional Distribution Patterns

In this section, we assume that the array to be distributed has N elements and there are P processors available. We also use 0-based indexing for both

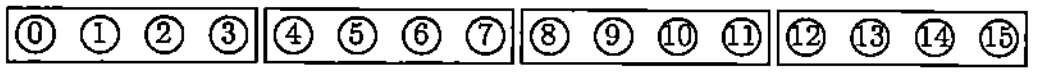

Figure 1: Block distribution of one-dimensional array

arrays and processors, making the index sets

$$\begin{aligned} Proc &= \{0, 1, 2, \ldots, P-1\} \\ Elem &= \{0, 1, 2, \ldots, N-1\} \end{aligned}$$

To avoid unnecessary complication of the formulas, we will assume that N is divisible by P.

The most common distribution pattern for one-dimensional arrays is *block* distribution. This pattern groups the array elements into contiguous subsets, storing each subset on a single processor. For example, if $N = 1000$ and $P = 10$, then processor 0 would store elements 0 through 99; processor 1 would store elements 100 through 199; and so on. Figure 1 illustrates this for $N = 16$ and $P = 4$. The corresponding *local* function is

$$local(p) = \left\{ i \;\middle|\; \frac{N}{P} \cdot p \leq i < \frac{N}{P} \cdot (p+1) \right\}$$

Storing contiguous groups of elements together in this way tends to reduce communication if many references are made to "neighboring" elements.

Cyclic distribution stores every Pth element on the same processor. For example, if $N = 1000$ and $P = 10$, then processor 0 would store elements 0, 10, 20, and so on; and processor 1 would store elements 1, 11, 21, etc. Figure 2 illustrates this for $N = 16$ and $P = 4$. Processors in that figure are labeled as "P0," etc.; notice that each processor appears several times. The *local* function is defined as

$$local(p) = \{ i \mid i \equiv p \pmod{P} \}$$

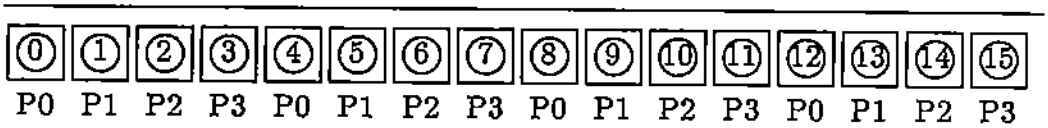

Figure 2: Cyclic distribution of one-dimensional array

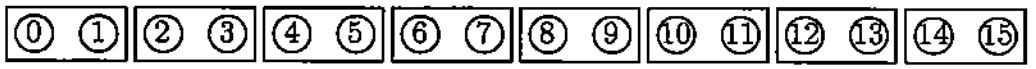

Figure 3: Block-cyclic distribution of one-dimensional array

This pattern is often useful if only a subrange of the original array will be used. Cyclic distribution then distributes the workload relatively evenly, while a block pattern would leave some processors idle.

Finally, the above patterns can be combined using a *block-cyclic* scheme. This pattern uses a parameter K. The array is divided into contiguous blocks of size K which are then distributed cyclically among the processors. For example, if $N = 1000$, $P = 10$, and $K = 200$ then processor 0 would store elements 0–19, 200–219, 400–419, 600–619, and 800–819. Other processors would have similar sets of elements. Figure 3 illustrates this pattern for $N = 16$, $P = 4$, and $K = 2$. The processors there are labeled "Proc0," etc.; note that each processor occurs twice. The corresponding *local* function is

$$local(p) = \left\{ i \;\middle|\; \left\lfloor \frac{i}{K} \right\rfloor \equiv p \pmod{P} \right\}$$

This pattern is a compromise between block and cyclic patterns, and has performance intermediate between them. In fact, block and cyclic distributions can be considered as special cases of block-cyclic distribution. When $K = 1$, the distribution is simple cyclic distribution, while block distribution occurs when $K = N/P$.

2.2 Multi-Dimensional Distribution Patterns

In this section, we examine distributions for multi-dimensional arrays. We assume that dimension i of the array has 0-based indexing of N_i elements. Thus, a two-dimensional matrix would have an index set of

$$Elem = \{0, 1, \ldots, N_1 - 1\} \times \{0, 1, \ldots, N_2 - 1\}$$

Similar assumptions hold for the processor index set. As before, we will assume that quantities are divisible when it simplifies the formulas.

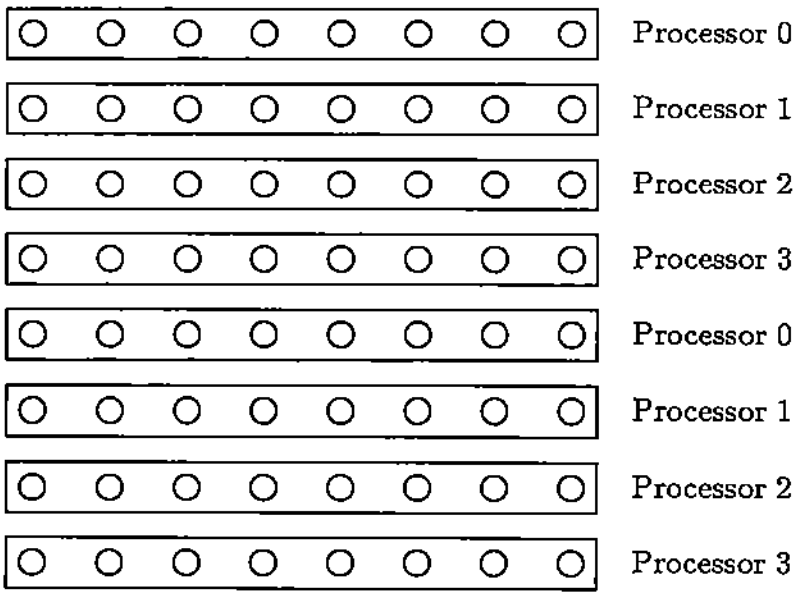

Figure 4: Cyclic distribution of rows

The simplest distribution patterns for multi-dimensional arrays are obtained by applying one-dimensional distribution patterns to a single dimension and not distributing any other dimension. For example, the rows of a matrix could be cyclically distributed using the function

$$local(p) = \{(i,j) \mid i \equiv p \pmod{P}\}$$

Figure 4 shows this distribution pattern for $N_1 = 8$ and $P = 4$; note that each processor stores two rows of the matrix. A generalization of this approach is to distribute several dimensions independently on a multi-dimensional processor set. An example of this is the *two-dimensional blocked* distribution shown in Figure 5 for $N_1 = N_2 = 8$ and $P_1 = P_2 = 4$. The *local* function in this case is

$$\begin{aligned} local(p_1,p_2) = \{(i,j) \mid\ & \frac{N_1}{P_1} \cdot p_1 \le i < \frac{N_1}{P_1} \cdot (p_1+1) \\ & \text{and } \frac{N_2}{P_2} \cdot p_2 \le i < \frac{N_2}{P_2} \cdot (p_2+1)\} \end{aligned}$$

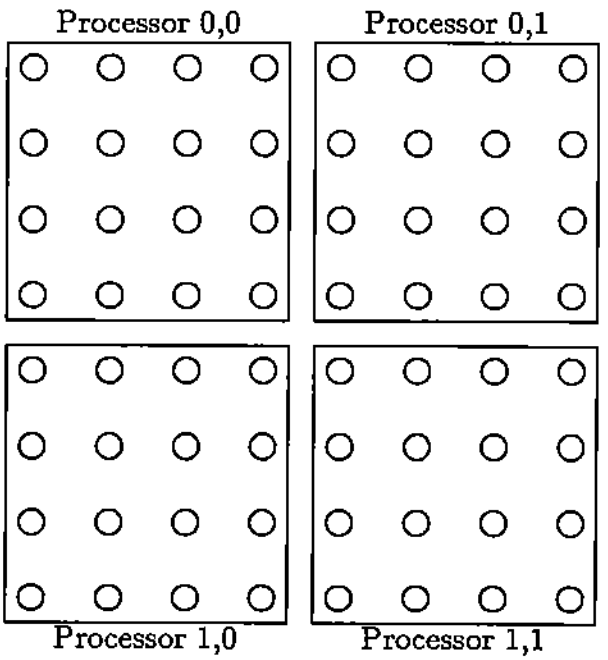

Figure 5: Two-dimensional blocked distribution

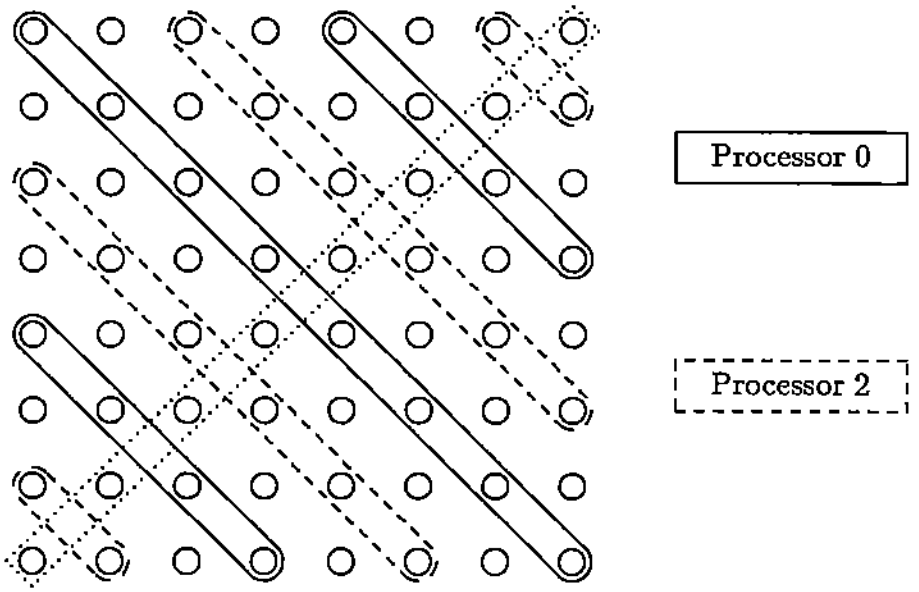

Figure 6: Skewed distribution

The *skewed* distribution pattern is often used to pipeline computations on a two-dimensional matrix. The basic idea is to give each processor several "slices" of the array parallel to the main diagonal. Figure 6 illustrates this idea for $N_1 = N_2 = 8$ and $P = 4$. Processor 0 stores the elements contained in the solid ovals while processor 2 stores those in the dashed ovals. The elements stored on processors 1 and 3 are not distinguished in the figure; they are the elements not in any oval. The relevant *local* function is

$$local(p) = \{(i,j) \mid i - j \equiv p \pmod{P}\}$$

This is an effective organization when the elements in the antidiagonal strips (for example, the dotted box in Figure 6) can be computed independently.

```
for i in range(A) loop
    A[ i ] := sin(B[i]) * cos(C[i]);
end;
```

(a)

```
coprocess p in 1..P on Proc[p] do
    for i in range(A,Proc[p]) loop
        A[ i ] := sin(B[i]) * cos(C[i]);
    end;
end;
```

(b)

Figure 7: Simple loop (a) before and (b) after strip mining

3 An Application of the Formalism

In this section, we present strip mining as an application of the *local* function formalism. Strip mining is a common transformation in converting sequential programs for use on parallel architectures [12]. The basic idea is to divide the iteration space of a loop into a number of "strips" and to schedule one such strip on each processor. Figure 7 shows a simple example of this transformation.[1]

Data distribution can be used in two ways in strip mining. If data are distributed among processor memories then it is advantageous to strip mine according to that distribution. Such a scheme allows more references to be made to local data than would be made by an arbitrary stripping scheme. Some references must still be made to non-local data, however; these must be copied to local temporary storage. The *local* function can be used both to guide the strip-mining itself and to generate the needed copying instructions.

We now describe strip mining for the loop shown in Figure 8. This is

[1]Throughout this section, we use the BLAZE language as a basis for our examples. The **coprocess** statement initiates a set of parallel processes; **range** gives the subscript range of an array with one argument and the subrange stored on a particular processor with two arguments; other constructs should be self-explanatory. More details about the language can be found in [6].

a generalization of the loop shown in Figure 7a. The analysis shown here assumes that each element is stored on only one processor (that is, $p \neq q$ implies $ker(p) \neq ker(q)$). Only minor additions to the analysis would be needed if this were not true. At one point we will also make the standard assumption that f is a linear function. This is by far the most common case in real programs, and is generally assumed by researchers working on parallelizing sequential code.

3.1 Strip Mining

The performance of a strip-mined program depends on choosing a "good" collection of strips. To enhance locality of reference, we will choose strips so that any *storing* of values can be done locally. If the loop contains only a single assignment to an array A, as does the pseudocode loop of Figure 8, this can be accomplished by using the set

$$ref(p) = f^{-1}(local_A(p))$$

as the strip. (To allow for loops which do not access all elements of array A, this strip must be intersected with the original loop range.) Given that the loop accesses $A[f(i)]$, restricting processor p to iterate over the set $ref(p)$ ensures that the set of array elements assigned to will be $f(ref(p)) = f(f^{-1}(local_A(p))) = local_A(p)$. The loop resulting from strip mining Figure 8 in this way is shown in Figure 9. This locality rule clearly is not sufficient for all loops; we will generalize it later.

Before accepting this transformation, we must ensure that the strip mined loop computes the same results as the original sequential loop. This can be guaranteed if there is no interaction between two strips scheduled on separate processors. This type of independence is captured by the notion of data dependence, as defined by Allen [2] and Wolfe [12]. Their theses showed how data dependence could be checked by the compiler; we give only an overview of that analysis here.

Data dependence analysis recognizes situations in which a memory location is referenced by two separate statements (or by one statement executed repeatedly, as in a loop). If one statement assigns to a memory location while a second uses the location's value, there is said to be a data dependence between the two statements. Two statements can be executed in

```
for i ∈ Range loop
    A[f(i)] := ... A[f(i)] ...
               ... A[g(i)] ...
               ... B[h(i)] ...
end;
```

Figure 8: Example loop for strip mining

```
coprocess p ∈ Procs do
    for i ∈ Range ∩ f^{-1}(local_A(p)) loop
      A[f(i)] := ... A[f(i)] ...
                 ... A[g(i)] ...
                 ... B[h(i)] ...
    end;
end;
```

Figure 9: Example loop after strip mining

parallel when there are no dependences between them. If the statements of a data dependence are in a loop, there are two possibilities: the references causing the dependence can be made during the same loop iteration or during different iterations. If the references are made during different iterations, the dependence is *loop-carried*; otherwise, it is *loop-independent*. In general, a loop can be executed in parallel without introducing synchronization instructions if and only if it does not carry any dependence.

For loop-carried dependences, the concept of distance vectors is important. For a single loop, if the first reference causing a dependence occurs on iteration i_1 and the second reference on iteration i_2, then the distance vector associated with the dependence is $i_2 - i_1$. In the case of nested loops, this idea is extended to the vectors of the loop indices, with the convention that the index of the outermost loop is first in the vector. Thus, if the first reference comes on iteration (i_1, j_1) and the second on iteration (i_2, j_2), then the distance vector is $(i_2 - i_1, j_2 - j_1)$.

From the above discussion, we see that strip mining will be valid if there are no data dependences between two sets $ref(p)$ and $ref(q)$ where $p \neq q$. This can be stated more formally as

> Strip mining according to data distribution is valid if, for any data dependence from iteration i to iteration j in the original loop and any processor p, $i \in f^{-1}(local(p))$ implies $j \in f^{-1}(local(p))$.

Note that if both iterations i and j are in the same strip, then the dependence is satisfied by sequential execution on processor p.

If f is a completely general function, then the above condition is all that can be said about validity, and theorem proving will be necessary to validate the transformation. Since f is linear, however, we can make a stronger statement. To do this, we define the kernel ker of a distribution as the set of vectors which "don't change the processor" when added to an array subscript. That is, for each processor p,

$$ker(p) = \{d \mid \forall i \in local(p),\ i + d \in local(p)\}$$

where the $+$ operator represents vector addition, if appropriate. The most intuitive example of such a kernel is the set of dimensions which are not

distributed in a multi-dimensional array. For example, if an array is distributed by blocks of rows using the distribution function

$$local(p) = \left\{(i,j) \;\middle|\; \frac{N_1}{P} \cdot p \leq i < \frac{N_1}{P} \cdot (p+1)\right\}$$

then changing the column of a reference has no bearing on the processor storing the reference. Thus, the vector $(0, j) \in ker(p)$ for any j or p.

We can now restate the validity condition for strip mining according to the distribution pattern.

> Strip mining is valid if, for all distance vectors d of data dependences in the original loop and all processors p in the processor array, $f(d) \in ker(p)$.

Intuitively, this says that if no loop over a distributed dimension of the array carries a data dependence, then the nested loop can be strip mined. As proof of the condition, consider a dependence from iteration i to iteration $i+d$. If processor p executes iteration i, then $f(i) \in local(p)$ by the definition of $ref(p)$. Since $f(d) \in ker(p)$, we have $f(i) + f(d) \in local(p)$. But since f is linear, $f(i) + f(d) = f(i + d)$, so $i + d \in f^{-1}(local(p))$. Thus, the new condition implies the old one.

Figure 10 shows two loops which can be strip mined using this condition. Let C be distributed by blocks of rows, as given by the $local_C$ function in part (a). In both loops, the only dependence d is carried by the "column" loop. Since the array is distributed by rows, such "column" dependences do not prevent strip mining. Loop (c) presents a particularly interesting case. The usual condition on strip mining a set of nested loops is that the outermost loop may carry no dependence. In Figure 10c, the outer loop does carry a dependence, preventing ordinary strip mining. Strip mining according to data distribution allows the transformation, however, since the dependence will be satisfied on each sequential processor after the transformation.

The basic strip mining strategy given at the beginning of this section can be generalized in several ways. If the loop has several assignments to arrays with the same distribution patterns and subscript expressions then the new loop bounds can be calculated based on any of the assignment statements. These calculations will clearly give the same

$$local_C(p) = \left\{(i,j) \;\middle|\; \frac{\mathrm{N}}{\mathrm{P}} \cdot p \le i < \frac{\mathrm{N}}{\mathrm{P}} \cdot (p+1)\right\}$$
$$ker_C(p) = \{(i,j) \mid i = 0\}$$

(a)

```
for i in 1..N loop
    for j in 1..N loop
        C[i,j] := C[i,j] + C[i,j-1];
    end;
end;
```

$d = (0,1)$ $\qquad f(i,j) = (i,j) \qquad f(d) = (0,1) \in ker_C(p)$

(b)

```
for j in 1..N loop
    for i in 1..N loop
        C[i,j] := C[i,j] + C[i,j-1];
    end;
end;
```

$d = (1,0)$ $\qquad f(j,i) = (i,j) \qquad f(d) = (0,1) \in ker_C(p)$

(c)

Figure 10: (a) Array distribution and (b), (c) two loops to strip mine

$ref(p)$ sets. Similarly, if there are assignments to $A[f_1(i)]$ and $B[f_2(i)]$ and $f_1^{-1}(local_A(p)) = f_2^{-1}(local_B(p))$, then the loop can be strip mined using either f_1 or f_2. This often occurs when $f_1(i) - f_2(i) \in ker(p)$ for all i and p; for example, when two arrays are distributed by rows, and the subscripts differ only in the column entries. Furthermore, these cases can easily be extended from the single **for** loop shown here to nested loops by considering i in the formulas to be a vector. More complex cases, in which arrays with different distribution patterns or subscript expressions are updated, are being studied.

3.2 Copying

By strip mining according to the left-hand side of assignments in the original loop, we have ensured that any store will be to a local location. The possibility of reading from non-local storage still exists, however. In the loop of Figure 9, for example, the references to $A[g(i)]$ and $B[h(i)]$ could reference non-local memory. In order to correctly and efficiently execute the program, such non-local references must be copied to local storage. Our *local* formalism can be used to guide this copying.

Consider the reference $A[g(i)]$ in Figure 9. For each processor p there will be an set

$$loc(p) = g^{-1}(local_A(p))$$

containing the indices for which $A[g(i)]$ will be a local reference on that processor. If $ref(p) \subseteq loc(p)$ for all p, then the reference will always be satisfied locally in the strip-mined loop. This will be true in the common case when $f = g$. If $f(i) - g(i) \in ker(p)$ for all i and p, then $ref(p) = loc(p)$ and the condition is again satisfied. This occurs when the subscripts are the same for all distributed dimensions, but differ in the undistributed dimensions. If $ref(p) \not\subseteq loc(p)$, then any index $i \in ref(p) - loc(p)$ will cause a non-local reference by processor p in the strip-mined loop. (Processor p will execute loop iterations in $ref(p)$ and reference local memory during $loc(p)$; other iterations will cause non-local accesses.) This element must be brought into local memory. A similar analysis can be made for $B[h(i)]$, using the sets $loc(p) = h^{-1}(local_B(p))$.

Up to now, all of our analysis has been the same for hybrid shared memory and non-shared memory architectures. At this point, however, we

```
coprocess p ∈ Procs do
    for i ∈ g(Range ∩ (f^-1(local_A(p)) − g^-1(local_A(p)))) loop
        Temp_A[i] := A[i];
    end;
    for i ∈ h(Range ∩ (f^-1(local_A(p)) − h^-1(local_B(p)))) loop
        Temp_B[i] := B[i];
    end;
      ⋮      (main loop)
end;
```

Figure 11: Shared memory copying for Figure 9

must make a distinction between the two. The fundamental distinction between anonymous copying and message passing must be addressed when the copying transformation is formulated. We first consider shared memory copying, then the more complex case of non-shared memory message passing.

For shared memory architectures, it suffices to determine which non-local elements of an array may be accessed. A straightforward copying operation can then move these elements to local storage. If a given non-local array element will only be accessed once, then there is no need to allocate temporary storage for it; the element may simply be accessed directly, and discarded after use. This is often the case for "border" elements when an array is shifted in storage. If the non-local element will be accessed frequently, however, it is better to copy it once from non-local to temporary local storage and then use only the local copy. This minimizes non-local references and improves performance. An example of such a situation is the pivot row in Gaussian elimination; every processor must access this row once for each row it eliminates.

To perform the copying, it is only necessary to determine which non-local elements may be accessed. For a particular reference like $A[g(i)]$ in Figure 9, processor p will reference non-local elements on loop indices in $ref(p) - loc(p)$. The elements accessed on these iterations will be $g(ref(p) - loc(p))$. Thus, those elements should be copied to temporary storage. The copying instructions are shown in Figure 11. The main loop body has been omitted from that figure for brevity. Note that the loop indices apply

```
coprocess p ∈ Procs do
    send( p − 1,
      A[g(Range ∩ f^{-1}(local_A(p − 1)) ∩ g^{-1}(local_A(p)))] );
    send( p − 1,
      A[g(Range ∩ f^{-1}(local_A(p − 1)) ∩ h^{-1}(local_B(p)))] );
    Temp_A[g(Range ∩ f^{-1}(local_A(p)) ∩ g^{-1}(local_A(p + 1)))]
      := recv(p + 1);
    Temp_B[g(Range ∩ f^{-1}(local_A(p)) ∩ h^{-1}(local_B(p + 1)))]
      := recv(p + 1);
        ⋮      (main loop)
end;
```

Figure 12: Non-shared memory message passing for Figure 9

the subscript functions g and h to the range to be copied; this eliminates multiple indices which map to the same array element.

To generate the message passing needed for copying on non-shared memory architectures, a finer-grained analysis is needed. Not only must non-local references be identified, but the processor storing the needed values must also be known. Thus, the analysis must find, for each pair of processors p and q, the set $ref(p) \cap loc(q)$. This is the set of loop iterations performed by processor p which reference an array element on processor q. If $p \neq q$ and $ref(p) \cap loc(q) \neq \emptyset$ then any element of $g(ref(p) \cap loc(q))$ must be passed from q to p as a message. Note that this requires code to be generated for both processors p (for receiving) and q (for sending). In practice, a processor will often need to exchange messages with only a few other processors. An important exception to this rule occurs when a single datum is needed by many or all processors, as the pivot row is in Gaussian elimination. When this occurs, a broadcast operation is of great benefit. Figure 12 shows the message passing code generated for the loop of Figure 9, under the assumption that each processor need only receive data from its neighbor to the right. Again, we omit the main loop body from the figure.

In either the shared memory or non-shared memory case, once the copying is done, the body of the loop must be rewritten to test the index before reference $A[g(i)]$ and satisfy the reference from either the original array or

```
coprocess p ∈ Procs do
        ⋮      (copying code)
    for i ∈ local_A(p) loop
        Temp_A[i] := A[i];
    end;
    for i ∈ local_B(p) loop
        Temp_B[i] := B[i];
    end;
    for i ∈ Range ∩ f^-1(local_A(p)) loop
      A[f(i)] := ...A[f(i)]...
                 ...Temp_A[g(i)]...
                 ...Temp_B[h(i)]...
    end;
end;
```

Figure 13: Copying arrays to temporary storage

the temporary, as appropriate. This test can be eliminated in either of two ways. Either the local section of the array can be copied to the temporary as well as the non-local elements, or the loop can be split into two loops, one always using the temporary and one always using the original array. Copying the original array is always valid, but adds a large overhead. On the other hand, data dependences may prevent loop splitting. Figures 13 and 14 show the copying and loop-splitting versions of the final program. The copying code has been omitted from these versions; the code of either Figure 11 or 12 could be used.

4 Conclusion

Our goal in this paper has been to develop a formalism for describing the distribution of data on parallel machines. The idea of a *local* function giving, for each processor p, the set of elements stored on p, captures the data distribution concisely. This approach is more general than the alternative formalism of relating each element to the processor. We have also demonstrated that the *local* function can be used in at least one application,

```
coprocess p ∈ Procs do
        ⋮      (copying code)
    Let IndexSet = Range ∩ f^{-1}(local_A(p))
        loc_A = g^{-1}(local_A(p))
        loc_B = h^{-1}(local_B(p))
    for i ∈ IndexSet ∩ loc_A ∩ loc_B loop
      A[f(i)] := ... A[f(i)] ...
                 ... A[g(i)] ...
                 ... B[h(i)] ...
    end;
    for i ∈ IndexSet ∩ loc_A − loc_B loop
      A[f(i)] := ... A[f(i)] ...
                 ... A[g(i)] ...
                 ... Temp_B[h(i)] ...
    end;
    for i ∈ IndexSet ∩ loc_B − loc_A loop
      A[f(i)] := ... A[f(i)] ...
                 ... Temp_A[g(i)] ...
                 ... B[h(i)] ...
    end;
    for i ∈ IndexSet − loc_A − loc_B loop
      A[f(i)] := ... A[f(i)] ...
                 ... Temp_A[g(i)] ...
                 ... Temp_B[h(i)] ...
    end;
end;
```

Figure 14: Loop splitting

namely strip mining sequential loops. Other applications are also possible, and will be examined in future papers.

The discussion of strip mining in this paper has been somewhat abstract, and the reader may question its practical value. Closed-form formulas describing the sets $ref(p)$, $ker(p)$, and $loc(p)$ may be obtained, however, for many common distribution patterns and subscript functions. These formulas are simple enough to be used in a compiler to automate the transformation. More details of this may be found in [5].

Our plans for future research on this topic include

- Using the *local* function in other parallelism-extracting transformations on sequential programs
- Developing heuristics for choosing a distribution pattern for an array guided by properties of possible *local* functions
- Extending the *local* function to cases where data distribution is not static, such as cache memory schemes

References

[1] *Multimax Multiprocessor System.* Encore Computer Corporation, Marlboro, MA.

[2] J. R. Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations.* PhD thesis, Rice University, Houston, TX, April 1983.

[3] E. Davidson, D. Kuck, D. Lawrie, and A. Sameh. *Supercomputing Tradeoffs and the Cedar System.* CSRD Report 577, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, May 1986.

[4] K. Gallivan, W. Jalby, and D. Gannon. On the problem of optimizing data transfers for complex memory systems. In *Conference Proceedings of theInternational Conference on Supercomputing*, pages 238–253, ACM Press, July 1988.

[5] C. Koelbel and P. Mehrotra. *Semi-automatic Process Decomposition for Non-shared Memory Machines.* Technical Report CSD-TR 802, Purdue University, West Lafayette, IN, August 16 1988.

[6] P. Mehrotra and J. V. Rosendale. The BLAZE language: a parallel language for scientific programming. *Parallel Computing*, 5:339–361, 1987.

[7] Douglas Pase and Allan Larrabee. *Programming Parallel Processors*, chapter Intel iPSC Concurrent Computer, pages 105–124. Addison-Wesley Publishing Company, 1988.

[8] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): introduction and architecture. In D. Degroot, editor, *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, Computer Society Press, August 1985.

[9] D. Reed, L. Adams, and M. Patrick. Stencils and problem partitioning: their influence on performance of multiprocessor systems. *IEEE Transactions on Computers*, C-36(7):845–858, July 1987.

[10] C. L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–32, January 1985.

[11] S. Thakkar, P. Gifford, and G. Fielland. Balance: a shared memory multiprocessor. In *Proceedings of the Second International Conference on Supercomputing*, Santa Clara, CA, May 1987.

[12] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois, Urbana, IL, October 1982.