View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Purdue E-Pubs

Purdue University
Purdue e-Pubs

Department of Computer Science Technical Reports
Department of Computer Science

1988

Semi-automatic Process Decomposition for Non-shared Memory Machines

Charles Koelbel

Piyush Mehrotra

Report Number:
88-802

Koelbel, Charles and Mehrotra, Piyush, "Semi-automatic Process Decomposition for Non-shared Memory Machines" (1988). *Department of Computer Science Technical Reports.* Paper 684.
https://docs.lib.purdue.edu/cstech/684

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

SEMI-AUTOMATIC PROCESS DECOMPOSITION FOR NON-SHARED MEMORY MACHINES

Charles Koelbel
Piyush Mehrotra

CSD-TR-802
August 1988

Semi-automatic Process Decomposition for Non-shared Memory Machines

Charles Koelbel   Piyush Mehrotra
Department of Computer Sciences
Purdue University

Abstract

To achieve high performance on non-shared memory machines one must carefully distribute the data and the work so as to keep the workload balanced while minimizing the access to non-local data. Process decomposition is the operation of writing an algorithm as a collection of tasks, each operating primarily on its own portion of the data, to carry out the computation in parallel. In this paper we consider a semi-automatic approach to process decomposition in which the compiler, guided by advice from the user, automatically transforms sequential programs into such a set of interacting tasks. This approach is illustrated with a Gaussian elimination example which is transformed into a task system maximizing locality of memory reference.

1 Introduction

Parallel computer architectures have the potential to supply the computing power needed for large scientific computations. Unfortunately, methods of programming these machines are still in their infancy. This is especially true for non-shared memory machines, where user programs must manage processes and communication between processes.

Currently there are two major approaches to programming parallel machines. The first requires the programmer to explicitly manage all of the

parallel activities in the program, usually by using a language with constructs for parallelism. Such languages include Ada [1], Force [7], and Pisces [14]. Unfortunately, explicitly parallel programs seem to be much harder to write than sequential ones. The alternate approach is to allow the programmer to write sequential code and transfer the burden of extracting parallelism onto the compiler. Work in this direction includes new languages such as ParAlfl [6], SISAL [11], and BLAZE [12] and restructuring compiler efforts [2,4,13] which target conventional languages such as FORTRAN. The implicit parallelism approach has only been applied to shared memory architectures, however, making it unclear whether it is appropriate for non-shared memory machines.

An important consideration when implementing a program using either programming paradigm is the distribution of data across the processor memories of the system. Because processors may only reference data stored locally, it is vital that each processor do as much computation as possible on the data which it stores. To do this, the program is decomposed into a set of parallel tasks, each of which operates mainly on its local data. If non-local data is needed, it is received as a message and a copy stored locally. A good choice of data distribution patterns allows the workload to be evenly balanced among processors while minimizing communication between processors. If either of these goals is not met, performance will suffer. An unbalanced workload implies that part of the machine is idle, and thus parallelism is reduced. On the other hand, excessive communication adds a great deal of overhead to the system. Balancing these goals is often a difficult task, but vitally important in non-shared memory systems.

In this paper we take the first step toward automatically rewriting sequential code for execution on non-shared memory parallel architectures. We present a method of transforming a sequential loop into a task system with message passing between the tasks, using data partitioning concepts. Previously this was done by hand; our method transforms the program automatically, given a very small amount of extra input. Our method is an extension of the region of locality method introduced in [8,9].

2 The Region of Locality Method

The region of locality method of transforming a sequential loop has five steps:

1. Data distribution decides how the program's data will be divided among the processors. As we discussed earlier, this choice determines the efficiency of the resulting program.

2. Strip mining converts the sequential loop into a set of parallel processes. This gives the rough form of the final program, but non-local references make this version unsuitable for direct execution on non-shared memory machines.

3. Subscript analysis identifies the array references which may access non-local data. No program transformation is done at this stage, but the information is used in the next two steps.

4. Message generation creates the message-passing statements needed to correctly implement the non-local references.

5. Loop decomposition breaks the strip mined loops into subloops with the same pattern of non-local references in order to optimize those references.

The next five sections of this paper present the region of locality method. Each section explains one step of the process, illustrating it on a local example. The step is then applied to the program of Figure 1, the factorization phase of Gaussian elimination without pivoting written in the BLAZE language. By the end of the discussion, we will have converted that program into a explicitly parallel program with primitives for non-shared memory execution.

Throughout this paper, all example programs will be written in BLAZE, a high-level language for scientific computation [12]. Although the language has no parallel tasking or communication constructs, it has been designed to allow compiler transformations to extract implicit parallelism. Transformation of a BLAZE program introduces explicit parallelism into the program. We use E-BLAZE, an extension of BLAZE, to express such explicitly parallel constructs. Thus, E-BLAZE provides a virtual architecture

```
var a : array[ 1..n, 1..n ] of real; --$dist [*,cyclic]

for k in 1..n-1 loop

   for i in k+1 .. n loop
        for j in k+1 .. n loop
           a[i,j] += - a[k,j] * a[i,k] / a[k,k];
        end;
   end;

end;
```

Figure 1: Gaussian elimination program for transformation

for the BLAZE transformation system. E-BLAZE uses the SPMD (Single Program Multiple Data) model of parallel computation, in which each processor executes the same program but may do so asynchronously from the other processors. The sequential features of BLAZE and E-BLAZE are fairly standard; we will introduce parallel features as they are needed throughout the paper. We emphasize that, although we use BLAZE as a platform for these transformations, our work is language-independent.

3 Data Distribution

The first step in our method is to choose the distribution patterns to be used for the arrays in the program. Subsection 3.1 below formally defines these distribution patterns. Using those definitions, Subsection 3.2 tells how the distribution patterns for a particular program may be chosen.

3.1 Defining a Distribution Pattern

Mathematically, the distribution pattern of an array can be defined as a function from processors to sets of array elements. If P is the set of processors and A the set of array elements, then we define

$$local : P \rightarrow 2^A$$

```
processors Procs : array [ 1..P ] with P in 1..max_procs;
var     A : array[ 1..N ] of real by [ block ] on Procs;
        B : array[ 1..N, 1..M ] of real by [ cyclic, * ] on Procs;
```

Figure 2: Distribution patterns in E-BLAZE

as the function giving, for each processor p, the set of elements of A which p stores locally. In this paper we will assume that the sets of local elements are disjoint; that is, if $p \neq q$ then $local(p) \cap local(q) = \phi$. This reflects the practice of storing only one copy of each array element. We also make the convention that collections of processors and array elements are represented by their index sets, which we take to be vectors of integers.

E-BLAZE provides notations for the most common distribution patterns. First, the available processors must be declared via a **processors** declaration as in Figure 2. This allocates P processors, where $1 \leq P \leq max_procs$. Given this declaration, the E-BLAZE run-time environment dynamically chooses the largest feasible value for P. Once the processor array is declared, data arrays can be distributed across it using **by** clauses in the array declarations, also shown in Figure 2. Array A is distributed by blocks, giving it a *local* function of

$$local(p) = \left\{ i \mid (p-1) \cdot \left\lceil \frac{N}{P} \right\rceil + 1 \leq i \leq p \cdot \left\lceil \frac{N}{P} \right\rceil \right\}$$

This assigns a contiguous block of array elements to each processor. Array B has its rows cyclically distributed; its *local* is

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

Here, if P were 10 processor 1 would store elements in rows 1, 11, 21, and so on, while processor 10 would store rows which were multiples of 10.

3.2 Choosing a Distribution Pattern

The goal of choosing data distribution patterns is to minimize references to non-local array elements. Unfortunately, it seems unlikely that a compiler could make an optimal decision based solely on analysis of the program.

```
processors Procs : array[ 1..NP ] with NP in 1..n;
var a : array[ 1..n, 1..n ] of real by [ *, cyclic ] on Procs;
```

Figure 3: E-BLAZE declarations for Gaussian elimination program

Mace [10] has shown that at least one form of the data distribution problem is NP-complete. Approximate methods must therefore be used to find distribution patterns that, while not necessarily optimal, produce reasonable results. Finding such heuristics is an important open research problem.

In our current system, we require the programmer to provide annotations describing the patterns desired, which appear as comments in the original program. In our system, the E-BLAZE **processors** declaration is generated based on the description of the target architecture and the distribution pattern. The BLAZE program annotation in Figure 1 is translated directly into an E-BLAZE array distribution pattern. In the Gaussian elimination example, the columns of a will be distributed cyclically across the processors. The corresponding *local* function is similar to that for B in Figure 2. The E-BLAZE declarations generated for the Gaussian elimination program are shown in Figure 3.

4 Strip Mining

Once the distribution patterns for the arrays in the program have been chosen, the loops in the program can be converted to parallel form. The transformation to do this is called *strip mining*, after a similar transformation in vectorizing compilers [15]. The fundamental idea of the transformation is to break the range of a **for** loop into subranges ("strips" in the original formulation). If the subranges are independent of each other, they may be executed in parallel. The strip mining transformation simply creates a parallel loop which runs over these subranges. The transformation is not valid if the subranges are not independent, since then they cannot safely be run in parallel.

The type of independence between subranges needed for strip mining is captured by data dependence. As shown in [3,15] data dependence analysis in the compiler can be used to check for independence of subranges. checked

by the compiler using data dependence analysis. We give only an overview of this 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 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. Dependences in which the references are made during the same iteration are called loop-independent; if the references are made during different iterations, the dependence is loop-carried. A loop can be executed in parallel without introducing synchronization instructions if and only if it does not carry any dependence. In the case of strip mining, this means that the loop may be executed in parallel only if there are no dependences from statements in one subrange to statements in another.

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 $(i_2 - i_1, j_2 - j_1)$. We will use this concept in formulating an exact validity condition for strip mining.

Strip mining is accomplished in a BLAZE program by using the E-BLAZE **coprocess** construct. This is an explicitly parallel loop which creates a process on each processor declared in a program. Each process executes one iteration of the **coprocess** and finishes by performing a barrier synchronization with all other processes. To strip mine a loop, a **coprocess** is created around the original **for** loop, and the loop bounds of the original **for** loop are modified to run over only a subrange. The new bounds are parameterized with respect to the **coprocess** index so that each **coprocess** iteration will execute one of the chosen subranges.

The subranges of the strip mined **for** loop must be chosen so that as many array accesses as possible are made to local data. In our work, we

```
                    for i ∈ range loop
                        A[ f(i) ] := ...
                    end;
```

(a)

```
coprocess p ∈ Procs do
    for i ∈ range ∩ f⁻¹(local(p)) loop
        A[ f(i) ] := ...
    end;
end;
```

(b)

Figure 4: Simple loop before and after strip mining

choose these subranges so that any storing of values can be done locally. If the loop contains only a single assignment to an array, as does the pseudocode loop of Figure 4a, the strip mining transformation produces the nested construct of Figure 4b. Given that the loop accesses $A[f(i)]$, restricting processor p to iterate over a subset of the set $f^{-1}(local(p))$ ensures that the set of array elements assigned to will be $f(f^{-1}(local(p))) = local(p)$. For notational convenience, we refer to the set $f^{-1}(local(p))$ as $ref(p)$.

In order for the transformation of Figure 4 to be valid, we must ensure that the subranges formed by $ref(p)$ are independent. 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, as in the E-BLAZE declaration

var A : **array**[1..N, 1..N] **of real by** [**block**, *] **on** Proc;

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

We can now state the validity conditions 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, $f(d) \in ker(p)$.

Under this condition, all dependences have both their beginning and ending on the same processor; thus, the original order of the execution is always preserved. 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.

The loop of Figure 4 can easily be generalized to the case when the loop has several assignments to arrays with the same distribution patterns and subscript expressions. The new loop bounds can then be calculated based on any of the assignment statements, since these will clearly give the same set. This is also true if the difference between any two subscripts is an element of $ker(p)$. For example, if the only two assignments in a loop are to $A[f_1(i)]$ and $A[f_2(i)]$ and $f_1(i) - f_2(i) \in ker(p)$ for all i and p, then the loop can be strip mined using either f_1 or f_2. Furthermore, these cases can easily be extended from the single **for** loop shown here to nested loops. More complex cases, in which arrays with different distribution patterns or subscript expressions are updated, are being studied.

Applying this transformation to the Gaussian elimination program of Figure 1 produces the program of Figure 5. The **for** k loop cannot be strip mined, because it carries data dependences between columns of the matrix. For example, the references to $a[i, j]$ and $a[i, k]$ produce a dependence with distance vector $(k, i, j) = (1, 0, -1)$. In this case, $f(k, i, j) = (i, j)$, so $f(1, 0, -1) = (0, -1) \notin ker(p)$. The **for** i and **for** j loops may be strip mined as shown, however. While transforming the inner loop, dependences carried by the outer loop may be ignored, as discussed in [3]. Two notational conveniences in the program should be explained. The E-BLAZE **range** primitive gives the range of indices of an array stored on a given processor. The intersection operator ^ takes the intersection of two ranges. The bounds of the resulting range can be easily computed, but are notationally complex. We therefore show only the intersected version.

```
processors Procs : array[ 1..NP ] with NP in 1..n;
var a : array[ 1..n, 1..n ] of real by [ *, cyclic ] on Procs;

for k in 1..n-1 loop

   coprocess p in 1..NP on Procs[p] do

         for i in k+1 .. n loop
            for j in k+1..n ^ range(a[,*],Procs[p]) loop
                  a[i,j] := a[i,j] - a[k,j] * a[i,k] / a[k,k];
            end;
         end;

   end;

end;
```

Figure 5: Strip-mined Gaussian elimination program

```
coprocess p in 1..NP on Procs[p] do
    for (i1, i2) ∈ range_orig ∩ f^-1(local(p)) loop
        ... R1 ...
        ... R2 ...
            ⋮
        ... Rn ...
    end;
end;
```

Figure 6: Pseudocode loop for subscript analysis

5 Subscript Analysis

Once the parallelism of the program has been expressed as explicit **coprocess** constructs, the question of memory locality can be addressed. This entails an analysis of the subscripts of array references to determine which ones may cause access to non-local elements. We will describe such an analysis in this section. The information gained will be used in the next two sections to transform the program to bring the non-local references together and make local copies of the non-local data.

The type of loop we are considering has the form shown in Figure 6. Each R_k represents an array reference, which may occur either on the right or left of an assignment. For simplicity, we will assume that only one array A is referenced. The general case of multiple arrays does not alter the goals of the analysis, although it may complicate the analysis itself if the arrays have different distribution patterns. We restrict the subscripts in the references to be linear functions of the loop indices. Thus, a reference to a two-dimensional array A will have the form

$$R = A[g(i_1, i_2)]$$

where g is defined as

$$g(i_1, i_2) = (g_1(i_1, i_2), g_2(i_1, i_2))$$

and the g_j functions have the form

$$g_j(i_1, i_2) = c_{j,0} + c_{j,1}i_1 + c_{j,2}i_2$$

The $c_{j,0}$ terms may contain loop invariants, but the coefficients of the indices must be compile-time constants. These restrictions are the same as those usually imposed by data dependence analysis [15], and cover the most common cases found in real programs.

For each processor p and reference R the set $g^{-1}(local(p))$ defines a subset of the iteration space such that R is always a local reference. Because g is linear, these sets may be calculated easily. If $g^{-1}(local(p)) \subseteq ref(p)$ for all p then the reference R can always be satisfied locally. Otherwise, any element a such that $a \in g^{-1}(local(p))$ but $a \notin ref(p)$ must be communicated to processor p via messages. The goal of subscript analysis is to find the appropriate sets $g^{-1}(local(p))$ for each p and determine how they intersect with the loop range sets $ref(p)$.

This process can be visualized easily for block distributions in two dimensions, as shown in Figure 7. The $local(p)$ sets are rectangles in the iteration space. Linear transformations of these such as $f^{-1}(local(p))$ become parallelograms in the same space. In the common special case of $c_{i,j} = 1$ for $j \neq 0$ the parallelograms are rectangles. These are represented by the large solid and dashed rectangles in the figure. Intersections between the sets are represented as overlapping regions. Thus, $B[i-1, j+3]$ is a non-local reference in areas 1, 2, and 3. Our subscript analysis would identify these regions using simple formulas.

Applying a similar analysis to the Gaussian elimination example produces the results in Figure 8. The analysis itself is harder to visualize, since the $local(p)$ sets no longer form contiguous regions in the iteration space. The formulas, however, can still be put in closed form to give the desired results. Note that only the references $a[i, k]$ and $a[k, k]$ can cause non-local references.

6 Message Generation

Using the information from subscript analysis, it is possible to determine which array elements must be moved into local memory. In non-shared memory machines, this means generating message-passing statements to send and receive non-local data. The basic idea is, for each array reference that may be non-local, to generate a pair of **send** and **recv** statements which communicates the appropriate array elements.

```
var A, B : array[ 1..N, 1..N ] of real by [ block, block ] on Procs;
coprocess p1 in 1..NP, p2 in 1..NP on Procs[p] do
    for i in range(A[*,],Procs[p1,p2]) loop
        for j in range(A[,*],Procs[p1,p2]) loop
            A[ i, j ] := A[ i, j ] - B[ i-1, j+3 ];
        end;
    end;
end;
```

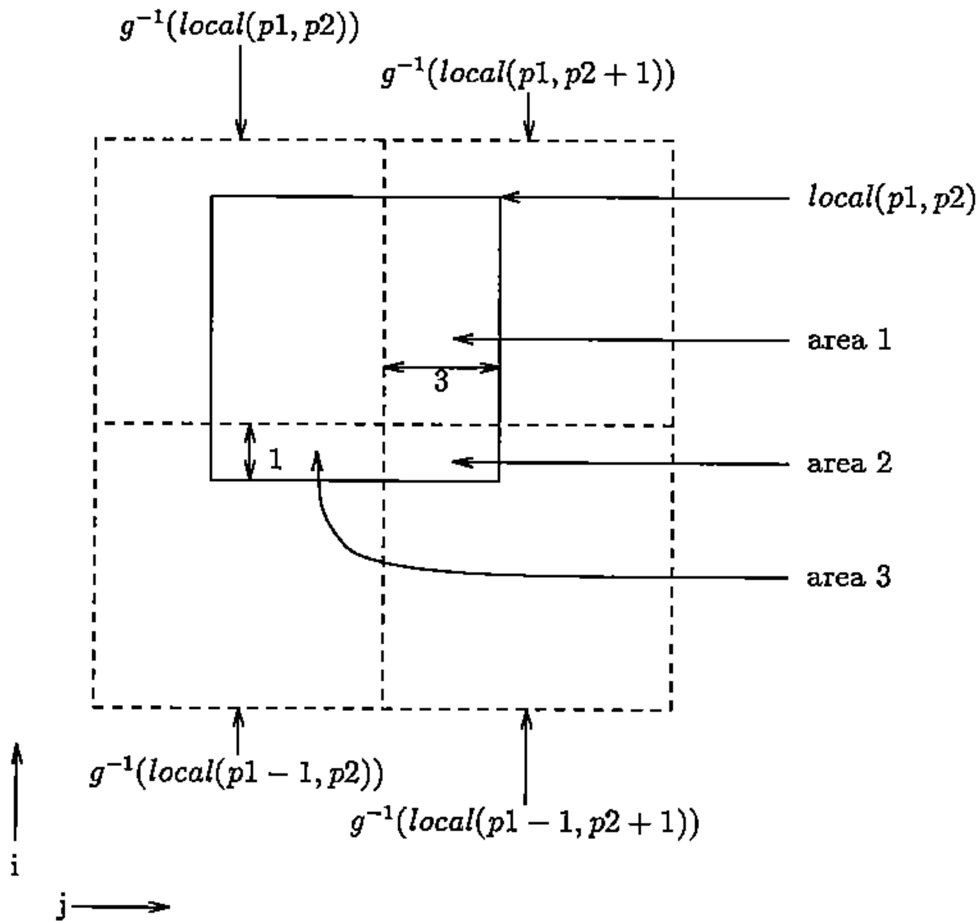

Figure 7: Visualizing subscript analysis

Reference	$g^{-1}(local(p))$	When Local
a[i,j]	$\{(i,j) \mid j \equiv p \pmod{NP}\}$	Always
a[k,j]	$\{(i,j) \mid j \equiv p \pmod{NP}\}$	Always
a[i,k]	$\{(i,j) \mid k \equiv p \pmod{NP}\}$	When $k \equiv p \pmod{NP}$
a[k,k]	$\{(i,j) \mid k \equiv p \pmod{NP}\}$	When $k \equiv p \pmod{NP}$

Figure 8: Subscript analysis summary for Gaussian elimination example

The major factor in generating the message passing instructions is to decide whether the reference may access a non-local element of the array. This will occur whenever $ref(p) \not\subseteq g^{-1}(local(p))$. This matching works in two ways. An iteration in $ref(p) - g^{-1}(local(p))$ will make a non-local reference on the current processor when it accesses $A[g(i)]$; this reference must be received as a message. In this case there must be a processor q which "owns" the element. Therefore, an equivalent condition is that any nonempty set $g(ref(p) \cap g^{-1}(local(q)))$ represents elements which must be sent from q to p. Similarly, an iteration in $g^{-1}(local(p)) - ref(p)$ produces a reference on another processor which is local to the current processor; this reference must be sent out. The alternate formulation is that if $g(ref(q) \cap g^{-1}(local(p)))$ is nonempty then that set must be sent from p to q. In the alternate formulations of these equations, there may be more than one q for a given p. When this is true, there must be one message generated for each q.

Once the non-local references have been identified, the message passing statements can be generated easily. In the Gaussian elimination example, there are two messages generated. Both are broadcasts of elements in the pivot row. The elements broadcast and the conditions for sending can be taken directly from the descriptions of the sets $g^{-1}(local(p))$ given in Figure 8 and the bounds of the **for** j loop. The resulting **send** and **recv** statements are shown in Figure 9.

7 Loop Decomposition

After data is moved into local memory, the strip mined loop must be decomposed to isolate non-local references. Since these references are satisfied from the original array and from temporary arrays at various times,

```
var tmp1 : array[ 1..n ] of real;
   tmp2 : real;

if ( k mod NP = p mod NP ) then
   send( Procs[1..NP], a[k+1..n,k] );          -- from reference a[i,k]
   send( Procs[1..NP], a[k,k] );               -- from reference a[k,k]
end;
tmp1[k+1..n] := recv( Procs[ (k−1) mod NP + 1 ] );
tmp2 := recv( Procs[ (k−1) mod NP + 1 ] );
```

Figure 9: Message passing in Gaussian elimination example

the same reference in the original program is mapped into references to two variables. If the loop is not split, each reference to the array must be preceded by a test to determine which variable to use. Decomposing the loop removes these tests, increasing run-time efficiency.

Figure 10 shows the results of loop decomposition on the Gaussian elimination example. The index range of the strip mined loop is divided into subranges such that an array reference is local for one subrange value if and only if it is local for all subrange values. The subranges used are exactly the sets used to generate the **recv** statements in the last section. New loops are then created running over each of these subranges. The bodies of the new loops are identical to the bodies of the old loops except for the non-local references, which are replaced by references to the appropriate arrays. For the Gaussian elimination example, no subranges are necessary, but this is a rare occurrence. The simple program in Figure 7, for example, decomposes into four loops corresponding to the four partitions of the $local(p1, p2)$ region.

8 Conclusions

In this paper we have shown how some loops written in a sequential language can be converted to run in parallel on a non-shared memory architecture. Our method is conceptually very simple, but we believe it is quite

```
processors Procs : array[ 1..NP ] with NP in 1..n;
var a : array[ 1..n, 1..n ] of real by [ *, cyclic ] on Procs;

for k in 1..n-1 loop

   coprocess p in 1..NP on Procs[p] do

        var tmp1 : array[ 1..n ] of real;
           tmp2 : real;
        if ( k mod NP = p mod NP ) then
           send( Procs[1..NP], a[k+1..n,k] );          -- from reference a[i,k]
           send( Procs[1..NP], a[k,k] );               -- from reference a[k,k]
        end;
        tmp1[k+1..n] := recv( Procs[ (k-1) mod NP + 1 ] );
        tmp2 := recv( Procs[ (k-1) mod NP + 1 ] );

        for i in k+1 .. n loop
           for j in k+1..n ^ range(a[,*],Procs[p]) loop
                 a[i,j] := a[i,j] - a[k,j] * tmp1[i] / tmp2;
           end;
        end;

   end;

end;
```

Figure 10: Final form of Gaussian elimination example

general and easily implementable. As evidence for its generality, we have used it to hand-translate several important algorithms, such as the Gaussian elimination example in this paper and picture smoothing algorithms. We are currently working on implementing our method in the BLAZE compiler being developed at Purdue; we will report on the results of this work at the conference.

Our work shares much with other work on automatic parallelization of sequential code. Two of the more prominent groups working in this area are the Cedar group headed by David Kuck at the University of Illinois [13] and Ken Kennedy at Rice University [4]. Our contribution to this has been to extend that work to include distributing the data. We have further extended the work by considering non-shared memory systems, where the techniques of automatic parallelization had not been brought to bear.

It is admittedly a shortcoming of this work that users must supply this information. We justify using program annotations on two grounds. First, the extra input required is very small. This seems to be an acceptable burden to place on the user, especially in light of some of the other annotations required by advanced compilers. Kennedy [5] has suggested that compilers in the future will become much more interactive to allow just this type of user input. The other justification is that our system is a useful back end to any new methods of choosing these patterns. Many groups are working on heuristics for pattern selection, and our method will be useful for implementing any of their ideas.

References

[1] *Reference Manual for the Ada Programming Language.* American National Standards Institute, Inc., ANSI/MIL-STD-1815A-1983 edition, February 1983.

[2] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. *An Overview of the PTRAN Analysis System for Multiprocessing.* Research Report RC 13115 (#56866), IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1987.

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

[4] J. R. Allen and K. Kennedy. *Automatic Translation of Fortran Programs to Vector Form.* Technical Report 476-029-4, Rice University, Houston, TX, October 1980.

[5] R. Allen, D. Callahan, and K. Kennedy. *Automatic Decomposition of Scientific Programs for Parallel Execution.* Computer Science Technical Report TR86-42, Rice University, Houston, TX, November 1986.

[6] P. Hudak. Parafunctional programming. *IEEE Computer*, 19:60–71, 1986.

[7] H. Jordan. Structuring parallel algorithms in an MIMD, shared memory environment. *Parallel Computing*, 3(2):93–110, May 1986.

[8] C. Koelbel, P. Mehrotra, and J. V. Rosendale. Semi-automatic domain decomposition in BLAZE. In Sartaj K. Sahni, editor, *Proceedings of the 1987 International Conference on Parallel Processing*, pages 521–524, Pennsylvania State University Press, August 1987.

[9] C. Koelbel, P. Mehrotra, and J. V. Rosendale. Semi-automatic process partitioning for parallel computation. *International Journal of Parallel Processing*, to appear.

[10] M. Mace. *Globally Optimal Selection of Memory Storage Patterns.* PhD thesis, Duke University, Durham, NC, May 1983.

[11] J. McGraw, S. Skedzielewski, S. Allan, R. Oldenhoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. *SISAL: Streams and Iteration in a Single Assignment Language: Language Reference Manual.* Report M-146, Lawrence Livermore National Laboratory, March 1985.

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

[13] D. A. Padua, D. J. Kuck, and D. H. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Transactions on Computers*, C-29(9):763–776, September 1980.

[14] T. W. Pratt. Pisces: an environment for paralle scientific computation. *IEEE Software*, 2:7–20, 1985.

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