Compiling for distributed memory architectures

Anne Rogers and Keshav Pingali

June 10, 1991

1 Introduction

For MIMD computers to become the computers of choice, compiler technology must improve to a point where the programmer can exploit parallelism without undue programming effort. This will be achieved when the programmer can get good performance from application programs written using high-level control and data abstractions, leaving it to the compiler and run-time system to worry about aspects of the architecture such as the number of processors and the interconnection topology, and intricacies of parallel execution such as synchronization and load balancing. Unfortunately, MIMD machines cannot be programmed like this at present. In distributed memory MIMD machines, like the Intel Hypercube and NCUBE, the programming model consists of multiple processes, each with its own address space, that communicate with each other by exchanging messages. The programmer must decompose his algorithm into processes, distribute data among the address spaces of these processes and insert message sends and receives wherever a process needs access to a non-local data item. The need to deal with multiple address spaces results in a loss of abstraction in programming. For example, if a process needs access to element X[i,j] of a global array X that is distributed across processes, it can read the value directly if the element is mapped into its own address space; otherwise, it must exchange messages with the process on which this element resides. Furthermore, synchronization between processes is also accomplished using message passing, and doing sends and receives in the wrong order can result in cycles of deadlocked processes. These problems are absent in shared memory MIMD machines because there is a single global address space. A typical shared memory multiprocessor today supports conventional imperative languages like FORTRAN extended with constructs like DO-ALL or DO-ACROSS for explicitly requesting parallel computation and constructs like locks for synchronization. Unfortunately, inserting correct synchronization code is as difficult as writing correct operating systems code on uniprocessors and reading and writing shared locations in parallel opens the way to inadvertent race conditions that are very difficult to find. We conclude that explicit management of parallelism in programs is too onerous and error-prone for most programmers.

One alternative is to program in a language without explicit parallelism such as FORTRAN or Id. This throws the burden of exposing and managing parallelism on the compiler and run-time system. For distributed memory architectures, the compiler must decide what should execute in parallel and how data should be decomposed across the address spaces of the processes, inserting message transfers wherever a process needs access to non-local data. An extremely important consideration is locality of reference — data should be mapped in such a way that processes access local data as far as possible. This is because message passing can cost as much as two orders of magnitude more than accessing local memory. On current generation machines, most of this cost is the software overhead of packing and unpacking the message, and the transmission time of the message itself is relatively small. Therefore, the topology of the network can be ignored and a reasonable approximation of a message-passing machine is a two-level memory hierarchy in which non-local accesses are about two orders of magnitude more expensive than local accesses. On such a hierarchy, it is clear that process decomposition must attempt to locate code and the data it references in the same process. The same considerations apply when compiling for shared memory machines. The single shared address space is usually implemented using memory that is physically distributed among the processors, so that a processor can access its local memory relatively quickly but must go across the network to access non-local memory. Such machines are called Non-uniform Memory Access (NUMA) machines and exploiting locality of reference is important on these machines as well. The importance of combining parallelism detection with proper data distribution to achieve locality of reference has been eloquently summarized by Karp[11] 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 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.

In this paper, we report on one such system. The cornerstone of our approach is to let the mapping of data on to processors drive process decomposition. The idea underlying our approach is to enable the programmer to write and debug his program in a high-level language using standard high-level abstractions such as loops and arrays. In addition, he specifies the *domain decomposition* — a mapping of the data structures on to the multiprocessor. In most programs we have looked at (such as matrix algorithms and SIMPLE[7]), this is quite straightforward since the programmer thinks naturally in terms of decompositions by columns, rows, blocks, and so on. Given this data decomposition, the compiler performs process decomposition by analyzing the program and specializing it, for each processor, to the data that resides on that processor. Thus, our approach to process decomposition is "data-driven" rather than "program-driven" as are more traditional approaches[1, 19].

1.1 System Overview

Our compiler takes a sequential program and a domain decomposition and generates C code for the Intel iPSC/2. The system generates code for each process based on the data residing in that process. In this section, we present an overview of our system which includes two code generation schemes and three message optimizations.

The difficulty of code generation arises from the need to generate code even for programs that cannot be analyzed well by the compiler, such as programs with complex array subscripts. It is not acceptable for a compiler to reject a program simply because it is too complex! Just as vector compilers generate scalar code as a fallback position, our compiler must be able to generate some code that is guaranteed to run correctly, if not at high speed, for any program. It was not immediately obvious that this would be possible. In the absence of a single address space, it is crucial for both the process that needs a data value and the process that owns that data value to know about each other so that communication can take place. This information may not be available at compile-time in programs that are hard to analyze. A code generation strategy called *run-time resolution* is the fallback position for our system. It is a simple but fairly inefficient strategy that is guaranteed to work for any program, no matter how complex.

The code produced by run-time resolution can be improved by *partial evaluation* at compile-time — the resulting code generation strategy is called *compile-time resolution.* With run-time resolution to fall back to, we can improve on the quality of code for programs whose communication-computation patterns can be analyzed by the compiler. Three such patterns that programmers use frequently in writing code are do-across style parallelism, global accumulation, and scatter-gather.

- Using *do-across* style parallelism in the presence of a data distribution consists of laying out the iterations of a loop based on the placement of data and using data synchronization to satisfy data dependences across processors.
- *Accumulation* (or *global combine*) is used to apply a commutative and associative operator (such as sum) to the elements of a matrix. Each process examines its local data to compute the local contribution and sends that value to a central location where the final value is

computed. The data distribution determines which process will examine a particular element of the matrix.

- *Scatter-gather* is a method that is used when the data dependences in a loop can only be determined at run-time. If the dependences arise from a matrix that is not updated in the loop, then the loop can be separated into two pieces. In the first piece, each process determines which elements it will need and sends requests for those elements to the processes where they reside. In addition, each process services requests for data from other processes. When all the data has been transferred, the processes synchronize and perform the second piece of the loop which does the computation.

A compiler must be able to exploit all three of these parallelization techniques to produce good code for a distributed memory machine.

Programmers carefully hand optimize their programs to reduce the cost of messages and to increase parallelism. The two code generation methods mentioned above, run-time resolution and compile-time resolution, do not handle messages nearly as well. We have incorporated two message optimizations, *pipelining* and *vectorization of messages*, into the compiler to improve the efficiency of the code we produce.

The first optimization concerns the placement of sends and receives in the generated code. In general, sends should be performed as early as possible to reduce waiting time for the receiving process. Using compile-time resolution, the send and receive commands for a non-local reference are inserted into the generated code at the point in the original program where the value is needed. This is a good place for the receive but not the best place for the send. The receiving process can be delayed unnecessarily if the sending process has work to do between the time that the value to be sent becomes available and the time that compile-time resolution dictates the send should be executed. A better scheme would execute the send as soon as the value to be sent is available. We call the optimization that inserts a send at the earliest possible point in the generated code *pipelining*.

The second optimization reduces overhead by combining multiple messages with similar sources and destinations into one message. Each message sent incurs a fixed overhead plus a length related cost. In most message-passing systems the fixed overhead dominates, making it sensible to try to pack messages together. The compiler optimization to accomplish this is called *vectorization of messages* and may be thought of as the vectorization of a read operation. It is performed before code generation. Just as with any potentially vectorizable operation, the operands (in this case, the referenced array element) must be checked to ensure that there is no cycle of data dependences within the loop. If no such cycle exists, the read may be converted to a vector read which in turn will be converted during code generation to block sends and block receives.

Currently, our compiler ignores the assignment of processes to processors: In fact, load balancing may cause a process to migrate among many different processors during its lifetime. This approach runs counter to more traditional approaches to programming distributed memory machines that focus on mapping the topology of problems (rings, trees, etc.) to the topology of machines (hypercube, shuffle-exchange etc.) to exploit nearest-neighbor communication[3, 23]. Exploiting this kind of locality is important for architectures with *multi-level* memory hierarchies like the Intel iPSC/1, but it is not important to our two-level hierarchy. While our system can be extended to include this kind of topological information, we believe that the performance advantages of dynamic load balancing outweigh any advantage gained from a static assignment of processes to processors.

1.2 Outlineof paper

The rest of this paper is organized as follows. In Section 2, we introduce the programming language and machine model. 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 Section 3, we discuss the simple but inefficient code generation algorithm called *run-time resolution*. Section 4 shows how compile-time resolution uses *partial evaluation* (evaluation at compile-time) to improve considerably the code generated by the run-time resolution algorithm. We also remark on some connections between our techniques and standard code generation strategies for languages with overloaded operators. An experiment that illustrates the need for our message optimizations is discussed in Section 5. To reduce the overhead of message-passing, it is preferable to combine messages, thereby reducing message traffic. However, this must be done judiciously since combining messages can have an adverse effect on parallelism. The placement of messages can also affect parallelism. The details of our vectorization of messages algorithm are presented in Section 7 and the details of our pipelining algorithm are presented in Section **??**. These optimizations attempt to strike a balance between reducing message overhead and increasing parallelism. An important optimization is the recognition and exploitation of reduction operations. Section 9 discusses how this is done in our compiler. Section 10 presents the results of using the compiler to parallelize SIMPLE[7], a large heat conduction/hydrodynamics benchmark from Los Alamos. We discuss related work in Section 11. Section 12 is a discussion of the scope and limitations of our approach and opportunities for further work.

2 Language and Machine Model

The example programs in this paper are written in a language similar to Id[17]. The programs differ from Id in two major ways: data distributions must be specified for most variables and the parallel programs include communication statements. This rest of this section describes the allowed data distributions and the machine model assumed by the compiler.

2.1 Data Distributions

All program variables must be assigned a data distribution, also known as a mapping. Scalar variables may be distributed in four ways. A variable may be replicated (`A@ALL`), that is, each processor has its own copy of the variable. Loop index variables are assigned a replicated mapping automatically. To map a variable to a specific process, the programmer can supply an explicit name (`A@P1`), an expression (`A@(j mod S)`), or an indirect mapping using `Same_As`. An expression supplied as a mapping must take on a value from `0` to `S-1` where `S` is the number of processors. The mapping `Same_As` ties the mapping of a scalar variable to that of an array reference. `A@Same_As(B, (i, j))` indicates that the scalar variable `A` should be mapped to the same process as array element `B[i, j]`. The *owner* of a scalar variable is the process to which it is mapped. Every process is an *owner* of a replicated scalar variable.

The array distributions supported in our system are a compromise between generality and regularity. As we discuss in Section 3, we can support very general, irregular data distributions in the sense that our compiler can generate correct code regardless of how complex the program or the data distribution is. However, irregular data distributions are not amenable to compile time analysis and the performance of the generated code may be poor. A regular data distribution offers more opportunities for optimization. Two such distributions are wrapped rows/columns and blocks.

Given P processes, a wrapped row distribution of an array would assign the rows of the array to processors in a round-robin way – thus, Process 1 would get row 1, (P+1), (2P+1) etc. A wrapped column distribution is similar except that columns are distributed rather than rows. For a block distribution, the array is divided into P blocks and each process is assigned one block. Blocks need not have the same length in every dimension. For example, one possible block decomposition of a 100 by 100 array among 10 processes is to assign the first 10 rows to Process 1, the next 10 rows to Process 2 and so on.

The data distribution or mapping of array assigns each element of an array to a process. In addition to knowing where an element resides, the compiler must be able to determine the location of the element in the local memory of that process and must know how much space is required in each process to store the portion of the array assigned there. An array mapping consists of three functions that provide the necessary information:

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 its owner's local memory.

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

In this paper, we will use block and wrapped distributions. The required functions for blocks are: BM is the map function, BL is the local function, and BA is the allocation function. Col-map, col-local and col-alloc are the required functions for wrapped mappings. The *owner* of an array element is the process to which it is mapped.

2.2 Machine Model

The examples in later sections assume a very simple machine model. There are n processors in the model, each of which executes one process.[1] Each process has its own address space and all communication requires explicit message passing. The primitives for message passing are:

- **send**(Ai, Pi) - The value of Ai is sent by the process executing this command to process Pi. The sending process does not have to wait for the receiving process to get the item. The value of Ai will be broadcast to all other processes from the process executing this command when **all** is used in place of Pi.

  - The process executing this command waits until it receives a message from Process Pi. The received value is stored in Ai. While the process is waiting, messages it may receive from other processes are ignored temporarily.

We use the standard vector notation, A[x..y], in place of Ai above to mean send the values of A[x] through A[y] or receive values into A[x] through A[y].

On the iPSC/2, the cost of a message of length $k * 4$ bytes is defined by the following equation:

$$cost(k) = t_{startup} + k * t_{send}$$

. For messages under 100 bytes, $t_{startup} = 350\mu s$ and $t_{send} = 0.8\mu s$, and $t_{startup} = 660\mu s$ and $t_{send} = 0.8\mu s$ for longer messages[4]. Most of the cost in sending a message is software overhead

[1]Strictly speaking, the iPSC/2 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. Hereafter, we use the words process and processor interchangeably.

related to packing and unpacking the message. The length related cost is minimal. From this we can conclude that when possible sending one long message will be more efficient than sending many small messages. Long messages are broken into packets automatically by the underlying implementation. Packet reassembly is handled similarly.

3 Run-time Resolution

The difficulty of code generation arises from the need to generate code even for programs that cannot be analyzed well by the compiler, such as programs with complex array subscripts — after all, it is not acceptable for a compiler to reject a program simply because it is too complex! Just as vector compilers generate scalar code as a fallback position, our compiler must be able to generate some code that is guaranteed to run correctly, if not at blinding speed, for any program. When we started this research, it was not immediately obvious that we would be able to do this. In the absence of a single address space, it is crucial for both the process that needs a data value and the process that owns that data value to know about each other, so that communication can take place. This information may not be available at compile-time in programs that are hard to analyze. Once we have such a fallback position, we can improve on the quality of code for programs whose communication-computation patterns can be analyzed by the compiler. This section discusses *run-time resolution*, a simple but fairly inefficient code generation strategy guaranteed to work for any program, no matter how complex.

Run-time resolution produces one program that is executed by all processes. The following four principles plus two language restrictions guide the generation of this program.

Principle 1 Every process examines each statement to determine its role (if any) in the execution of the statement by using the three principles below.

Principle 2 The process that owns a variable or array element is responsible for computing its defining expression and recording its value.

Principle 3 The process that owns a variable or array element must communicate its value to any process that needs the value.

Principle 4 Conditionals, function calls, and loops are executed by all processes.

We restrict the use of mappings in two ways. The first restriction is that both sides of an array assignment must have the same mapping; an explicit copy is required to redistribute an array. The second restriction concerns function calls and is best explained in terms of scalar functions. Let function f return a single scalar value to Process P1. Everywhere a call to f appears, the process that needs f's return value (according to the principles stated above) must be Process P1. A simple transformation can convert a program that does not meet this restriction into one that does.

These principles dictate which processes will do the work associated with an expression or statement. A simple compilation strategy is to translate the program one statement at a time using these principles. We examine a simple example to illustrate run-time resolution. Consider the code fragment in Figure 1. Applying run-time resolution to this fragment yields the code in Figure 2. The translation of the first two statements requires a straightforward application of Principle 2; the process that owns the left-hand side of an assignment statement should evaluate the right-hand side and perform the assignment. The third statement requires a more elaborate translation. Following Principle 4, the conditional statement is executed by all processes. Therefore, the value of `b` needed to evaluate the predicate must be broadcast from b's owner (P2) to every other process. To translate the statement in the then branch we use principles 2 and 3. According

```
a:P1, b:P2, c:P3

a = 5;
b = 7;
if (b > 10)
   c = 2*a;
else c = a/2;
```

Figure 1: Simple example

to Principle 2 both the assignment and multiplication should be performed by the owner of c (P3). Principle 3 is used to translate the reference to a; Process P3 needs to obtain the value of a from its owner (P1). The translation of the else branch is similar.

For future reference, we will define the *evaluator set* of an expression to be the set of processes that execute the expression. In our example, the evaluator set of the predicate of the conditional is the set P1,P2,P3 while the evaluator set of the expression 2*a is P3. If we think of assignment as an operator, we can extend the concept to assignment statements in the obvious way — the evaluator set of an assignment statement is the set of processes that perform the assignment. In run-time resolution, only two kinds of evaluator sets are possible — a singleton process and the set of all processes.

3.1 Discussion

The key feature of run-time resolution is that the owner of a variable or array element does not have to be known to the compiler since this information is determined dynamically. This makes it possible to generate code even in the presence of complicated array subscripts and complex mappings of arrays to processors. Note that run-time resolution code does not send any more messages than code generated using perfect knowledge of ownership of all variables and array elements. This is a useful fact to keep in mind when comparing run-time resolution to other schemes such as compile-time resolution which we discuss in the next section. Our basic code generation strategy can be changed in many ways. For example, predicates in conditionals in the source code can be translated differently — it is possible to make one process compute the predicate and broadcast the value to all other processes. More importantly, by keeping track of which values have been sent to which processes, the translator can avoid message transfers if the value needed by a process has been sent to it earlier in the program. Run-time resolution code has a lot of conditional tests in it, and many of these tests involve the same predicate. 'Conditional jamming' of conditionals with the same tests can improve the code considerably.

4 Compile-time Resolution

Given the mappings of variables and arrays, the code generated by run-time resolution can be improved vastly for many programs since many of the ownership tests can be done at compile-time. A useful analogy here is the resolution of overloading in languages like Lisp. When compiling languages like Lisp without explicit type declarations, an overloaded operator such as + is compiled

```
P1, P2, P3:

if (my_id == owner(a)) a = 5;
if (my_id == owner(b)) b = 7;
/* Broadcast the value of b */
if (my_id == (owner(b)) {
  send(b, ALL);
  t1 = b;}
else
  receive(t1, owner(b));

if (t1 > 10) {
  /* coerce a */
  if (my_id == owner(a) and my_id == owner(c)
  t2 = a;
  elseif (my_id == owner(a))
        send(a, owner(c));
  elseif (my_id == owner(c))
        receive(t2,owner(a));}

  /* perform computation */
  if (my_id == owner(c))
    c = 2 * t2;
  }
else {
  /* coerce a */
  if (my_id == owner(a) and my_id == owner(c)
  t2 = a;
  elseif (my_id == owner(a))
        send(a, owner(c));
  elseif (my_id == owner(c))
        receive(t2,owner(a));}

  /*perform computation */
  if (my_id == owner(c))
    c = t3/2;
  }
```

Figure 2: Run-time resolution code for the simple example

P1:

```
a = 5
receive(t1, P2);
if (t1 > 10)
  send(a, P3);
else
  send(a, P3);
```

P2:

```
b = 7;
send(b, P1);
send(b, P3);
```

P3:

```
receive(t1, P2);
if (t1 > 10) {
  receive(t2, P1);
  c = 2 * t2; }
else {
  receive(t3, P1);
  c = t3/2; }
```

Figure 3: Compile-time resolution code for the simple example

into a case statement that tests the type of the arguments and calls the appropriate type specific addition routine. The naive code generated by this general 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 call to the relevant addition routine. This technique of code improvement through partial evaluation of generic code can be applied profitably in our context as well. The code generated by run-time resolution is like generic code that can be specialized to each process using data distribution information. For example, note that partial evaluation of the code in Figure 2 can eliminate many of the tests that determine where work should be done.

Compile-time resolution is essentially partial evaluation of the run-time resolution code, coupled with a more sophisticated handling of control structures such as conditionals and loops. To understand the treatment of control structures, consider a conditional expression. Under run-time resolution, all processes must evaluate the predicate, and partial evaluation cannot reduce this set of processes. This is unfortunate since only those processes that might have work to do in one of the branches need to evaluate the predicate. To accomplish this, compile-time resolution takes a more sophisticated view of conditionals and loops. Like run-time resolution, compile-time resolution requires the owner of a variable to compute its value and communicate it to any other processes that needs it. To determine who should evaluate the predicate of a conditional, the compiler first analyzes the two branches of the conditional, and generates code in which only processes whose participation is required to evaluate the branches perform the test. The compile-time resolution of the simple example in Figure 1 is shown in Figure 3. Note that process P2 does not evaluate the test in the conditional statement.

Like conditionals, loops must be given special treatment. Run-time resolution code requires all processes to execute all iterations, but it is better that a process should execute only those iterations in which it has work to do. A subtle point is that the work assigned to a given process during the execution of a loop can vary from iteration to iteration. For example, let `A` be a wrapped vector of size 16 in the following code fragment:

```
for i = 2 to 16 do
    A[i] = 0.5 * A[i-1];
```

```
for i′= 2 to 13 by 4 do {
    /* iteration 2, 6, 10 — (i − 1) mod 4 = 1 */
    i = i′;
    t1 = A[Col-local(i-1)];
    send(t1,P2);

    /* Process 1 has no work to do in iterations, 3,4,7,8,11,12 */

    /* iteration 5, 9, 13 — i mod 4 = 1 */
    i = i′+ 3;
    receive(t2, P0);
    A[Col-local(i)] = 0.5 * t2;
}
/* iteration 14 — (i − 1) mod 4 = 1 */
i = 14;
t1 = A[Col-local(i-1)];
send(t1, P2);

/* Process 1 has no work to do in iterations 15, 16 */
```

Suppose there are 4 processors. Consider Process 1's role in the evaluation of this loop. In any particular iteration, the work assigned to Process 1 will be one of the following: a receive followed by some computation and an assignment or a send or nothing. Our goal is to generate code that includes exactly the work assigned to a process and no more. The following code fragment satisfies this goal:

To review, two issues must be resolved to generate code for loops: how to determine which iterations a process must participate in and how to determine the work assigned to a process in an iteration. Recall that the translation of a conditional expression depends on the processes required to evaluate the branches. The translation of a loop also depends on the set of processes required to evaluate its constituent statements. Returning to our example, the set of processes that must participate in the ith iteration (that is, the evaluator set for that iteration) is $\{i \, mod \, 4, (i-1) \, mod \, 4\}$ since $i \, mod \, 4$ is the owner of `A[i]` and $(i-1) \, mod \, 4$ is the owner of `A[i-1]`. Let e_1 denote expression $\{i \, mod \, 4\}$ and e_2 denote expression $\{(i-1) \, mod \, 4\}$; note that i is free in both of these expressions. To determine which iterations Process 1 must participate in, we must determine those iterations i in which $i \, mod \, 4 = 1 \, \text{or} \, (i-1) \, mod \, 4 = 1 \, \text{and} \, 2 \leq i \leq 16$. Each of these equations can be solved to find an arithmetic sequence which can be represented as a starting point, ending point and step size; in our example, we get the sequences (5,13,4) and (2,14,4). To generate code for the loop, we find the least common multiple (lcm) of the step sizes and unroll the loop that many times; in our example, we would unroll the loop 4 times. Since the number of iterations to be done in the loop is not necessarily a multiple of the lcm of the step sizes, we can have a number of trailing iterations to be done outside the loop. For our example, the unrolled loop would start at $i = 2$ and end at $i = 13$, and the iterations $i = 14, 15, 16$ would be done outside the unrolled loop. In the unrolled code, every reference will owned by just one process (or by all processes) and we can generate code just as for acylic code.

In our implementation, we generate the final code without actually unrolling the loop and then specializing it. To permit this, we keep information about sequences and the role of each process in the evaluation of the loop. This information contains the starting point (α), ending point (β) and period (step size) (γ) of the unrolled loop; in our example, these are 2, 13 and 4 respectively. In addition, for each process P, we keep a *pattern* which is a list of pairs $< j, \{...e_k...\}$ where $0 \leq j < \gamma$ and $e_k[i \leftarrow j] = \mathrm{P}$; intuitively, this list of pairs tells us the references in the unrolled loop which are owned by process P; for our example, the list of pairs for process 1 is $\{< 0, \{e_2\} >, < 1, \{\} >, < 2, \{\} >, < 3, \{e_1\} >\}$, showing that in the unrolled loop, process 1 has work to do in the first and fourth copies of the original loop. By keeping this summary information, we generate the right code without actually unrolling the loop and then specializing the body of the unrolled loop. The reader who is interested in the details is referred to [20].

Unlike wrapped mappings, block mappings yield loops in which the step size is 1. An expression (e_1) will equal a process number, P, for some range of integer values (x..y). To generate code given a set of such expressions, we need to determine where these ranges overlap. The result will be a set of loops with period 1. For example, given a set $\{e_1, e_2\}$ where $e_1 = P$ from 1..10 and $e_2 = P$ from 5..16, the code will contain three loops: one loop from 1 to 4 with period 1, a second from 5 to 10 with period 1 and a third from 11 to 16 with period 1. A simple line sweep algorithm is used to compute the beginning and ending of each new loop.

4.1 Discussion

The code produced by compile-time resolution is a vast improvement over that produced by run-time resolution. Many of the conditional tests of ownership in run-time resolution code are removed by making use of data mappings. In addition, compile-time resolution treats control structures in a more sophisticated way - conditionals and loop iterations are executed only by those processes that have work to do in the control structure. In the next section, we examine the timing results for an example compiled using run-time resolution and compile-time resolution. This simple experiment points out several inefficiencies in the code generated by compile-time resolution. We discuss several message optimizations needed to improve the efficiency of the code generated by our compiler.

5 Message Optimizations

Even though the code produced by compile-time resolution is much better than the code produced by run-time resolution, the handling of messages leaves something to be desired. The program in Figure 4 provides a good basis for discussion. The compile-time resolution code for a four processor system assuming that `N=128` is shown in Figure 5.[2] We first performed experiments to determine how the code generated using run-time and compile-time resolution compares with handwritten code. The run-time resolution programs for a 128 by 128 integer grid took 24 seconds no matter how many processors were involved. Similarly, the compile-time resolution programs took 15 seconds no matter how many processors were used. The lowest curve in Figure 6 shows the performance of the handwritten code. 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] The absence of speedup arises from the fact that there is

[2] Recall that given the index expressions for an array reference `Col-local` computes the local offset for the referenced array element. `Col-alloc` is the allocation function for the column mapping. It will allocate enough space for a process's portion of the array.

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

no parallelism being exploited in this program — all the processes go through all the statements in the program. The compile-time implementation is more encouraging (as we expected) but it is still poor compared to the handwritten program. There are two reasons for this. First, it exchanges as many messages as the run-time resolution program, many more than the handwritten code. Second, and more surprisingly, this program does not exploit any of the parallelism in the problem, since execution time does not decrease as the number of processors increases!

To understand these two issues, it is useful to contrast the treatment of messages in the compile-time resolution code and a handwritten implementation (see Appendix A). In the compile-time resolution code, the values in the `Old` column are sent one at a time. The handwritten implementation attempts to reduce the number of messages by combining messages that have the same source and destination processors — for example, values in the `Old` column are sent through a single *send* command. This is useful because of the relatively high startup cost of messages on the iPSC/2. To reduce the number of messages, we need to use block sends and receives wherever possible. The topmost curve in Figure 6 shows the improvement that results from combining all the values of the `Old` column into a single message. The compiler optimization to accomplish this is performed before compile-time resolution and may be thought of as *vectorization* of a read operation. Just as with any potentially vectorizable operation, the operands (in this case, the referenced array element) must be checked to ensure that there is no cycle of data dependences within the loop. Intuitively, if no such cycle of dependences exists, the read may be converted to a vector read. This in turn will be converted during code generation to block sends and receives.

Even after this optimization is performed, there is no speed-up when the number of processors is increased. This came as a surprise initially, but closer examination of the code generated by compile-time resolution revealed the reason for this. In the code generated by compile-time resolution, the owner of a variable computes its value but does not immediately send it to other processes that may need it — rather, the process computes until the point in the original computation where the value is used and then sends it. In our program, this results in each new column being computed in its entirety by a process before that column is transmitted to other processes. Only one process is active at a time! To remedy this, computation of the elements of the column and communication of these values should be overlapped. Not surprisingly, the most impressive gains in our experiments were obtained by pipelining computation and communication.

Pipelining the Gauss-Seidel program results in a program in which a value is sent as soon as it is computed. Note that this increases the number of messages over a scheme in which the entire column is sent over in one message. Thus, pipelining of communication and computation has to traded off against reducing the number of messages sent. The handwritten implementation achieves a compromise 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. By 'blocking' these values, we obtain the curve of Figure 6 which has the best performance — the block size is a compromise between decreasing the number of messages and exploiting parallelism. The optimal block size is a combination of the machine dependent cost of message overhead and the amount of computation between sends. We have not implemented this optimization in our compiler.

It is reasonable to wonder if the optimizations in this section are required only because of idiosyncrasies of the iPSC/2. The answer to this question is an emphatic no. Let us first consider other message-passing machines. Pipelining of computation and communication is necessary to exploit parallelism on any machine — this has nothing to do with the overhead of message passing. Vectorization and blocking are required because the overhead associated with a long message is less than the overhead of many small messages. This is likely to remain true for message-passing machines.

In the context of shared memory machines with two level memory hierarchies, these optimiza-

```
Def GS-iteration(Old@column)@column
{ New@column = array(N,N)

  for i = 2 to N-1 do
      New[i, 1] = 0.5;

  for j = 2 to N-1 do {
      for i = 2 to N-1 do {
          New[i, j] = 0.5 * (New[i-1, j] + New[i, j-1] + Old[i+1, j] + Old[i, j+1]);
      }
  }

  return New }
```

Figure 4: Sequential version of Gauss-Seidel iteration

tions become a matter of chosing the appropriate granularity for synchronization between processes. In any situation in which one process is a producer of a sequence of values and another is a consumer of these values, there is need for synchronization. The granularity of this synchronization affects parallelism and the cost of synchronization. If the processes synchronize on every value, parallelism may be enhanced but the cost of synchronization goes up and vice versa.

Vectorization of messages and pipelining communication with computation are both essential for generating efficient code. We discuss these optimizations in more detail in following sections.

6 New Intermediate Code for Optimizations

Before discussing the vectorization of messages and pipelining algorithms, we introduce several new intermediate language constructs. The motivation for these constructs will become clear as we discuss the details of the message optimizations. In the following, let process Q be the owner of variable x.

- make-send(x, <t, readers>) – send the value of x from Q to all processes, P, such that P ϵ readers and P $\neq$ Q.
- make-receive(x, <t, readers>) – receive the value of x from its owner, Q, on process P, if P ϵ readers and P $\neq$ Q.
- local(x, <t, readers>) – for all processes, P ϵ readers, read the local value of x. If P = Q, the local value will be x. If P $\neq$ Q, the local value will be stored in the temporary variable t.

Any scalar variable or array reference can be replaced by a combination of make-send, make-receive, and local constructs. The readers set is determined by the processes that need the value of the referenced scalar variable or array element (in run-time resolution parlance, readers is just

```
Def GS-iteration(Old)
{ New = col-alloc(128, 128);

  /* Process 3 has no work to do in the initialization of Column 1 */
  for j′= 2 to 125 by 4 do {
      /* (j+1) mod 4 = 3 */
      j = j′;
      for i = 2 to 127 do
          send(Old[col-local(i, j+1)], j mod 4);

      /* j mod 4 = 3 */
      j = j′+ 1;
      for i = 2 to 127 do {
          t1 = New[col-local(i-1, j)];
          receive(t2, (j-1) mod 4);
          t3 = Old[col-local(i+1, j)];
          receive(t4, (j+1) mod 4);
          New[col-local(i, j)] = c*(t1 + t2 + t3 + t4);
      }

      /* (j-1) mod 4 = 3 */
      j = j′+ 2;
      for i = 1 to 127 do
          send(New[col-local(i, j-1)], j mod 4);
  }

  /* (j+1) mod 4 = 3 */
  j = 126;
  for i = 2 to 127 do
      send(Old[col-local(i, j+1)], j mod 4);

  /* j mod 4 = 3 */
  j = 127;
  for i = 2 to 127 do {
      t1 = New[col-local(i-1, j)];
      receive(t2, (j-1) mod 4);
      t3 = Old[col-local(i+1, j)];
      receive(t4, (j+1) mod 4);
      New[col-local(i, j)] = c*(t1 + t2 + t3 + t4);
  }


  return New; }
```

Figure 5: Code generated for Process 3 using compile-time resolution

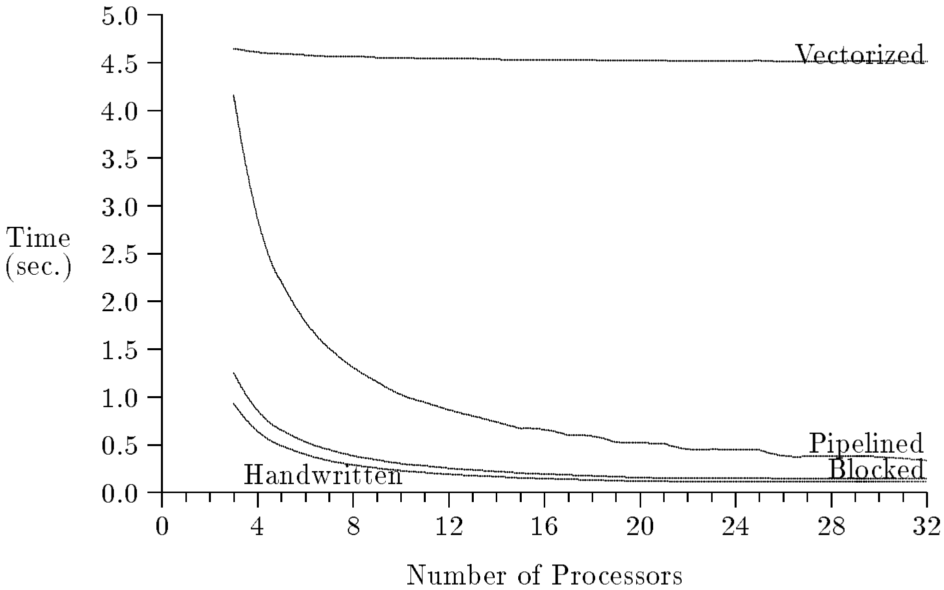

Figure 6: Effect of message-passing optimizations

the evaluators set of the reference). Figure 7 displays the Gauss-Seidel example expressed using make-send, make-receive, and local.

7 Vectorization of Messages

Vectorization of messages combines messages that arise from a single array reference within a loop. This optimization may be thought of as the vectorization of a read operation. Just as with any potentially vectorizable operation, the operands (in this case, the referenced array element) must be checked to ensure that there is no cycle of data dependences within the loop. Intuitively, if no such cycle of dependences exists, the reference may be converted into a set of local references, vector receive statements, and vector send statements during code generation.

Vectorization of messages has an extra twist introduced by the need to guarantee that the code generated for the sender matches the code generated for the receiver. If a group of sends is converted into a vector send in the code for the sending process, the corresponding receives must be converted into a vector receive in the code for the receiving process to prevent deadlock. Compile-time resolution installs the communication statements for a non-local reference at the site of the reference. If a send to process p occurs in iteration i on process q, then a receive from process q will occur in iteration i on process p. Therefore, a group of sends that occur in contiguous iterations in the original loop can be vectorized because the corresponding receives are guaranteed to occur in those same iterations in the receiving process. Unfortunately, sends from non-contiguous iterations cannot be vectorized because the same guarantee does not hold, the receiver may have receives from other processes interspersed. This restriction can be explained in terms of the translation of loops. If the period of the inverse of the set $\{\text{owner}\} \cup$ readers is 1, then the communication arising from this reference will occur in contiguous iterations.

We present an algorithm that vectorizes messages from a restricted class of array references. An array reference is *vectorizable* if

1. The referenced array is read-only in this function,

```
Def GS-iteration(Old@column) @column
{ New@column = array(128,128)

  for i = 2 to 128-1 do
      New[i, 1] = c;

  for j = 2 to 128-1 do {
      for i = 2 to 128-1 do {
          make-send(New[i-1, j], <t1, {j mod 4}>);
          make-receive(New[i-1, j], <t1, {j mod 4}>);

          make-send(New[i, j-1], <t2, {j mod 4}>);
          make-receive (New[i, j-1], <t2, {j mod 4}>);

          make-send(Old[i+1, j], <t3, {j mod 4}>);
          make-receive(Old[i+1, j], <t3, {j mod 4}>);

          make-send(Old[i, j+1], <t4, {j mod 4}>);
          make-receive(Old[i, j+1], <t4, {j mod 4}>);

          New[i, j] = c * ( local(New[i-1, j], <t1, {j mod 4}>)
                          + local(New[i, j-1], <t2, {j mod 4}>)
                          + local(Old[i+1, j], <t3, {j mod 4}>);
                          + local(Old[i, j+1], <t4, {j mod 4}>);
      }
  }

  return New }
```

Figure 7: Gauss-Seidel expressed using new intermediate language constructs

2. the reference is not in the scope of a conditional,

3. and any communication that arises from this reference occurs in a contiguous set of iterations from the original loop, that is, the period of the inverse of the set {owner} $\cup$ readers is 1.

The rest of this section contains a discussion of the basic vectorization algorithm followed by a suggested extension and a summary.

7.1 Algorithm

The first step of our vectorization of messages algorithm is to determine which array references meet the definition of vectorizable. Once a vectorizable reference has been identified, we replace it with a local expression and introduce make-send and make-receive statements. Figure 7.1(a) contains result of replacing the vectorizable reference `Old[i, j+1]` with local, make-send, and make-receive in a much simplified version of the Gauss-Seidel example discussed earlier. Once we have introduced these new statements, we distribute the inner-most loop over the make-send and make-receive while leaving the local expression in the original context (see Figure 7.1(b)). Notice that we have used the vector form of make-send and make-receive. The translation the vector versions of make-send and make-receive uses ideas from the translation of their scalar equivalents and from the translation of loops. As we saw earlier, a scalar make-send is translated into a send on process P, only when Process P owns the value which is needed by another process Q, that is Q ϵ readers and Q $\neq$ P. To translate a vector make-send we need to determine which elements of the vector are owned by process P but needed by other processes. We achieve this by computing the inverse of the set {owner} $\cup$ readers which is just the evaluators set of the loop inherent in the vector form of the make-send. For our example, the set is $\{(j-1)\, mod\, 4, j\, mod\, 4\}$ and its inverse for Process 3 is $\alpha = 2$, $\beta = 127$, $\gamma = 1$ and pattern $= \{< 0, \{(j+1)\, mod\, 4\} >\}$. The translation follows directly from the pattern ($\{< j, \{\ldots e_k \ldots\} >\}$): if this process owns a group of the referenced array elements (owner $\epsilon \{\ldots e_k \ldots\}$) while some other process needs their values (readers $\cap$ $\{\ldots e_k \ldots\} = \emptyset$), then a vector send is needed. In all other cases, no code is required. The translation of make-receive is similar. A vector receive is needed when this process needs the values of some of the referenced array elements (readers $\cap$ $\{\ldots e_k \ldots\} \neq \emptyset$) but does not own them (owner $\not\epsilon$ $\{\ldots e_k \ldots\}$).

Figure 9 contains the result of translating the intermediate code from Figure 7.1(b). Notice that the make-send and make-receive have been translated into vector send and vector receive statements and that the local expression has been converted into a reference to the received vector. There are two additional things to notice about this example. First, all of the values in the vector send go to the same process. This is not always true. When a block mapping is used, the expression that describes the reader can take on different values. For example, given the statement `make-send(A[i-1, 1..7], <t, {BM(i-1, 1..7)}>)`, process 0 will need to send `A[4, 1..3]` to Process 2 and `A[4, 4]` to Process 3 from a single vector send. When this occurs, we generate multiple vector sends. One for each of the possible value of the reader expression. The same holds for vector receives. When different elements are owned by different processes, we generate one vector receive for each such process. Second, it would be more efficient to perform all the sends as soon as possible. To accomplish this we distribute the both loops over the make-send rather than just the innermost loop. Figure 10 contains the intermediate code that results from this additional optimization and Figure 11 contains the final result.

```
for j = 2 to 128-1 do {
    for i = 2 to 128-1 do {
        make-send(Old[i, j+1], <t, { j mod 4}>)
        make-receive(Old[i, j+1], <t, { j mod 4}>)
        New[i, j] = c * local(Old[i, j+1], <t, { j mod 4}>);
    }
}
```

(a) Intermediate code

```
for j = 2 to 128-1 do {
    make-send(Old[2..127, j+1], < T[2..127], { (j mod 4}>);
    make-receive(Old[2..127, j+1], <T[2..127], { j mod 4}>);
    for i = 2 to 128-1 do
        New[i, j] = c * local (Old[i, j+1], <T[i], {j mod 4}>);
}
```

(b) Intermediate code after loop distribution

Figure 8: Vectorization example

```
for j'= 2 to 122 by 4 do {
    /* (j+1) mod 4 = 3 */
    j = j';
    send(Old[2..127, j+1], j mod 4);

    /* j mod 4 = 3 */
    j = j'+ 1;
    receive(T[2..127], (j+1) mod 4);
    for i = 2 to 127 do
        New[col-local(i, j)] = c * T[i];

}
/* (j+1) mod 4 = 3 */
j = 126;
send(Old[2..127, j+1], j mod 4);

/* j mod 4 = 3 */
j = 127;
receive(T[2..127], (j+1) mod 4);
for i = 2 to 127 do
    New[col-local(i, j)] = c * T[i];
```

Figure 9: Generated Code for vectorization of messages example.

```
for j = 2 to 128-1 do
    make-send(Old[2..127, j+1], < T[2..127], { (j mod 4}>);
for j = 2 to 128-1 do {
    make-receive(Old[2..127, j+1], <T[2..127], { j mod 4}>);
    for i = 2 to 128-1 do
        New[i, j] = c * local (Old[i, j+1], <T[i], {j mod 4}>);
}
```

Figure 10: Intermediate code with extra loop distribution.

```
for j′= 2 to 122 by 4 do {
    /* (j+1) mod 4 = 3 */
    j = j′;
    send(Old[2..127, j+1], j mod 4);
for j′= 2 to 122 by 4 do {
    /* j mod 4 = 3 */
    j = j′+ 1;
    receive(T[2..127], (j+1) mod 4);
    for i = 2 to 127 do
        New[col-local(i, j)] = c * T[i];

}
/* (j+1) mod 4 = 3 */
j = 126;
send(Old[2..127, j+1], j mod 4);

/* j mod 4 = 3 */
j = 127;
receive(T[2..127], (j+1) mod 4);
for i = 2 to 127 do
    New[col-local(i, j)] = c * T[i];
```

Figure 11: Final code for vectorization of messages example.

```
Def test1 B@block(4, 4, 16, 16)
{ A@block(4, 4, 16, 16) = matrix(16, 16);
  For i=2 to 16 do
      For j=2 to 16 do
          A[i, j] = B[i-1, j] + B[i, j-1]; }
```

Figure 12: Example of degenerate reference

7.2 An Extension

Our algorithm, as described, cannot handle one commonly occurring class of references. The function in Figure 12 displays an example of this class of reference. The reference `B[i, j-1]` will generate only one send for the "j" loop. In a handwritten implementation the reference would be vectorized at the "i" loop level to compensate.

We can modify our algorithm to handle these kinds of references. Rather than distributing only the inner loop over the make-send and make-receive nodes, we distribute the two innermost loops over these nodes. When generating code, if the inner loop produces only one send, then we may be able to vectorize it at the next loop level out. We refer to a single send that is the sole result of translating an innermost loop as *degenerate*. Presently, we promote (vectorize at the next level out) only a very restricted class of degenerate references. Degenerate sends/receives can vectorized, if in the iteration following the iteration that produced the degenerate send, the process that owns the reference and the process that need it are the same. The reference to `B[i, j-1]` in Figure 12 is such a reference.

7.3 Summary

Our vectorization of messages algorithm handles messages from a restricted class of array references; namely, references to read-only arrays that appear inside loops but not inside conditional expressions. Our algorithm generates code for these references without introducing deadlock and without coordinating code generation across processes. A necessary criterion to determine if a reference is vectorizable is: does the communication that arises from this reference occur in contiguous sets of iterations from the original loop.

8 Pipelining

Pipelining is a transformation that overlaps communication and computation, thereby reducing the time a processor spends waiting for data to be sent to it. Both run-time and compile-time resolution introduce send and receive commands in places that correspond to non-local data references in the original program. There are other choices for the placement of these commands. In particular, the send can be moved up in the program so that it is performed as soon as the value to be sent has been computed. This can reduce the time the receiving processor spends waiting for the message; in the best case, message transmission is complete by the time it executes the receive command. An example of this is shown in Figure 13. Process P2 cannot progress until it receives the value of `a`. Unfortunately, compile-time resolution inserts the send command after the big computation of `c`. By 'bubbling' the send up in the program, we get the code in Figure 13(c) in which P2 does not have to wait for this computation to be complete. This is a simple example of pipelining. In

```
a @P1 = exp1;
b @P1 = exp2;
c @P1 = some-big-computation;
d @P2 = 2*b + a;
```

(a) Original code

```
P1:                                P2:
a = exp1;                          receive(t1, 1, P1)
b = exp2;                          receive(t2, 2, P1)
c = some-big-computation;          d = 2 * t1 + t2;
send(b, 1, P2);
send(a, 2, P2);
```

(b) Compile-time resolution code

```
P1:                                P2:
a = exp1;                          receive(t1, 1, P1);
send(a, 2, P2);                    receive(t2, 2, P1)
b = exp2;                          d = 2*t1 + t2;
send(b, 1, P2);
c = some-big-computation;
```

(c) Pipelined code

Figure 13: Simple example

Section 8.1, we study the pipelining of scalar computations. In Section 8.2, we give an algorithm for the more complicated case of pipelining of array values and in Section 8.3 we discuss two extensions to this algorithm. Finally, in Section 8.4, we discuss similar transformations that have been studied in other contexts.

8.1 Pipelining of Scalar References

Pipelining of scalar values in acyclic code is more or less a trivial application of *def-use* chains[4]. The compile-time resolution algorithm can be modified so that when it encounters a non-local use of a variable x, it inserts a receive command at the use, and then traverses use-def chains to insert send's immediately after every assignment to x that reaches this use. Since we are using a

[4]An assignment to a variable is said to *reach* a point in a program if execution of the assignment can be followed by control reaching that point without the execution of any intermediate assignments to that variable. Def-use chains are a standard compiler data structure that associate each assignment to a variable with uses of the variable that it reaches.

```
a @P1 = f(1);
b @P2 = f(2);
c @P2 = if (b > 10) then b else a;
```

(a) Original code

```
P1:                          P2:
a = f(1);                    b = f(2);
send(a, 1, P2);              c = if (b > 10)
                                   then { receive(t1, 1, P1);
                                          in b }
                                   else { receive(t1, 1, P1);
                                          in t1 }
```

(b) Pipelined Code

Figure 14: Second example

functional language, there will be only one assignment to a variable x. This algorithm can be used in an imperative context by representing the program in Static Single Assignment form[?] and inserting the send after the last ϕ function for x that occurs before the use. This works in simple cases, such as Figure 13, in which there are no conditionals between these assignments and the use.

Before we consider conditionals, notice that the two send commands in Figure 13 have been reordered in the pipelined code. We introduce mailboxes to allow sends to be reordered. Every process has a set of numbered mailboxes. Messages are tagged with a mailbox number as well as the names of the sending and receiving processes. We modify both the send and receive commands to include a mailbox number. Even though the messages may arrive out of order, they will be received in order because a receive command selects a message from the buffer of waiting messages using both the sender and mailbox number specified in the command. In our implementation, each textual occurrence of a non-local reference in the program is assigned a unique mailbox number, so sends can be reordered without altering the result of the computation.

The picture is more complex when conditionals are taken into account. In Figure 14(a), Process P2 uses the value of `a` on only one branch of the conditional. To maintain correctness, we must guarantee the *message consumption property* — every message that is sent must also received. If we insert a send immediately after the computation of `a`, we must install a dummy receive of `a` in the then branch of the conditional (see Figure 14(b)). Alternatively, we can evaluate the predicate on process P1 as well, and send the message only if the predicate is false. Clearly, there is a trade-off here between doing extra computation and sending unnecessary messages.

```
For i = 1 to n do
     A[i] = ...
...
For j = 1 to m do
     ... = ...A[2j] ...
```

Figure 15: A Simple Example of Loop Pipelining

8.2 Pipelining of Array Elements

Array element pipelining is like constructing a stream of values that connects the producer of an array to its consumer. Consider the program shown in Figure 15. The first loop produces the elements of an array A and the second loop consumes every second element of this array. In the code produced by compile-time resolution, the whole array A is produced before it is consumed. To pipeline the production and consumption of array elements, we replace the reference `A[2j]` with a receive, and we send values of A whenever they are produced by the first loop. Note that not all values produced by the first loop are required by the second loop. It is easy to verify that a send must take place in the i^{th} iteration of the producer loop if i/2 is an integer between 1 and m. Therefore, in each iteration of the producer, we must evaluate this condition, and perform a send only if the condition is satisfied.

In the example just cited, the producer generated values in the order required by the consumer. Allowing different production and consumption orders is possible but problematic. We solved the out-of-order send problem in the scalar case by using mailbox numbers to identify values. The same solution could be used here by assigning a different mailbox number to each array element. Unfortunately, these numbers can become quite large. To avoid this problem we restrict our attention to references that consume values in the same order as they are produced.

We place the following restrictions on the producer and consumer loops:

1. All assignments to the array must be local to the procedure in which the reference occurs. We do not perform inter-procedural program analysis. Therefore, we will not, for example, pipeline array references if the array is passed to another procedure where some of its elements are defined.

2. Array elements must be produced in the same order in which they are consumed, since pipelining requires tight 'coupling' between producers and consumers, the only slack being that some elements may be produced but not sent to the consumer. For example, we will not perform pipelining if the array elements are produced in the order 1 to 100, and consumed in the order 100 to 1. However, we do not require that there be exactly one producer for an array; for example, we can have one loop that produces the first column of an array, and another loop that produces the rest of the array.

The first condition is easy to check using standard dataflow analysis. The second condition is somewhat more involved, and we discuss it next. To keep matters simple, let us assume for now that there are no conditionals surrounding either the producers or the consumer, and that we have nested loops surrounding the producers and the consumer, as in Figure 16.

$$
\begin{array}{l}
\text{For } i_1 = 1 \text{ to } n_1 \text{ do} \\
\quad \text{For } i_2 = 1 \text{ to } n_2 \text{ do} \\
\quad\quad \ldots \\
\quad\quad \text{For } i_k = 1 \text{ to } n_k \text{ do} \\
\quad\quad\quad A[f_1(i_1, \ldots, i_k), \ldots, f_t(i_1, \ldots, i_k)] = \ldots \qquad (1) \\
\text{For } j_1 = 1 \text{ to } m_1 \text{ do} \\
\quad \text{For } j_2 = 1 \text{ to } m_2 \text{ do} \\
\quad\quad \ldots \\
\quad\quad \text{For } j_l = 1 \text{ to } m_l \text{ do} \\
\quad\quad\quad A[g_1(j_1, \ldots, j_l), \ldots, g_t(j_1, \ldots, j_l)] = \qquad (2) \\
\quad\quad\quad\quad\quad \ldots A[h_1(j_1, \ldots, j_l), \ldots, h_t(j_1, \ldots, j_l)] \ldots \qquad (3)
\end{array}
$$

Figure 16: Simple template

The second condition can be stated as follows. To avoid out-of-order sends, we restrict the set of references that can be pipelined to those that meet a simple monotonicity condition. Using the standard ordering on tuples the criteria are:

1. The elements produced by each definition are in monotonically increasing order,
2. the elements referenced by the use are in monotonically increasing order,
3. and the array elements defined by the outside definition (1) occur earlier in the element ordering than those elements defined by the inside definition (2).

Note that we could also pipeline if the conditions were reversed, that is, the elements are produced and consumed in monotonically decreasing order. These conditions are easy to check when loop bounds are constant, which is the common case.

When a value is computed at run-time in the definition loop, we must determine whether this value is needed at the use site, and if so, which process needs it. To do this, we express the use loop variables in terms of the definition loop variables. Using these expressions, we can compute the value of the use loop variables at run-time. If the computed values are integers that fall within the correct ranges, then there is a dependence between the definition and the use and the value must be sent. Once we have expressions for the use loop variables in terms of the definition loop variables, we can determine where the value needs to be sent by computing the evaluator set at the definition site.[5] To accomplish all this, we proceed as follows. Note that the producer must send a value if the following system of equations has a solution

$$
\begin{aligned}
h_1(j_1, \ldots, j_l) &= f_1(i_1, \ldots, i_k) \\
&\vdots \\
h_d(j_1, \ldots, j_l) &= f_d(i_1, \ldots, i_k)
\end{aligned}
$$

[5] We assume that the evaluator set for the original reference contains expressions written in terms of the use loop variables.

and loop variables are in the proper range.

If the functions $f_1, \ldots, f_d$ and $h_1, \ldots, h_d$ compute linear combinations of the loop variables, $i_1, \ldots, i_k$ and $j_1, \ldots, j_l$ respectively, then we have a system of linear equations that relates the use loop variables to the definition loop variables. This system has the form:

$$\begin{bmatrix} u_1 & v_1 & \ldots & w_1 \\ u_2 & v_2 & \ldots & w_2 \\ & & \vdots & \\ u_d & v_d & \ldots & w_d \end{bmatrix} \begin{bmatrix} j_1 \\ j_2 \\ \vdots \\ j_l \end{bmatrix} = \begin{bmatrix} f_1(i_1, \ldots, i_k) - x_1 \\ f_2(i_1, \ldots, i_k) - x_2 \\ \vdots \\ f_d(i_1, \ldots, i_k) - x_d \end{bmatrix}$$

Note that the entries in the right-hand vector are expressions written in terms of the definition loop variables. Therefore, if there is a unique solution to this system of equalities, it will consist of a vector of linear combinations of the definition loop variables.

Having addressed the problem of expressing the use loop variables in terms of the definition loop variables, we can now state the final condition for pipelineablity of array elements. A array reference is *pipelineable* if for every definition that reaches it, the associated system of equations has a unique solution. Once all conditions are met, a reference can be pipelined by inserting a guarded send at each definition site and the receive at the use. The guard computes the use loop variables and checks that there is a run-time dependence between the definition and the use. Figure 17 contains the pipelined intermediate code for a simplified version of the Gauss-Seidel relaxation example discussed earlier. Note that the primed variables contain the values of the loop variables at the use. 18 display the final code for this example. Notice that the send and receive specify a mailbox number in addition to the name of the sending or receiving process.

8.3 Extensions

The previous discussion handled the most straight-forward situation. What kind of complications can arise? We will discuss two: a conditional surrounding the use and extra loops surrounding both loop nests. The solution suggested for conditionals in the scalar case is also valid here. We can either move the birth point of the definition into the innermost conditional branch or we can insert dummy receives into the other branches.

The extra loop case (see Figure 19) is not substantially different. The pipelining algorithm proceeds as described earlier except that an underdetermined system of equations no longer automatically disqualifies a potentially pipelineable read. If the index variable from the extra loop does not appear in the index expressions of either definition or the array reference and if removing it from the system of equations yields a system that has a unique solution, then the array reference can be pipelined.

8.4 Discussion

Pipelining is similar to reorganization of instructions for pipelined machines. Just as in that case, resource requirements may go up after reorganization. We must be careful not to deadlock due to overcommitment of buffer space. We have not faced the problem so far since we have run medium sized problems, but it may become an issue for large programs.

9 Accumulation

Programmers use a programming paradigm known as *accumulation* or *global combine* to implement reductions. Accumulation is used to apply a commutative and associative operator (such as sum)

```
Def GS-iteration(Old@column) @column
{ New@column = array(128,128)

  for i = 2 to 127 do {
      New[i, 1] = 0.5;
      i′= i + 1;
      j′= 1;
      if (is-int(i′) && is-int(j′) && (2 ≤ i′≤ 127) && (2 ≤ j′≤ 127))
        make-send(New[i′-1, j′], <t1, {j′mod 4}>);

      i′= i;
      j′= 2;
      if (is-int(i′) && is-int(j′) && (2 ≤ i′≤ 127) && (2 ≤ j′≤ 127))
        make-send(New[i′, j′-1], <t2, {j′mod 4}>);
  }

  for j = 2 to 127 do {
      for i = 2 to 127 do {
          make-receive(New[i-1, j], <t1, {j mod 4}>);
          make-receive (New[i, j-1], <t2, {j mod 4}>);

          New[i, j] = 0.5 * ( local(New[i-1, j], <t1, {j mod 4}>)
                              + local(New[i, j-1], <t2, {j mod 4}>))
          i′= i + 1;
          j′= j;
          if (is-int(i′) && is-int(j′) && (2 ≤ i′≤ 127) && (2 ≤ j′≤ 127))
            make-send(New[i′-1, j′], <t1, {j′mod 4}>);

          i′= i;
          j′= j+1;
          if (is-int(i′) && is-int(j′) && (2 ≤ i′≤ 127) && (2 ≤ j′≤ 127))
            make-send(New[i′, j′-1], <t2, {j′mod 4}>);
      }
  }

  return New }
```

Figure 17: Pipelined intermediate code for simplified Gauss-Seidel

```
Def GS-iteration(Old)
{ New = array(128,128)
  /* Process 3 has no work to do in the initialization of Column 1 */

  for j″= 2 to 122 by 4 do {
      j = j″;
      /* (j+1) mod 4 = 3 – no work */

      /* j mod 4 = 3 */
      j = j″+ 1;
      for i = 2 to 128-1 do {
          receive (t2, 1, j mod 4);
          New[col-loc(i, j)] = 0.5 * (New[col-loc(i-1, j)] + t2);

          i′= i + 1; j′= j;
          /* make-send → null operation */

          i′= i; j′= j+1;
          if (is-int(i′) && is-int(j′) && (2 ≤ i′≤ 127) && (2 ≤ j′≤ 127))
             send(New[col-loc(i′, j′-1)], 1, j′mod 4);
      }

      j = j″+ 2;
      /* (j-1) mod 4 = 4 – no work */
  }
  /* j mod 4 = 3 */
  j = 126;
  for i = 2 to 128-1 do {
      receive (t2, 1, j mod 4);
      New[col-loc(i, j)] = 0.5 * (New[col-loc(i-1, j)] + t2);

      i′= i + 1; j′= j;
      /* make-send → null operation */

      i′= i; j′= j+1;
      if (is-int(i′) && is-int(j′) && (2 ≤ i′≤ 127) && (2 ≤ j′≤ 127))
         send(New[col-loc(i′, j′-1)], 1, j′mod 4);
  }

  j = 127;
  /* (j-1) mod 4 = 4 – no work */
  return New
}
```

Figure 18: Final code for pipelined Gauss-Seidel for Process 3

```
For u = 1 to N do
    For i_1 = 1 to n_1 do
        ...
        For i_k = 1 to n_k do
            A[f_1(u, i_1, ..., i_k), ..., f_t(u, i_1, ..., i_k)] = ...
    For j_1 = 1 to m_1 do
        ...
        For j_l = 1 to m_l do
            A[g_1(u,, j_1, ..., j_l), ..., g_t(u, j_1, ..., j_l)] =
                    ...A[h_1(u, j_1, ..., j_l), ..., h_t(u, j_1, ..., j_l)] ...
```

Figure 19: Extra loop template

to the elements of a matrix. Each process examines its local data to compute the local contribution and sends that value to a central location where the final value is computed. The data distribution determines which process will examine a particular element of the matrix. In this section, we briefly describe how to incorporate accumulation into compile-time resolution.

Compile-time resolution assigns the work of computing a variable to the process that owns it. As a result, the translation of a reduction produces code in which all values needed to compute the reduction are sent to one process. A much better scheme would be to recognize that the operation being performed is commutative and associative and therefore each process can compute a partial accumulation and send that value to a central location where the final value will be computed.

Fortunately, accumulation fits nicely into compile-time resolution. If the partial values of the circulating variable are not used in any statement in the loop body other than the reduction and if the operation is commutative and associative, then a different rule can be used to assign the work for the statement. Instead of assuming that the owner of the left-hand side should do the work, it is assigned to the process that appears most often as the owner of references in the right-hand side of the statement. In addition, the loop is tagged to indicate that the partial values collected by each process should be accumulated at the end of the loop. The collection of the partial values can be pushed out to the outermost loop in which the intermediate values of the variable are not used, to further reduce message traffic. Accumulation has been incorporated into our compiler.

10 SIMPLE

SIMPLE is a program that simulates the behavior of fluid in a sphere. The fluid is in motion and the physical phenomena being simulated are the propagation of shock waves and conduction of heat through the fluid. By taking advantage of rotational symmetry and transforming co-ordinates appropriately, we can reformulate the problem over a mesh divided into rectangles. The rectangles are called *zones* and the corners are called *nodes*. Physical quantities such as position, velocity, temperature and pressure are associated with nodes and zones. The main computation of SIMPLE is a loop that involves the determination of these quantities over a number of discrete time steps where the size of each step is adjusted each time through the loop to guarantee stability. For lack of space, we will not describe the details of the various computations in SIMPLE here, and refer the interested reader to an associated technical report [18].

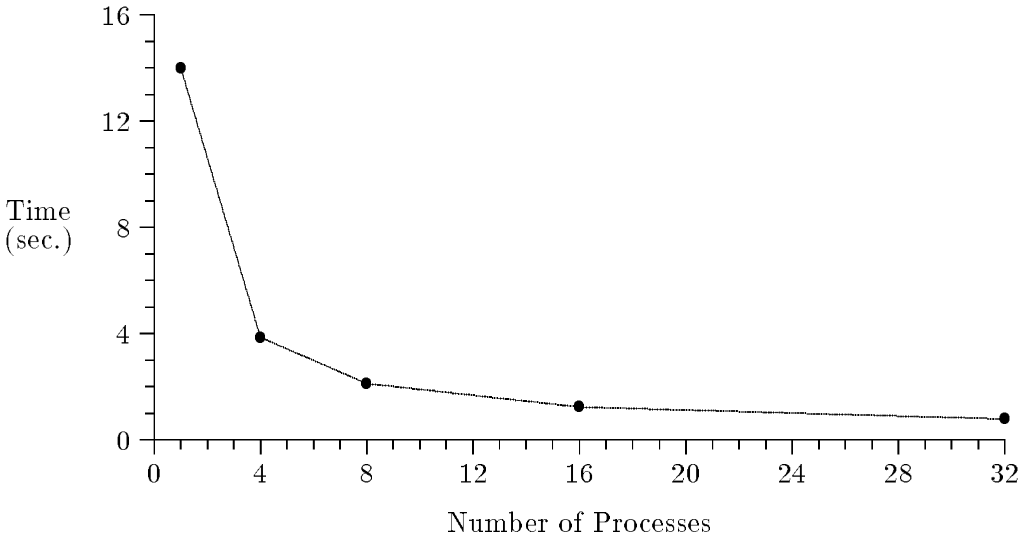

Figure 20: Running times for SIMPLE on a 64 x 64 grid

We ran a set of experiments using an implementation[6] of one iteration of SIMPLE for a 64 by 64 grid. With the exception of the coefficient matrices, all of the matrices in the program were decomposed into roughly square blocks. Our compiler generates C code[7] for the Intel iPSC/2.

Carefully tuned, handwritten programs provide the best comparison for any parallelizing compiler. Unfortunately, we have been unable to locate a handwritten implementation of SIMPLE for the Intel iPSC/2. This is not surprising; decomposing SIMPLE by hand would be a tedious process. The next best alternative is to develop a model that estimates the behavior of a good handwritten program. In this section we present our experimental results, a model for a handwritten implementation, and a comparison of the two.

The graph in figure 20 displays the timing results for SIMPLE on a 64 by 64 grid using roughly square block mappings for the matrices. The implementation for the one process case was obtained by mapping of each matrix into a single block of size 64x64. The resulting sequential program has no communication statements and is the similar to the original sequential program.

10.1 Model of a handwritten implementation

How do we model the total amount of work done by a parallel system? The total parallel time can be represented by the formula

$$TPT = \sum_{i=1}^{n} T^i_{comp} + \sum_{i=1}^{n} T^i_{msg} + \sum_{i=1}^{n} T^i_{system} + \sum_{i=1}^{n} T^i_{idle}$$

where T^i_{comp} is the computation time for the ith process, T^i_{msg} is the time spent by process i in sending/receiving messages, T^i_{system} is the time spent by process i doing system chores like paging and context switching, and T^i_{idle} is the time process i spends waiting for messages. Some of these quantities are difficult to obtain because the time measurement tools on the iPSC/2 are primitive.

[6]We started with an implementation of written in Id[17] by K. Ekanadham[8]. This program makes heavy use of higher-order functions. Since our compiler does not yet handle such functions, we replaced all higher-order functions by equivalent first-order code. Also, our compiler can handle only flat arrays. Since the coefficient matrices were implemented as matrices of matrices, we replaced them with flattened (two-dimensional) versions.

[7]The C programs were compiled using the -O option of the Greenhills C compiler.

Our goal is to develop a model that estimates the running time of a handwritten program. With this goal in mind we can safely ignore the two parameters that are hardest to measure, namely system time and idle time. The resulting model is still valid being on the conservative side of less accurate.

For our particular problem, $\sum_{i=1}^{n} T_{comp}^{i}$ can be estimated by the number of floating point operations[8] necessary times the cost of an average floating point operation, $\sum_{i=1}^{n} T_{comp}^{i} = 1,552,883 * C$. About one third of the operations are multiplies, the rest are adds, compares etc. We estimate the cost of a floating point operation as $C = .33 * 8.52 + .66 * 6.64$. The operation costs used are the costs reported for a the multiplication/addition of two double length floating point numbers[4]. This estimate of the floating point work does not taken into account pipelining.

Communication overhead must be measured with respect to a particular mapping. To estimate this overhead, we counted the number of messages and the size of each message in a four process system. From this we can extrapolate to estimate the communication cost for larger numbers of processes. Our four process estimate is $\Sigma_{i=1}^{n} T_{msg}^{i} = 207.305ms$. This estimate assumes that a value is sent from one process to another process at most once and all the values from a row/column of a matrix are transmitted as one message (except in the sweep phases). Values from different arrays are assumed to be sent in separate messages.

System costs are not completely immeasurable. We can measure the cost of allocating space for arrays. Many functions return arrays as results, so the matrices are stored on the heap instead of the stack. To reduce the cost of allocation, a large chunk of space is allocated at the start of the program. Each individual array allocation requires only a few pointer operations. The allocation cost for a four block configuration is $482ms$ per process.

Once we have an estimate for the total work done in the system, it is easy to model parallel time and expected speed-up. The best possible situation would result in a perfect division of the work. Parallel time, PT, is therefore estimated as $PT = TPT/N$ where N is the number of processes. Speed-up is sequential time divided by parallel time:

$$speed - up = (1,552,883 * C + 1760)/PT$$

Sequential time is the cost of floating point operations plus the allocation cost (1760). The graph in Figure 21 displays three curves: perfect speed-up, the speed-up projected by the model, and our speed-up using the roughly square block decomposition.

There is a substantial gap between the speed-up projected by the model and the speed-up obtained by the programs the compiler generates. Three factors account for the discrepancy. First, our programs send more messages than the projected number. We do not do interprocedural common subexpression elimination to determine when to combine sends of the same value from different procedures. Second, the model assumes that the workload is perfectly balanced. It is not, so all processes slow down to wait for the one with the most work. The blocks on the perimeter have less work than internal blocks because in most cases the work associated with the boundaries is trivial. Also, the southern and western perimeter blocks have more work than the northern and eastern blocks because of the way nodes and zones fit into a matrix. The third factor comes from idle time introduced by the sweep phases in heat conduction. This is unavoidable and is not factored into the model. In summary, our results could be improved by performing interprocedural common subexpression elimination and the accuracy of the model could be improved by incorporating a better estimate of the workload balance and the idle time.

[8] The count of the number of floating point operations was obtained by Jamey Hicks using the MIT Gita dataflow simulator.

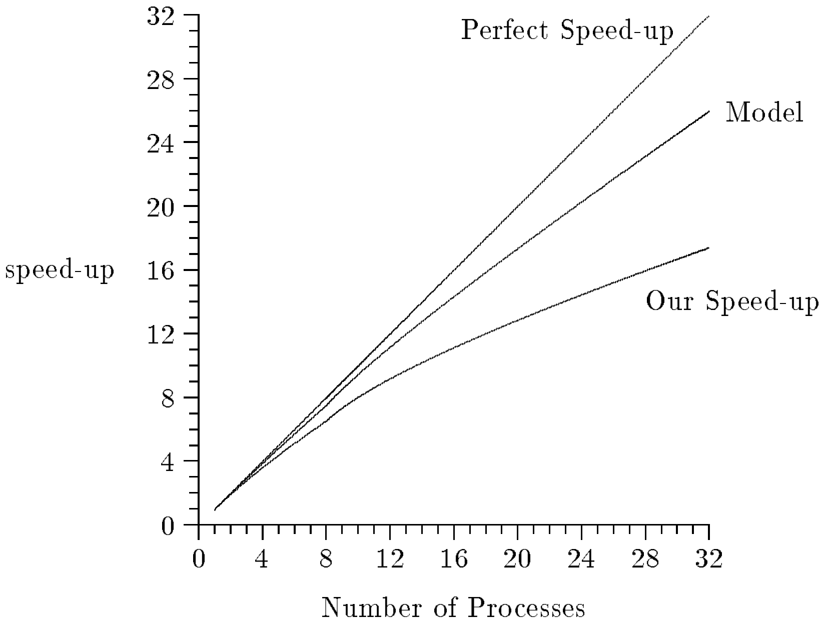

Figure 21: SIMPLE Speed-up

11 Related Work

Several other groups[5, 6, 14, 21, 25, 26] have concurrently developed compilers that use code generation methods similar to our run-time and compile-time resolution. We briefly describe the different ways these groups have built upon the basic code generation schemes.

Rice

A group led by Ken Kennedy[5] at Rice University is considering the question of how to determine good mappings. In Balasundaram et al.[2] they propose an interactive tool to assist the user in selecting the partition and distribution of each array in a program. The basic idea is to estimate statically the performance of a program under various distributions.

Superb

The Superb project[10, 26] led by Hans Zima at the University of Vienna has built a compiler that includes a vectorization of communication optimization, as well as the standard code generation schemes. This optimization is built upon the concept of an overlap area. The *overlap area* of a block contains all non-local elements needed by a process to which this block is local. An overlap area is a block big enough to include all the original block elements plus any necessary non-local elements. It may include elements that are not referenced by the process. The space allocated for a block in a process includes space for the elements in the overlap area. Once a non-local value is transmitted from its owner, it can be stored in the space allocated and referenced simply by translating the reference to the global array into a reference to the local portion of the array. The Superb compiler's basic code generation scheme translates an array reference into a reference to the local array and a statement, called an EXSR, that transmits a value from its owner to any process that has the element in its overlap area. Vectorization of communication uses loop distribution to isolate these statements into loops containing only a single EXSR statement. If the elements referenced by the EXSR can be described by a simple array section, then the loop and the EXSR can be transformed into a single vector EXSR statement.

The overlap concept simplifies the process of vectorizing communication statements by admitting a uniform naming scheme but at the expense of transmitting data that may not be needed. The overlap areas as defined do not extend nicely to wrapped mappings. For instance, the overlap area for an array used in the wrapped version of Gauss-Seidel would be the whole array!

Kali

Koelbel, Mehrotra and van Rosendale [12, 13, 14] have developed Kali, a system that compiles a functional language with a **forall** construct into a language that includes constructs for explicit process creation, data storage layout, and interprocessor communication and synchronization. Recently, this group has been focusing on the problem of generating efficient code for programs that are hard to analyze at compile-time. Their goal is to reduce the overhead of determining what communication is necessary to execute a forall loop. They exploit repeated patterns by saving information about the loop's communication patterns between executions. Rather than recomputing the communication pattern, they try to use the saved information. Saltz et al.[22] have taken a similar approach.

AL

P.S. Tseng [24, 25] has written a compiler that generates W2[15] code for the Warp, a programmable linear systolic array, from programs written in a simple sequential language called AL. This approach may require more than just domain decomposition information from the programmer. The compiler can infer which portion of the data is needed by one process but resides on another, in some cases [25]. When the compiler cannot infer this information, the user must supply it. With this information the AL compiler can generate code using a method similar to ours. One difference is that the basic code generation scheme sends data as soon as it is available rather than waiting until it is needed. The advantage of this method is that the data is made available as soon as possible without explicitly pipelining. The disadvantage is that when the data dependences in a program are too complex for the compiler to analyze, it will generate code that errs on the side of sending any data that might be needed. The cost of sending unneeded values could be high.

Crystal

The Crystal project[6, 16] led by Marina Chen at Yale has designed a compiler that focuses on reducing communication overhead. Program references are analyzed and matched against a set of syntactic patterns that represent a set of aggregate communication routines that can be implemented efficiently. These aggregates include broadcast, reduction, and shift. Some data layouts will match the combination of the program reference pattern and communication aggregates better than others. The compiler tries to chose a decomposition for the data that minimizes the time spent on communication. A decomposition is chosen by searching through the space of possibilities and computing the communication cost for each possible decomposition. The programmer can bypass this search by specifying a data decomposition explicitly.

One of the main ideas behind Crystal is that the programmer should be able to write programs in a very high level language. Ideally, the compiler will be able to generate an efficient implementation for the target architecture. Crystal includes a meta-language that allows the programmer to direct the transformation process. Practically, this transformation process may require insight from the programmer.

Dino

The Dino Project[21] led by R. Schnabel at Colorado takes a more language oriented approach to the problem of programming distributed memory machines. The Dino language consists of a set of parallel constructs added to C. These constructs allow the programmer to distribute data and to specify composite procedures that act upon the distributed data. A composite procedure is one called by several processes concurrently. Communication among processes is not specified by explicit sends and receives. Instead, the semantics of the language require that any actual parameter data needed for a composite call will be transmitted to the processes from the host when the procedure is called and any results will be transmitted back upon return from the call. Also, the programmer can explicitly indicate that a piece of data should be read or written remotely by tagging the reference with a '#'. This approach gives the programmer more control over the final code but at the expense of managing the details of a parallel program.

12 Conclusions

In this paper we have described a compilation method for distributed memory machines that is based on the idea that parallelization should be driven by the placement of data rather than the text of a program. We have built a compiler that takes a sequential program augmented with data decomposition information and performs process decomposition inserting synchronization where necessary. We use two different strategies for generating code. The first, run-time resolution, is very general; it is guaranteed to generate correct if somewhat inefficient code for any program, no matter how complex. The principle behind run-time resolution is that the process that owns a value must do any work necessary to compute the value and it must participate in any computation involving the value. Every process examines each statement to determine its role in the computation. This code can be partially evaluated at compile-time to improve its efficiency. Our second strategy, compile-time resolution, specializes the code generated by run-time resolution to each process. As many tests that determine where work is to be done are removed as possible. The resulting code is guaranteed to be at least as efficient as the run-time resolution code and in many cases is substantially more efficient.

We showed that the improvements gained by compile-time resolution are not enough to compete with hand-written code and we described two message passing optimizations that deal with the issues of overhead and synchronization in message passing. Vectorization of messages combines messages that arise from a single array reference within a loop to reduce the overhead attributed to message passing. Pipelining moves sends to the earliest point in the program where the value is available to optimize the synchronization point of the message which increases the parallelism displayed by the final code. Finally, we recognized reductions involving commutative and associative operators like addition, and showed how to exploit that structure in the code.

We believe that this approach to compiling for parallel machines with a memory hierarchy is suitable for a large class of applications that can be characterized roughly as dense problems in which load balancing and locality of reference are not in conflict. We demonstrated this by automatically generating code for a large scientific benchmark called SIMPLE, representative of the computations our system can parallelize well. We estimate that the resulting code is within sixty to seventy percent as efficient as a carefully hand-written program. However, there are applications for which our approach may not do as well. Compile-time resolution will not be helpful for applications that require complex data structures, such as sparse problems or those requiring irregular meshes, since the data references may not be analyzable by the compiler. One approach to these problems is to use variations on scatter and gather, as suggested by Saltz et al [22]. Problems in which load balancing and locality of reference are in conflict, such as particle-in-the-cell codes, may also not yield to our techniques. The particle-in-the-cell problem deals with a bunch of charged particles on a grid; in each time step, the electric field at a point in the grid is determined by looking at the particles near that point. The resulting electric field is used to compute how far the particles travel in that time-step. The problem here is that particles tend to travel in bunches. Consider what happens if the mesh is distributed in some regular way among the processes. If the particles are distributed evenly among the processes, processes may have to update the field at mesh points far away from themselves. If, however, a process is given all the particles close to the mesh points it owns, some processes may have little work to do. One approach is to use complex partitioning strategies, as suggested by Azari and Lee [**?**]. We would like to emphasize that our system will produce correct code even for these kinds of problems; however, the quality of the code produced by the compiler will probably not be competitive with hand-written code. We leave these problems for future research.

A Handwritten Version of Gauss-Seidel

The following is a handwritten implementation of Gauss Seidel for a system with S processors for a non-boundary processor.

```
/* S is the number of processes in the system */
Defconstant LEFT = (p-1) mod S;
Defconstant RIGHT = (p+1) mod S;
Defconstant blksize = 8;

Def GS-iteration(Old) =
{ /* All array allocations have been omitted due to lack of space */

  /* 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 */
      For i = 1 to N-1 do
          Old-out[i] = Old[i, j];
      send(old-out, 5, LEFT);
      receive(old-in, 5, RIGHT);

      /* 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 */
          receive(rnewvalues, 3, 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]
                                  + Old-in[i] + Old[i+1, j]);
              snewvalues[i mod blksize] = New[i,j]; }
          /* send these values to the RIGHT */
          send(snewvalues, 3, RIGHT); } }
  return(New); }
```

References

[1] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the ptran analysis system for multiprocessing. *J. of Parallel and Distributed Computing*, 5:617–640, 1988.

[2] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In *to appear in Proceedings of the Fifth Distributed Memory Computing Conference*, April 1990.

[3] S. Bokhari. *Assignment Problems in Parallel and Distributed Computing.* Kluwer Academic Publishing, 1987.

[4] L. Bomans and D. Roose. Benchmarking the iPSC/2 hypercube multiprocessor. *Concurrency Practice and Experience*, 1(1), September 1989.

[5] D. Callahan and K. Kennedy. Compiling programs for distributed memory multiprocessors. *The Journal of Supercomputing*, 2(2), October 1988.

[6] M. Chen, Y. Choo, and J. Li. Theory and pragmatics of compiling efficient parallel code. Technical Report YALEU/DCS/TR-790, Yale Univeristy, December 1989.

[7] W.P. Crowley, C.P. Hendrickson, and T.E. Rudy. The SIMPLE code. Technical Report UCID-17715, Lawrence Livermore Laboratory, 1978.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F.K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, 1989.

[9] K. Ekanadham and Arvind. SIMPLE Part I: An exercise in future scientific programming. Technical Report RC12686, IBM, April 1987.

[10] M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems.* PhD thesis, Univeristy of Bonn, 1989.

[11] A. Karp. Programming for parallelism. *IEEE Computer*, 20(5), May 1987.

[12] C. Koelbel. *Compiling Programs for Nonshared Memory Machines.* PhD thesis, Purdue University, West Lafayette, IN, August 1990.

[13] C. Koelbel, P. Mehrotra, and J. van Rosendale. Semi-automatic domain decomposition in Blaze. In *Proceedings of the International Conference on Parallel Processing*, 1987.

[14] C. Koelbel, P. Mehrotra, and J. van Rosendale. Supporting shared data structures on distributed memory architectures. In *Proceedings of the Second ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, 1990.

[15] M. Lam. *A Systolic Array Optimizing Compiler.* PhD thesis, Carnegie Mellon University, 1987.

[16] J. Li and M. Chen. Synthesis of explicit communication from shared-memory program references. Technical Report YALEU/DCS/TR-755, Yale Univeristy, May 1990.

[17] R. Nikhil, K. Pingali, and Arvind. Id Nouveau. Technical Report CSG Memo 265, MIT Laboratory for Computer Science, 1986.

[18] K. Pingali and A. Rogers. Compiler parallelization of SIMPLE for a distributed memory machine. Technical Report 90-1084, Department of Computer Science, Cornell University, 1990.

[19] C. Polychronopoulos, M. Girkar, M. Haghighat, C.L. Lee, B. Leung, and D. Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, August 1989.

[20] A. Rogers. *Compiling for Locality of Reference.* PhD thesis, Cornell University, August 1990.

[21] M. Rosing, R. Schnabel, and R. Weaver. The DINO parallel programming language. Technical Report CU-CS-457-90, Univeristy of Colorado at Boulder, April 1990.

[22] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[23] H. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *ACM Transactions on Software Engineering*, 1977.

[24] P.S. Tseng. *A Parallelizing Compiler for distributed memory parallel computers.* PhD thesis, Carnegie-Mellon University, May 1989.

[25] P.S. Tseng. Compiling programs for a linear systolic array. In *Proceeding of ACM Symposium on Programming Language Design and Implementation*, June 1990.

[26] H. Zima, H. Bast, and M. Gerndt. SUPERB:a tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6(1):1–18, 1988.