International Journal of Parallel Programming; Vol. 20, No. 5, 1991

A Portable Implementation of SIMPLE

Calvin Lin[1] and Lawrence Snyder

Received December 1991; Revised June 1992

This paper describes how the SIMPLE fluid dynamics benchmark is programmed using a new language that promotes portability. Beginning with the mathematics behind the SIMPLE simulation, we present in detail the process by which a portable Orca program is written. We provide evidence that our program is portable across the Intel iPSC/2, the nCUBE/7, the Sequent Symmetry, the BBN Butterfly, and a simulated Transputer-based nonshared memory machine. In addition, this paper illustrates that language constructs can be provided to ease the burden of programming with message passing.

KEY WORDS: Portable parallel programming; SIMPLE; performance results.

1. INTRODUCTION

The importance of portability is not in dispute. Program portability is beneficial because its allows the cost of program development to be amortized over a long lifetime that spans many machines. Portability also allows and encourages program sharing. Unfortunately, portability is not as easily accomplished today as it has been in the past.

Portability in the parallel world is difficult because of the diversity of available architectures. A program that relies on the low-level details of one machine is likely to find those details nonexistent or inefficiently implemented on another machine. While programming at too low a level is problematic, so too is programming at too high a level. A high level of abstraction can be realized on all machines, but the gap between the abstraction and the machine is often too large to be bridged by compilers and runtime systems alone; the result is poor performance.

This paper introduces the Orca programming language. Orca

[1] Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, Washington 98195.

0885-7458/91/1000-0363$06 50/0

programs are portable because they only require facilities that are efficiently realizable on all MIMD multiprocessors. Moreover, certain key aspects of Orca programs are parameterized, yielding programs that are flexible enough to adapt to different architectures. In this paper we describe the Orca specification of SIMPLE. This example demonstrates the flexibility of Orca programs and shows how an appropriate choice of language constructs leads to message passing programs that are clean and concise.

The importance of SIMPLE as a parallel application comes from the substantial body of literature already devoted to its study. SIMPLE is a computational fluid dynamics code that was introduced in 1977 as a benchmark to evaluate new computers.[1] Since its creation it has been widely studied, both to illustrate new programming approaches[2–5] and to demonstrate program performance.[4,6–10] This paper follows both traditions, using SIMPLE to illustrate the Orca language and to demonstrate the portability of SIMPLE across a variety of MIMD machines.

This paper is organized as follows. Section 2 provides background on the Orca language. Section 3 presents the SIMPLE computation, first in terms of the mathematics involved and then in an algorithmic form. Section 4 describes how to program SIMPLE in Orca C. The final three sections give performance results and discuss the Orca C language with respect to ease of programming and program performance.

2. THE ORCA LANGUAGES

The Orca family of languages are based on Phase Abstractions,[11–13] a MIMD message passing model that aims to support scalable and portable scientific applications.

At a high level, a parallel program consists of logical units of computation known as *phases*. A phase is a parallel algorithm: a set of concurrent processes working together to accomplish some common goal. Each process is defined by program text which in the most primitive case differs from sequential code only in the existence of message passing operations. This paper discusses one of many possible languages that can be used to specify these process codes, but rather than provide the programmer with a sequential language with message passing libraries simply added on, Orca C was designed with parallelism in mind. (Another such language, based on Fortran, is under construction. These Orca languages share a common set of abstractions but each attempts to maintain the flavor of the language from which it is derived.) In the parlance of the Phase Abstractions,[12] the individual processes are part of the *X level* of specification, and languages that describe them are *X level* languages. The remaining two

levels of the Phase Abstractions, the *Y level* and *Z level* are introduced later.

As a distributed memory programming language, an Orca program contains global data structures that are logically (and often physically) distributed across the various processors. (For shared memory machines the data may not be physically distributed but performance advantages—in the form of improved locality of reference—may still exist.[14]) For performance reasons it is useful to create a correspondence between individual processes and the data on which they operate. Both of these activities—the decomposition of data and the binding of processes to this data—are supported by *ensembles.*

An ensemble is defined to be a set with a partitioning; the resulting partitions are called *sections.*[12] For example, a *data ensemble* decomposes the global data structures into sections that represent local memory. Similarly, a *code ensemble* specifies the assignment of process instances to sections. Together, these two entities specify which processes operate on which sections of data. Finally, since the processes must communicate, *port ensembles* are used to specify each phase's communication structure. Orca C programs communicate with each other through named ports, so a port ensemble is a partitioned graph that binds ports of one section to those of another.

Once all of the phases have been defined, they can be combined to solve a problem. The invocation of phases is ordered with the same control flow constructs that are typically found in sequential imperative languages. Similarly, phase invocation uses the same syntax as procedure calls in sequential languages.

We can now summarize the structure of an Orca program:

- The *X* level consists of sequential code (but loop-level and instruction-level parallelism are not precluded) that defines the behavior of processes.
- At the *Y* level, ensembles specify how processes are composed to form a coherent parallel algorithm. While a phase may at present have at most one code ensemble, it may consist of multiple data ensembles—one for each global data structure it accesses—and multiple port ensembles—for cases where a phase logically utilizes more than one communication structure. For each section, the process defined by the code ensemble operates on local data as defined by the data ensembles and communicates with other processes through the ports defined by the port ensembles.
- Finally, phases are invoked at the *Z* level to solve the overall problem. A *Z* level program controls the execution sequence of the constituent parallel algorithms.

Subsequent sections illustrate the details of Orca C by describing their use in the SIMPLE computation.

2.1. The *X* Level or Orca C

This section discusses the *X* level features of Orca C in general terms.

Message Passing. Message passing in Orca C is based on ports and invoked by the transmit operator (⇐), for which a port name on the left specifies a send of the data on the righthand side, and a port on the right indicates a receive into the variable on the lefthand side. The semantics are that sends transmit immediately, with data buffered at the destination, and receives remove data from the buffer in order of arrival, blocking on empty. For example, the following code fragment sends the value of the *pressure* variable to whatever section is connected to the *East* port by the port ensemble:

East ⇐ pressure;

The adjacent port might receive this value into a variable, *edgePressure*, with a statement such as the following:

edgePressure ⇐ West;

In addition to support for ensembles and message passing, Orca C possesses several features that facilitate parallel programming at the *X* level, nemaly, array operators, support for boundary conditions, and *fluff* (defined later).

Array Constructs. Array operators provide a concise way to apply a single operator to multiple elements of a set. These operations are defined only for conformable pairs of arrays, that is, arrays with the same number of dimensions and the same number of elements per dimension. Furthermore, Orca C allows for the manipulation of subportions of arrays using *slices*. For example, $a[x:y]$ specifies the xth through yth elements of array a, inclusive. This notion applies to all dimensions of an array. Empty brackets ([ ]) are shorthand for the entire dimension but may only be used in cases where the array bounds are known at compile time. As an example of an aray operation, the following computes the element-wise sum of the b and c arrays and stores the result to the a array.

$$a[\,][\,] = b[\,][\,] + c[\,][\,];$$

Besides assignment and addition, array operators are also supported for the other C arithmetic operators. The order of evaluation for the

constituents of an array operation is not defined. In particular, operations on overlapping areas of memory should be avoided. For example, the semantics of the following are implementation dependent:

$$a[1:10] = a[0:9].$$

Array slices are particularly useful in conjunction with message passing. In addition to improving the clarity of the source code, slices have performance implications. Compilers can easily gather array slices into a single message, thus minimizing the number of messages sent. Without array slices, a programmer must either gather the data explicitly or send the elements of the array as individual components. In either case the meaning of the code is clearer with slices, and the latter case requires the existence of an optimizing compiler if good performance is expected. For example, the following code fragment sends the left column of the a array to the West neighbor as a single message.

$$\text{West} \Leftarrow a[\ ][0];$$

Since this slice is not contiguous in memory, the compiler copies it to temporary storage before sending it to the West port.

Support for Boundary Conditions. Boundary conditions represent a problem in parallel programming because they lead to the proliferation of special case code.[15,16] Orca C provides a mechanism known as *derivative functions* that allows boundary conditions to be specified at the Y level as part of the problem specification. Together, the port ensembles and derivative functions shelter the X level source code from boundary conditions and lead to uniform code. This feature is discussed and illustrated in Section 4.2.

Fluff. The nonshared memory programming paradigm encourages users to program for locality.[13,14] The general approach is to cache values whenever possible: First obtain local copies of remote values, then compute. The Orca term for these cached non-local values is *fluff* (see Fig. 1). This notion is similar to Overlaps[17] and Guard Strips.[18] To ease

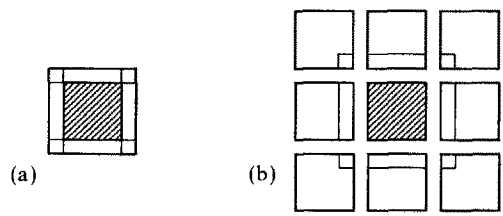

Fig. 1. Fluff. (a): Sections without fluff. (b): A local section with fluff.

the indexing of fluff it is useful to allocate memory that is contiguous to the local data and that can be referenced using the same naming scheme as the local data. Orca C's data ensembles provide a clean mechanism to define fluff. Examples are given in Section 4.2.

2.2. The *Y* Level of Orca C

Phases, or parallel algorithms, are defined at the *Y* level. A phase is defined by the combination of data ensembles, a code ensemble, and a port ensemble. The ensembles making up a phase must all have the same number of sections (partitions), allowing them to be placed into one-to-one correspondence. This establishes the correspondence between data and a process instance, as well as specifying neighbor relationships among sections for communication purposes.

Each section of an ensemble represents a local thread that executes independently. Therefore, through parameterization of critical features of the computation such as the number of data values, the number of processors, etc., the ensembles control the logical concurrency of an Orca program.

2.3. The *Z* Level of Orca C

The *Z* level of Orca C is programmed using the C language. Syntactically, a *Z* level program looks like a standard C program. Semantically, procedure invocation is replaced by phase invocation. Control flow is based on scalar values that are local to the *Z* level—that is, these scalars are not visible to the *X* level processes. The *Z* level program specifies the order in which the various phases are invoked, but there is no implied synchronization among these phases. If synchronization is desired, barrier synchronization can be explicitly specified at the *Z* level. Finally, the *Z* level program can access the data ensembles using the same array constructs that are available at the *X* level. However, there are two differences between ensemble access at the *Z* and *X* levels. First, the *Z* level uses a global indexing scheme. Second, *X* level access is guaranteed to involve only local memory access, which is not the case at the *Z* level. An example of a *Z* level program is given in Fig. 2 and is discussed in Section 4.

2.4. The Ensembles

While the ensembles exist across all three levels of programming, each level sees a different view. For example, the *Z* level sees a global view of

```
Load(x, u, a, ...);
while (error > δ)
{
    Delta (x, rho, J, time, iter);
    Hydro (p, rho, J, ...);
    Heat (x, rho, e, J, ...);
    Energy1 (x, u, ...);
    error := Energy2 (en_error);
}
Output (x, u, ...);
```

Fig. 2. Z level program body for SIMPLE.

the data ensembles, but each X level process sees only local portions of the ensembles. Because of these different views, the Z and X levels can use different indexing schemes to refer to the same data. The local view at the X level is useful because it allows each process to execute the same source code. Functions are provided to map between global and local indices.

Note that ensembles exist permanently and do not "belong" to any of the different programming levels. To limit their scope at the X level, data ensemble are logically passed from the Z to X levels in much the same way that parameters are passed to procedure calls. The mechanism is pass by reference. At present, if two phases require different data partitionings, the programmer must explicitly move the data. For example, a Transpose phase could be used to transpose the elements of a distributed matrix. Details concerning the alignment of local and global views are presented in Section 4.

2.5. The Overall Orca C Program

The overall program structure is given in Fig. 3. The ⟨ *parameter list* ⟩ specifies arguments to the computation, including computation-specific information such as the convergence limit and environment characteristics such as the number of processors on the host machine. This data, plus

```
program <name> (<parameter list>)
    configuration and constraint computations;
    (<configuration parameter list>)
    data ensemble definitions;          /* Y Level */
    port ensemble definitions;
    code ensemble definitions;
    phase definitions;
    process definitions                 /* X Level */
begin
    program body                        /* Z Level */
end.
```

Fig. 3. Schematic of Orca Program Structure.

other information that might be read in from external media, such as the size of the data sets, are input to configuration computations that the programmer defines. These computations determine how the structures of the program are configured to respond to the execution. Typical of a configuration computation is the determination of the number of sections the computation should have, i.e. how much logical concurrency is appropriate for the prevailing conditions. Once computed, the parameters are explicitly given in the ⟨*configuration parameter list*⟩ and are followed by the definition of the parallel program.

3. THE SIMPLE COMPUTATION

The SIMPLE computation simulates the hydrodynamics of a pressurized fluid inside a spherical shell. The state of the simulation is maintained by recording the values of various physical quantities at a number of points inside the shell. As simulated time progresses, these values—representing such entities as pressure, density, viscosity and temperature—are iteratively updated.

The algorithm is based on Lagrangian hydrodynamics, which gives the following set of equations.

$$\frac{d}{dt}(\rho V) = 0 \tag{1}$$

$$\rho \frac{d\vec{u}}{dt} + \vec{\nabla}(p+q) = 0 \tag{2}$$

$$\frac{d\epsilon}{dt} + (p+q)\frac{d\tau}{dt} = 0 \tag{3}$$

$$\frac{d\vec{x}}{dt} - \vec{u} = 0 \tag{4}$$

$$q = q(\rho, \delta u) \tag{5}$$

$$p = p(\rho, \epsilon) \tag{6}$$

$$\frac{\partial \epsilon}{\partial t} = \left(\frac{\partial \epsilon}{\partial \theta}\right)\frac{d\theta}{dt} + \left(\frac{\partial \epsilon}{\partial \tau}\right)\frac{d\tau}{dt} \tag{7}$$

where

$\vec{x}$ is position vector,
$\vec{u}$ is velocity vector,
ρ is mass density,
τ is specific volume,
ϵ is specific internal energy,
q is artificial viscosity,
p is pressure,
θ is temperature,
κ is heat conductivity and
t is time.

A cylindrical coordinate system is used to model the problem state. Because of the spherical symmetry of this problem, the physical domain of the problem is reduced to a quarter of an annular region (Fig. 4a) by first projecting the shell onto a two-dimensional plane and then taking the upper right quadrant of the projected annular region. Consequently, vectors such as velocity have only an r component along the radius direction and a z component along the vertical direction.

In order to solve the equations that simulate the motion of the fluid, both the time and the physical domain are discretized. The time, t, is discretized into a sequence of steps and the physical domain is discretized into a finite number of nodes. For the purposes of computer simulation, this 2D

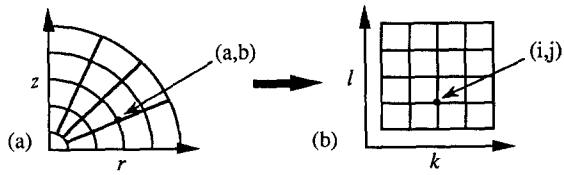

Fig. 4. Mapping of physical domain to computation domain.

projection can be transformed into Cartesian coordinates as shown in Fig. 4b.

Two types of boundary conditions occur in SIMPLE: (1) pressure may be applied to any surface and (2) the component of acceleration normal to the surface may be zero along any surface. In this paper a type 1 boundary condition is chosen for the inner surface and a type 2 boundary condition is chosen for the outer surface.

What follows is the basic algorithm to solve the preceding set of equations.[19] For clarity the code to deal with boundary conditions is omitted here. A detailed description can be found in the literature.[1,4]

In the algorithm below, the notation $V_{i,j}$ is used to denote the physical variable of node (i, j) in the computation grid. It is assumed that the pressure, density, Jacobian and viscosity are constants inside any square surrounded by nodes (i, j), $(i, j+1)$, $(i+1, j+1)$ and $(i+1, j)$ and that they are represented as values in the node at the upper right corner of the square, node $(i+1, j+1)$. Additionally, r and z denote the r and z components of the coordinate, u and w denote the r and z components of the velocity, and a^r and a^z denote the r and z component of the acceleration. Finally, *max_K* and *max_L* refer to the largest indices of the simulated problem space in the k and l dimensions, respectively.

The SIMPLE algorithm.

First compute the initial coordinates of all nodes and initialize the variables of all nodes. Then iteratively carry out the following sequence of steps until the error is sufficiently small:

1. Compute the next time step (δt).
   The standard rule, known as the Courant condition, is used. That is, the time step should not be so large that a speed-of-sound signal can move across a grid cell in one time step. So,

$$\delta t := \min_{i,j} \left[\frac{0.5\, J_{ij}}{C_A [\Delta r_{ij}^2 + \delta r_{ij}^2]^{1/2}} \right]$$

Here, the following notation is used:

$$2\Delta f_{ij} = f_{i,j} + f_{i-1,j} - f_{i,j-1} - f_{i-1,j-1}$$

$$2\delta_{ij} = f_{i,j} + f_{i,j-1} - f_{i-1,j-1} - f_{i-1,j}$$

where f stands for any point quantity such as r, z, u, w. C_A is the local speed of sound and can be computed as follows, where γ is the specific heat ($\gamma = 1.4$ for air):

$$C_A := \sqrt{\gamma \frac{p_{ij}}{\rho_{ij}}}$$

2. Compute the new acceleration (a). The derivative in Eq. 2 is replaced by a contour line integral according to Green's theorem. Furthermore, because the physical domain is discretized, the line integral is reduced to a summation. Let f denote $p + q$.

$$a_{ij}^r := \frac{\begin{pmatrix} f_{i,j}(z_{i-1,j} - z_{i,j-1}) + f_{i,j+1}(z_{i,j+1} - z_{i-1,j}) \\ + f_{i+1,j+1}(z_{i+1,j} - z_{i,j+1}) + f_{i+1,j}(z_{i,j-1} - z_{i+1,j}) \end{pmatrix}}{0.5\,(\rho_{i,j}J_{i,j} + \rho_{i,j+1}J_{i,j+1} + \rho_{i+1,j+1}J_{i+1,j+1} + \rho_{i+1,j}J_{i+1,j})}$$

$$a_{ij}^z := \frac{\begin{pmatrix} f_{i,j}(r_{i-1,j} - r_{i,j-1}) + f_{i,j+1}(r_{i,j+1} - r_{i-1,j}) \\ + f_{i+1,j+1}(r_{i+1,j} - r_{i,j+1}) + f_{i+1,j}(r_{i,j-1} - r_{i+1,j}) \end{pmatrix}}{0.5\,(\rho_{i,j}J_{i,j} + \rho_{i,j+1}J_{i,j+1} + \rho_{i+1,j+1}J_{i+1,j+1} + \rho_{i+1,j}J_{i+1,j})}$$

3. Compute the new velocity (**u**) and new coordinates (**x**).

$$\mathbf{u}_{i,j} := \mathbf{u}_{i,j} + \delta t\, \mathbf{a}_{i,j}$$

$$\mathbf{x}_{i,j} := \mathbf{x}_{i,j} + \delta t\, \mathbf{u}_{i,j}$$

4. Compute the new Jacobian (J) and volume of revolution (new_S).

$$tmp\_J1_{i,j} := \tfrac{1}{2}[r_{i,j}(z_{i,j-1} - z_{i-1,j}) + r_{i,j-1}(z_{i-1,j} - z_{i,j}) + r_{i-1,j}(z_{i,j} - z_{i,j-1})]$$

$$tmp\_J2_{i,j} := \tfrac{1}{2}[r_{i,j-1}(z_{i-1,j-1} - z_{i-1,j}) + r_{i-1,j-1}(z_{i-1,j} - z_{i,j-1}) + r_{i-1,j}(z_{i,j-1} - z_{i-1,j-1})]$$

$$J_{i,j} := temp\_J1_{i,j} + tmp\_J2_{i,j}$$

$$old\_S_{i,j} := new\_S_{i,j}$$

$$new\_S_{i,j} := \tfrac{1}{3}[(r_{i,j} + r_{i,j-1} + r_{i-1,j})\, tmp\_J1_{i,j} + (r_{i,j-1} + r_{i-1,j-1} + r_{i-1,j})\, tmp\_J2_{i,j}]$$

5. Compute the new density (ρ) and artificial viscosity (q).

$$\rho_{i,j} := \rho_{i,j} \frac{old\_S_{i,j}}{new\_S_{i,j}}$$

$$tmp1 := \begin{cases} \dfrac{[\Delta r\, \delta w - \Delta z\, \delta u]^2}{\Delta r^2 + \Delta z^2} & \text{if } [\,] < 0 \\ 0 & \text{otherwise} \end{cases}$$

$$tmp2 := \begin{cases} \dfrac{[\Delta u \delta z - \Delta w\, \delta r]^2}{\delta r^2 + \delta z^2} & \text{if } [\,] < 0 \\ 0 & \text{otherwise} \end{cases}$$

$$q_{i,j} := 1.5\rho_{i,j}(tmp1 + tmp2) + 0.5\rho_{i,j} C_A \sqrt{tmp1 + tmp2}$$

6. Compute the new energy (ε) and pressure (p).

$$\varepsilon_{i,j} := \varepsilon_{i,j} - (p_{i,j} + q_{i,j})\, delta\_\tau_{i,j}$$

$$tmp_{i,j} := (\gamma - 1)\, \varepsilon_{i,j} \rho_{i,j}$$

$$\varepsilon_{i,j} := \varepsilon_{i,j} - (\tfrac{1}{2}(tmp_{i,j} + p_{i,j}) + q_{i,j})\, delta\_\tau_{i,j}$$

$$p_{i,j} := (\gamma - 1)\, \varepsilon_{i,j} \rho_{i,j}$$

where *delta_*τ is the difference between the new τ (specific volume) and the τ in the previous iteration. $\tau = 1/\rho$.

7. Compute the new temperature (θ) and heat (*heat*).
The heat equation (Eq. 7) can be separated into 2 equations, one for the k dimension and one for the l dimension. Since the physical domain is discretized, both equations can be reduced to a set of linear equations with a tridiagonal matrix. Two passes are needed to solve this set of linear equations. The first pass transforms the tridiagonal matrix to an upper-right triangle matrix and the second pass directly computes the solution beginning with the last equation.

Heat flow into the shell is also calculated in this step.

for each pair (i,j) **do**

$$J_{i,j} := (r_{i,j} - r_{i,j-1})(z_{i,j-1} - z_{i,j}) - (z_{i,j-1} - z_{i,j})(r_{i,j} - r_{i-1,j}) \quad (8)$$

$$\sigma_{i,j} = 0.1\rho_{i,j} r_{i,j} J_{i,j}/\delta t \quad (9)$$

$$CC_{i,j} := 0.0001\theta_{i,j}^{5/2}/J_{i,j} \quad (10)$$

$$KJ_{i,j} := \frac{CC_{i,j} CC_{i,j+1}}{CC_{i,j} + CC_{i,j+1}} \quad (11)$$

$$R_{i,j} := (r_{i,j} + r_{i,j-1})\left((r_{i,j} - r_{i,j-1})^2 (z_{i,j} - z_{i,j-1})^2\right) KJ_{i,j} \quad (12)$$

end for
for j := 0 **to** max_K **do**
  for each i **do**

$$D_{i,j} := \sigma_{i,j} + R_{i,j} + R_{i,j-1}(1 - \alpha_{i,j-1}) \tag{13}$$

$$\alpha_{i,j} := R_{i,j} / D_{i,j} \tag{14}$$

$$\beta_{i,j} = \frac{R_{i,j-1}\beta_{i,j-1} + \sigma_{i,j}\theta_{i,j}}{D_{i,j}} \tag{15}$$

  end for

end for
for j = max_K **to** 0 **do**
  for each i **do**

$$\theta_{i,j} = \alpha_{i,j}\theta_{i,j+1} + \beta_{i,j} \tag{16}$$

  end for
end for
for i := 0 **to** max_L **do**

$$heat_{i,0} := (\theta_{i,0} - \theta_{i,1})\, R_{i,0}\, \delta t \tag{17}$$

end for

Repeat the calculations of statements 13–16 for the l dimension.

8. Compute the energy (*energy*) and work (*work*).

for each pair (i,j) **do**

$$m_{i,j} := \rho_{i,j} S_{i,j}$$

$$energy_{i,j} := \epsilon_{i,j} m_{i,j} + \frac{1}{8}(m_{i,j} + m_{i,j+1} + m_{i+1,j+1} + m_{i+1,j})(u_{i,j}^2 + w_{i,j}^2)$$

$$tmp_{i,j} := \frac{1}{4}\delta t\,(p_{i,j} - p_{i,j+1})\,(r_{i,j-1} - r_{i,j})\,[(r_{i,j} - r_{i,j-1})(u_{i,j} + u_{i,j-1}) - (z_{i,j} - z_{i,j-1})(u_{i,j} + u_{i,j-1})]$$

$$work_{i,j} := \begin{cases} -tmp_{i,j} & \text{if (i,j) is on the west boundary of the computation grid} \\ 0 & \text{otherwise} \end{cases}$$

end for

9. Compute the error.

$$total\_energy := \sum_{i,j} energy_{i,j}$$

$$total\_work := \sum_{i} work_{i,0}$$

$$total\_heat := \sum_{i} heat_{i,0}$$

$$error := total\_energy - total\_work + total\_heat$$

The Parallel Algorithm.

Each of these nine steps could be a phase of a Z level program. Closer inspection, however, reveals that this problem is naturally composed of just five phases. Note that these equations only involve local neighbor values. For example, the first step computes a global minimum; this computation can be achieved using only communication with neighboring data points. In the second step, the acceleration at point (i, j) is based on the (i, j) value and six neighbors: $(i+1, j)$, $(i+1, j+1)$, $(i, j+1)$, $(i-1, j)$, $(i-1, j-1)$ and $(i, j-1)$, which we refer to as the North, NorthEast, East, South, and SouthEast neighbors, repectively. Step 4 requires the West, South, and SouthWest neighbors to compute the value of the Jacobian, while Step 7 requires four neighbors. Finally, Step 9 accumulates the sum of the error values over the entire data space.

When this algorithm is parallelized, each type of data dependency induces a communication pattern. Since Steps 2 and 4 share the same data dependencies, and since Steps 3, 5, and 6 have no dependencies, these steps can be combined into a single phase, yielding the following phases, where each phase is characterized by a single data dependency pattern (see Fig. 5):

Delta Phase:	Step 1
Hydro Phase:	Steps 2, 3, 4, 5 and 6
Heat Phase:	Step 7
Energy1 Phase:	Step 8
Energy2 Phase:	Step 9

Note that this algorithm falls in the class of CAB algorithms as described by Nelson.[20] The Hydro, Heat and Energy1 phases are compute phases, while the Delta and Energy2 phases are Aggregate and Broadcast phases.

4. THE SIMPLE PROGRAM

Having presented the SIMPLE algorithm in high level terms, the Orca Z level program follows naturally (see Fig. 2). The computation

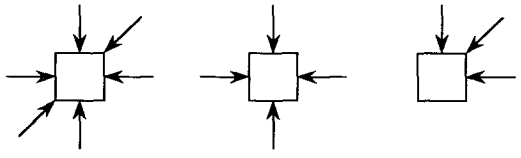

Fig. 5. Data dependencies in SIMPLE.

begins by invoking a phase that loads the initial problem state. Then, a series of five phases—*Delta*, *Hydro*, *Heat*, *Energy1*, and *Energy2*—is iteratively invoked until convergence is achieved, at which point the output of the simulation is produced. Since we presume that input and output are phases provided by the system, the programmer's job is to define the five computational phases.

We proceed by defining the data structures required by this program, since these apply to all phases. Then, the specifics of each individual phase will be addressed in turn.

4.1. Data Ensembles

Most programs will require many data ensembles for each phase. Since all the ensembles of a phase must have the same partitioning, it is most convenient to define all the data structures first, then define a single partitioning, and finally apply the partitioning to all data structures to form ensembles.

The arrays used to capture the state of the SIMPLE computation are given in Table I along with short descriptions of what they represent. The elements of the x, u and a arrays are two element double precision vectors representing the r and z components in the physical domain. Not all items

Table I. Array Values in SIMPLE

Type	Variable	Description
Vector	*x*[*rows*][*cols*]	Position vectors
Vector	*u*[*rows*][*cols*]	Velocity vectors
Vector	*a*[*rows*][*cols*]	Acceleration vectors
double	*rho*[*rows*][*cols*]	Fluid density
double	*p*[*rows*][*cols*]	Fluid pressure
double	*q*[*rows*][*cols*]	Fluid artificial viscosity
double	*delta_tau*[*rows*][*cols*]	Difference in specific volume
double	*e*[*rows*][*cols*]	Energy
double	*theta*[*rows*][*cols*]	Temperature
double	*J*[*rows*][*cols*]	Jacobian of transformation
double	*S*[*rows*][*cols*]	Volume of revolution
double	*delta_t*[*rows*][*cols*]	Time step
double	*heat*[*rows*][*cols*]	Heat flow across boundary
double	*en_error*[*rows*][*cols*]	Energy check error (Energy1)
double	*int_en*[*rows*][*cols*]	Internal energy (Energy1)
double	*kin_en*[*rows*][*cols*]	Kinetic energy (Energy1)
double	*work*[*rows*][*cols*]	Work done at boundary (Energy1)
double	*mass*[*rows*][*cols*]	Zonal mass (Energy1)

are used in each phase. For example, the last five items are needed only in the Energy1 phase.

Our parallel implementation partitions the arrays into contiguous two-dimensional subarrays (blocks). This choice reflects two assumptions: First, contiguous blocks give the greatest locality of reference; second, because data dependencies are local in nature, blocks will minimize the amount of communication among sections. While these assumptions may not hold universally, they have been empirically confirmed for the execution of SIMPLE on a number of machines.[21] Further discussion on data partitioning is given in Section 7.

The block partitioning of the pressure array is specified with the following data ensemble declaration:

$$\textit{partition} \quad \textit{block}[\bar{r}][\bar{c}] \qquad \textit{double } p[\textit{rows}][\textit{cols}];$$

which states that the array p has global dimensions $rows \times cols$ and will be partitioned onto a section array (process array) of size $\bar{r} \times \bar{c}$. Here, *partition* is a keyword for ensemble declarations and *block* is simply a name used to identify this partition. Figure 6 shows graphically how the pressure array is converted into an ensemble. (Section 6.1 shows how an alternate decomposition is declared.)

A property of this global view of data decompositions is that local sections are implicitly defined to have size $s \times t$, where $s = rows/\bar{r}$ and $t = cols/\bar{c}$. (If $\bar{r}$ does not divide *rows* evenly, some sections will have $s = \lfloor rows/\bar{r} \rfloor$ while others will have $s = \lceil rows/\bar{r} \rceil$. The value of t is treated analogously.) This means the X level processes contain no assumptions about the data decomposition; thus the program scales in both the number of logical processors and in the problem size. These meanings of *rows*, *cols*, $\bar{r}$, $\bar{c}$, s, and t are established in the *configuration* portion of the program, and we will refer to these variables throughout this paper.

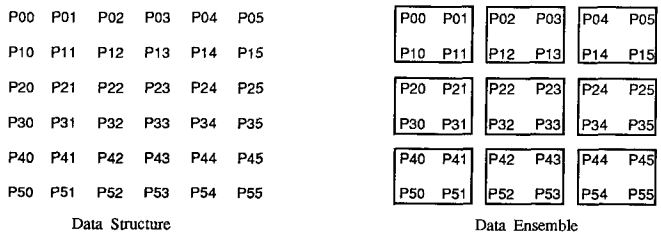

Fig. 6. The Pressure Array, P, and its ensemble, where $rows = cols = 6$, $\bar{r} = \bar{c} = 3$, $s = t = 2$.

Table II. Scalar Values for SIMPLE

Type	Variable	Description
double	*time*	Time
int	*iter*	Current iteration
double	*bound_p*	Pressure at the inner shell

This block partitioning is applied to each of the arrays of Table I. In addition to the arrays, three global scalar values are used (see Table II). It is helpful to assign a copy of the global scalars to each section for use in local computations. The last item in Table II is used only in the Hydro phase and then only for computations along the "west wall."

4.2. The Delta Phase

The Delta phase computes *delta_t*, the size of the next time step of the simulation, by computing the value of *delta_t* at each point and then taking the minimum of these values. To compute this minimum value over all sections, the sections are organized to form a binary tree where minimum *delta_t* values are passed up the tree starting from the leaves. See Fig. 7.

Port Ensemble. The following ports are declared for the Delta phase as part of the *Y* level specification:

$$Delta.portnames \leftrightarrow P, L, R$$

This binds each section's ports to the (possible different) names used in the process' port declaration. The pairing of port names to define a binary tree is specified as shown next. This syntax is only intended to be one of several methods of specifying the communication graph. In the future, rather than defining the graph is this textual manner, we envision using tools—

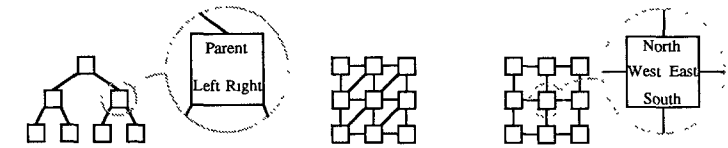

Fig. 7. Communication graphs and port ensembles for SIMPLE.

graphical (such as ParaGraph[22]) and otherwise—to produce these specifications.

$$Delta[i].port.R \leftrightarrow Delta[2*i].port.P \qquad \text{where} \quad 0 \leqslant i < \frac{\bar{r} \times \bar{c}}{2} - 1$$

$$Delta[i].port.L \leftrightarrow Delta[2*i+1].port.P \qquad \text{where} \quad 0 \leqslant i < \frac{\bar{r} \times \bar{c}}{2} - 1$$

Note that this port ensemble declaration associates only a subset of the ports, namely, those that are connected as in Fig. 7. The remaining ports, those on the boundary of the problem space, can be bound to *derivative functions*. These functions compute the boundary conditions using data local to the section. For example, the specification

$$Delta[i].port.L\ receive \leftrightarrow Largest\_Value(\ ) \quad \text{where} \quad \frac{\bar{r} \times \bar{c}}{2} - 1 \leqslant i < Processors$$

$$Delta[i].port.R\ receive \leftrightarrow Largest\_Value(\ ) \quad \text{where} \quad \frac{\bar{r} \times \bar{c}}{2} - 1 \leqslant i < Processors$$

states that for leaf sections, a "receive" from the *Left* port will return the value computed by the *Largest_Value*( ) function. In this case, the derivative function is trivial:

```
double Largest_Value( )
{
    return (MaxDouble);
}
```

Observe that derivative functions lead to code uniformity. In this example a single X level source program suffices for all processes despite the fact that not all processes have the same number of children. Of course, uniformity could be achieved by having the X level program check for the existence of children, but such tests for special cases add clutter, increase compile and/or execution time, and can be error prone. Furthermore, as will be shown in the next section, even modestly more complicated boundary conditions lead to the proliferation of special case code.

Finally, we mention that in a mature version of Orca C, global reduction operators such as Max, Min, and Sum would be implemented as primitives of the Z level language, greatly simplifying this entire phase (see Section 4.6).

Code Ensemble. Having defined the data and port ensembles, the next task is to provide processes that operate on each section. This is the role of the code ensemble (see Fig. 8).

$$Delta[i].code := xDelta(\ );\qquad \text{where}\quad 0 \leqslant i < \bar{r} \times \bar{c}$$

This declaration specifies that each of the $\bar{r} \times \bar{c}$ sections is assigned an instance of the *xDelta*( ) code, which is shown in Fig. 9.

Several features of the X level code are noteworthy:

- *parameters*—The arguments to the process are formals that establish a correspondence between local variables and the sections of the ensembles. For example, the local x array will be bound to a block of ensemble x.
- *fluff*—The size of the local array is based on the size of the data ensemble and is thus logically an input to the process. Furthermore, the local array may contain fluff, as shown by the local declaration $x[-1:s][-1:t]$ and the formal declaration $x[1:s][1:t]$, where the upper bounds of these array declarations are inclusive. Thus the local x array contains extra rows and columns to hold values from neighboring sections. The data is aligned so that extra rows and columns are placed at the top, bottom, left, and right of the ensemble data, and the compiler checks that these values do not exceed the amount of fluff that is declared globally at the Y level. This language support for fluff allows all parts of the array, including fluff, to be accessed with the same indexing scheme.
- *local computation*—The logic of the process is essentially the sequential computation of the original program. A sequential execution can be obtained by setting the number of partitions to be one.
- *array operators*—The last line of the program shows an example of an array assignment where all elements of the *delta_t* array are assigned the same value.

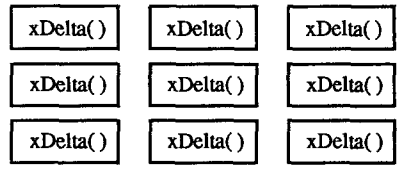

Fig. 8. The code ensemble for the Delta phase.

```
xDelta(x[0 s−1][0.t−1], rho[0 s−1][0:t−1], J[0:s−1][0 t−1], delta_t[0·s−1][0 t−1], time, iter)
    double  x[−1:s][−1 t];              /* fluff of 1 on all four sides */
    double  rho[0:s][0 t];              /* fluff of 1 along two sides */
    double  J[0.s][0.t];
    double  delta_t[0.s−1][0:t−1];      /* no fluff */
    double  time,
    int     iter;
    port    Parent, Left, Right;
{
    int     i, j;
    double  ca;         /* speed of sound */
    double  temp,   r_del,   r_delta   lm_delta_t,

    /* Compute delta_t */
    for (i=0; i<s; i++)
    {
        for (j=0, i<t; i++)
        {
            ca = sqrt(GAMMA * p[i][j]  rho[i][j]); /* GAMMA: a constant for specific heat of air */
            r_del=   x[i+1][j+1] r + x[i][j+1].r − x[i+1][j].r − x[i][j].r,
            r_delta= x[i+1][j+1] r + x[i+1][j] r − x[i][j] r − x[i][j+1] r;
            delta_t[i][j] = C0 * J[i][j]  (ca * sqrt(r_del * r_del + r_delta * r_delta));
        }
    }

    /* Compute the local minimum */
    lm_delta_t = delta_t[0][0],
    for (i=0; i<s; i++)
    {
        for (j=0; i<t, i++)
        {
            if (delta_t[i][j] < lm_delta_t) lm_delta_t = delta_t[i][j],
        }
    }

    /* Compute the global minimum */
    temp      <== Left;                 /* receive */
    lm_delta_t  =  Min(temp, lm_delta_t);
    temp      <== Right;                /* receive */
    lm_delta_t  =  Min(temp, lm_delta_t);
    Parent    <== lm_delta_t,           /* send */

    /* Broadcast the result */
    lm_delta_t <== Parent;              /* receive */
    Left      <== lm_delta_t;           /* send */
    Right     <== lm_delta_t,           /* send */

    delta_t[][]  =  lm_delta_t;         /* array assignment */
}
```

Fig. 9. *X* level code for the Delta phase.

The Complete Phase. To summarize, the data ensembles, the port ensemble, and the above code ensemble collectively define the Delta phase. With this specification and the above definition of the corresponding *X* level code, the Delta phase can now be compiled for execution. Upon execution, a number of sections will exist as specified by parameters to the program, and these sections are logically connected to form a binary tree.

Each section is assigned to a processor for execution and contains one process that performs the code of Fig. 9. The end result is a parallel algorithm that computes the global minimum of all *delta_t* values.

4.3. The Hydro Phase

This phase calculates the new position, velocity, acceleration, Jacobian, viscosity, density, pressure, and energy of each node after the next time step. The computation is divided into two steps. In the first step density, pressure, viscosity and the Jacobian are passed to the West, SouthWest, and South neighbors (see Fig. 10a). These values are used to compute the new acceleration, velocity and position of each node. In the second step, the newly computed position and velocity values are sent to the North, NorthEast, East, South and West neighbors so that they may be used to compute the new values of the Jacobian, viscosity, density, pressure and energy at each node (see Fig. 10b).

Port Ensemble. The data motion shown in Fig. 10 leads to an overall communication structure for this phase that is a 6-mesh, as shown in Fig. 7 and specified by the ensemble here.

$$Hydro\,.\,portnames \leftrightarrow N, NE, E, S, SW, W$$

This Y level declaration is bound with the X level declaration of ports using positional correspondence. The X level declaration is shown in Fig. 13 and is reproduced here. Note that the Y level port names (stated earlier) and the X level port names (next) can differ.

```
xHydro( p[0:s-1][0:t-1], rho[0:s-1][0:t-1], J[0:s-1][0:t-1]...bound_p)
        ...
    port  North, NorthEast, East, South, SouthWest, West;
{
        ...
}
```

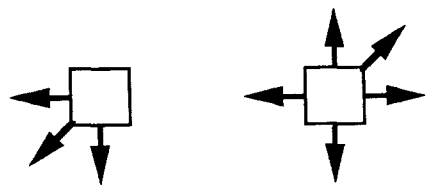

Fig. 10. Data motion in the Hydro phase.

This binding provides a level of indirection that allows X level codes of different phases to share port ensembles if their communication structures are identical. This is important because it allows X level code to be developed in isolation while still allowing them to reuse existing port ensembles.

The pairing of port names to define a communication channel is specified as follows:

$$Hydro[i][j].port.N \leftrightarrow Hydro[i-1][j].port.S \quad \text{where} \quad 1 \leqslant i < \bar{r}, \quad 0 \leqslant j < \bar{c}$$

$$Hydro[i][j], port.W \leftrightarrow Hydro[i][j-1].port.E \quad \text{where} \quad 0 \leqslant i < \bar{r}, \quad 1 \leqslant j < \bar{c}$$

$$Hydro[i][j-1], port.NE \leftrightarrow Hydro[i-1][j], port.SW \quad \text{where} \quad 1 \leqslant i < \bar{r}, \quad 1 \leqslant j < \bar{c}$$

The boundary conditions are set by binding the following functions to ports on the edges.

$Hydro[s][i].port.N\,(positionR)$ $receive \leftrightarrow North\_xr(\,)$; where $0 \leqslant i < \bar{c}$

$Hydro[i][t].port.E\,(positionR)$ $receive \leftrightarrow East\_xr(\,)$; where $0 \leqslant i < \bar{r}$

$Hydro[i][0].port.W\,(positionR)$ $receive \leftrightarrow West\_xr(\,)$; where $0 \leqslant i < \bar{r}$

$Hydro[0]][i].port.S\,(positionR)$ $receive \leftrightarrow South\_xr(\,)$; where $0 \leqslant i < \bar{c}$

$Hydro[s][t].port.NE\,(positionR)$ $receive \leftrightarrow NE\_xr(\,)$;

$Hydro[0][0].port.SW\,(positionR)$ $receive \leftrightarrow SW\_xr(\,)$;

The implementation of the *East_xr( )* derivative function is shown in Fig. 11; the other boundary functions are similar. This example illustrates the use of typed messages, which in Orca C are called *flavors* to differentiate them from types of the language. Flavors provide a way to define *types* for messages. They are needed because the X level code typically uses a single port to transmit different kinds of data. For example, Fig. 12 shows four different receives on the East port, each expecting to receive a *double*

```
East_xr()
{
    return (x[][t].r + (x[][t].r - x[][t-1].r));
}
```

Fig. 11. Example of derivative function definition.

```
/* The X level code: */
for (i=1; i<s+1; i++)
{
    x[i][t].r <== (positionR) East;      /* Expecting a message of flavor positionR */
    x[i][t].z <== (positionZ) East;
    u[i][t].r <== (velocityR) East;
    u[i][t].z <== (velocityZ) East;
}
```

Fig. 12. Implicit invocation of derivative functions in X level code.

but each representing a logically different kind of data; these different logical types are the message's flavor. In this case, if there were no flavors there would be no way to differentiate the types of the different messages. Flavors are currently defined by using C's typecast mechanism. The above example shows how flavors are specified when binding derivative functions to ports; this flavor specification is required whenever multiple derivative functions returning the same language type are bound to the same port of the same phase, that is, whenever there is ambiguity as to which derivative function to use.

```
xHydro(p[0:s-1][0:t-1], rho[0:s-1][0:t-1], J[0:s-1][0:t-1] ... bound_p)
    double   p[0:s][0:t];
    double   rho[0:s][0:t];
    double   J[0:s][0:t];
        . . .
    double   bound_p;
    port     North, NorthEast, East, South, SouthWest, West;
{
    int      i, j;
    double   denom;

    /* Receive from the East a column of the rho array */
    /* and place it in the rightmost column of rho. */
    rho[0:s][t] <== East;                          /* array slice */

    /* other communication . . . */

    /* Compute acceleration */
    for (i=0; i<s; i++)
    {
        for (j=0; j<t; i++)
        {
            denom = (rho[i][j]     * J[i][j]     + rho[i][j+1] * J[i][j+1] +
                     rho[i+1][j+1] * J[i+1][j+1] + rho[i+1][j] * J[i+1][j]) / 2;
            . . .
        }
    }
    /* other computation . . . */
}
```

Fig. 13. Sketch of X level code for the Hydro phase.

Again, we emphasize that a more sophisticated programming environment will provide alternate methods of specifying the port ensembles and boundary conditions.

Code Ensemble. The code ensemble for this phase is identical to that of the Delta phase except each section is assigned an instance of the *xHydro*( ) function instead of the *xDelta*( ) function. The *xHydro*( ) code is too large to be given in complete detail, but Fig. 13 shows a schematic of the process code. Note the following features:

- *array slices*—Slices are both a notational convenience and an efficiency optimization. These provide a clean way to refer to an entire row (or in general, a *d*-dimensional block) of data. Furthermore, when slices are used in conjunction with the transmit operator (⇐), the compiler can perform "message vectorization" to send the entire block as a single message, thus minimizing the number of messages transmitted.
- *no special case code*—Typically, processes on the edge of the processor array must be treated separately. In Orca C, boundary conditions are handled by derivative functions and port ensembles, reducing the need for special case code.

4.4. The Heat Phase

The Heat phase calculates the temperature and heat of each node after the next time step. Because the heat equation is separable in two dimensions, the computation of the heat phase is divided into two steps. The first step solves the equation in the k direction and the second step solves the equation in the l direction.

Port Ensemble. The data motion for the Heat phase is depicted in Fig. 14. Because of data dependencies between loops, each sweep must be completed before the next one begins, which means that this phase will see limited speedup after parallelization. This data motion requires the following port declaration and port ensemble.

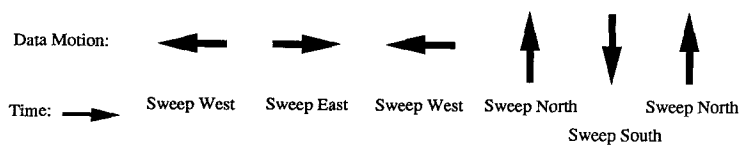

Fig. 14. Data motion in the Heat phase.

$$Heat.portnames \leftrightarrow N, E, S, W$$

$$Heat[i-1][j].port.S \leftrightarrow Heat[i][j].port.N \quad \text{where } 1 \leqslant i < \bar{r},\ 0 \leqslant j < \bar{c}$$

$$Heat[i][j-1].port.E \leftrightarrow Heat[i][j].port.W \quad \text{where } 0 \leqslant i < \bar{r},\ 1 \leqslant j < \bar{c}$$

Code Ensemble. The code ensemble is conceptually identical to those of the previous phases. A high-level description of the X level code is presented in Fig. 15.

4.5. The Energy1 Phase

The purpose of the Energy1 phase is to compute the energy and work and to check that energy has been conserved in the calculations. The port

```
xHeat()
    port    North, East, South, West;
{
    loop                                /* West-East Sweep */
    {
        receive R from West;
        compute equations 13, 14, 15; /* See Section 3 */
        send R to the East;
    }

    loop                                /* East-West Sweep */
    {
        receive theta from East;
        compute equations 16,           /* See Section 3 */
        send theta to the West;
    }

    Compute heat flow at West Border of the Grid,

    loop                                /* South-North Sweep */
    {
        receive R from South;
        compute equations 13, 14, 15; /* See Section 3 */
        send R to the North;
    }

    loop                                /* North-South Sweep */
    {
        receive theta from North;
        compute equations 16;           /* See Section 3 */
        send R to the South;
    }

    Compute heat flow at South Border of the Grid;

}
```

Fig. 15. Sketch of X level code for the Heat phase.

ensemble for the Hydro phase can be reused here because these phases have the same communication structure. Only the X level code changes. A major portion of this code is shown in Fig. 16.

4.6. The Energy2 Phase

Finally, the Energy2 phase sums the error of each node to compute the total error in the SIMPLE computation. Like the Delta phase, the Energy2 phase performs an aggregate and broadcast. This phase can thus share the port ensemble of the Delta phase. The X level code differs chiefly in the use of addition—rather than minimum—as the global reduction operator.

As mentioned in Section 4.2, global reduction operators are frequently used in scientific computations and a mature version of Orca will provide these as primitives of the language. Figure 17 shows how a global reduction operator simplifies the Energy2 phase to a single statement of the Z level program. Here the *GlobalSum* operator accepts the *en_error* ensemble as a parameter and returns the result in the same ensemble. Note that this

```
xEnergy1(x[0 s-1][0:t-1], u[0 s-1][0 t-1],    mass[0 s-1][0 t-1])
    Vector  x[-1 s][-1 t],
    Vector  u[-1 s][-1 t],
    double  rho[0 s][0 t],

    double  mass[0 s][0 t],
    port    North, NorthEast, East, South, SouthWest, West,
{
    int     i, j,

    for (i=0, i<s, i++)
    {
        for (j=0, i<t, i++)
        {
            mass[i][j] = rho[i][j] * S[i][j][0],
        }
    }

    /* Send left column of mass[][] West */
    West          <==   mass[0 s-1][0].
    mass[0 s][t]  <==   East,

    /* Send bottom-left corner of mass[][] SouthWest */
    SouthWest     <==   mass[0][0],
    mass[s][t]    <==   NorthEast,

    /* Send bottom row of mass[][] South */
    South         <==   mass[0][0 t-1],
    mass[s][0 t]  <==   North,

    /* Compute internal and kinetic energy */
    for (i=0, i<s, i++)
    {
        for (j=0, i<t, i++)
        {
            int_en[i][j] = e[i][j] * mass[i][j],
            kin_en[i][j] = ((mass[i][j] + mass[i][j+1] mass[i+1][j+1] + mass[i+1][j]) /8) *
                           (u[i+1][j+1] r * u[i+1][j+1] r + u[i+1][j+1] z * u[i+1][j+1] z),
        }
    }
    /* Compute work, heat and energy      */
}
```

Fig. 16. Sketch of X level code for the Energy1 phase.

```
Load(x, u, a, ..),
while (error > δ)
{
    Delta (x, rho, J, time, iter);
    Hydro (p, rho, J, . .);
    Heat (x, rho, e, J, .. );
    Energy1 (x, u, . .),
    error := Global_Sum (en_error);
}
Output (x, u, ...);
```

Fig. 17. The Energy2 phase using global reduction.

phase has no corresponding *user-defined* Y or X level code. Furthermore, these operators provide a high-level abstraction whose implementation could be optimized once for each individual machine.

5. PROGRAMMABILITY ISSUES

In SIMPLE, the processes forming each phase are instances of a single process. For example, the Delta phase uses instances of the *xDelta*( ) process. Thus, SIMPLE appears not to require the full MIMD capability of code ensembles—where the instances can be instances of different processes—but requires only the Single Program Multiple Data capability. This apparent uniformity derives largely from mechanisms provided by Orca C.

Typically, processes on the edge of the processor array must be treated separately. A receive into the East port must be conditionally executed because processes on the East edge have no eastern neighbors. (Although our reference to the "receive" operation implies a message passing language, shared memory programs also have to deal with these special cases.) Isolated occurrences of these conditionals pose little problem, but since in SIMPLE there can be up to nine different cases—depending on which portions of the boundaries are contained within a process—these conditional cases can lead to convoluted code.[15,16] For example, suppose a program in its conditional expression assumes that the process is either a NorthEast, East, or SouthEast section, as shown here:

```
if (NorthEast)
{
    /* special case 1 */
}
else if (East)
```

```
        /* special case 2 */
    }
    else if (SouthEast)
    {
        /* special case 3 */
    }
```

A problem arises if the programmer now decides that a vertical strips decomposition would be more efficient. This code assumes that exactly one of the three boundary conditions holds. But in the vertical strips decomposition there is only one section on the eastern edge, so all three conditions apply, not just one. Therefore, the change in data decomposition forces the programmer to rewrite this boundary condition code.

In Orca C, this scenario poses no problem because processes send and receive data through ports which in some cases involve interprocess communication and in other cases invoke derivative functions. Since the *X* level source code does not know what's on the other end of a port, the handling of boundary conditions has been decoupled from the *X* level source code. Note that instead of cluttering up the process code, special cases due to boundary conditions are handled at the problem level where they naturally belong.

Orca C's array operators and array slices are high-level constructs that eliminate the low-level chores of iterating over arrays and bundling messages. Such higher-level programming is syntactically cleaner and less error prone than the use of *for* loops.

Finally, Orca C incorporates the notion of fluff, which is a user managed cache that is particularly useful in applications with local neighbor computations. Orca C's data ensemble declarations provide an easy way to declare these extra data buffers and to specify their geometric relation to the local sections.

6. PERFORMANCE RESULTS

This section presents evidence for the claim that programs based on the Phase Abstractions—and therefore those written in Orca C—are portable.[23] Our approach was to take a single portable implementation of SIMPLE and execute it on several multiprocessors. Speedups were computed for each machine and compared against one another.

There is no currently agreed upon definition of portability for parallel

programs. Clearly, the ability to execute on different machines is a necessity. In addition, a portable program must *run well* on these machines. For this discussion, we consider a program that achieves similar speedups across a set of machines to be portable across these machines. The justification is that similar speedup curves are an indication that the program has extracted the same amount of parallelism from each machine. To be sure, there are problems with this approach, so we must be aware of the definition's limitations when analyzing the results. The main pitfall is that one machine may be inherently better suited for one application than another, in which case we would not expect to see identical speedups.

Note that other commonly used metrics such as MFLOPS or program execution time are too machine dependent to be the basis for a definition of portability; these machine dependencies make comparison across machines difficult. For example, comparing machines that are identical except for their clock speeds would produce different performance results even though this program is clearly portable between the two. For this reason, speedup—which eliminates some variables such as clock speed and, to some extent processor power—was chosen as our metric. Actual execution times are given in the Appendix A. Machine characteristics are given in Appendix B.

Experimental Setup. A variety of multiprocessors were used in the experiment, along with a detailed simulator of a Transputer-based machine. One multiprocessor is a Sequent Symmetry Model A, which has 20 Intel 80386 processors connected by a shared bus to a 32 MB memory module. Each processor has a unified 64K cache and an 80387 floating point accelerator.[24]

A second machine is a 24 node BBN Butterfly GP1000. Each node has a Motorola 68020 processor, 4 MB of local memory, and a processor node controller that interacts with an omega network to make remote references when needed. Together, the memory modules, the process node controllers, and the network form a single shared memory that all processors can access. Local memory access is about 12 times faster than remote access.[25]

Two machines are 32 node Intel iPSC/2 hypercubes in which all interprocessor communication is through message passing.[26] Both have 32 nodes with 80386 processors and a 64KB unified cache. They differ in that one has an iPSC SX floating point accelerator and 8MB of memory on each node (we refer to this as iPSC/2 F), while the other (iPSC/2 S) has the slower Intel 80387 floating point coprocessor and only 4MB of memory per node.

The 64 node nCUBE/7 is a nonshared memory hypercube in which each node has a custom main processor and 512 KB of memory.[27]

Finally, we have a detailed simulator of a Transputer-based nonshared memory machine. Using detailed information about arithmetic, logical and communication operators of the T800,[9] this simulator executes a Poker C program and produces time estimates for the program execution. Poker C is the precursor to Orca C[28]; the two languages share the same message passing semantics.

Our implementation of SIMPLE is structured in the manner discussed in Section 3. The entire program was written in C, including code to support a primitive form of ensembles. (C was used because the Orca C compiler is still under development.) The Orca message passing interface was implemented on all machines. These were directly supported on the non-shared memory machines, while on the Sequent and Butterfly they were written using shared memory.

Results. Figure 18 shows that similar speedups were achieved on all machines. Speedup values were computed based on a sequential version of our program. As mentioned earlier, many hardware characteristics can affect speedup, and these can explain the differences among the curves. In this discussion we concentrate on communication costs, the feature

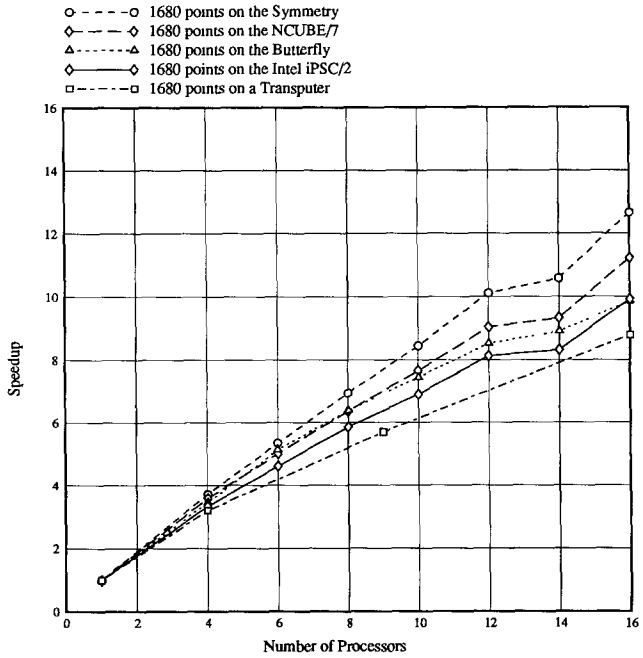

Fig. 18. SIMPLE speedup on various machines.

that best distinguishes these machines. For example, the iPSC/2 F and nCUBE/7 have identical interconnection topologies but the ratio of computation speed to communication speed is greater on the iPSC/2.[29,30] This has the effect of decreasing speedup since it diminishes the percentage of time spent computing, and this is where parallelism occurs. In other words, the larger fraction of time spent on communication increases the fraction of non-computation overhead. Similarly, since message passing latency is lowest on the Sequent's shared bus, the Sequent shows the best speedup. This claim assumes little or no bus contention, which is a valid assumption considering the modest bandwidth required by SIMPLE. After considering these machine differences, our claim of portability appears to be accurate.

As a reference point, Fig. 19 shows the results of Hiromoto *et al.*[8] on a Denelcor HEP using 4096 data points. (In our experiments, changing the problem size from 1680 to 4096 points involved the modification of compile-time constants and recompilation. When Orca C is complete, the problem size can be a runtime parameter.) These results are included only to show that our portable program is competitive with machine-specific code. The many differences with our results—including different problem sizes, different architectures, and possibly even different problem specifications—make it difficult to draw any stronger conclusions. As another reference point, Fig. 19 compares our results on the iPSC/2 S

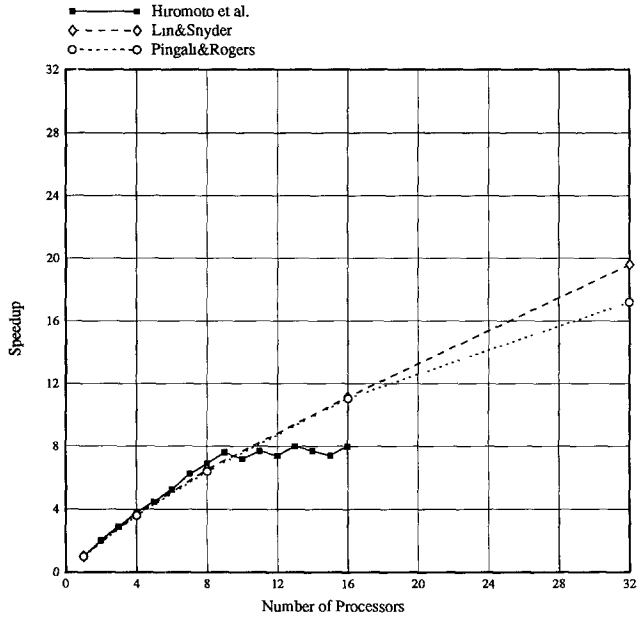

Fig. 19. SIMPLE with 4096 points.

against those of Pingali and Rogers' parallelizing compiler for a functional language.[10] Both experiments were run on iPSC/2's with 4MB of memory and 80387 floating point units. The largest potential difference lies in the performance of the sequential programs on which speedups are computed. All other parameters appear to be identical.

The SIMPLE program doesn't exercise all the features of Orca C, partly because of the low communication costs of SIMPLE and partly because of the small sizes of our multiprocessors. The next section discusses features of Orca C programs that make them flexible, and thus portable. These features become more significant in the context of newer, larger machines (particularly mesh machines), and for programs that require larger communication costs. We first give one detailed example of the flexibility of Orca programs.

6.1. Data Partitioning Experiment

The choice of data partitioning can significantly affect program performance. For SIMPLE, two obvious choices are a block decomposition and a strip decomposition. Because all communication is with nearest neighbors, blocks yield less overall data transmission but more messages (in the Hydro and Energy1 phases each interior section has six neighbors). With strips, each section has at most two neighbors, so fewer messages are sent. However, more data is transmitted because each section has a larger perimeter to area ratio.

Pingali and Rogers[10] pose the question of whether squares or strips are better. Data ensembles ease the task of changing data partitions and provide a mechanism for studying this question. The *Block* partitioning is the data ensemble described in previous sections. This will be compared against the *Strip* partitioning in which each section contains a vertical strip of the data ensembles.

Recall that the data ensembles discussed earlier create $\bar{r} \times \bar{c}$ arrays of blocks. With the Phase Abstractions, the *Strip* partitioning is easily derived from the *Block* partitioning by setting $\bar{r} = 1$ and $\bar{c} = Processors$ in the program's configuration computation section. The actual data ensemble declaration—shown again here—does not change.

$$partition \quad block[\bar{r}][\bar{c}] \qquad Vector\, x[rows][cols];$$

In addition, *Strips* require that each process have only East-West neighbors instead of the six neighbors used in *Block*. By using the port ensembles to bind functions to unused ports—in this case the North,

South, NorthEast and SouthWest ports—the program can easily accommodate this change in the number of neighbors. *No other source level changes are required.*

Figure 20 shows our results for problem sizes of 1K and 2K points on four machines. The *Block* partitioning performed better in every case, and the difference between the two strategies generally increases as the number of processors grows. This means that the overhead of sending more messages in *Blocks* is offset by the fact that *Block* transmits less overall data than *Strips*. Thus, we expect *Block*'s performance advantage to increase with the problem size since such changes do not alter the number of messages sent, but only increase the size of these messages. Our results appear to confirm this intuition. We conclude that for SIMPLE, partitioning by blocks is superior to partitioning by strips.

7. PORTABILITY ISSUES

For a parallel program to be portable it must not be tied to the details of any one architecture: It must be flexible and adapt to different machine characteristics. We now show how Orca C provides a flexible program structure.

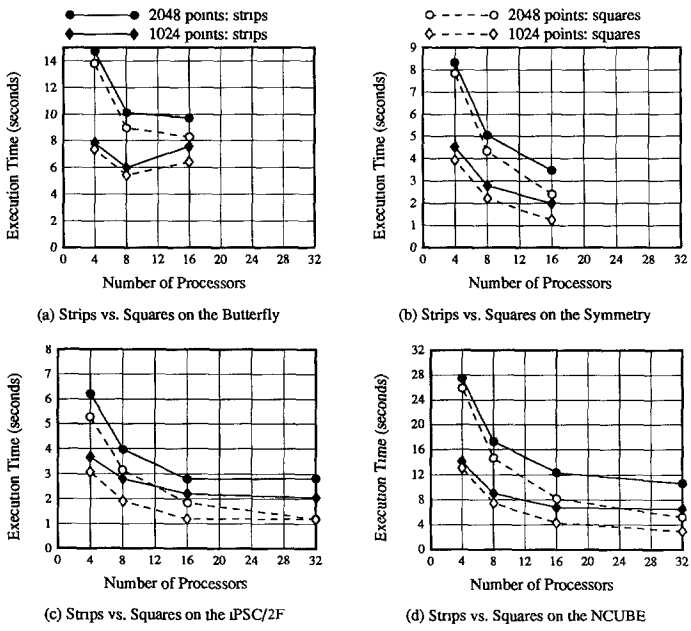

Fig. 20. Vertical strips vs. blocks.

Scaling. The data ensembles directly support scaling. Changes to both the problem size and the number of processors are accommodated implicitly in the data ensemble declaration. Recall that the section size is defined as:

$$s = \frac{rows}{\bar{r}}$$

$$t = \frac{cols}{\bar{c}}, \quad \text{where} \quad (s \times t) \text{ defines the section size,}$$

$(rows \times cols)$ defines the problem size, and

$(\bar{r} \times \bar{c})$ defines the number of sections.

We see that doubling the problem size in the *rows* dimension implicitly doubles s, creating sections that are twice as high. Similarly, increasing the number of sections by changing the value of $\bar{r}$ or $\bar{c}$ implicitly decreases the section size.

Granularity. The same mechanism that provides implicit scaling can be used to explicitly control granularity. The size of each section can be controlled by either changing the number of processors or by changing the number of sections in the ensemble declaration.

Multiple Threads. Each section corresponds to a logical thread of concurrency. By creating more sections than there are processors, Orca C programs can make use of multiple threads. This technique can help hide communication latency and can take advantage of the fast hardware context switching provided by architectures such as the Tera Computer[31] and the MIT Alewife Machine.[32]

Data Partitioning. Perhaps the most important characteristic in achieving good performance is the data partitioning. In general, the best choice of partitions depends on details of both the machine and the problem to be solved. For example, in some cases a strips decomposition is desirable while in other cases a block decomposition is better.[33]

In Orca C changes to the data partitioning are localized to the ensemble declarations. This was illustrated in the previous section where a one line change in the configuration section was sufficient to convert from a block to a strip decomposition.

Of course, the change in data partitioning leads to a different communication pattern. In this case, sections of the vertical strips partitioning have no North, South, NorthEast or SouthWest neighbors. Note, however, that the port ensemble, as declared in Section 4.3, requires no changes, nor does the specification of the boundary conditions or process code.

Changes in the Communication Graph. Ideally, portable parallel programs are written without assumptions about the underlying architecture. On the other hand, knowledge of machine details can be used to optimize program performance, such as when the logical communication graph is made to match an architecture's communication structure. Consider embedding the binary tree of the Delta phase onto a mesh architecture. Some logical edges will necessarily span multiple physical links (that is, dilation is greater than one), so a better way to implement the global minimum on a mesh is to use the "rows and columns" approach where values are first compared along each row of processors, then the minimum of each row is compared along a single column (see Fig. 21). With proper foresight an Orca C program can be written to realize either approach.

Instead of a binary tree, the "rows and columns" approach uses an n-ary tree, so the X level code is parameterized, as shown in Fig. 22. With the code suitably parameterized, this program can now execute efficiently on a variety of architectures. The "rows and columns" approach uses the following port ensemble declaration:

$$Delta[i][j].port.P \leftrightarrow Delta[i][j-1].port.C[0] \qquad 0 \leqslant i \leqslant \bar{r}, \quad 1 \leqslant j < \bar{c}$$

$$Delta[i][0].port.P \leftrightarrow Delta[i-1][0].port.C[1] \qquad 1 \leqslant i < \bar{r}$$

Locality. Notice that sections capture the important notion of locality. Because of the high latency of message passing, the need for locality of reference on nonshared memory machines is clear. But because all multiprocessors have some type of memory hierarchy, locality of reference can usually be exploited even on shared memory machines.[14]

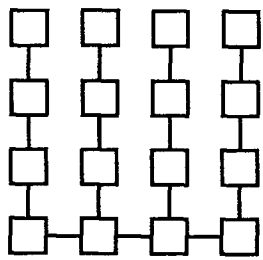

Fig. 21. Rows and columns to compute the global minimum.

```
xDelta(x[0:s-1][0.t-1], . )
    double  x[-1:s][-1:t];
        . .
{
    int     i, j;
    double  temp, . . . .

        . .
    /* Compute the global minimum */
    for (i=0; i<n, i++)
    {
        temp      <== Child[i];                    /* receive */
        lm_delta_t  =  Min(temp, lm_delta_t);
    }
    Parent    <== lm_delta_t;                      /* send */

    /* Broadcast the result */
    lm_delta_t <== Parent,                         /* receive */
    for (i=0; i<n; i++)
    {
        Child[i]   <== lm_delta_t;                 /* send */
    }

    delta_t[][]  =  lm_delta_t;                    /* array assignment */
}
```

Fig. 22. Parameterized X level code for the Delta phase.

Process to Processor Mapping. When the logical communication structure of a program does not match the physical communication structure of the underlying machine, the processes must be mapped to the processors and the choice of mapping can influence program performance. A good mapping will achieve low interprocessor communication while maintaining good load balance. While Orca C does not solve this mapping problem, the port ensembles do provide information—the logical communication graph—that is essential to performing such a mapping.

Our experiments with different mesh-to-hypercube mappings showed negligible performance differences for three different mappings: a Random mapping, a Default mapping, and a Binary Reflected Grey Code mapping. To compute the Default mapping the nodes of a mesh are assigned numbers in row major order and the nodes of a hypercube are numbered according to their position in the cube. Node 1 of the mesh is then mapped to node 1 of the hypercube, node 2 of the mesh is mapped to the node 2 of the hypercube, etc. Because of the relatively slow processors and small machine diameters, the performance differences were very small. For example, using 32 processors on the iPSC/2 F, the Random mapping was never more than 1.9% slower than the other mappings. Presumably, the importance of good mappings will become more significant as processor speeds increase and machines grow in diameter. For the shared memory machines in our experiments the issue of mapping processes to processors doesn't exist.

8. CONCLUSION

In this paper we have presented Orca C and shown how it is used to program SIMPLE. We have supplied evidence that this Orca program is portable across a variety of multiprocessors, and we have discussed the features of Orca in terms of ease of programming and portability. Finally, we speculate that the flexibility of Orca C programs will become even more important in the future. As the trend towards larger machines continues, the increased cost of communication will force programs to deal effectively with issues of data motion, granularity, and locality.

ACKNOWLEDGMENTS

It is a pleasure to thank Jinling Lee and Kevin Gates for implementing SIMPLE. We thank the members of the Orca Project—Langdon Beeck, George Forman, Scott Hauck, and Ton Ngo—for their contributions in designing Orca C. We also thank the referees for their careful reading and detailed comments. Finally, we wish to thank Hans Mandt and the Advanced Systems Laboratory of Boeing Computing Services for providing access to their Butterfly multiprocessor, and Walter Rudd and others at the Oregon Advanced Computing Institute for providing access to their iPSC/2 and nCUBE/7. This research was supported in part by Office of Naval Research Contract N00014-89-J-1368.

APPENDIX A.

Execution Time in Milliseconds[a]

1680 points					4096 points	
P	Symmetry	nCUBE/7	Butterfly	iPSC/2 F	P	iPSC/2 S
1	65310	173968	60940	34199	1	107924
4	17530	48535	41471	10267	4	29398
6	12200	34667	29899	7398	8	16550
8	9410	27339	23901	5838	16	9683
10	7720	22716	21422	4959	32	5510
12	6460	19252	19482	4205		
14	6170	18650	18966	4108		
16	5160	15497	18015	3445		

[a] All programs compiled without the -O flag on (no optimizer). Times given for 11 iterations of SIMPLE, not including, initialization time (memory allocation, file I/O).

APPENDIX B.

Machine Characteristics

model	Intel iPSC/2	Intel iPSC/2	nCUBE/7	Butterfly GP1000	Sequent Symmetry Model A
organization	OACIS	Univ. of Washington	OACIS	Boeing Computer Services	Univ. of Washington
cpu	80386	80386	custom	68020	80386
FPU	Intel SX	80387	?	none	80387
# of processors	32	32	64	24	20
memory	8MB per node	4MB per node	512KB per node	4MB per node	32MB total
cache	64KB per node	64KB per node	?	none	64KB unified per processor

REFERENCES

1. W. Crowley, C. P. Hendrickson, and T. I. Luby, The SIMPLE Code, Technical Report UCID-17715, Lawrence Livermore Laboratory (1978).
2. K. Ekanadham and Arvind, SIMPLE: Part I, An Exercise in Future Scientific Programming, Technical Report CSG Technical Report 273, MIT (1987).
3. D. Gannon and J. Panetta, SIMPLE on the CHiP, Technical Report 469, Computer Science Department, Purdue University (1984).
4. D. Gannon and J. Panetta, Restructuring Simple for the CHiP Architecture, *Parallel Computing*, **3**:305–326 (1986).
5. J. M. Meyers, Analysis of the SIMPLE Code for Dataflow Computation, Technical Report MIT/LCS/TR-216, MIT (1979).
6. T. S. Axelrod, P. F. Dubois, and P. G. Eltgroth, A Simulator for MIMD Performace Prediction—Application to the S-1 MkIIa Multiprocessor, *Proc. of the Int'l. Conf. on Parallel Processing*, pp. 350–358 (1983).
7. D. E. Culler and Arvind, Resource Requirements of Dataflow Programs, *Proc. of the Int'l. Symp. on Computer Architecture*, pp. 141–150 (1988).
8. R. E. Hiromoto, O. M. Lubeck, and J. Moore, Experiences with the Denelcor HEP, *Parallel Computing*, **1**:197–206 (1984).
9. T. J. Holman, Processor Element Architecture for Nonshared Memory Parallel Computers, PhD Thesis, University of Washington, Department of Computer Science (1988).
10. K. Pingali and A. Rogers, Compiler Parallelization of SIMPLE for a Distributed Memory Machine, Technical Report 90-1084, Cornell University (1990).
11. G. Alverson, W. Griswold, D. Notkin, and L. Snyder, A Flexible Communication Abstraction for Nonshared Memory Parallel Computing, *Proc. of Supercomputing '90* (November 1990).
12. W. Griswold, G. Harrison, D. Notkin, and L. Snyder, Scalable Abstractions for Parallel Programming, *Proc. of the Fifth Distributed Memory Computing Conference*, Charleston, South Carolina (1990).
13. L. Snyder, Applications of the "Phase Abstractions" for Portable and Scalable Parallel Programming, in *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*, Joel Saltz and Piyush Mehrotra, Eds., North Holland (1992).
14. C. Lin and L. Snyder, A Comparison of Programming Models for Shared Memory Multiprocessors, *Proc. of the Int'l. Conf. on Parallel Processing*, Vol. II, pp. 163–180 (1990).
15. K. Gates, SIMPLE: An Exercise in Programming in Poker, Technical Report, Applied Mathematics Department, University of Washington (1989).
16. D. Notkin, D. Socha, M. Bailey, B. Forstall, K. Gates, R. Greenlaw, W. Griswold, T. Holman, R. Korry, G. Lasswell, R. Mitchell, P. Nelson, and L. Snyder, Experiences with Poker, *Proc. of the ACM SIGPLAN Symp. on Parallel Programming: Experience with Applications, Languages, and Systems* (July 1988).
17. M. Gerndt, Updating Distributed Variables in Local Computations, *Concurrency—Practice and Experience*, **2**(3):171–193 (September 1990).
18. S. Otto, MetaMP: A Higher Level Abstraction for Message-Passing Programming. Technical Report CS/E 91-003, Oregon Graduate Institute of Science and Technology (1991).
19. J. Lee, Extending the SIMPLE Program in Poker, Technical Report 89-11-07, Department of Computer Science and Engineering, University of Washington (1989).
20. P. Nelson, Parallel Programming Paradigms, PhD Thesis, University of Washington, Department of Computer Science (1987).

21. J. Lee, C. Lin, and L. Snyder, Programming SIMPLE for Parallel Portability, in *Languages and Compilers for Parallel Computing*, U. Banerjee, D. Gelernter, A. Nicolav, and D. Padua, Eds., Springer-Verlag (1992).
22. D. Bailey, J. Cuny, and C. Loomis, ParaGraph: Graph Editor Support for Parallel Programming Environments, Technical Report 89-53, Department of Computer and Information Science, University of Massachusetts, Amherst (August 1989).
23. C. Lin and L. Snyder, Portable Parallel Programming: Cross Machine Comparisons for SIMPLE, *Fifth SIAM Conference on Parallel Processing* (1991).
24. T. Lovett and S. Thakkar, The Symmetry Multiprocessor System, *Proc. of the Int'l. Conf. on Parallel Processing*, pp. 303–310 (1988).
25. G. Alverson, Abstractions for Effectively Portable Shared Memory Parallel Programs, PhD Thesis, University of Washington, Department of Computer Science and Engineering (1990).
26. Intel Corporation, *iPSC/2 User's Guide* (October 1989).
27. NCUBE Corporation, *NCUBE Product Report*, Beaverton, Oregon (1986).
28. L. Snyder, Parallel Programming and the Poker Programming Environment, *Computer*, pp. 27–36 (July 1984).
29. T. Dunigan, Hypercube Performance, *Proc. of the 2nd Conf. on Hypercube Architectures*, pp. 178–192 (1987).
30. T. Dunigan, Performance of the Intel iPSC/860 and NCUBE 6400 Hypercubes, Technical Report ONRL/TM-11790, Oak Ridge National Laboratory (1991).
31. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, The Tera Computer System, *Int'l. Conf. on Supercomputing*, pp. 1–6 (June 1990).
32. A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B.-H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung, The MIT Alewife machine: A Large Scale Distributed-Memory Multiprocessor, *The Workshop on Multithreaded Computers at Supercomputing '91* (November 1991).
33. J. Saltz, V. Naik, and D. Nicol, Reduction of the Effects of the Communication Delays in Scientific Algorithms on Message Passing MIMD Architectures, *SIAM Journal of Statistical Computing*, **8**(1):118–134 (January 1987).