NASA Contractor Report 181633

ICASE REPORT NO. 88-16

ICASE

SEMI-AUTOMATIC PROCESS PARTITIONING FOR PARALLEL COMPUTATION

Charles Koelbel

Piyush Mehrotra

John Van Rosendale

(NASA-CR-181633) SEMI-AUTOMATIC PROCESS PARTITIONING FOR PARALLEL COMPUTATION Final Report (ICASE) 23 p CSCL 09B

N89-25565

Unclas

G3/61 0209831

Contract No. NAS1-18107

February 1988

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

NASA

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

SEMI-AUTOMATIC PROCESS PARTITIONING FOR PARALLEL COMPUTATION

Charles Koelbel[1], *Piyush Mehrotra*[1,2]

Department of Computer Science,
Purdue University,
West Lafayette, In. 47907.
and

John Van Rosendale[2]

Department of Computer Science,
University of Utah,
Salt Lake City, Ut. 84112.

Abstract

On current multiprocessor architectures one must carefully distribute data in memory in order to achieve high performance. Process partitioning is the operation of rewriting a 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 partitioning in which the compiler, guided by advice from the user, automatically transforms programs into such an interacting task system. This approach is illustrated with a picture processing example written in BLAZE, which is transformed into a task system maximizing locality of memory reference.

[1] Research was supported under NASA Contract No. 520-1398-0356.

[2] Research was supported under the National Aeronautics and Space Administration under NASA Contract No. NAS1-18107 while the second and third authors were in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23665.

1. Introduction

The shared memory paradigm is one of the most useful models of parallel computation. A parallel machine having a single homogeneous memory, randomly addressable by all processors, is relatively easy to program, and makes an effective target architecture for compilers restructuring sequential programs.[6, 1] Unfortunately, this model does not accurately reflect most current machines. On most multiprocessor architectures, each processor has attached local memory. Access to this local memory is faster, and involves less contention than access to remote memory. This speed difference makes it important for the data used in a calculation to be in local memory, whenever possible. Treating an architecture having local memories as a pure shared memory architecture results in inferior performance.

Process partitioning is the process of assigning data to processors to exploit locality of reference. Until now it has been done manually by coding algorithms in a parallel language giving the user explicit control over the location of all computations and data. While this strategy is effective, it suffers from two major drawbacks. First it is time consuming and error prone for the programmer, who must supply a great deal of detail about data and task allocation having little to do with the underlying algorithm being implemented. Second manual process partitioning generally produces non-portable programs. While the data may be divided in precisely the same manner on both a shared memory and a non-shared memory machine, the syntax for non-local accesses will almost certainly be different.

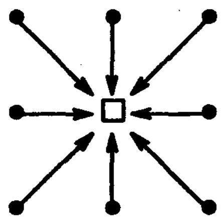
Our purpose in this paper is to show that, with a small amount of help from the user, the compiler can automatically perform process partitioning. The only extra information that the programmer must provide is a general statement of the data distribution pattern. The remaining aspects of process partitioning are essentially just mechanical details, which can easily be left to the compiler. Eventually even specifying data distribution patterns can be left to the compiler, either via an expert system front end, or through more thorough analysis than is currently available.

While this approach does not fully automate process partitioning, it makes it relatively painless to programmers. They are then free to focus on the higher-level aspects of program and algorithm design. In this paper, we consider a simple example, showing the sequence of transformations a BLAZE compiler would follow in generating efficient code for a shared memory multiprocessor.

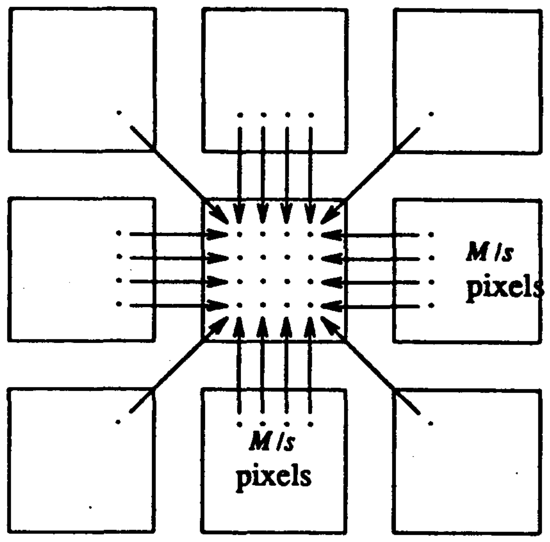
Locality of memory reference is important on both shared memory and non-shared memory architectures, but is more critical on non-shared memory architectures, such as hypercubes. However, even in the simpler shared memory case, effectively treating memory locality issues is still relatively complex. It is our eventual goal to treat general non-shared memory architectures; however there are a number of complex issues that have not been resolved yet. These issues will be treated in a future paper.

2. Process Partitioning

As an example We consider here a standard process partitioning example, a smoothing algorithm used in picture processing and other numerical applications. Extracting parallelism from this type of algorithm is a well studied problem.[5, 4] In this algorithm, a picture, represented as two dimensional array of pixel values, is modified to produce a new "improved" array. This process is repeated a number of times. In one approach, the pixel values in each new array are formed by taking a weighted average of four neighboring pixel values as shown in the picture below.



Suppose the pixel matrix is $M \times M$ and there are N^2 processors available. The data in this case should be grouped into $M/N \times M/N$ blocks so that each processor contains a contiguous subarray of the original matrix. This allows most of the array references in the formula to be made locally. Only the edge pixels in each processor's subarray require non-local accesses as shown below.



Our transformations automatically changes a program expressing the smoothing algorithm whose data flow pattern is shown in the first figure to a task system like that illustrated in the second.

3. The BLAZE Environment

BLAZE is a new programming language intended for programming scientific applications on parallel machines.[2] It is intended to be a simple and elegant way to express scientific programs in a way that compilers can easily transform them to execute on a variety of parallel architectures. In this section, we describe a few of the features of BLAZE, and also describe E-BLAZE, the target language of the BLAZE transformation system.

3.1. BLAZE Features

While closely related to data flow languages, BLAZE uses relatively conventional syntax. The listing below represents the picture processing algorithm that we have been considering.

```
procedure smooth (w)  returns : w ;

param w : array[0..NP : integer, 0..NP] of real;
var ws : like w;
const N := NP – 1;
begin

 for k in 1..50 loop
  ws := w;

  forall (i, j) in [1..N] * [1..N] do
   w[i, j] := c*( w[i–1, j–1] + w[i–1, j] + w[i–1, j+1]
           + w[i,  j–1] + w[i,  j] + w[i,  j+1]
           + w[i+1, j–1] + w[i+1, j] + w[i+1, j+1] );
  end; – –forall loop

 end;

end;  – – procedure smooth
```

The **forall** loop here is similar to conventional **for** or **do** loops. The difference is that in a **forall** loop all invocations can run in parallel. The *ws* array is declared via the **like** type specification, which gives it the same type as the array *w*.

3.2. E-BLAZE Constructs

BLAZE is a high level language designed to permit program portability across a variety of parallel architectures. E-BLAZE is a lower-level explicit-tasking language providing a virtual target architecture for our program transformation system. BLAZE itself contains no explicit parallel constructs, except the **forall** loop. E-BLAZE is a superset of BLAZE, consisting of BLAZE together with constructs for allocating processes, specifying array storage patterns, scheduling loops, and performing interprocess communication and synchronization.

The following declaration:

```
processors procs : array[ 1..np, 1..np] with np in 1..P;
```

allocates an array of np^2 processors, where np is in the range $1..P$. The run-time environment dynamically chooses the largest feasible value for np.

To use the above processors for computation, a **coprocess** loop must be used. This is an explicitly parallel version of the **forall** loop, with the same copy-in-copy-out semantics. Each iteration of the loop is performed on a separate processor. For example, given the above declaration of procs, the E-BLAZE statement

```
coprocess (p, q) in [1..np] * [1..np] on procs[p, q] do
...
end;
```

would start np^2 processes in parallel, each of which would execute one **coprocess** iteration. The processes all execute the same code, and finish by performing a barrier synchronization.

Arrays are declared as stored across E-BLAZE processors in one of several ways. Each array declaration must have a **by** clause describing its distribution. The example below shows two possible **by** clauses.

```
processors procs : array[1..np] with np in 1..10;
var  A : array[1..200] of integer by [ block ] on procs;
     B : array[1..200, 1..200] of integer
            by [ block, *] on procs;
```

If *procs* were allocated ten processors, each processor in *procs* would be assigned a consecutive block of twenty elements of A. The two-dimensional array *B* is distributed by blocks of rows across *procs*. There are other distribution patterns in E-BLAZE, such as cyclic distribution, which are not discussed here.

4. Transformation of BLAZE to E-BLAZE

In this section we consider the transformation of the picture smoothing algorithm for multiprocessor execution. One of the critical decisions required in transforming such programs is the choice of data distribution pattern employed. In our current research, we assume that the programmer provides annotations describing the data distribution patterns desired. Thus, in the above picture processing example, the programmer might provide the annotation:

```
param w: array[0..NP, 0..NP] of real;  -- distribute by [block, block]
```

To simplify the presentation here, we look at a one dimensional version of the smoothing algorithm as shown below.

```
procedure smooth (w)  returns : w ;

param w : array[0..NP : integer] of real;  -- distribute by [block]
var ws : like w;
const N := NP – 1;
begin

 for k in 1..50 loop
  ws := w;

  forall i in [1..N] do
    w[i] := c*(w[i–1] + w[i] + w[i+1]);
  end; --forall loop

 end;

end;  --procedure smooth
```

All transformations described here apply equally to higher dimensional cases.

4.1. Transforming forall Loops

The first step in the transformation process is to restructure the logically parallel **forall** loops into the explicitly parallel **coprocess** loops. This requires the introduction of the processor declaration and the distribution of the arrays across the processors.

The original **forall** loop is then "strip-mined" so as to generate a **coprocess** with a nested **forall** performing the same computation. The bounds of the generated **coprocess** loop are based on the processor declaration while those of the nested **forall** need to be generated such that most of the array accesses refer to data local to that processor. In the smoothing example, there is only one statement within the **forall** loop, wherein only one array is referenced. In such a situation, the **forall** loop bounds are generated based on the array reference on the left hand side of the statement. More complex loops, in which more than one array is referenced or the reference pattern is more complex are being studied. Thus the result of the above transformations result in the following E-BLAZE program:

```
procedure smooth (w) returns : w ;

processors P : array[1..m] with m in [1..NP];
param w : array[0..NP : integer] of real by [ block ] on P;
var ws : like w;
const N := NP – 1;

begin

 for k in 1..50 loop

   coprocess q in 1..m on P[q] do
     ws[range(ws[*], P)] := w[range(w[*], P)];
   end; – – coprocess loop

   coprocess q in 1..m on P[q] do
     forall i in (1..N) ^ range(w[*], P) do
       w[i] := c*(ws[i–1] + ws[i] + ws[i+1]);
     end;
   end; – – coprocess loop
 end;

end; – – procedure smooth
```

The **range**(*w[*], P*) primitive here provides the bounds of the subvector of *w* stored on the processor of the array P on which this particular **coprocess** loop is executing. The intersection operator ^, provides the intersection of this range with the original **forall** range.

5. Subscript Analysis

Once the parallelism in the program has been expressed as explicit **coprocess** loops, the question of memory locality can be addressed. This requires an analysis of the local and non-local memory references occurring in each loop, followed by a sequence of program transformations based on this analysis. This section describes this analysis and the subsequent program transformation.

The type of loop we are looking at has the basic form

```
coprocess ... P[ ... ] do
 forall (i, j) in product_range do
     ...
     ... $R^1$ ...
     ... $R^2$ ...
     ...
     ... $R^n$ ...
     ...
 end;
end;
```

where each R^k is an array reference. Array references can occur on both the left and right sides of statements, and a number of arrays may be referenced in the loop. For simplicity, we assume that only one array A is referenced, and also assume a two dimensional **forall** loop, as shown. The general case is no more difficult, but complicates notation.

Our analysis is applicable to the commonly occurring case in which array subscripts are linear functions of the loop indices. We also require the assumption that each array subscript depends on at most one loop index. Thus all array references have the form:

$$R = A[\, f_1(i,j),\ f_2(i,j)\,]$$

where f_m has either the form:

$$f_m(i,j) = c_0 + c_1\, \mathrm{i}$$

or the form

$$f_m(i,j) = c_0 + c_1 j$$

Now assume that array A is distributed by blocks across the processors in the processor array P. For example, A might have the declaration

A:array[1..N,1..N] of real by [block,block] on P;

or

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

depending on whether P is one dimensional or two dimensional. Then there exist two intervals, I_1 and I_2, such that if (s,t) is in the rectangle

$$I_1 \times I_2$$

then $A[s,t]$ will be a local reference. If either s or t were outside the corresponding interval, $A[s,t]$ would not be a local reference.

Given our assumptions on the form of subscript functions occurring, for each array reference R^k, there is a rectangle G^k, such that if the loop indices (i,j) are within G^k, then the reference will be local. That is,

$$(i,j) \varepsilon G^k$$

implies

$$(f_1(i,j),\ f_2(i,j)) \ \varepsilon\ I_1 \times I_2$$

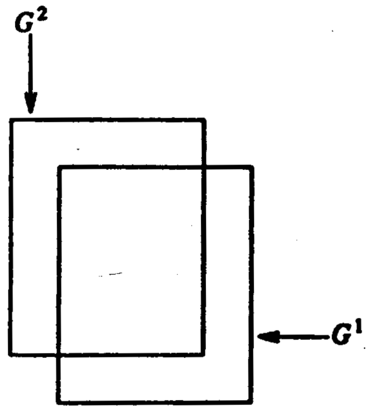
Note that in degenerate cases the "rectangles" may be empty, may be the whole plane, or may be an infinite slab. The slab case, for example, occurs with array references such as $A[j,j]$, in which one index is omitted.

5.1. Loop decomposition

Once the rectangles, G^k are computed for each reference R^k, they may be combined into a single diagram, called the "reference locality diagram." As a simple example, consider the loop:

```
forall (i, j) in [a..b] * [c..d] do
   A[i, j] := A[i, j] - A[i+3, j+3]
end;
```

has the the reference locality diagram

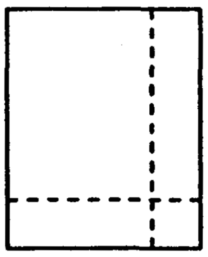


where the correspondence between the rectangles shown and the array references occurring is:

```
A[i, j] := A[i, j] -  A[i+3, j+3]
  G¹        G¹          G²
```

The rectangle G^1 corresponds both to the first reference here, and to the index range of the forall loop.

The next step in the transformation process is to extend the edges of the rectangles in the reference locality diagram, so that the index range is decomposed into a disjoint union of rectangles. We also discard the portion of each rectangle lying outside the index range. Applying this operation to the simple example here, yields the following diagram.



Each rectangle in this diagram corresponds to a loop over a subrange of the original loop. In this case there would be four subloops, as shown.

```
forall i, j in [a..b-3] * [c..d-3] do
 A[i, j] := A[i, j] - A[i+3, j+3]
end;

forall i, j in [a..b-3] * [d-2..d] do
 A[i, j] := A[i, j] - !A[i+3, j+3]
end;

forall i, j in [b-2..b] * [c..d-3] do
 A[i, j] := A[i, j] - !A[i+3, j+3]
end;

forall i, j in [b-2..b] * [d-2..d] do
 A[i, j] := A[i, j] - !A[i+3, j+3]
end;
```

From the rectangle inducing each loop, one can tell which array references in the assignment statement are local. Non-local references are flagged with an exclamation marks.

Finally, in many cases one can merge loops, reducing the amount of loop overhead. In this case, for example, the last two loops could be merged into a single loop over the index range:

[b-2..b] * [c..d]

5.2. Picture Processing Example

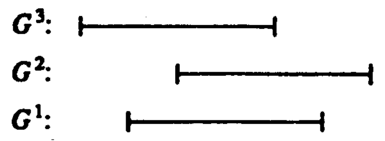
Let us now apply the above analysis on the one dimensional picture processing under consideration. The important **forall** loop is given below:

```
forall i in range(w[*], P) do
  w[i] := c*(ws[i-1] + ws[i] + ws[i+1]);
end;
```

The three references to the array *ws* give rise to the following (degenerate) rectangles:

```
w[i] := c*(ws[i-1] + ws[i] + ws[i+1]);
```

G^1 G^2 G^1 G^3



where G^1 corresponds to the index range of the **forall** loop.

The disjoint rectangles can be easily determined and give rise to the following three **forall** loops:

```
procedure smooth (w)  returns : w ;

processors P : array[1..m] with m in [1..NP];
param w : array[0..NP : integer] of real
                    by [ block ] on P;
var ws : like w;
const N := NP - 1;

begin

 for k in 1..50 loop

   coprocess q in 1..m on P[q] do
     ws[range(ws[*], P)] := w[range(w[*], P)];
   end;  -- coprocess loop

   coprocess q in 1..m on P[q] do
     var s, t : integer;

     s := lower(ws[*], P);
     t := upper(ws[*], P);

     if (s >= 1 and s <= N) then
       w[s] := c*(!ws[s-1] + ws[s] + ws[s+1]);
     end;

     forall i in [1..N] ^ [s+1..t-1] do
       w[i] := c*(ws[i-1] + ws[i] + ws[i+1]);
     end;

     if (t >= 1 and t <= N) then
       w[t] := c*(ws[t-1] + ws[t] + !ws[t+1]);
     end;

   end;  -- coprocess loop
 end;

end;  -- procedure smooth
```

At this point, it is fairly simple for a compiler to generate good code for the non-local references.

6. Conclusions

Careful distribution of data is essential for high performance on many parallel machines. This paper has demonstrated that automatic process partitioning by a series of source-to-source translations is feasible. Our method seems to be quite general, is easy to implement, and should substantially reduce users programming effort.

The work here falls in the framework of automatic parallelization of sequential code. Our transformations are related to the concept of "strip mining" one of the loop transformations studied by program restructuring. Much work in this area has been done by David Kuck and the Cedar group at the University of Illinois[3] and Ken Kennedy at Rice University.[1] We have extended that work to include distributing data and performing the computations on multiprocessors.

A major shortcoming of our work is that the general array distribution pattern must be specified by the programmer's annotations. Ideally, the compiler should also determine the storage pattern of all arrays. This is a complex optimization problem and is being studied.

Even though we have concentrated on shared memory machines, extensions of the ideas here will apply to non-shared memory architectures, such as the hypercubes. However, a number of aspects of these extensions are unresolved, so we will report on this research later.

7. References

1. Kennedy, K., "Automatic Translation of Fortran Programs to Vector Form," Rice Technical Report 476-029-4, Rice University (October 1980).
2. Mehrotra, P. and Van Rosendale, J., "The BLAZE Language: A Parallel Language for Scientific Programming," *Parallel Computing* **5** pp. 339-361 (1987).
3. Padua, D. A., Kuck, D. J., and Lawrie, D. H., "High-Speed Multiprocessors and Compilation Techniques," *IEEE Transactions on Computers* **C-29** (9) pp. 763-776 (September 1980). Special issue on Parallel Processing
4. Reed, D., Adams, L., and Patrick, M., "Stencils and Problem Partioning: Their Influence on Performance of Multiple Processor Systems," *IEEE Transactions on Computers (to appear)*, Also available as ICASE Report 86-24, (May 1986).
5. Siegel, H. J., Siegel, L. J., Kemmerer, F. C., Mueller, P. T. Jr., Smalley, H. E. Jr., and Smith, S. D., "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition," *IEEE Transactions on Computers* **C-30** (12) pp. 934-947 (December 1981).

Report Documentation Page

1. Report No. NASA CR-181633 ICASE Report No. 88-16	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle SEMI-AUTOMATIC PROCESS PARTITIONING FOR PARALLEL COMPUTATION		5. Report Date February 1988
		6. Performing Organization Code
7. Author(s) Charles Koelbel, Piyush Mehrotra, and John Van Rosendale		8. Performing Organization Report No. 88-16
		10. Work Unit No. 505-90-21-01
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225		11. Contract or Grant No. NAS1-18107
		13. Type of Report and Period Covered Contractor Report
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225		14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor: Richard W. Barnwell

Submitted to Parallel Programming

Final Report

16. Abstract

On current multiprocessor architectures one must carefully distribute data in memory in order to achieve high performance. Process partitioning is the operation of rewriting 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 partitioning in which the compiler, guided by advice from the user, automatically transforms programs into such an interacting task system. This approach is illustrated with a picture processing example written in BLAZE, which is transformed into a task system maximizing locality of memory reference.

17. Key Words (Suggested by Author(s)) programming languages, parallel computers	18. Distribution Statement 61 - Computer Programming and Software; Unclassified -unlimited

19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of pages 22	22. Price A02