Process Decomposition Through Locality of Reference

Anne Rogers
Keshav Pingali*

TR 88-935
August 1988

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

* This research is supported by NSF grant CCR-8702668.

Process Decomposition Through Locality of Reference*

Anne Rogers
Keshav Pingali
Department of Computer Science,
Cornell University,
Ithaca, NY 14853.

August 24, 1988

Abstract

In the context of sequential computers, it is common practice to exploit temporal locality of reference through devices such as caches and virtual memory. In the context of multiprocessors, we believe that it is equally important to exploit spatial locality of reference. We are developing a system which, given a sequential program and its domain decomposition, performs process decomposition so as to enhance spatial locality of reference. We describe an application of this method - generating code from shared-memory programs for the (distributed memory) Intel iPSC/2.

Submitted to the Third Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS).

*This research is supported by NSF grant CCR-8702668.

1 Introduction

Fundamental limits on the switching times and integration densities of devices constrain the computational speeds of single processors. To achieve the computation rates required for large problems such as PDE solutions, it is necessary to harness the power of multiprocessing. The trend towards multiprocessing is very evident in the market-place - the best-selling CRAY machine is the CRAY-XMP which obtains its performance from four processors (each of which is only 30% faster than the original CRAY-1) while multiprocessors with less complex processing elements, such as the BBN Butterfly and Intel Hypercube, are becoming increasingly popular. The major barrier to widespread acceptance of multiprocessors is the primitive state of parallelizing compilers. Most vendors have taken the easy way out and have simply added parallel constructs to an existing language like C or FORTRAN. The compiler provides little or no help in parallelism detection or ensuring correct synchronization, leaving all that entirely to the programmer. This places a severe burden on the programmer and opens the door to time-dependent bugs such as races between reads and writes which are extremely difficult to track down. It is no exaggeration to say that compiler technology for multiprocessors is in the same sorry state that vectorizing compilers were in ten years ago. The CRAY-1 of the mid-70's sold well in spite of poor vectorizing software mainly because it could be used as a very fast scalar machine. Most multiprocessors, on the other hand, are being built from microprocessors such as the Intel 80386 or the Motorola 68020 which are not exactly supercomputers! The only way such machines will gain acceptance is if compiler technology improves to a point where programmers can exploit a large number of processing elements without undue programming effort. This will be achieved when the programmer can write his application program using standard high-level control and data abstractions such as procedures, loops, and arrays, leaving it to the compiler and run-time system to worry about lower level details such as process decomposition, synchronization, and load balancing.

Our approach to the compilation problem is based on exploiting *locality of reference*. In the context of uniprocessors, it is commonplace to exploit *temporal* locality of reference through devices such as caches, virtual memory, and translation look-aside buffers. We believe that it is just as important to exploit *spatial* locality of reference in the context of multiprocessors.

This concept can be explained with reference to the way multiprocessors are organized: as *message-passing* machines and as *shared-memory* machines. In message-passing machines, like the Intel Hypercube and the NCUBE, each process has its own address space and processes must communicate by explicitly sending and receiving messages. Message-passing systems typically take hundreds to thousands of cycles to deliver messages[1]. Thus, a process can access local data items (*i.e.*, data items in its own address space) very efficiently, but access to non-local data items, which must be choreographed through an exchange of messages, is rather inefficient. Clearly, exploiting locality of reference is critical in message-passing architectures. In shared-memory machines, such as the BBN Butterfly and the IBM RP3, there is a single, global address space that is shared by all processes. Inter-process communication is accomplished simply by reading and writing of memory locations. The single, shared address space is usually an illusion presented to the programmer by the operating system since most shared memory systems are implemented as a number of processor-memory pairs interconnected through some network. The cost of accessing a non-local data item (*i.e.*, across the network) is on the order of tens of cycles. Therefore, even in shared-memory machines, spatial locality of reference is extremely important for good performance[1]. In a recent paper on programming multiprocessors, Karp summarizes the importance of locality of reference as follows: "... we see that data organization is the key to parallel algorithms even on shared memory systems. It will take some retraining to get FORTRAN programmers to plan their data first and their program flow later. The importance of data management is also a problem for people writing automatic parallelization compilers....A new kind of analysis will have to match the data structures to the executable code in order to minimize memory traffic."[1]

These admonitions have not been heeded by most researchers working on automatic parallelization. The most popular approach to automatic parallelization is the 'program-driven' approach, a typical example of which is the Camp system of Peir and Gajski[3]. Their strategy is to parallelize the program by distributing loop iterations among processors. Synchronization

[1] The only exception to this is the Ultracomputer [2] in which all memory is equally far away from all processors. This uniformity is achieved by making all accesses equally expensive!

is required for loops with loop-carried dependencies and is implemented through complex bit-masks at every word of memory. A similar approach is being pursued in the CEDAR system at Illinois. Most of these efforts discuss locality but do very little to exploit it. In contrast, exploiting locality of reference is the cornerstone of our approach. The intuitive idea is the following. The programmer writes and debugs his program in a high-level language such as Id Nouveau[4] using standard high-level abstractions such as loops and arrays. Once this is accomplished, he specifies the *domain decomposition* - that is, how data structures are to be distributed across the multiprocessor. In most programs we have looked at (such as matrix algorithms, SIMPLE, and particle-in-the-cell), this is quite straight-forward since the programmer thinks naturally in terms of mapping by columns, rows, blocks, etc. Given this data decomposition, the compiler performs process decomposition by analyzing the program and specializing it to the data that resides at each processor. Thus, our approach to process decomposition is 'data-driven' rather than 'program-driven'. An interesting facet of our technique is that it can be viewed formally as a novel kind of type inference for overloaded operators.

The rest of the paper is organized as follows. In Section 2, we introduce the programming language and the machine model we use in this paper. The programming language is a functional language augmented with I-structures, an array construct borrowed from logic programming languages. The machine model is a simple message-passing model similar to that supported by the Intel Hypercube or the Ncube. In this section, we also introduce the 'wavefront' problem which we use as a running example in this paper. In Section 3, we discuss our code generation strategy. The basic strategy is embodied in a simple but inefficient algorithm called *run-time resolution.* The code generated by this algorithm can be improved considerably by *partial evaluation* (evaluation at compile-time) and incorporating this optimization leads to an improved algorithm that we call *compile-time resolution.* We also remark on some connections between our techniques and standard code generation strategies for languages with overloaded operators. In Section 4, we discuss other optimizations that must be performed to generate good code. To reduce the overhead of message passing, it is preferable to combine messages together, thereby reducing message traffic. However, this must be done judiciously since combining messages can have an adverse effect on parallelism. The transformations in Section 4 attempt to strike a balance between these two concerns. We present experimental results that highlight the importance of these transformations. We discuss some extensions in Section 5, related work in Section 6, and conclusions in Section 7.

2 Language and Machine Model

The programming language we use in this paper is Id Nouveau [4] which is a functional language with an array construct called *I-structures* borrowed from logic programming languages. Our techniques work equally well for an imperative language such as FORTRAN. The machine model is a simple message-passing model like the one supported on the Intel Hypercube or the Ncube. We chose to work with this model since we are implementing the system on the Intel iPSC/2. However, our techniques can also be used to exploit spatial locality of reference in code for shared-memory machines such as the BBN Butterfly.

2.1 Programming Language

Id Nouveau is a functional language augmented with I-structures, an array construct borrowed from logic programming languages. The rationale behind this integration of functional and logic programming constructs is to permit the programmer to define large arrays and matrices incrementally without incurring the copy overhead of functional arrays. We assume that the reader is familiar with functional languages; therefore, we will describe only I-structures. In a functional language, the allocation of storage for an array is inseparable from the definition of the array elements. This makes it difficult to write programs in which arrays must be defined incrementally. I-structures get around this problem by separating the allocation of storage from the definition of the array elements. This is similar to imperative arrays; however, unlike in imperative arrays, an element of an I-structure cannot be redefined once it has been given a value. Looked at another way, I-structures are 'write-once' arrays. For the sake of completeness, we describe the primitives for manipulating two-dimensional I-structures.

`matrix(e1,e2)` The expressions `e1` and `e2` must evaluate to positive integers. A matrix of that size is allocated.

`A[i1,i2] = e` The expression `e` is evaluated and the resulting value is stored into `A[i1,i2]`. If `A[i1,i2]` has already been written into, a run-time error occurs.

`A[i1,i2]` The contents of `A[i1,i2]` are returned. If `A[i1,i2]` is undefined, a run-time error occurs.

For a complete description of Id Nouveau, we refer the reader to [5]. Figure 1 shows the Id Nouveau program for the Gauss-Seidel relaxation method applied to a grid in normal order. In this program, procedure init-boundary initializes the boundary of the array *New*. Interior elements in the new matrix are computed by averaging neighbors, two from the old matrix and two from the new as illustrated in Figure 2a. The code in italics specifies the domain decomposition and will be explained later.

2.2 Machine Model

As we stated in the introduction, our techniques work equally well for shared memory as well as message-passing machines. For the sake of concreteness, we will assume a message-passing model similar to that provided by the Intel iPSC or the NCube. There are n processors in the model, each of which executes one process[2]. We assume that communication between two processors is independent of the identities of the processors. This is a reasonable assumption because in most message-passing systems, the time for packing and unpacking a message dominates the 'time-of-flight' of the message. Thus, the cost of accessing a data item is 'binary' - local access is more efficient than non-local access, but all non-local accesses are equally expensive.

2.3 Domain Decomposition

To exploit spatial locality, a programmer first decomposes the data in a manner appropriate for the problem and the architecture. For example, given an architecture that contains a ring of size S, a good data organization for executing this version of Gauss-Seidel in parallel is to wrap the

[2]Strictly speaking, the iPSC permits multiple processes to execute on a processor but we can take that into account simply by increasing the number of processors in our model.

columns of the matrix around a ring like a dealer deals cards, one column to each processor in turn until all of the columns have been distributed. In general, column j is assigned to processor j *mod* s. In Figure 1, which is the Gauss-Seidel program discussed earlier, the italicized portion of the program specifies the domain decomposition. Parallelism in this program is along a wavefront (see Figure 2b).

```
Procedure GS-iteration(Old:column) : column
    Let New = matrix(N,N) : column  in
      init–boundary New;
      for j = 2 to N-1 do
          for i = 2 to N-1 do
            New[i, j] = c × (New[i–1, j] + New[i, j–1] +
                                              Old[i+1, j] + Old[i, j+1]);
    return New
```

Figure 1: Sequential version of Gauss-Seidel Iteration

In general, domain decomposition is specified as follows. Variables may be mapped to either a single processor (a:P1) or to all processors (a:ALL). This processor is said to *own* the variable. Array mappings consist of three functions:

Map Given the indices of an array reference, map computes the processor on which the element resides.

Local Given the indices of a reference, local computes the location of the reference in the processor on which it resides.

Alloc Given the subscript ranges for the original array, allocate an appropriately sized local array.

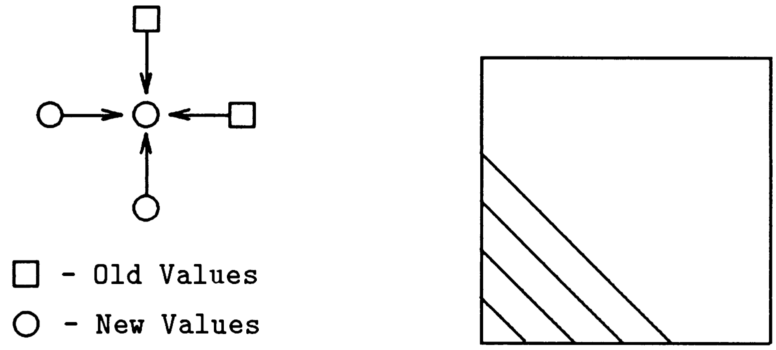
The *owner* of an array element is the processor to which it is mapped.

For example, the wrapped columns discussed earlier are defined as:

Column = <col-map, col-local, col-alloc>

Col-map(i, j) = j **mod** s

$$\text{Col-local}(i, j) = (i-1)\times N/s + (j \textbf{ div } s)$$
$$\text{Col-alloc}(N, N) = \text{matrix}(N, N/S)$$



(a) Data Dependencies (b) Wavefront

Figure 2: Wavefront Program Data Dependencies

Our prototype compiler accepts programs such as the one in Figure 1 as input. Our goal is to produce C code for the iPSC/2 that does as well as a handwritten program. For the purpose of discussion, Figure 3 contains a handwritten version of Gauss-Seidel written in pseudo-Id with explicit message passing constructs. The matrix is wrapped by column around a ring of size S. The complexity of this program is caused by the need to reduce the number of message exchanged between neighboring processors. Values from the *old* matrix are sent a column at a time to the left. To exploit wavefront parallelism, it is necessary to pipeline the computation and communication of the *new* values. To accomplish this, we compute a block of *new* values and send the block in one message to the right. The best block size depends on the size of the matrix.

In the remainder of the paper, we discuss how a compiler can generate the code similar to that of Figure 3 from the program of Figure 1.

```
LEFT = (p−1) mod s;
RIGHT = (p+1) mod s;

Procedure GS−iteration(Old)
  Let New = matrix(N, N/S)
      rnewvalues = vector[blksize]
      snewvalues =vector[blksize] in

      init−boundary(New);
          { For every column residing in this process }
      for j =   1 to N/S   do
            { Send column j of Old values to the LEFT
                and receive column j+1 of Old values
                from the RIGHT   }
        send(Old[1..N, j], LEFT);
        t[1..N] = receive(Old[1..N, j+1], New);

            { The new values for column j are computed
                and communicated in blocks of size blksize }
        for k = 0 to N/blksize do
              { Receive a block of new values for column j−1 }
          rnewvalues[1..blksize] = receive(snewvalues[1..blksize],
                                                          LEFT);

              { Compute a block of new values for column j }
          for i = k×blksize + 2 to min((k+1)×blksize+1, n−1) do
            New[i, j] = c × (rnewvalues[i mod blksize] +
                                    New[i−1, j] + t[i] + Old[i+1, j]);
            snewvalues[i mod s] = New[i,j];

              { send these values to the RIGHT }
          send(snewvalues[1..blksize], RIGHT);

      return(New);
```

Figure 3: Handwritten version of Gauss-Seidel iteration for a non-boundary processor, p

3 Code Generation

We first discuss *run-time resolution* which is a simple but fairly inefficient implementation. Next, we show how this code can be improved by *partial evaluation* at compile-time - the resulting code generation strategy is called *compile-time resolution.* We also point out some connections between this problem and the problem of code generation for languages with overloaded operators.

3.1 Run-time Resolution

Our first method, called *run-time resolution*, produces the same program for each processor. Three simple rules drive the generation of code.

1. The owner of a variable or array element computes its value.
2. The owner of a variable or array element is responsible for communicating its value to any processor that requires it.
3. Every statement is examined by every processor to determine its role (if any) in the execution of the statement.

For example, in the program of Figure 4a, the first statement will be executed by processor `P1` since the identifier `a` is mapped onto processor `P1`. Similarly, the expression on the right hand side of the third statement is computed by `P3`, with `P1` and `P2` participating only in the communication of the values of `a` and `b` respectively. `Coerce` sends a value from the processor that owns it to the processor that needs it. These processors may be the same, in which case just a read is performed.

Figure 4b shows the code generated by the run-time resolution strategy from the program of Figure 4a. The code in Figure 4b is executed by all processors. **mynode** is a procedure that is executed by a processor to determine its own identity.

3.2 Compile-time Resolution

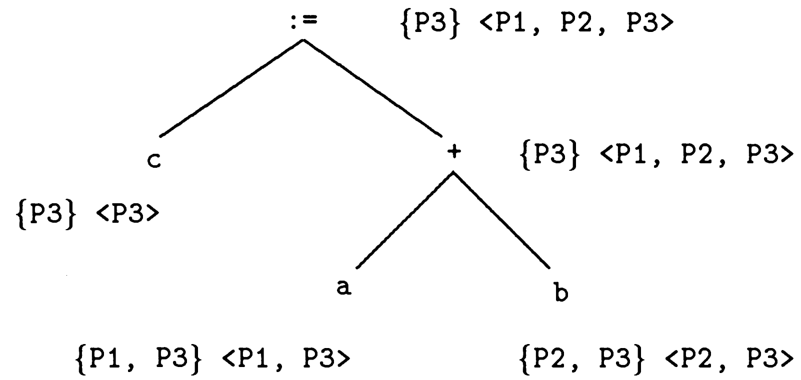
Run-time resolution inserts many extra lines of code into each processor. In our example, none of the tests that follow the first coercion in the code

```
a:P1, b:P2, c:P3
a := 5;
b := 7;
c := a + b;
```

(a) Sequential code

```
P1, P2, P3:
if P1 = mynode() then a := 5;
if P2 = mynode() then b := 7;
t1 := coerce(a, P1, P3);
t2 := coerce(b, P2, P3);
if P3 = mynode() then t3 := t1 + t2;
if P3 = mynode() then c := t3;
```

(b) Code generated by run-time resolution



(c) Evaluators and participants computations

```
P1:
a := 5;
send(a, P3);
```

```
P2:
b := 7;
send(b, p3);
```

```
P3:
t1 := receive(a, P1);
t2 := receive(b, P2);
t3 := t1 + t2;
```

(d) Code generated by compile-time resolution

Figure 4: Simple example

will evaluate to true for processor P1. Techniques similar to those used to resolve overloading in conventional compilers can be used to generate less extraneous code. When compiling languages like Lisp, an overloaded operator like $+$ is usually compiled into a case statement that tests the type of the arguments and dispatches to the appropriate type specific addition routine. The naive code generated by this strategy can be improved considerably if the compiler knows the types of the arguments or the result (for example, through type declarations) since the case statement can be replaced by a dispatch to the relevant addition routine. This kind of code improvement through 'specialization' of generic code can be used profitably in our context as well. Our compiler uses mapping information to eliminate many tests by specializing, for each processor, the 'generic' code produced by the run-time resolution strategy (which runs on all processors). This approach is called *compile-time resolution.*

When generating code for each processor using compile-time resolution, the compiler examines each statement to determine the processor's role in the evaluation of that statement. This is done in two stages. The compiler uses conventional abstract syntax trees as the internal representation of programs. In the first stage, the user's mapping information is propagated through the program's abstract syntax tree. In the second stage, this information is used to generate code. Each node of the abstract syntax tree has two attributes named *evaluators* and *participants.* The *evaluators* of a node in the abstract syntax tree is the set of processors that perform the operation defined by the node. The *participants* of a node, n, in the abstract syntax tree is the set of processors that must participate in the evaluation of some node in the subtree rooted at the node, i.e. the union of the evaluators of the nodes in the subtree rooted at n.

Figure 4c shows these sets for the our simple example; the evaluators are enclosed in braces and the participants are enclosed in angle brackets. In this simple example, only processor names appear in the evaluators and participants. This is not necessarily the case since the mapping for an array reference will be an equation that may include program variables. For example, if a matrix, `A`, is declared to be mapped by `column`, the evaluators for the reference, `A[i, j+1]`, would include `(j+1) mod S`.

The set of participants is used to determine the evaluators for some types of nodes, such as conditionals. The union of the participants of the then-branch and else-branch defines the evaluators for a conditional expression. The participants for a function definition is also a function. To determine the evaluators of a particular function call, the participants function is symbolically applied to the actual parameters of the call.

The information collected in the propagation stage is used to generate code. Given a processor name and a tree node, the compiler tries to determine if the processor is a member of the evaluators of the node. Three outcomes are possible: true, false, and inconclusive. True means that the processor must perform the operation defined by the node. False means it need not. Inconclusive means that run-time resolution must be applied because the compiler cannot analyze the mappings sufficiently. This evaluation will require techniques such as subscript analysis that are commonly used in vectorizing compilers. The code generation phase produces code for each processor by walking the annotated abstract syntax tree while applying this evaluation scheme at each node. Figure 4d contains the code that this method generates for our simple example.

Returning to the Gauss-Seidel example discussed earlier, Figure 5 contains the code that would be generated by compile-time resolution for a non-boundary processor p. Since the goal of compile-time resolution is to have each processor participate in only those computations for which it has data, it is important to ensure that a processor executes only required loop iterations, rather than go through all iterations looking for work. To compute the required set of iterations for a given processor, we set the equations in the evaluators equal to the processor name and solve for the loop variable.

4 Optimizations

While Figure 5, the end result of compile-time resolution, resembles the handwritten program (Figure 3) to some extent, there are important differences in the treatment of messages in the two programs. By combining messages that have the same source and destination processors, the handwritten version attempts to cut down on the number of messages that must be sent - for example, values in the *Old* column are sent through a single *send* command, rather than one element at a time. This is useful because of the relatively high start-up cost of messages on the iPSC/2. On the other hand, combining messages may impact adversely on parallelism - for

```
Procedure GS–iteration(Old)
  Let New = col–alloc(n, n) in
      init–boundary(New);
      for j = p to N by S do
        for i = 1 to N do
            send(Old[col–local(i, j)], (j–1) mod S);
        for i = 1 to N do
            t1 = New[col–local(i–1, j)];
            t2 = receive(New[i, j–1], (j–1) mod S);
            t3 = Old[col–local(i+1, j)];
            t4 = receive(Old[i, j+1], (j+1) mod S);
            New[col–local(i, j)] = c x (t1 + t2 + t3 + t4);
        for i = 1 to N do
            send(New[col–local(i, j)], (j+1) mod S);
      return(New)
```

Figure 5: Gauss-Seidel code by compile-time resolution for a non-boundary processor

example, if the *New* column is passed only after all of its elements have been computed, there is no parallelism in the execution of the program. Thus, communication and computation have to be pipelined in order to achieve the best tradeoff between minimizing the number of messages and exploiting parallelism. The handwritten version achieves this by sending the *new* elements in blocks of size 8, a compromise between sending them one at a time and sending them all at once.

How important are these optimizations? To understand this issue, we performed a set of preliminary experiments to determine how the code generated using run-time and compile-time resolution compares with handwritten code. Figures 6 and 7 contain graphs of the results for an 128 x 128 integer grid. Compared to the handwritten version, the run-time resolution code performs rather poorly. This was to be expected because it exchanges many more messages than the handwritten code [3] and messages on the Intel iPSC/2 are very expensive. The relatively flat shape of the curve arises from the fact that there is no parallelism being exploited in this program. The compile-time implementation is more encouraging but still bad. It exchanges as many messages as the run-time version but each processor only participates in those iteration for which it has data. However, it does not exploit any parallelism either. Figure 6 (Optimized I) shows the improvement that results from combining all the values of the *Old* column into a single message. The most impressive gains are demonstrated by Figure 6 (Optimized II) which shows the improvements due to pipelining of computation and communication. In this program, a *new* value is sent as soon as it is computed. This leads to a lot of message traffic. By 'blocking' these values, we obtain the curve of Figure 6 (Optimized III) which has the best performance - the block size is a compromise between decreasing the number of messages and exploiting parallelism.

The effect of these optimizations can be obtained through standard transformations on the code produced by compile-time resolution. The techniques are *vectorization*, *loop jamming*, and *strip mining* (see [6]). Vectorizing the *send*'s of the elements of the *Old* column has precisely the effect of combining all these messages into a single message. This is a standard transformation based on recognizing that the old values are not changed

[3]31,752 messages for the run-time resolution code versus 2142 messages for the handwritten code.

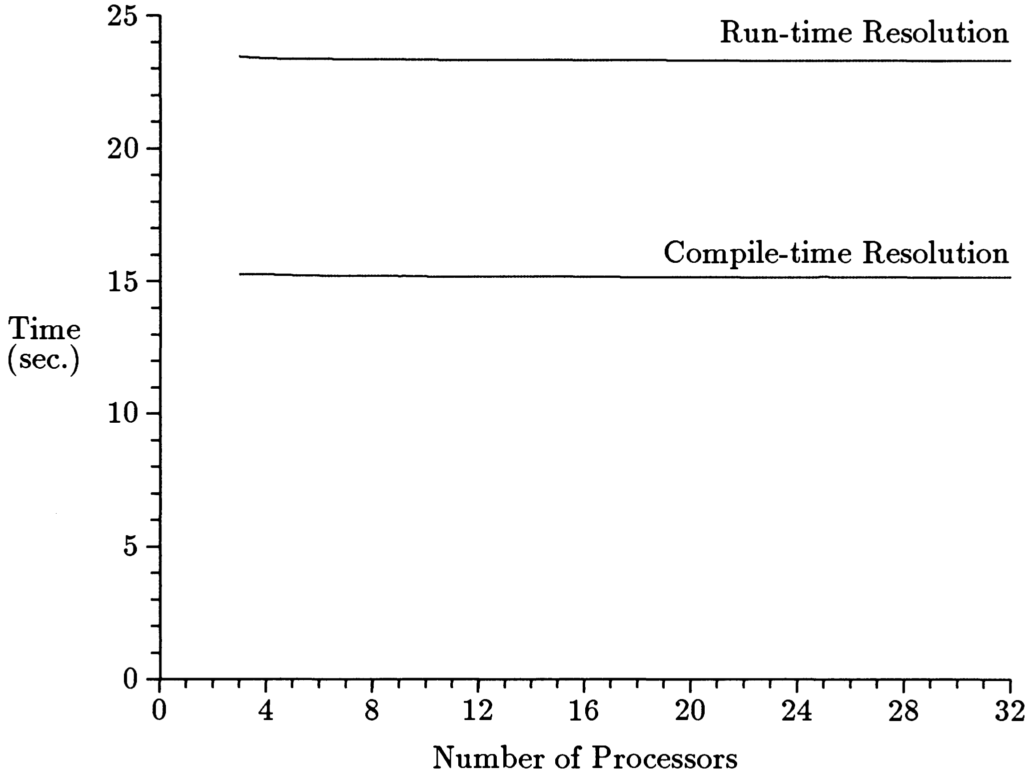


Figure 6: Effect of Compile-time and Run-time Resolution

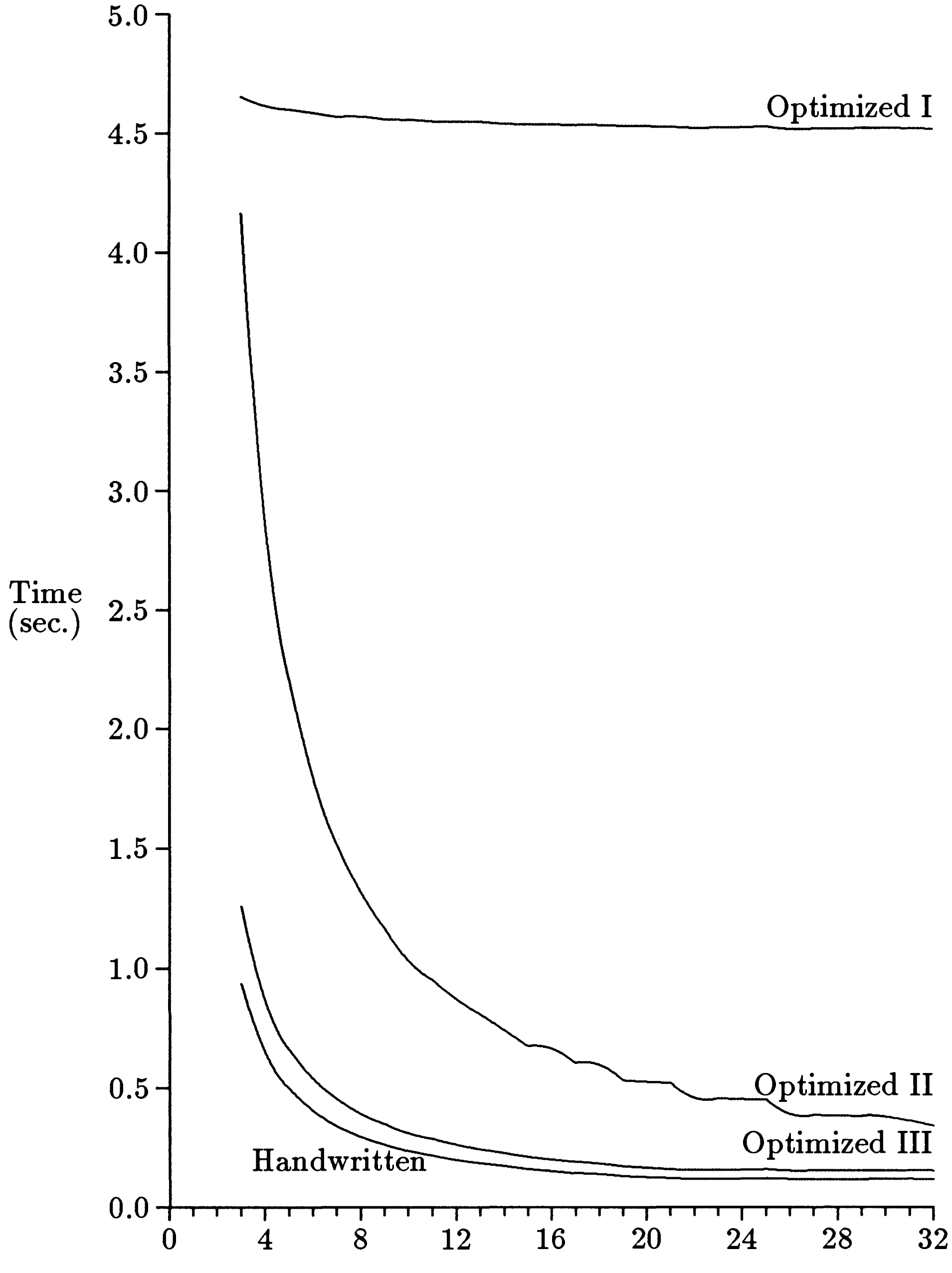


Figure 7: Effect of Message-Passing Optimizations

during the execution of the loop. Next, applying loop jamming to the computation and communication loops in the vectorized compile-time version yields a program in which new values are sent off as soon as they are computed. The final optimization is to stripmine this loop, resulting in the desired program. One issue that is still under study is the determination of the block size to obtain the best trade-off between minimizing message traffic and exploiting parallelism. Appendix A has a more complete discussion of the effects of these transformations, including the resulting programs.

We plan to automate these transformations in the next phase of our compiler development. In addition to these transformations, we will also include transformations to align the order of the computation with the mapping of the data. For example, if the sequential version of Gauss-Seidel had had the i and j-loops reversed then generated code would not have shown any parallelism, so loop interchange would be required.

5 Extensions

This section briefly presents several extensions we plan to incorporate into our system: mapping polymorphism, higher order functions, dynamically spawned processes, and load balancing.

5.1 Mapping Polymorphism

One disadvantage of the system described above is that a procedure must have a fixed mapping - if the programmer wants two different mappings for a procedure, he must make two copies of the code of procedure. This is, of course, analogous to the situation in ordinary type specification - a PASCAL programmer who wants to sort both integer lists and floating point lists must write two procedures with similar code but different type specifications. To get around this problem, we can introduce *mapping polymorphism* by permitting the abstraction of mapping specifications, in much the same way that abstracting types from procedures yields polymorphic type systems. The constants include the single processor mappings as well as the special constant ALL. The type of a function describes the mappings of the arguments and the return value along with their conventional types. A difference arises between the mapping and the conventional view of array. The mapping of an array element depends on which element it is, i.e. it depends on the values of the indices into the array. Such mappings may be modeled with a dependent product[4]. A simple example illustrates why this is of more than theoretical interest. Consider the identity function $f = \lambda a : P1.a$ and the code fragment

b:P2;
c:P3;

...f(b)...
...f(c)...

Applying compile-time resolution would generate the three processes in Figure 8.

P1:	P2:	P3
$f_{P1}(a) = a;$		
...		
receive(temp1, P2);	...	...
temp2 = f_{P1}(temp1);	**send**(b, P1);	**send**(c,P1);
send(temp2, P2);	**receive**(temp, P1);	**receive**(temp,P1);
...	...	...
receive(temp3, P3);		
temp4 = f_{P1}(temp3);		
send(temp4, P3);		
...		

Figure 8: Code Generated in Absence of Mapping Polymorphism

Even though processes `P2` and `P3` own the only values used in each of the function calls, the calls must be performed by processor `P1` because `f` is defined to take an argument owned by processor `P1`. If instead we allowed the polymorphic identity function $\lambda P.\lambda a : P.a$ and the code fragment

[4]The members of a dependent product type x:A$\sharp$B are the terms <a, b> with $a \in A$ and $b \in B[a/x]$. The type of the second component may depend on the first component[7].

```
b : P2;
c : P3;

...f(P2)(b)...
...f(P3)(c)...
```

the generated code (see Figure 9) would exhibit much more parallelism. Not only can `f(b)` and `f(c)` be done in parallel but also four messages have been eliminated.

P1:	P2:	P3:
	f_{P2}(a:P2) = a;	f_{P3}(a:P3) = a;
	...	...
	f_{P2}(b);	f_{P3}(c)
	...	...

Figure 9: Code Generated with mapping Polymorphism

We do not as yet understand the full ramifications of mapping polymorphism. We believe that there are connections to work done by Lucassen and Gifford on using mapping specifications to perform alias detection[8].

5.2 Higher Order Functions

Higher order functions are those that may take functions as arguments and return functions as results. Run-time resolution can handle higher-order functions without any modifications. To permit good compile-time resolution, some restrictions must be placed on functions. All functions passed as an argument to a given function must have:

- the same argument types and return type,
- the same argument mappings and return mapping,
- and the same participants function.

Without these restrictions, compile-time analysis may produce code that is not much better than run-time resolution. To allow functions as return values second order participants functions are needed. We do not as yet know if these restrictions are overly constraining.

5.3 Dynamically Spawned Processes

Our current scheme is based on a fixed number of processes. We would like to expand our mapping language to allow the user to specify that a new process should be allocated for a particular set of data. This would allow the user to specify some process graph parallelism without needing to be concerned about process decomposition and synchronization. The code for the new process will be determined by the data mapped to it, using the usual rules. Synchronization is a bit more tricky; any existing process that owns data required by the new process must participate in the spawning of the new process, so that its existence and need for data are known.

In addition to user specified dynamically spawned processes, our compiler will need to spawn new processes to exploit process graph parallelism. Since Id Nouveau is a functional language, this kind of parallelism is easy to detect but we need some metric to determine when spawning a new process is worth the overhead.

5.4 Load Balancing

A good process decomposition places several processes on one processor to ensure that when one process needs to wait for a remote reference the processor running it will have work to do. In addition to hiding memory latency, having multiple processes on a node facilitates load balancing. Processes may be shuffled from overloaded to underloaded nodes without slowing their execution if the data associated with a process is moved along with the code. We would like to experiment with a simple load balancing scheme that moves a process and its data together. Preliminary work on this problem has been done by Fox et al [9] and we are studying their work to see if it can be adapted to our situation.

6 Related Work

Callahan and Kennedy are studying similar techniques for compiling a version of FORTRAN 77 that includes annotations for specifying a data decomposition, for the Intel iPSC/2. In [10] they describe at length a method quite similar to our run-time resolution. They also discuss how existing transformations may be used to improve their generated code. Our methods are equally applicable to FORTRAN.

FORTRAN 77 does not have recursion. If code space is not at a premium, function calls may be expanded inline. As a result, work on parallelizing FORTRAN often does not deal explicitly with functions. In our setting, inline expansion is not possible since we have recursion. Instead, compile-time resolution must perform interprocedural analysis to determine which processes need to participate in a particular function call.

Koelbel, Mehrotra, and Van Rosendale [11] at Purdue are translating Blaze, a functional language with a forall construct, into an extension, E-Blaze, that includes constructs for explicit process creation, data storage layout, and interprocessor communication and synchronization. They use programmer supplied data decomposition information to schedule forall loops to exploit spatial locality.

7 Conclusions

In this paper, we presented a compilation strategy for multiprocessors that exploits locality of reference. The ideas in this paper are applicable to both shared-memory processors as well as message-passing processors. We discussed an implementation on the Intel iPSC/2.

8 Acknowledgements

Many thanks to Laurie Hendren for typesetting the graphs and to Tom Coleman for looking at our programs.

A Optimizing the Compile-time Resolution Code

Section 4 discusses the optimizations we applied to the compile-time resolution code to reduce message traffic and to pipeline computation and communication. In this appendix, we present each of the four versions of the wavefront iteration code with an explaination of the transformation applied and its effects. The programs in this appendix are written in C for the iPSC/2. `is_read` and `is_write` are macros from our run-time system that perform I-structure reads and writes.

A.1 Compile-time resolution code

The code shown here was generated using the compile-time resolution rules for a non-boundary processor p.

```
istructure wavefront (Old, n)
istructure a;
int n, p;

{ int c, i, j, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7, left, right;
  istructure New;

  left = p - 1;
  right = p + 1;

  c = 1;
  New = column_alloc(n, n);
  init_boundary(New, n, p);

  for (j=p <=n-3; j = j + S)  {
     for (i=2; i <= n-1; i++)
        { tmp1 = is_read(Old, column_local(i, j+1));
          csend(p, &tmp1, sizeof(int), left, 0); }

     for (i=2; i <= n-1; i++)
        { tmp2 = is_read(New, column_local(i-1, j+1));
          crecv(right, &tmp3, sizeof(int));
          tmp4 = is_read(Old, column_local(i+1, j+1));
          crecv(left, &tmp5, sizeof(int));
```

```
        tmp6 = c * (tmp2 + tmp3 + tmp4 + tmp5);
        is_write(New, column_local(i, j+1), tmp6); }

    for (i=2; i <= n-1; i++)
       { tmp7 = is_read(New, column_local(i, j+1));
         csend(p, &tmp7, sizeof(int), right, 0);  } }

  return(New); }
```

A.2 Applying Vectorization

In the original compile-time resolution version, each element of a column of the *Old* matrix is sent to the left in its own message. It is straightforward to recognize that these sends may be vectorized, since the *Old* values do not change during the computation. Vector sends and receives are inserted and every send and receive of an old value is converted to access these vectors.

```
istructure wavefront (Old, n)
istructure a;
int n;

{ int c, i, j, tmp2, tmp4, tmp5, tmp6, tmp7, left, right, *oldvalues;
  istructure b;

  left = p - 1;
  right = p + 1;

  c = 1;
  New = column_alloc(n, n);
  init_boundary(New, n, p);
  oldvalues = (int *) calloc(n, sizeof(int));

  for (j=p; j <=n-3; j = j + S)  {
          /* fill in the vector with Old values and send it. */
    for (i=2; i <= n-1; i++)
       oldvalues[i] = is_read(Old, column_local(i, j+1));
    csend(p, oldvalues, sizeof(int)*n, left, 0);

       /* Recieve a column of Old values */
    crecv(right, oldvalues, sizeof(int)*n);
```

```
    for (i=2; i <= n-1; i++)
       { tmp2 = is_read(New, column_local(i-1, j+1));
         tmp4 = is_read(Old, column_local(i+1, j+1));
         crecv(left, &tmp5, sizeof(int));
             /* Read Old value from the received column */
         tmp6 = c * (tmp2 + oldvalues[i] + tmp4 + tmp5);
         is_write(New, column_local(i, j+1), tmp6); }

    for (i=2; i <= n-1; i++)
       { tmp7 = is_read(New, column_local(i, j+1));
         csend(p, &tmp7, sizeof(int), right, 0);  } }

  return(New); }
```

A.3 Applying Loop Jamming

As discussed in section 4, the vectorized version is significantly faster but still exhibits no parallelism. This arises from the fact that every element of a *New* column is computed before any of them are sent. Pipelining the computation and communication of the elements in a column is achieved by applying loop jamming to the computation and communication loops.

```
istructure wavefront (Old, n)
istructure a;
int n;

{ int c, i, j, tmp2, tmp4, tmp5, tmp6, tmp7, left, right, *oldvalues;
  istructure b;

  left = p - 1;
  right = p + 1;

  c = 1;
  New = column_alloc(n, n);
  init_boundary(New, n, p);
  oldvalues = (int *) calloc(n, sizeof(int));
```

```
  for (j=p ; j <=n-3; j = j + S)  {
     for (i=2; i <= n-1; i++)
        oldvalues[i] = is_read(Old, column_local(i, j+1));
     csend(p, oldvalues, sizeof(int)*n, left, 0);

     crecv(right, oldvalues, sizeof(int)*n);
         /* Pipeline the computation of New[i, j] with its
                communication to the left */
     for (i=2; i <= n-1; i++)
        { tmp2 = is_read(New, column_local(i-1, j+1));
          tmp4 = is_read(Old, column_local(i+1, j+1));
          crecv(left, &tmp5, sizeof(int));
          tmp6 = c * (tmp2 + oldvalues[i] + tmp4 + tmp5);
          is_write(New, column_local(i, j+1), tmp6);

          tmp7 = is_read(New, column_local(i, j+1));
          csend(p, &tmp7, sizeof(int), right, 0);  } }

  return(New); }
```

A.4 Applying Strip Mining

The final optimization, strip mining, further reduces message traffic without nullifying the effects of pipelining. Rather than sending each *New* value as it is computed, a block of *New* values is computed and then sent in a single message. The transformation necessary to achieve this is strip mining. The outer-loop (k-loop) steps through a column in blocks while the inner-loop (i-loop) processes each element in a block. We use two arrays, `snewvalues` and `rnewvalues`, to hold the blocks of communicated values to make the code easier to follow. One would suffice.

```
istructure wavefront (Old, n)
istructure a;
int n;

{ int c, i, j, k, tmp2, tmp4, tmp6, left, right, *oldvalues,
           snewvalues[blksize], rnewvalues[blksize];
  istructure b;
```

```
left = p - 1;
right = p + 1;

c = 1;
New = column_alloc(n, n);
init_boundary(New, n, p);
oldvalues = (int *) calloc(n, sizeof(int));

for (j=0; j <=n-3; j = j + S)  {
   for (i=2; i <= n-1; i++)
      oldvalues[i] = is_read(Old, column_local(i, j+1));
   csend(p, oldvalues, sizeof(int)*n, left, 0);

   crecv(right, oldvalues, sizeof(int)*n);
        /* Walk through column j in blocks of size blksize */
   for (k=0; k <= n/blksize; k++) {
           /* Receive a block of  newvalues */
      crecv(left, rnewvalues, sizeof(int)*blksize);
      for (i=k*blksize+2; i <= min((k+1)*blksize+1, n-1); i++)
         { tmp2 = is_read(New, column_local(i-1, j+1));
           tmp4 = is_read(Old, column_local(i+1, j+1));
           tmp6 = c * (tmp2 + oldvalues[i] + tmp4 + rnewvalues[i%blksize]);
           is_write(New, column_local(i, j+1), tmp6);

           snewvalues[i%blksize] = is_read(New, column_local(i, j+1)); }
          /* Send New values in a block */
      csend(p, snewvalues, sizeof(int)*blksize, right, 0);  } }

return(New); }
```

References

[1] A. Karp, Programming for Parallelism, *IEEE Computer, May 1987.*

[2] A. Gottlieb et al., The NYU Ultracomputer– Designing an MIMD Shared Memory Parallel Computer, *IEEE Transactions on Computers, 1983.*

[3] J. Peir and D. Gajski, Camp: a programming aide for multiprocessors, *Proceedings of the International Conference on Parallel Processing, 1986.*

[4] Arvind, R. Nikhil and K. Pingali, Id Nouveau: Language and Operational Semantics, *CSG Memo*, M.I.T., September 1987.

[5] Arvind, R. Nikhil, K. Pingali, I-structures: Data Structures for Parallel Computing, *Proceedings of Workshop on Graph Reduction, Santa Fe, NM. Springer-verlag LNCS 279 (September 1986).*

[6] D. Padua and M. Wolfe, Advanced Compiler Optimizations For Supercomputers, *Communications of the ACM, December 1986.*

[7] R. L. Constable et al, *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[8] J. Lucassen and D. Gifford, Polymorphic Effect Systems, *Proceedings of the Fifteenth Annual Symposium on Principles of Programming Languages.*

[9] G. Fox et al, *Solving Problems on Concurrent Processors, Volume 1*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[10] D. Callahan and K. Kennedy, Compiling Programs for Distributed-Memory Multiprocessors, to appear in *The Journal of Supercomputing.*

[11] C. Koelbel, P. Mehrotra, and J. Van Rosendale, Semi-Automatic Domain Decomposition in Blaze, *Proceedings of the International Conference on Parallel Processing, 1987.*