Compiler Parallelization of SIMPLE for a Distributed Memory Machine

Keshav Pingali*
Anne Rogers**

TR 90-1084
January 1990

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

*Keshav Pingali was supported by an NSF Presidential Young Investigator award and by grants from the General Electric Corporation, the Digital Equipment Corporation, and the Math Sciences Institute, Cornell.
**Anne Rogers was supported by a DARPA Assistantship in Parallel Processing, Grant number 26974C.

Compiler Parallelization of SIMPLE for a Distributed Memory Machine

Keshav Pingali[*]
Anne Rogers[†]
Department of Computer Science
Cornell University
Ithaca, New York

January 16, 1990

Abstract

In machines like the Intel iPSC/2 and the BBN Butterfly, local memory operations are much faster than inter-processor communication. When writing programs for these machines, programmers must worry about exploiting spatial locality of reference. This is tedious and reduces the level of abstraction at the which the programmer works. We are implementing a parallelizing compiler that will shoulder much of that burden. Given a sequential, shared memory program and a specification of how data structures are to be mapped across the processors, our compiler will perform process decomposition to exploit locality of reference. In this paper, we discuss some experiments in parallelizing SIMPLE, a large scientific benchmark from Los Alamos, for the Intel iPSC/2.

[*]Keshav Pingali was supported by an NSF Presidential Young Investigator award and by grants from the General Electric Corporation, the Digital Equipment Corporation, and the Math Sciences Institute, Cornell.

[†]Anne Rogers was supported by a DARPA Assistantship in Parallel Processing, Grant number 26974C.

1 Introduction

In distributed memory machines like the Intel iPSC/2 or the NCube, each process has its own address space and inter-process communication takes place through sending and receiving messages. Typically, passing messages is about 10 to 100 times slower than reads and writes out of local memory. For example, on the Intel iPSC/2, local memory access takes about a microsecond but passing a message can take 300 microseconds even if there is no congestion in the network. From the perspective of the programmer, this means that a process can access local data items (*i.e.*, data items in its own address space) very fast, but access to non-local data items, which must be choreographed through an exchange of messages, can be one or two orders of magnitude slower. Therefore, it is important to exploit locality of reference when programming distributed memory machines.

The solution adopted by manufacturers of distributed memory machines is to provide C or FORTRAN with message-passing extensions. The programmer writes CSP-like programs in which he has control over the distribution of data and code across processes. Unfortunately, this results in a loss of abstraction in programming. For example, consider a matrix X that is distributed across the processes by rows or columns to get effective parallel execution. To access element X[i,j], the programmer cannot just write X[i,j] as he could in C or FORTRAN on a sequential machine — if it is a local data element, he can read it directly, but if it is a non-local data item, he must put in calls for sending and receiving messages. This is a big burden on the programmer.

As an aside, we note that exploiting spatial locality of reference is important even on shared-memory machines. In shared-memory machines, such as the BBN Butterfly and the IBM RP3, there is a single, global address space that is shared by all processes. Inter-process communication is accomplished by reading and writing memory locations. The single, shared address space is usually implemented physically as a number of processor-memory pairs interconnected through some network. The cost of accessing a non-local data item (*i.e.*, across the network) can be an order of magnitude more than accessing local data. Therefore, even in shared-memory machines, exploiting spatial locality of reference is important for good performance[1]. One way to reduce the impact of non-uniformity of memory access is caching. However, caching in multiprocessors introduces the problem of cache coherence for which practical, scalable solutions are not yet known.

Can the problem of exploiting locality of reference be tackled by the compiler? Most work to date on compiling for multiprocessors focuses on parallelization of code using techniques like distributing loop iterations among processors. An example of this approach is the Camp system of Peir and Gajski[14]. Parallelization is achieved by distributing loop iterations among processors; synchronization required for loops with loop-carried dependencies is implemented through complex bit-masks at each word of memory. A similar approach is being pursued in the CEDAR system at Illinois. We characterize these approaches as 'code-driven' because they pay little attention to data partitioning - a processor may execute an iteration for which the data is not local.

We are implementing a system in which data partitioning plays a central role because it is used to drive the parallelization of code. The intuitive idea is the following. The programmer writes and debugs his program in a high-level language using standard high-level abstractions such as loops and arrays. Once this is accomplished, he specifies the *domain decomposition* - that is, how data structures are to be distributed across the multiprocessor. Given this data decomposition, the compiler performs process decomposition by analyzing the program and specializing it to the data that resides at each processor. Thus, our approach to process decomposition is 'data-driven' rather than 'program-driven'. It is important to understand that our work is orthogonal to earlier studies on so-called 'assignment problems' that have focused on mapping the topology of the problem onto the interconnection topology of the machine[1,2]. These studies assume a model in which the major factor in the cost of communication is the distance between the source and destination of messages. This model is valid in machines like the Intel iPSC/1 in which a message interrupts every processor on the way from the source to the destination (routing at intermediate nodes was performed by the processor). In more recent machines like the Intel iPSC/2, routing of messages at intermediate nodes is handled by com-

[1]An exception to this is the Ultracomputer [7] in which all memory is equally far away from all processors. This uniformity is achieved by making all accesses equally expensive!

munications coprocessors. On the iPSC/2, the cost of start-up and receipt of the message is about 350 microseconds, while the time per hop is only about 10-20 microseconds. As mentioned earlier, the cost of a local access is less than a microsecond. Therefore, for any machine of reasonable size, there is a big difference between the cost of a local access and the cost of sending a message, but once a message has to be sent, the distance it has to travel is relatively unimportant. There are secondary considerations such as bandwidth - if messages have to travel less on an average, the probability of saturating the available network bandwidth when a lot of messages are sent simultaneously is reduced. All things considered, we decided not to worry about mapping the topology of the problem to the topology of the machine.

In our opinion, the crux of the problem is to achieve locality of reference by matching data distribution to executable code in order to reduce the number of messages sent. However, concerns of locality must be balanced against the overall goal of achieving parallel execution - after all, a simple way to minimize the number of messages sent is to map all the code and data to a single processor! In fact, in some 'particle-pushing' codes, it is difficult to achieve locality of reference and load balancing simultaneously. These codes simulate the motion of charged particles in a grid. Each processor is assigned a region of the grid and it keeps track of physical variables associated with that region. In many of these problems, the particles tend to move together in a bunch. If each processor is assigned the work of pushing particles in its region, the computational load will be unbalanced; if the work of pushing particles is divided equally among all the processors, a processor may have to push a particle in a region residing on some other processor, which leads to loss of locality of reference. The code generation techniques we describe in this paper will work even for such programs but it is unclear to us that the resulting code will have good performance. In fact, some researchers have even questioned the suitability of parallel processing for such codes[17].

In the very large class of 'continuum' problems[10], on the other hand, locality of reference and load balancing are not usually in conflict. SIMPLE is an example of such a problem. In this paper, we discuss experiments in using our compiler to parallelize SIMPLE, a large hydrodynamics application, which is about a thousand lines of FORTRAN[5]. This program incorporates many of the computation and communication paradigms typical of scientific code; therefore, it is a popular benchmark. There is plenty of parallelism in SIMPLE, but it is not an 'embarrassingly parallel' application in the sense that there is a lot of movement of data during the execution of the problems. These aspects of SIMPLE are discussed in Section 2. A simple machine model is presented in Section 3. Section 4 is a discussion of various methods of distributing data across processors. In Section 5, we use parts of SIMPLE to explain our compiling techniques. Some preliminary results were reported in an earlier paper [16]; however, this paper is self-contained. We also report performance results on the Intel iPSC/2 (without vector boards). On a 32 processor Intel iPSC/2, we have obtained about 2MFlops on a 64x64 problem. In Section 6, we present a simple performance model to explain the observed performance. We conclude in Section 7 with a discussion of some extensions to our system.

2 What is SIMPLE?

In this section, we describe SIMPLE and discuss the computation and communication behavior of the various phases in this program. We have followed the presentation of Ekanadham and Arvind[6].

2.1 Overview

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, we can confine our attention to a semi-circular annulus divided into *zones* by radial and axial lines as shown in Figure 1. The annulus is defined by a minimum radius *kmin* and a maximum radius *kmax*. Similarly, the radial lines are numbered from *lmin* to *lmax*. Corners of zones are called *nodes*. Nodes are referred to using indices *(k,l)*. The same co-ordinate system is used for zones by giving the zone the label its leading corner — for example, zone *(k,l)* has corners labeled *(k,l)*, *(k-1,l)*, *(k,l-1)* and *(k-1,l-1)*.

To take boundary conditions into account, *ghost zones* are added on the periphery of the annulus as shown in Figure 1. Values of physical variables like temperature, pressure etc. at ghost zones are chosen to simulate desired

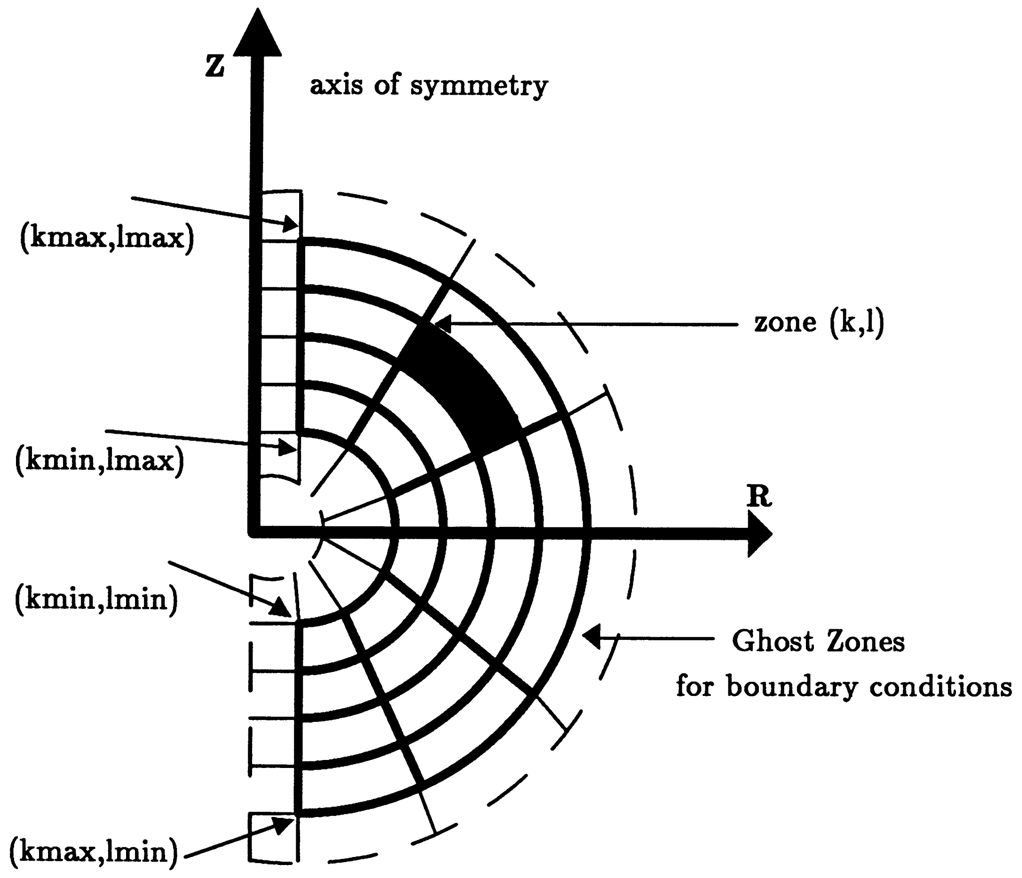


Figure 1: Zones in Semi-circular Cross-section

boundary conditions.

The following physical quantities are computed for each node and zone:

Node:	Zone:
Co-ordinates X: (r,z)	Area: a
Velocity V: (u,w)	Volume: s
	Density: ρ
	Artificial viscosity: q
	Energy: ϵ
	Temperature: θ
	Pressure: p

Since the fluid is in motion, the nodal and zonal attributes vary with time. The mass in each zone is fixed, and the co-ordinates of the four corner nodes of the zone are updated at each time step to reflect the motion of the fluid. The main computation of SIMPLE is a loop that involves the determination of the physical quantities over a number of discrete time steps where the size of each step is adjusted each time through the loop to guarantee stability. The steps involved in each iteration of the loop and the associated data flow is shown in Figure 2.

2.2 Computation and communication in each step

In this subsection, we discuss the dataflow in each step of the main loop of SIMPLE. At the beginning of the iteration, it is assumed that the values of the physical quantities at each node and zone and the size of the time step are given. At the end of the iteration, the new values of these physical quantities have been computed together with a new step step and an estimate of the error introduced by various approximations in the computation.

Step 1: Newton's second law of motion is used to compute the acceleration at each node. The forces that act on the fluid mass in a zone are due to pressure and artificial viscosity. The resulting acceleration at a node is computed by averaging over effects on the four zones adjacent to the node. Thus, the acceleration at a node is a function of the pressure, artificial viscosity, density and volume of its adjacent zones and the positions of its four nearest neighbour nodes. The new velocity is computed from this acceleration and the size of the time step. There are no dependencies between the velocity computations of different nodes and all these computations can be done in parallel. The dataflow for this step is shown in Figure 3(a).

Step 2: Given the velocity of a node, its new position at the end of the time step is computed by adding the product of the velocity and time step to its position at the start of the time step. This is a simple pointwise computation at each node. For boundary nodes, the computation is more complex and requires some communication between adjacent zones.

Step 3: The new area, volume and density of each zone is computed in this step. Given the positions of the four nodes that form the corners of the zone, and the density and volume of the zone, the new area, volume and density of the zone can be computed. The dataflow for this step is shown in Figure 3(c). There are no dependencies between the computations of the densities and volumes of different zones and all of them can be done in parallel.

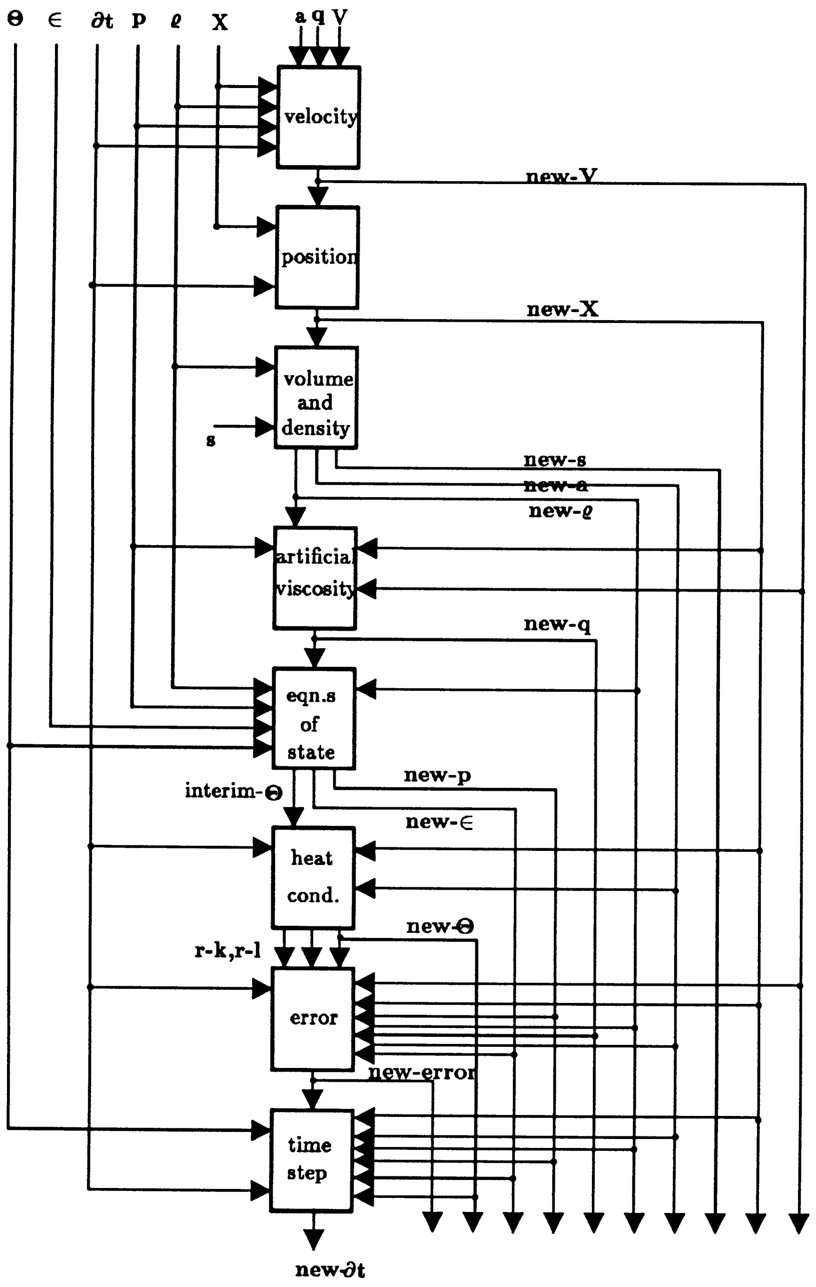
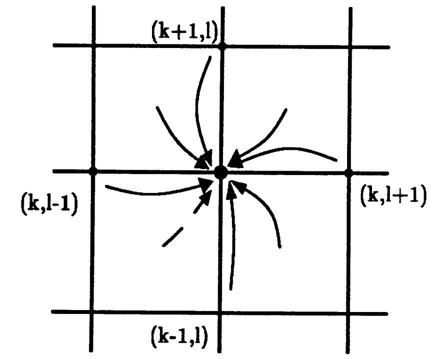
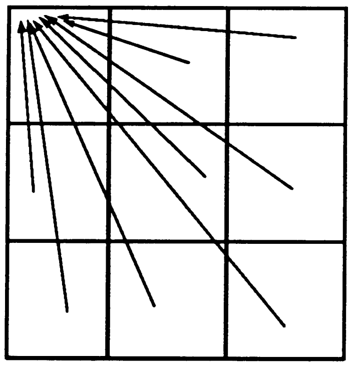


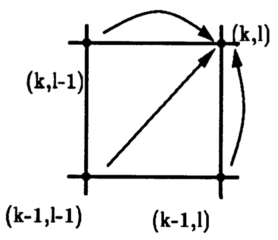
Figure 2: Data flow in Main Loop of SIMPLE



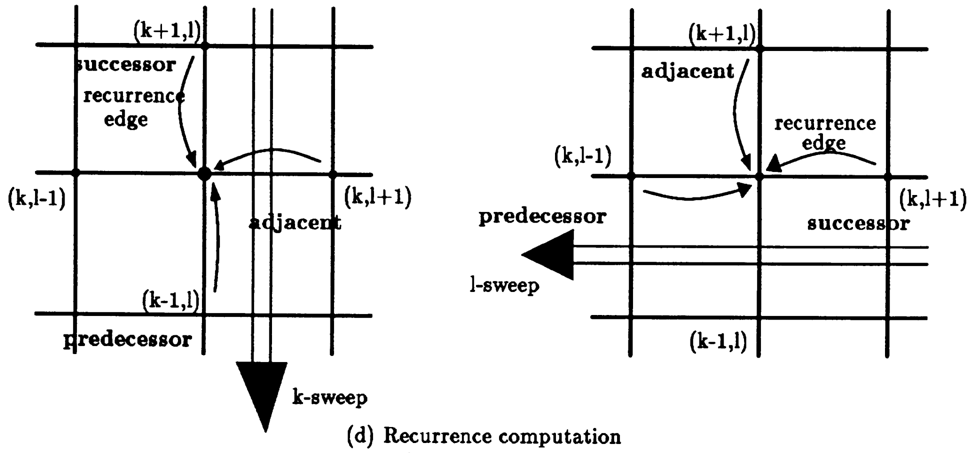
(a) Nearest Neighbour Communication
(Velocity Computation)



(b) Global accumulation
(Energy Balance)



(c) Zone corners communication
(Density computation)



(d) Recurrence computation
(Heat conduction)

Figure 3: Patterns of Communication in SIMPLE

Step 4: The new artificial viscosity in a zone is computed from the new positions and velocities of the corner nodes of the zone, the new density of the zone as computed in the previous step, and the pressure in the zone. The data flow in this step is shown in Figure 3(c).

Step 5: The pressure, energy and temperature in each zone are related by the so-called equations of state. The new energy is first calculated by taking into account the work done by the fluid in the zone against pressure and artificial viscosity. This is a pointwise computation. The new pressure and an interim value for the temperature are obtained by another pointwise computation which uses two polynomials. The polynomials are piecewise continuous and their coefficients depend on the density and temperature. Therefore, the ρ–θ plane is divided into regions and a table is provided which has the coefficients for each region. Given the density and temperature, this table is first indexed by the density and temperature to retrieve the relevant coefficients of the polynomials which are then used in a pointwise computation to compute the new energy, new density and an interim value of the new temperature in the zone. Thus, the computation in this step is pointwise.

Step 6: In this step, the effect of heat conduction across zones is computed. The inputs to this step are the new volumes and positions as well as the interim temperature computed in Step 5. The outputs are the new temperature of the zones and two matrices r–k and r–l which are used in Step 7. In the beginning of this step, some pointwise computation is first performed to compute intermediate values. The crux of the heat conduction computation involves two 'sweeps' called the *k-sweep* and *l-sweep*. The data flow in these sweeps is shown in Figure 3(d). The new temperature at each zone is computed by a simple recurrence which is a function of the new temperature in the *successor* zone and various intermediate values at the *predecessor* zone and the *adjacent* zone. Notice that there are two sweeps in this step - one in the k direction and one in the l direction. Because of the recurrences, these computations cannot all be done in parallel. Notice that values in the new temperature matrix are being used to compute other values in the same matrix. This requires the use of non-strict array constructors which are not supported in some functional languages like SISAL[8]. In spite of the recurrence, there is parallelism that can be exploited during the sweeps - during the k and l sweeps, each column and each row can be computed in parallel respectively.

Step 7: This step involves some book-keeping to check the balance of energy of the internal and kinetic energies in the interior zones and the work done and heat lost at the boundaries. At each step, the energy balance is computed to ensure that the error is within tolerance limits. The internal energy computation involves pointwise computations of internal energies at each node and then a global accumulation to add all the contributions. The dataflow of the global accumulation is shown in Figure 3(b). Kinetic energy requires values from the four adjacent zones of a node followed by a global accumulation. The dataflow in this step is like that of Figure 3(a) followed by Figure 3(b). Boundary heat and boundary work require a global accumulation of values computed for nodes at the boundary. A small amount of communication between adjacent nodes is also required.

Step 8: The value of the time step for the next iteration is computed. The computation involves checking that the Courant condition is met (sound should not cross any zone in a single time step) and that the time increment should not exceed the relative temperature change in any zone. The dataflow for checking the Courant condition requires a dataflow like that of Figure 3(c) followed by a global accumulation as in Figure 3(b) to determine a global minimum. The maximum relative change in temperature is a pointwise computation followed by a global accumulation to determine the maximum.

2.3 Our Implementation

We started with an implementation of SIMPLE written in Id[12] by K. Ekanadham[6]. 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 non-higher-order code. Also, our compiler can handle only flat arrays. The coefficient matrices were implemented as matrices of matrices. We replaced these with flattened (two-dimensional) versions.

3 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[2]. Each process has its own address space and all communication is done using explicit message passing. Also, each process also has a set of mailboxes for receiving messages. The primitives for message passing are:

- **send**(Bi, Ai, Si, Pi) - The bytes in memory locations Ai to Ai + Si - 1 are sent by the process executing this command to mailbox Bi at process Pi. The sending process does not have to wait for the receiving process to get the message.

- **receive**(Bi, Ai, Si) - The process executing this command waits until it receives a message in box number Bi. The first Si bytes of the message are stored in locations Ai to Ai + Si - 1. While it is waiting, messages it may receive that are destined for other mailboxes are ignored temporarily.

The order that messages arrive at a process is determined by the order in which they are sent but also on how they collide with other messages in transit. Mailbox numbers let the receiving process focus its attention on a message of a particular type from a particular process rather than having to look at the first message that comes along. Mailbox numbers also make it easier to reorder sends within a process. As long as two sends have either different destinations or the same destinations but different mailbox numbers, they may be reordered. Each textual reference in the program has a number associated with it and all communication that arises from a given reference is tagged with a combination of the associated number and the process name. Therefore messages arising from different array references can always be reordered.

Long messages are automatically broken into packets by the underlying implementation. Packet reassembly is handled similarly . In most message passing systems sending one long message is less expensive than sending several small ones. We assume this in our model.

[2]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.

The examples in the next section are written in C. They were generated by our compiler and then cleaned-up to make them more readable. The loops have been un-normalized and the temporary variable names have been shortened. **Mynode()** is a system routine that returns the name of the executing process.

4 Data Distribution

The data distributions supported in our system are a compromise between generality and regularity. As we discuss in Section 5, 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 leave much to be desired. A regular data distribution offers more opportunities for optimization. Three such distributions for SIMPLE are wrapped rows/columns and two styles of block distributions.

Given P processes, a wrapped row distribution of the 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 X 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.

Determining the best data distribution for a given problem is not an exact science but there are some rules of the thumb people use when writing code for distributed memory machines. For the dataflow patterns of Figure 3(a) and (c), block decompositions are preferred over wrapped row/column mappings. Since the computations at various points are independent of each other, either kind of distribution leads to a well-balanced decomposition (barring edge effects at boundaries). On the other hand, a row/column distribution can result in more communication than a block decomposition. This is the familiar boundary length to area argument - for a block decomposition, values at interior points of a block do not have to be communicated using messages[9]. Therefore, a block decomposition is preferred over a row/column distribution for this situation.

The dataflow patterns in SIMPLE are amenable to two kinds of block decompositions: roughly square and horizontal/vertical strips. A block decomposition qualifies as roughly square if there are approximately the same number of blocks in each dimension. Perfectly square blocks are possible only when the desired number of blocks is a perfect square. A strip decomposition breaks the matrix into blocks of contiguous rows or columns. The boundary length argument mentioned earlier dictates that roughly square blocks incur lower communication overhead. This is certainly true if each value that flows across a boundary is sent in a separate message. However, for most of the data flow patterns in SIMPLE, all of the values from a given matrix can be sent across a boundary in a single message. The marginal cost for each additional value is much lower than the start-up cost associated with any message. A strip mapping would have slightly higher communication overhead but not prohibitively so.

Not all of the computation in SIMPLE follows the dataflow patterns of Figure 3(a) and (c). The sweep phases use very simple recurrences, data flows in only one dimension at a time. During k-sweep the data flows along the columns and during the l-sweep it flows along the rows (see Figure 3(d). A strip mapping in the appropriate dimension reduces the communication overhead of such a recurrence to zero. Unfortunately, the correct dimension for one sweep is not the correct dimension for the other sweep. The communication costs and the forced idle time would be larger in the off-dimension sweep than for a roughly square decomposition. Whether the communication costs saved by the on-dimension sweep outweigh the extra cost for the off-dimension sweep depends heavily on the computation done during the sweeps. If the off-dimension computation is very expensive, it might be worth remapping from one strip dimension to another.

For more complex recurrences, blocks are not the best choice. When the parallelism in the most expensive part of the computation falls along a wavefront, wrapped mappings will yield better results.

The above discussion is meant to give the reader a feeling for the trade-offs involved in choosing a mapping. Without a clear choice, we chose to follow conventional wisdom and use a roughly square block decomposition for our final implementation of SIMPLE.

5 Code Generation

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 fall-back position, 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 *doacross* 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 dependencies across processors. *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 is what determines which process will examine a particular element of the matrix. *Scatter-gather* is a method that is used when the data dependencies in a loop can only be determined at run-time. If the dependencies 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 of 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 of these techniques to produce good code for a distributed memory machine.

In this section we first discuss *run-time resolution* which is a simple but fairly inefficient code generation strategy that is guaranteed to work for any program, no matter how complex. Next, we show how this code can be improved by *partial evaluation* at compile-time - the resulting code generation strategy is called *compile-time resolution.* We also point out some connections between this problem and the problem of code generation for languages with overloaded operators. Even though the results of compile-time resolution are superior to those of run-time resolution, there is still room for improvement. We discuss two extensions, *vectorization* of messages and *accumulation*, that are required to parallelize SIMPLE successfully. We motivate both the code generation schemes and the extensions with examples taken from SIMPLE.

5.1 Run-time Resolution

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

- Every process examines each statement to determine its role (if any) in the execution of the statement by using the two rules below.
- The process that owns a variable or array element is responsible for computing its defining expression and recording its value.
- The process that owns a variable or array element must communicate that value to any process that needs it.

The procedure `Make_AB_North` in Figure 4 is part of a routine from the heat conduction phase of SIMPLE. Notice that all of the arrays are to be decomposed into four blocks of size 32x32. Three functions are associated with a block mapping: *BM* determines the owner of an array element from the index expressions of the reference, *BL* computes the offset into the local portion of an array from the index expressions of the reference, and *BA* allocates the local portion of an array. Figure 5.1 contains the result of applying run-time resolution this procedure. `Coerce` is a macro for the reading and transmission of a value (see Figure 6 for its definition). Run-time resolution produces one program that is executed by all of the processes. The iterations during which a process has real work to do are

```
defconstant kmin = 2
defconstant kmax = 63
defconstant lmin = 2
defconstant lmax = 63
defconstant bs1 = 32
defconstant bs2 = 32

def Make_AB_North R_bar:realarray@block(bs1, bs2)
                       Sigma:realarray@block(bs1, bs2)
                       a:realarray@block(bs1, bs2)
                       :void =
{ /* Fill in boundary elements */
  call set_boundary_zones A 0.0;
  { for l = lmin+1 to lmax do
      { for k = kmin+1 to kmax do
          d:real@SameAs(A, (k, l)) = Sigma[k,l] + R_bar[k-1, l] * (1 - A[k-1, l]);
          A[k, l] = R_bar[k,l]/d; } }
}
```

Figure 4: Make_AB_North routine from Heat Conduction Phase

not known at compile-time, therefore every process must participate in every iteration of both loops of Make_AB_North. In each iteration, one process will own d and A[k, l]. That process will collect the required values, do the arithmetic, and perform the assignment. A second process may own the required values from R_bar, and a and if so, will send those values to the process that owns d and A[k, l]. The rest of the processes will have nothing to do.

5.2 Compile-time resolution

Run-time resolution generates inefficient code. In our example, a process will spin through many unnecessary iterations checking for work to no avail. Techniques similar to those used to resolve overloading in conventional compilers can be used to generate better code. When compiling languages like Lisp, an overloaded operator like + is usually compiled into a case statement that tests the type of the arguments and dispatches to the appropriate type specific addition routine. The naive code generated by this strategy can be improved considerably if the compiler knows the types of the arguments or the result (for example, through type declarations) since the case statement can be replaced by a dispatch to the relevant addition routine. This kind of code improvement through 'specialization' of generic code can be used profitably in our context as well. The code generated by run-time resolution is like generic code that can be specialized to each process by using the mapping information. This approach is called *compile-time resolution.*

When generating code for each process using compile-time resolution, the compiler examines each statement to determine the process's role in the evaluation of that statement. This is done in two stages. The compiler uses conventional abstract syntax trees as the internal representation of programs. In the first stage, the user's mapping information is propagated through the program's abstract syntax tree. In the second stage, this information is used to generate code. Each node of the abstract syntax tree has two attributes named *evaluators* and *participants.* The *evaluators* of a node in the abstract syntax tree is the set of processes that perform the operation defined by the node. The *participants* of a node, n, in the abstract syntax tree is the set of processes that must participate in the evaluation of some node in the subtree rooted at the node, i.e. the union of the evaluators of the nodes in the subtree rooted at n. For lack of space, we do not give the

```
Make_AB_North(R_bar, Sigma, a)
real_istructure a, Sigma, R_bar;

{ int k, l;
  double t1, t2, t3, t4, t5, t6;

  set_boundary(A, 0.0) ;

  { for (l = 3; l <= 63; l = l + 1)
      { double d;
        for (k = 3; k <= 63; k = k + 1)
          { if (Mynode() == BM(k, l)) {
              /* this process owns d - get necessary values and
                 do computation */
              coerce(t1, Sigma[BL(k, l)], BM(k, l), BM(k, l));
              coerce(t2, R_bar[BL(k-1, l)], BM(k-1, l), BM(k, l));
              coerce(t3, A[BL(k-1, l)], BM(k-1, l), BM(k, l));
              t4 = t1 + t2 * (1 - t3);
              d = t4; }
            else {
              /* this process does not own d - send any values
                 that are necessary which this process owns */
              coerce(t1, Sigma[BL(k, l)], BM(k, l), BM(k, l));
              coerce(t2, R_bar[BL(k-1, l)], BM(k-1, l), BM(k, l));
              coerce(t3, A[BL(k-1, l)], BM(k-1, l), BM(k, l)); }

            if (Mynode() == BM(k, l)) {
              /* this process owns A[k,l] - get necessary values and
                 do computation */
              coerce(t5, R_bar[BL(k, l)], BM(k, l), BM(k, l));
              coerce(t6, d, BM(k, l), BM(k, l));
              A[BL(k, l)] = t5/t6; }
            else {
              /* this process does not own A[k,l] - send any values
                 that are necessary which this process owns */
              coerce(t5, R_bar[BL(k, l)], BM(k, l), BM(k, l));
              coerce(t6, d, BM(k, l), BM(k, l)); }
          } } }
}
```

Figure 5: Run-time resolution code for Make_AB_North

```
macro coerce(ref, var, owner, eval) =
{
  if ((mynode() == owner) and (mynode() == eval))
    var = ref
  elseif (mynode() == owner) {
    t1 = ref;
    send(boxnum(ref), &t1, sizeof(typeof(ref)), eval); }
  elseif (mynode() == eval)
    recv(boxnum(ref), &var, owner);
}
```

Figure 6: Definition of Coerce

details of the determination of the evaluators and participants attributes. For the most part, these rules are quite straight-forward; the only complication is that the set of participants is used to determine the evaluators for some types of nodes, such as conditionals - the union of the participants of the then-branch and else-branch defines the evaluators of the boolean test in a conditional expression.

The information collected in the first phase is used to generate code. Given a process name and a tree node, the compiler tries to determine if the process is a member of the evaluators of the node. Three outcomes are possible: true, false, and inconclusive. True means that the process must perform the operation defined by the node. False means it need not. Inconclusive means that run-time resolution must be applied because the compiler cannot analyze the mappings sufficiently. This evaluation will require techniques such as subscript analysis that are commonly used in vectorizing compilers [13]. The code generation phase produces code for each process by walking the annotated abstract syntax tree while applying this evaluation scheme at each node.

Figure 7 contains the code that this method generates for our example for process zero. Notice that the loops are restricted to those iterations in which process zero needs to be involved. Since process zero owns rows one to thirty-two and columns one to thirty-two of matrices `A`, `R_bar`, and `Sigma`, it owns the locations to be updated and all the values required for iterations

($3 \leq l \leq 32, 3 \leq k \leq 32$). In addition, process zero owns two values that are needed iteration 33 of the outer loop. To determine which iterations a given process must participate in, the set $\{i | f(i) = p \, for \, f \, \epsilon \, evaluators\}$ is computed. Computing this inverse is nontrivial; the subset of the evaluators set that is equal to p for a given iteration, i, is associated with that iteration and is what determines the work done during that iteration. Also notice that, the code for process zero contains a call to `Set_boundary_zones`. The code for this function is not shown. It just fills in the boundary elements for the rows and columns that process zero owns.

For details concerning the how loops and procedure calls are handled and for all of the rules for computing evaluators and participants, we refer the interested reader to the forthcoming dissertation of one of the authors[15].

5.3 Message Optimizations

Programmers use many tricks to decrease the cost of messages and increase parallelism. A compiler must be able to do the same kinds of optimizations to produce good code. We have implemented two of these optimizations in our compiler: *pipelining* and *vectorization* of messages. Message vectorization is crucial for generating good code for SIMPLE.

Using compile-time resolution, the send and receive commands for a non-local reference are inserted into the generated code where the value is needed. This is a fine place for the receive but a non-optimal place for the send. The receiving process can be delayed unnecessarily if the sending process has the value available at an earlier point but has work to do between then and when the send is executed. A better scheme would execute the send as soon as the value is available. The optimization that moves the send to the earliest possible point in the generated code is called *pipelining.* We discuss the need for pipelining in detail in an earlier paper [16].

Vectorization combines messages with a similar source and destination into a single message to reduce overhead. Each message sent incurs a fixed overhead plus a cost per message byte. 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 performed before code generation and may be thought of as *vectorization* of a read operation.

```
Make_AB_North(R_bar, Sigma, a)
real_istructure A, Sigma, R_bar;

{ int k, l;

  double t29, t30, t31, t32, t44, t45,

  set_boundary_zones(A, 0.0) ;

  { for ( l= 3; l<= 32; l= l+ 1)
      { double d;
        for ( k= 3; k<= 32; k= k+ 1)
          { t29 = Sigma[BL(k, l)];
            t30 = R_bar[BL(k-1, l)];
            t31 = A[BL(k-1, l)];
            d = t29 + t30 * (1 - t31);
            t32 = R_bar[BL(k, l)];
            A[BL(k, l)] = t32 / d);
          } ;
        t44 = R_bar[BL(32, l)];
        send(13, &t44, sizeof(double ), BM(33, l));
        t45 = A[BL(32, l)];
        send(16, &t45, sizeof(double ), BM(33, l));
}
```

Figure 7: Compile-time resolution code for `Make_AB_North`.

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 dependencies within the loop. If no such cycle of dependencies exists, the read may be converted to a vector read. This in turn will be converted during code generation to block sends and receives or will be removed by copy elimination, if the process that owns the vector is the same as the process that needs it.

Most of the data dependencies in SIMPLE are either from one time step to the next or from one phase in the computation to a later phase in the computation.[3] Only in the heat conduction routines are there data dependencies between array elements of the same matrix in the same time step. Precomputed values should always be sent as a vector. Our example illustrates this. The array `R_bar` is computed by an earlier routine in the heat conduction phase. By the time `Make_AB_North` is called `R_bar` will already have been computed. A look back at the compile-time resolution code shows that one value from the bottom row in the block is sent to process two in each iteration of the outer loop. A much more sensible implementation would be to send all of the required `R_bar` values to process two in one message.

The code that results from using compile-time resolution augmented with vectorization can be seen in Figure 8. Notice that the whole last row of zero's block of `R_bar` is packed into a vector and sent to process two . Also notice that in the situation where process zero owns both `A[k, l]` and `R_bar[k-1, l]` nothing has changed from the straight compile-time resolution code even though this reference arises from the same original reference as the values being sent.

5.4 Accumulation

The code generation methods presented above implement "do-across" style parallelism. To generate efficient code for all of SIMPLE, a programming paradigm known as *accumulation* or *global combine* is required.

Consider the function, `compute_internal`, in Figure 9. Starting from the code shown, our compiler using just compile-time resolution would gen-

[3]For example, the new position matrices computed by make_position are used by almost all of the subsequent phases in a time step.

```
Make_AB_North(R_bar, Sigma, a)
real_istructure A, Sigma, R_bar;

{ int k, l, t4;

  double t29, t30, t31, t32, t45, T8[30]

  for ( t4= 3; t4 <= 32; t4 = t4+ 1)
    { t7= R_bar[BL(32, t4)];
      T8[VEC_SEND_Local(t4-3)] = t7; };
  send(13, T8, 30* sizeof(double), 2);

  { for (l = 3; l<= 32; l = l+ 1)
      { double d;
        for (k = 3; k<= 32; k = k+ 1)
          { t29 = Sigma[BL(k, l)];
            t30 = R_bar[BL(k-1, l)];
            t31 = A[BL(k-1, l)];
            d = t29 + t30 * (1 - t31);
            t32 = R_bar[BL(k, l)];
            A[BL(k, l)] = t32 / d);
          } ;
        t45 = A[BL(32, l)];
        send(16, &t45, sizeof(double ), BM(33, l));
}
```

Figure 8: Vectorized code for Make_AB_North

```
def compute_internal new_alpha:realarray@block(bs1, bs2, size1 , size2)
                     new_rho:realarray@block(bs1, bs2, size1, size2)
                     new_epsilon:realarray@block(bs1, bs2, size1, size2)
                     :real@P0 =

{ internal:real@P0 = 0;

  { for k = kmin+1 to kmax do
     { for l = lmin+1 to lmax do
        temp1:real@((k-1) div bs1*nb + (l-1) div bs2) =
              (new_rho[k,l] * new_alpha[k,l] * new_epsilon[k, l]);
        next internal = internal + temp1 } }

  in internal
}
```

Figure 9: Accumulation example

erate code that sends all of the values for `temp1` to process zero. Recall that the process, p, that owns the left-hand side of an assignment will perform the work necessary for the right-hand side as well as the assignment itself. All other processes participate by sending any necessary values to process p. In our example process zero will do all of the work because it owns `internal`. The rest of the processes will madly send values to process zero. 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 sum and send that value to process zero where the final sum can be computed. Figure 10 illustrates the difference between the two implementations. This paradigm known as accumulation or global combine is well known to programmers of parallel machines.

Fortunately, accumulation fits very nicely into our model. If the partial values of the circulating (next) variable are not used (ie. the variable is not live in the body of the loop) and if the operation is commutative and associative, then a different rule can be used to determine where the work should be done. Instead of assuming that the owner of the left-hand side

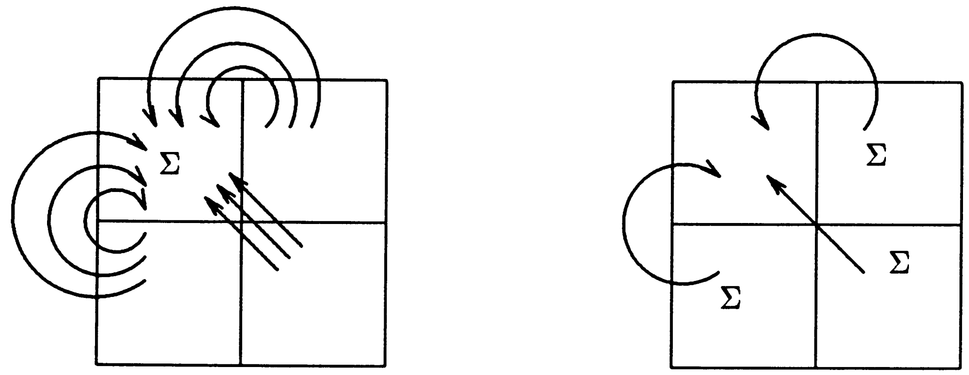


Figure 10: Two methods for computing an accumulation

should do the work for an assignment, the work and the assignment are assigned to the process that appears most often in the participants set of the right-hand side. In addition, the loop is tagged to indicate that the partial values collected by each process should be accumulated at end of the loop. This accumulation can be pushed out to the outermost loop in which the variable is not live to further reduce message traffic. Accumulation has been incorporated into the compiler. Figure 11 contains the code generated for `compute_internal` using code generation augmented to recognize opportunities for accumulation. For larger number of processes, a fan-in of the partially accumulated values would be beneficial.

5.5 Scatter-gather

The coefficient tables used by the pressure/temperature/energy computation are accessed through another table. This raises a subtle issue regarding the distribution of the coefficient table. If the table is large in size, it may have to be distributed across the processes. If so, it will not be possible to predict precisely which processes will need access to which coefficients. This means that send/receive pairs cannot be programmed into the code and it is necessary to simulate shared memory. Since the table is read-only, a second possibility is to give a copy of the table to each process. This is particularly useful when the table is small, which is the case in SIMPLE (12x15). We chose this solution in our implementation.

A second possibility is to perform a gather phase in which processes

```
double compute_internal(new_alpha, new_rho, new_epsilon)
real_istructure new_epsilon, new_rho, new_alpha;

{ int k, l;
  double internal, t12, t13, t14, t17;

  internal = 0;

  /* accumulate local contribution for internal */
  { for ( k = 3; k <= 32; k = k+ 1)
      { double templ;
        for (l = 1; l <= 30; l = l+ 1)
          { t12 = new_rho[BL(k, l)];
            t13 = new_alpha[BL(k, l)];
            t14 = new_epsilon[BL(k, l)];
            templ = t12 * t13 * t14;
            internal = internal + templ; } } }

  /* collect other contributions to the final value for internal */
  recv(319, &t17, sizeof(double));
  internal = internal + t17;
  recv(219, &t17, sizeof(double));
  internal = internal + t17;
  recv(119, &t17, sizeof(double));
  internal = internal + t17; }

  return(internal);
}
```

Figure 11: Code generated by compile-time resolution with accumulators

send requests for the coefficients they want to appropriate processes and respond to requests from other processes for the coefficients they happen to own. It is necessary for processes to synchronize before entering and leaving the gather phase.

5.6 Discussion

We have presented two code generation methods. *Run-time resolution* produces inefficient code but guarantees that we can generate code for any program. *Compile-time resolution* uses partial evaluation to produce more efficient code. A natural question is, "Are there any places in SIMPLE where run-time resolution is needed?" The answer is no. The reason is that except in a few limited situations the array subscripts are very simple. Only the accesses to the coefficient tables in the pressure/temperature/energy computation are too complex for the compiler to analyze. Fortunately, these tables are small enough to place a copy on each process which solves the problem.

The computation of the boundary elements is another issue that might cause concern. The program we started with separated the boundary element computations from the interior element computations. The result was more straightforward inner loops. The quality of the input code definitely effects the quality of the generated code.

6 Results and Analysis

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 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.

We ran a set of experiments using an implementation of one iteration of SIMPLE for a 64x64 grid. With the exception of the coefficient matrices, all of the matrices in the program were decomposed into roughly square

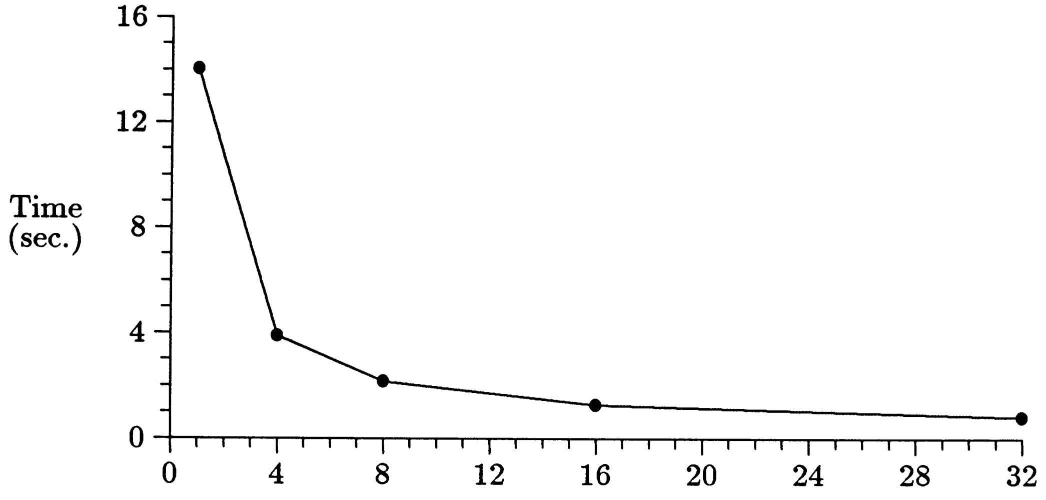


Figure 12: Running times for Simple on a 64 x 64 grid

blocks. Our compiler generates C code[4] for the Intel iPSC/2.

The graph in figure 12 displays the timing results from our experiments. The implementation for the one process case was obtained by mapping of each matrix into a single block of size 64x64. The resulting program has no communication statements and looks quite like the original sequential program.

6.1 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

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

quantities are difficult to obtain because of the primitive nature the time measurement tools on the iPSC/2. Our goal is to develop a model that estimates the running time of a hand-written 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[5] 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[3]. 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. The cost of message that is of length *k**4 bytes is defined by the following equation:

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

For the iPSC/2, $t_{startup} = 350\mu s$ and $t_{send} = 0.8\mu s$ for message under 100 bytes and $t_{startup} = 660\mu s$ and $t_{send} = 1.44\mu s$ for longer messages[3]. From this we can extrapolate to estimate the communication cost for larger numbers of processes. The following table

N	*bs1*	*bs2*	$\sum_{i=1}^{n} T_{msg}^{i}$	Per process cost
4	32	32	207.305	51.826
8	32	16	421.858	52.732
16	16	16	675.932	42.245
32	16	8	1,307.099	40.846

displays the projected communication costs (in milliseconds) for the configurations in question. The four process estimate assumes that a value is sent from a one process to another process at most once and the all the values

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

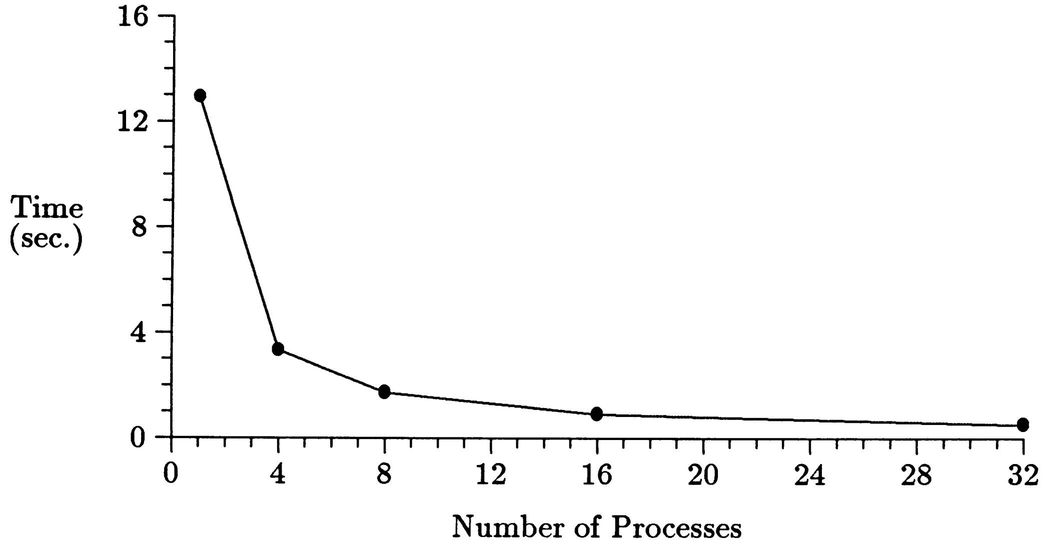


Figure 13: Projected running times for Simple

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 unmeasurable. We can measure the cost of allocating space for arrays. Many functions return arrays as result, 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 beginning. Each individual array allocation requires only a few pointer operations. The following table lists the values we measured for the per process allocation costs.

N	*bs1*	*bs2*	*Per process allocation cost*
4	32	32	482
8	32	16	269
16	16	16	161
32	16	8	108

The allocation costs do not decrease linearly because a few of the arrays are replicated, i.e. each process has a copy.

Once we have an estimate for the total work done in the system, it is easy to model parallel time and expected speedup. The best possible situation would result in a perfect division of the work. Parallel time is therefore estimated as $PT = TPT/N$ where N is the number of processes. The graph in figure 13 shows the projected running times for both decompositions. Speedup is sequential time divided by parallel time:

$$speedup = (1,552,883 * C + 1760)/PT$$

Sequential time is the cost of the floating point operations plus the allocation cost. The graph in figure 14 displays three curves: perfect speedup, the speedup projected by the model, and our speedup using the roughly square block decomposition.

6.2 Comparison of the Results

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 that 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 of the processes slow down to the 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.

Preliminary experiments indicate that using a strip decomposition rather than a roughly square decomposition for the matrices in SIMPLE produces similar results to those reported above.

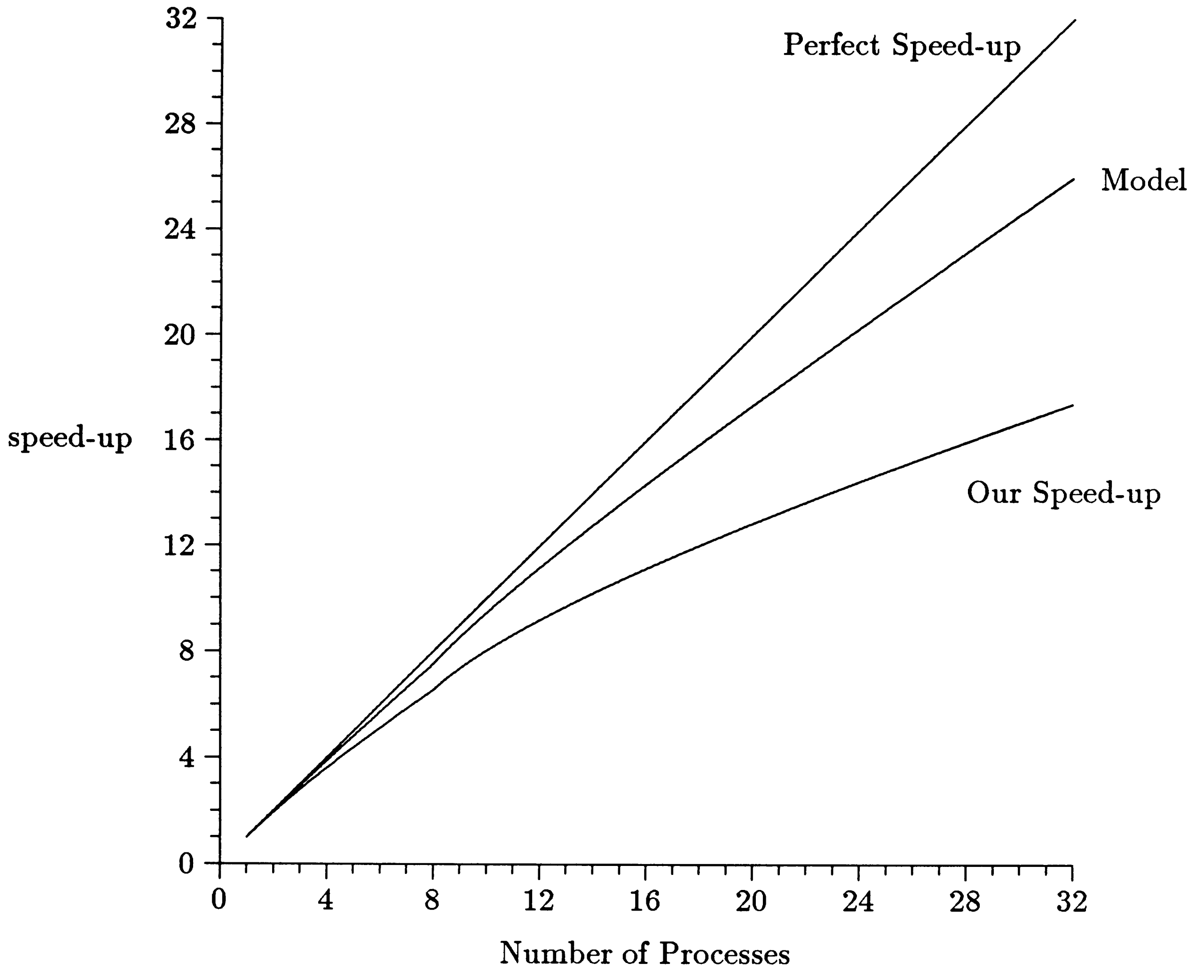


Figure 14: Simple Speedup

7 Summary

Our experiments with SIMPLE have been very beneficial. Our basic methods have been validated on a large program and we have gained insight into how to expand our system to generate better code. The paradigms that programmers employ by hand such as accumulation and scatter-gather are quite useful. As we discussed, accumulation fits quite naturally into our scheme. We have not yet implemented scatter-gather as we do not yet understand where the cross-over point between replicating arrays and using scatter-gather lies.

We have implemented some techniques that reduce the number of messages in the system. Vectorization of messages is the most notable for SIMPLE. Reducing the number of messages sent still requires work. A handwritten program would send fewer messages than code we currently generate. Programmers easily recognize that if a value is used by a process in several different procedures it need only be transmitted once. A compiler must perform interprocedural common subexpression recognition to be able to come the same conclusion.

As we have seen, the heat conduction routines throw a wrench into the nice regular pattern of most of SIMPLE. The amount of work done in this phase is small enough that we decided not to try to shuffle the data around to get better performance. However, it is not hard to imagine cases where the work load involved would be large enough that the cost of data movement would be worth the gain in efficiency. Efficient remapping routines need to be added to our system.

Three other groups have taken similar approaches to the problem of compiling for locality. Koelbel and Mehrotra [11] at Purdue are translating Blaze, a functional language with a forall construct, into an extension of Blaze that includes constructs for explicit process creation, data storage layout, and interprocessor communication and synchronization. They use programmer supplied data decomposition information to schedule forall loops to exploit spatial locality. A group led by Kennedy and Zima at Rice University are studying similar techniques for compiling a version of FORTRAN 77 that includes annotations for specifying a data decomposition, for the Intel iPSC/2. [4] describes a method quite similar to our run-time resolution. They also discuss how existing transformations may be used to improve their generated code. Our methods are equally applicable to FOR-

TRAN. Tseng [18] takes an approach that requires more information from the programmer. In addition to a domain decomposition, the programmer must describe the portion of data that is needed by one process but resides on another process. The basic method uses this information to decide what data to send and when. When the data dependencies in a program can not be sufficiently analyzed by the compiler, the fallback position is to err on the side of sending the data. The advantage to this is that all values can be sent as soon as they become available. This disadvantage is that values that are not needed may be sent needlessly.

Acknowledgements We would like to thank Kattamuri Ekanadham for the original Id version of SIMPLE, Jamey Hicks for running SIMPLE on Gita for us, and the Cornell Theory Center for use of the iPSC/2.

References

[1] F. Berman and et al. Prep-P: a mapping preprocessor for CHiP architectures. In *Proceedings of the International Conference on Parallel Processing*, 1985.

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

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

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

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

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

[7] A. Gottlieb et al. The NYU ultracomputer– designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, 1983.

[8] J. McGraw et al. SISAL: Streams and Iteration in a Single Assignment Language, language reference manual, version 1.2. Technical Report M-146, LLNL, March 1985.

[9] G. Fox and et al. *Solving Problems on Concurrent Processors.* Prentice-Hall, 1988.

[10] T. Hoshino. Invitation to the world of Pax. *Computer*, 19(5), May 1986.

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

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

[13] David Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12), December 1986.

[14] J. Peir and D. Gajski. Towards computer-aided programming for multi-processors. *Parallel Algorithms and Architectures*, 1986.

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

[16] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on programming language design and implementation*, 1989.

[17] H. Stone. *High-performance Computer Architecture.* Addison-Wesley, 1987.

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