Programming SIMPLE for Parallel Portability*

Jinling Lee
Calvin Lin
Lawrence Snyder
Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195

Abstract

The Phase Abstractions formulation of Livermore's SIMPLE computation is described in detail. This new program is of interest for three reasons. First, it enables the Phase Abstractions approach to be easily compared with the many other programming styles illustrated by SIMPLE in the past. Secondly, this program has recently been reported to execute on five diverse MIMD computers, in all cases achieving efficiency in excess of 50%. Thirdly, the Phase Abstractions permit easy revisions of the program that enable performance experimentation. Specifically, "array blocks" are compared to "column strips" as a means of array decomposition. Results comparing the two approaches are presented for the Sequent Symmetry, BBN Butterfly, Intel iPSC/2 and NCUBE/7. The results confirm the intuition that blocks are superior.

1 Introduction

The SIMPLE program, a "simple" computational fluid dynamics code, was released by Lawrence Livermore National Laboratory in 1978 [5] as a benchmark program to evaluate new computers and Fortran compilers. Since that time the program has frequently been used to illustrate new computational approaches [7, 8, 9, 18] and to estimate machine performance [3, 6, 9, 11, 13]. This paper follows both themes by presenting a version of SIMPLE written using the Phase Abstractions [2, 12, 22] as well as certain performance data.

The Phase Abstraction version of SIMPLE is notable because it has recently been shown to be portable across a wide variety of MIMD parallel computers [16]. Figure 1 shows the speedups achieved for the BBN Butterfly, the Intel iPSC/2, the NCUBE/7, the Sequent Symmetry, and a detailed simulator for a Transputer-based multicomputer. Though the machines differ substantially, e.g. in memory structure, the speedups fall roughly within the same range.

*This research was supported in part by Office of Naval Research Contract N00014-89-J-1368.

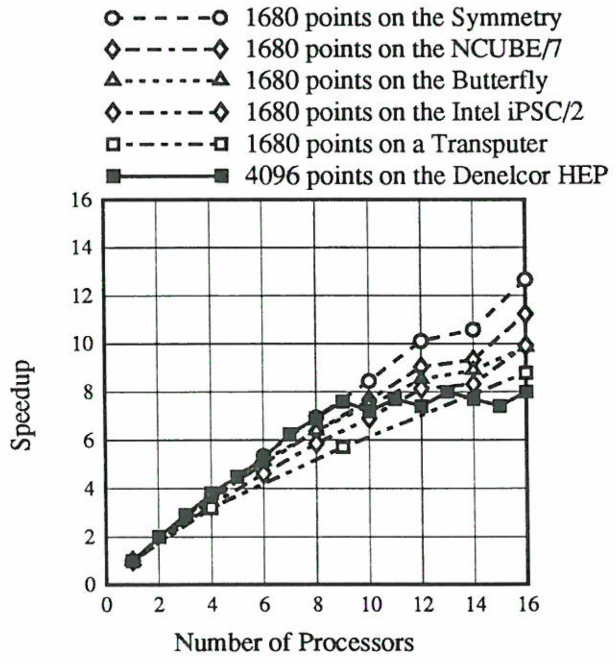


Figure 1: SIMPLE on Various Machines

The Phase Abstractions – the XYZ programming levels and the ensemble structures – provide critical information about a program that can be used by a compiler to generate efficient code for the target machine. These include locality, scalable concurrency, granularity and communication patterns. It is the goal of this paper to present SIMPLE and to illustrate how these features are described and controlled to achieve portability

To illustrate how abstractions can structure a program to improve portability, consider the grain size of SIMPLE.

The program presented here is the latest in a series of improvements beginning with the work of Gannon and Panetta [8], who developed a version of SIMPLE suitable for the CHiP architecture and programmed it using early Poker software. Their emphasis [9] was on extracting the essence of the computation from the original Fortran version and reformulating it using highly parallel algorithms suitable for the nonshared memory model of computation. Their code used a one-point-per-process approach, i.e. each process retains the values describing a single point of the state space, rather than, say, a region of points. A similar approach was taken by Gates [10] in his later Poker version of SIMPLE. Such fine grain solutions achieve a very high degree of concurrency and are conceptually easy to program, but they have a serious drawback: They can be too concurrent!

Since problems tend to have many more data points than physical processors, the parallel execution of a one-point-per-process program requires many logical processes to be executed on each physical processor. This generally requires that processes be multiplexed, though compilation techniques are now being developed that can in some cases "aggregate" fine grain processes into coarser grain processes to avoid multiplexing [23]. Multiplexing, whether done in hardware [24] or software [4], incurs overhead because at the very least there is context switching overhead, and in the usual case many instructions are executed which are superfluous in the absence of physical concurrency.

But if the finest granularity is not ideal, neither is any specific choice of coarse granularity. Clearly, the problem size and number of processors available will change. Moreover, though multiple processes are useful for hiding communication latency, machines differ greatly on their ability to benefit from them.

The Phase Abstractions approach supports the definition of variable grain size programs. The processes are parameterized so that grain size can be customized based on the problem size, number of processors, operating characteristics of the target architecture, etc. The customization is performed by the compiler, loader, or runtime system depending on when sufficient information is known to define the granularity. The Phase Abstraction mechanisms that support this control of granularity are the ensembles and the X programming level. Both are described in the next section.

2 Phase Abstractions

Recall [2, 12, 22] that the term Phase Abstractions encompasses both the XYZ programming levels and the ensemble structures.

The XYZ programming levels are an abstraction that recognizes that the instructions of a parallel program serve different purposes. Beginning at the lowest level, instructions are used to form *processes* (X level), which are the basic building blocks of a computation. Processes are then composed to form concurrent *phases* (Y level). Phases correspond to our informal notion of a parallel algorithm. The *problem level* (Z level) controls the overall phase invocation to solve the user's problem.

The XYZ decomposition of the SIMPLE computation is (partially) illustrated in Figure 2. The computation begins by invoking a phase that reads in the problem state and initializes program variables. Then a series of five phases – *Delta*, *Hydro*, *Heat*, *Energy1*, and *Energy2* – are iteratively invoked to implement the state changes required in one logical iteration of the algorithm. When convergence is achieved, an output phase is invoked. Each phase is a parallel algorithm composed of processes, as explained next.

Phases are defined using ensembles. An *ensemble* is a set with a partitioning. The set is generally composed of names having additional structure and the partitioning describes how it is decomposed into constituent parts. A partition is called a *section*.

Three kinds of ensembles are used to define a phase: A *data ensemble* is a data structure with a partitioning. A *code ensemble* is a set of process instances with a partitioning. A *port ensemble* is a set of adjacency names with a partitioning. Ensembles will be illustrated extensively in subsequent sections, but illustrations of a data ensemble can be seen in Figure 4, a code ensemble in Figure 5, and a port ensemble in Figure 9.)

```
data := Load();
while (error > δ)
{
    Delta(data);
    Hydro(data);
    Heat(data);
    Energy1(data);
    error := Energy2(data);
}
Output (data);
```

Figure 2: Z Level for SIMPLE

To define a phase, the partitionings of the data, code and port ensembles must be isomorphic. This requirement permits the process(es) of each section of a code ensemble to be associated with the data of the corresponding section of the data ensemble and with the ports of the corresponding section of the port ensemble. Each section will be allocated to a processor for execution: The process executes on that processor, the data can be stored in memory local to that processor, and ports support interprocessor communication.

The data ensemble provides a (logically) global view of the problem state as represented by its data structures, but it does so in a way that permits data to be allocated to separate address spaces if necessary. The code ensemble gives a (logically) global view of the processes performing the parallel computation; when the process instances differ the model is MIMD, but it can be SPMD if they are all identical. Finally, the port ensemble gives the overall communication structure of the phase, which is extremely useful in cases where interprocessor communication can be optimized.

To complete the illustration begun in the last section, the "size" of a section, that is, the amount of data allocated to the section in the data ensemble and the amount of computation required to execute the process in that section, defines the granularity of the parallel computation. Changes to the data ensemble – total number of partitions or the amount of data allocated to a section – are the means of changing the granularity of the computation. Notice that the section also captures the important notion of locality.

3 The SIMPLE Computation

The goal of SIMPLE is to simulate the flow of a pressurized fluid as it moves inside a spherical shell. This section describes the computation. For a more detailed description of the derivation of the formulas, see Crowley, Gannon, and Gates [5, 9, 10].

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

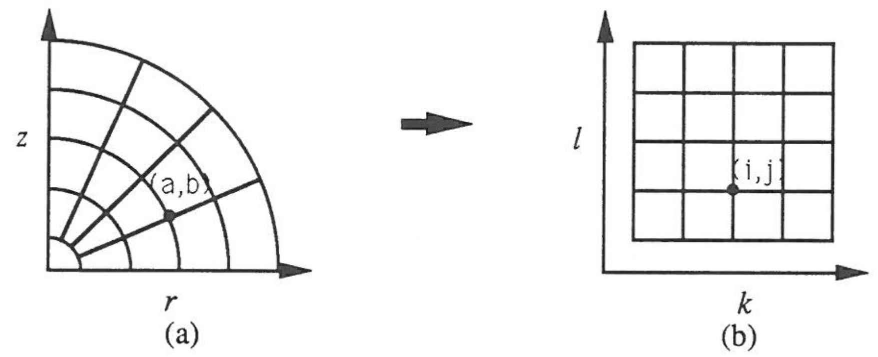


Figure 3: Mapping of Physical Domain to Computation Domain

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.

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.

Because of the spherical symmetry of this problem, a cylindrical coordinate system is used. The physical domain of the problem is reduced to a quarter of an annular region

(Figure 3 (a)) by first projecting the shell onto a 2 dimensional plane and then taking the upper right quadrant of the projected annular region. Consequently, vectors such as velocity only have 2 components in the newly reduced physical domain: r, the radius, and z, the height.

In order to solve the equations which 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 convenience, the physical domain is mapped to the computation domain as depicted in Figure 3. In the following, the notation $V_{i,j}$ is used to denote the physical variable of node (i, j) in the computation grid (Figure 3 (b)). 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 the upper right corner of the square, node $(i+1, j+1)$.

What follows is the basic algorithm to solve the set of equations. For clarity the code to deal with boundary conditions is omitted here. The next section gives a brief description of boundary conditions. A more detailed description can be found in [5, 9]. In the following r and z denote the r and z components of the coordinate, u and w denote the r and z component of the velocity and a^r and a^z denote the r and z component of the acceleration.

The SIMPLE algorithm:

First compute the initial coordinates of all nodes and initialize the variables of every node, then iteratively carry out the following sequence of steps:

1. Compute the next time step.

   A standard rule is that the time step should not be so large that a speed-of-sound signal can move across a grid cell in one time step. This is called the Courant condition. So,

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

   Here, the following notation is used:

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

   where f stands for any point quantity such as r, z, u, w, and C_A is the local speed of sound, which can be computed as follows:

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

   .

2. Compute the new acceleration.

   The derivative in equation (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 $p + q$.

$$a^r_{ij} := \frac{f_{i,j}\,(z_{i-1,j} - z_{i,j-1}) + f_{i,j+1}\,(z_{i,j+1} - z_{i-1,j}) + f_{i+1,j+1}\,(z_{i+1,j} - z_{i,j+1}) + f_{i+1,j}\,(z_{i,j-1} - z_{i+1,j})}{0.5\,(\rho_{i,j}\,J_{i,j} + \rho_{i,j+1}\,J_{i,j+1} + \rho_{i+1,j+1}\,J_{i+1,j+1} + \rho_{i+1,j}\,J_{i+1,j})}$$

$$a^z_{ij} := \frac{f_{i,j}\,(r_{i-1,j} - r_{i,j-1}) + f_{i,j+1}\,(r_{i,j+1} - r_{i-1,j}) + f_{i+1,j+1}\,(r_{i+1,j} - r_{i,j+1}) + f_{i+1,j}\,(r_{i,j-1} - r_{i+1,j})}{0.5\,(\rho_{i,j}\,J_{i,j} + \rho_{i,j+1}\,J_{i,j+1} + \rho_{i+1,j+1}\,J_{i+1,j+1} + \rho_{i+1,j}\,J_{i+1,j})}$$

3. Compute the new velocity and new coordinates.

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

4. Compute the new Jacobian and volume of revolution.

$$tmp\_J1_{i,j} := \frac{1}{2}\left[r_{i,j}\,(z_{i,j-1} - z_{i-1,j}) + r_{i,j-1}\,(z_{i-1,j} - z_{i,j}) + r_{i-1,j}\,(z_{i,j} - z_{i,j-1})\right]$$
$$tmp\_J2_{i,j} := \frac{1}{2}\left[r_{i,j-1}\,(z_{i-1,j-1} - z_{i-1,j}) + r_{i-1,j-1}\,(z_{i-1,j} - z_{i,j-1}) + r_{i-1,j}\,(z_{i,j-1} - z_{i-1,j-1})\right]$$
$$J_{i,j} := tmp\_J1_{i,j} + tmp\_J2_{i,j}$$
$$old\_S_{i,j} := new\_S_{i,j}$$
$$new\_S_{i,j} := \frac{1}{3}\left[(r_{i,j} + r_{i,j-1} + r_{i-1,j})\,tmp\_J1_{i,j} + (r_{i,j-1} + r_{i-1,j-1} + r_{i-1,j})\,tmp\_J2_{i,j}\right]$$

5. Compute the new density and artificial viscosity.

$$\rho_{i,j} := \rho_{i,j}\,\frac{old\_S_{i,j}}{new\_S_{i,j}}$$
$$tmp1 := \begin{cases} \frac{[\Delta r\delta w - \Delta z\delta u]^2}{\Delta r^2 + \Delta z^2} & \text{if } [\,] < 0 \\ 0 & \text{otherwise} \end{cases}$$
$$tmp2 := \begin{cases} \frac{[\Delta u\delta z - \Delta w\delta r]^2}{\delta r^2 + \delta z^2} & \text{if } [\,] < 0 \\ 0 & \text{otherwise} \end{cases}$$
$$q_{i,j} := 1.5\rho_{i,j}\,(tmp1 + tmp2) + 0.5\rho_{i,j}C_A\sqrt{tmp1 + tmp2}$$

6. Compute the new energy and pressure.

$$\epsilon_{i,j} := \epsilon_{i,j} - (p_{i,j} + q_{i,j})\,delta\_\tau_{i,j}$$
$$tmp_{i,j} := (\gamma - 1)\,\epsilon_{i,j}\rho_{i,j}$$
$$\epsilon_{i,j} := \epsilon_{i,j} - \left(\frac{1}{2}\,(tmp_{i,j} + p_{i,j}) + q_{i,j}\right)delta\_\tau_{i,j}$$
$$p_{i,j} := (\gamma - 1)\,\epsilon_{i,j}\rho_{i,j}$$

where $delta\_\tau$ is the difference between the new τ (specific volume) and the τ in the previous iteration. γ is the specific heat. $\tau = 1/\rho$, $\gamma = 1.4$ for air.

7. Compute the new temperature and heat.

The heat equation (equation 7) can be separated into 2 equations, one in the k direction and one in the l direction. 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
    $J_{i,j} := (r_{i,j} - r_{i,j-1})(z_{i,j-1} - z_{i,j}) - (z_{i,j-1} - z_{i,j})(r_{i,j} - r_{i-1,j})$    (8)
    $\sigma_{i,j} := 0.1\rho_{i,j} r_{i,j} J_{i,j}/\delta t$    (9)
    $CC_{i,j} := 0.0001\theta_{i,j}^{5/2}/J_{i,j}$    (10)
    $KJ_{i,j} := \frac{CC_{i,j}CC_{i,j+1}}{CC_{i,j} + CC_{i,j+1}}$    (11)
    $R_{i,j} := (r_{i,j} + r_{i,j-1})\left((r_{i,j} - r_{i,j-1})^2 (z_{i,j} - z_{i,j-1})^2\right) KJ_{i,j}$    (12)
end for
for j := 0 to max_K do
    for each i do
        $D_{i,j} := \sigma_{i,j} + R_{i,j} + R_{i,j-1}(1 - \alpha_{i,j-1})$    (13)
        $\alpha_{i,j} := R_{i,j}/D_{i,j}$    (14)
        $\beta_{i,j} := \frac{R_{i,j-1}\beta_{i,j-1} + \sigma_{i,j}\theta_{i,j}}{D_{i,j}}$    (15)
    end for
end for
for j = max_K to 0 do
    for each i do
        $\theta_{i,j} := \alpha_{i,j}\theta_{i,j+1} + \beta_{i,j}$    (16)
    end for
end for
for i := 0 to max_L do
    $heat_{i,0} = (\theta_{i,0} - \theta_{i,1}) R_{i,0}\ \delta t$    (17)
end for
```

Do the same calculation (except statement 17) in the l direction.

8. Compute the energy and work. Check for error.

```
for each pair (i,j) do
    $m_{i,j} := \rho_{i,j} S_{i,j}$
    $energy_{i,j} := e_{i,j} m_{i,j} + \frac{1}{8}(m_{i,j} + m_{i,j+1} + mi + 1, j + 1 + mi + 1, j)\left(u_{i,j}^2 + w_{i,j}^2\right)$
    $tmp_{i,j} := \frac{1}{4}\delta t (p_{i,j} - p_{i,j+1})(r_{i,j-1} - r_{i,j})[(r_{i,j} - r_{i,j-1})(u_{i,j} + u_{i,j-1}) - (z_{i,j} - z_{i,j-1})(u_{i,j} + u_{i,j-1})]$
    $work_{i,j} := \begin{cases} -tmp_{i,j} & \text{if (i,j) is on the west boundary of the computation grid} \\ 0 & \text{otherwise} \end{cases}$
end for
```

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

$$
\begin{aligned}
total\_work &:= \sum_{i,j=0} work_{i,j} \\
total\_heat &:= \sum_{i,j=0} heat_{i,j} \\
error &:= total\_energy - total\_work + total\_heat
\end{aligned}
$$

4 The SIMPLE Program

The problem level (Z level) logic of the SIMPLE program has already been illustrated in Figure 2. Since we presume that input and output are handled by phases provided by the system, it remains to define the five computational phases.[1] To do so, we define ensembles, which in turn require that we define data structures, processes, and communication graphs. The three types of ensembles will be discussed in their own subsection.

Data Ensembles. Practical programs will usually require many data ensembles for each phase. Since all of the ensembles will require the same partitioning in order to be part of the phase, it is most convenient to define all of 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 represent the state of the SIMPLE computation are given in Table 1 together with a short description 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 are used in each phase, and the last five items are used only in the *Energy1* phase.

In addition to the arrays, three global scalar values are used (see Table 2). The last item is used only in the *Hydro* phase and then only for computations along the "west wall".

The arrays will be partitioned into contiguous two dimensional subarrays (blocks). Recall that partitions are called *sections*. This choice reflects the assumption that the locality of the computation is greatest when one process updates all points of a contiguous block. It is not immediately obvious that this is true (though it is commonly assumed) and we return to this issue in a later section. The appropriate specification of the block partitioning is given below:

$$
\begin{aligned}
rows &= \bar{r} \cdot s \\
cols &= \bar{c} \cdot t \\
\forall\, i &\in 0:(\bar{r}\text{-}1), \quad \forall\, j \in 0:(\bar{c}\text{-}1) \\
&\quad block[i][j] = [i \cdot s + [0:s\text{-}1]]\ \ [j \cdot t + [0:t\text{-}1]]
\end{aligned}
$$

which states that a $rows \cdot cols$ array is to be partitioned into a collection of $\bar{r} \cdot \bar{c}$ blocks each of size $s \cdot t$. This partitioning is applied to each of the arrays. For example, we indicate that the pressure array, p, is converted into an ensemble by specifying *block*(p). This has the

[1] At this time no language implementation of the Phase Abstractions exists, so the program text given in this paper is all expressed in pseudocode. To achieve the results presented in Figure 1, this pseudocode was hand translated into code acceptable to each object machine's C compiler.

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]	diff 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)

Table 1: Array Values in SIMPLE

Type	Variable	Description
double	time	time
int	iter	current iteration
double	bound_p	pressure at the inner shell

Table 2: Scalar Values for SIMPLE

effect of associating $p[i{\cdot}s + x][j{\cdot}t + y]$ with element x, y of section i, j. Figure 4 illustrates the *pressure* array converted into an ensemble.

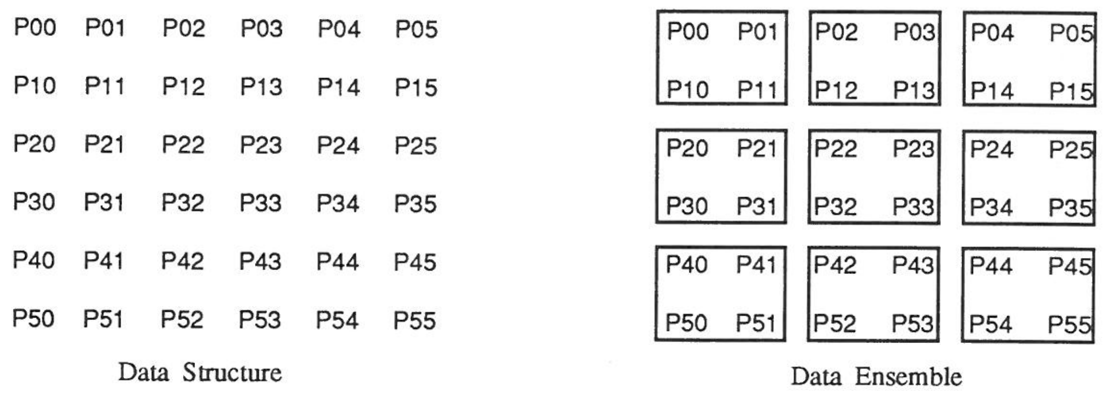


Figure 4: The *Pressure* Array and the *Pressure* Ensemble, where rows=cols=6, r=3, c=3, s=2, t=2.

In addition to converting the data structures into data ensembles, it is useful to assign a copy of each of the global scalars to each section to be available for local computations.

Code Ensembles. Having defined the data partitioning, the next task is to provide processes to operate on each region. This is the role of the code ensemble. A code ensemble is simply a set of process instances with a partitioning.

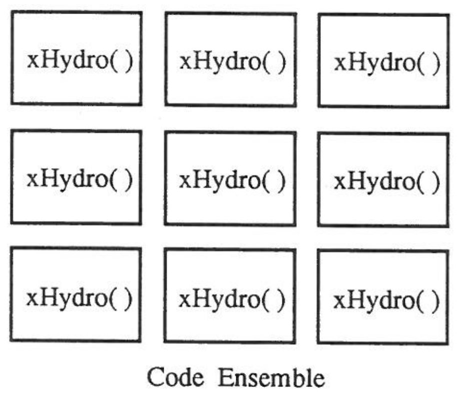


Figure 5: The Code Ensemble for the Hydro Phase.

In SIMPLE, the processes forming each phase are instances of a single process. For example, the *Hydro* phase uses instances of the *xHydro()* process. Thus, SIMPLE appears not to require the full MIMD capability of code ensembles – where the instances can be instances of different processes – but rather requires only the SPMD capability. This apparent uniformity derives to a considerable extent from mechanisms provided by the Phase Abstractions. Specifically, to handle the rather complex boundary conditions, it has been necessary in the past [10] to provide nine different process types, corresponding to which portion, if any, of the perimeter is included in the section, e.g. North edge, NorthEast corner, etc. The processing of the boundary code is what makes each routine unique. But with Phase Abstractions only a single code defining the activity on interior nodes is required. Boundary computations are specified using "port bindings," which are explained below.

The specification of a code ensemble for a phase, say the *Hydro* phase, is accomplished by declaring the process instance that is assigned to the section:

$$Hydro[i][j].code := xHydro(); \qquad [0 : \bar{r}\text{-}1]\ [0 : \bar{c}\text{-}1]$$

Each of the $\bar{r} \cdot \bar{c}$ sections[2] will be assigned an instance of the *xHydro()* code. More complex code ensembles are possible, e.g. with multithreading, but they are not needed in SIMPLE.

The *xHydro()* code, which computes the point updates for the blocks of data assigned to a section, is too large to be given in complete detail. Figure 6 shows a schematic of the process code. Several features are noteworthy:

- *parameters* – The arguments to the process are formal parameters that establish the correspondence between the local names for variables and the blocks of the ensembles. For example, in the phase declaration given below the local *pressure* array will be bound to the ensemble *p*.

- *ports* – The names of the neighboring sections are listed in the port declaration. These names are the destinations and sources of sends and receives.

- *local declarations* – Space is allocated for the local portion of a data ensemble, resulting in a more conveniently indexed arrangement of data. Notice that the size of the array will be determined by the size of the block in the data ensemble and is thus logically an input to the process. Furthermore, the size of the formal parameters may be smaller than that of its corresponding local array. For example, the local *pressure* array contains extra rows and columns to hold values from neighboring sections; the formal declaration $p[1 : s][1 : t]$ and the local declaration $pressure[0 : s + 1][0 : t + 1]$ specifies this mapping.

- *sends/receives* – Communication is implemented with the transmit operator ($<-$) for which a port name on the left specifies a send of the righthand side, and a port on

[2]Throughout this paper, the notation [*lower* : *upper*] specifies a range of values. Similarly, the notation $[l' : u'][l'' : u'']$ specifies a 2D range of values. In the above example, the 2D range specification indicates that for the array on the left, $Hydro[i][j]$, the index of the first dimension, i, takes on all values in the range $[0 : \bar{r}\text{-}1]$, and that the index of the second dimension, j, takes on all values in the range $[0 : \bar{c}\text{-}1]$.

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.

- *local computation* – The logic of the process is essentially the sequential computation of the original program, a characteristic that allows the number of partitions in the ensembles to go to one, yielding serial computation.

Notice, finally, that the process achieves a certain amount of encapsulation and captures locality.

The syntax in this paper is intended to illustrate the full flexibility of the ensembles. In practice the ensembles may be specified in some higher level manner so the programmer will not have to specify these bindings and ensembles in such detail. For example, there may be a *Block* ensemble which defines a 2D array of square sections. Similarly, we envision a *Strips* ensemble which defines an array of long narrow sections. Also, the neighbor communication might also be specified implicitly, freeing the programmer from the details of message passing.

Port Ensembles. The process instances of the code ensembles must communicate with one another. The overall structure of this communication for the five phases is illustrated in Figure 7, where the boxes represent process instances and the edges indicate the pairs of processes that must exchange information. The port ensemble is used to specify this association.

The *Hydro* phase, for example, uses a hex mesh where each section has six ports specified in the port ensemble:

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

which binds the section's ports to the (possibly different) names used in the process' port declaration. The pairing of port names to define a communication channel is specified as follows:

$$\begin{array}{lcll} Hydro[i\text{-}1][j].port.S & \leftrightarrow & Hydro[i][j].N & [1:\bar{r}\text{-}1]\ [0:\bar{c}\text{-}1] \\ Hydro[i][j\text{-}1].port.E & \leftrightarrow & Hydro[i][j].W & [0:\bar{r}\text{-}1]\ [1:\bar{c}\text{-}1] \\ Hydro[i][j\text{-}1].port.NE & \leftrightarrow & Hydro[i\text{-}1][j].SW & [1:\bar{r}\text{-}1]\ [1:\bar{c}\text{-}1] \end{array}$$

This specification associates only a subset of all of the ports, namely, those that are connected as in Figure 7. The remaining ports, those that are on the boundary of the problem space, can be bound to functions. These functions compute the boundary conditions using data local to the section. For example, the specification

$$Hydro[i][t].port.E \text{ receive} \quad \leftrightarrow \quad Eboundary(); \qquad [0:\bar{r}\text{-}1]$$

```
xHydro(p[1:s][1:t], rho[1:s][1:t], J[1:s][1:t] ... bound_p)
    double  pressure[0:s+1][0:t+1];
    double  rho[0:s+1][0:t+1];
    double  J[0:s+1][0:t+1];
        . . .
    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+1] <- East;

    /* other communication . . . */

    /* Compute acceleration */
    for (i=0; i<s; i++)
    {
        for (j=0; i<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 . . . */
}
```

Figure 6: X Level Code for the Hydro Phase

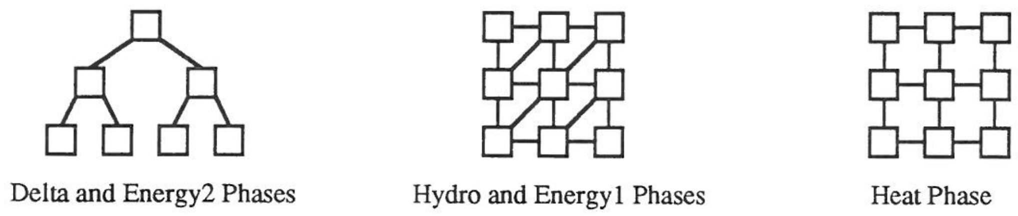


Figure 7: Instances of the Communication Graph Families for SIMPLE

states that for sections along the east boundary, a "receive" from the E port will return the value computed by the function *Eboundary()*. Figure 8 shoes the computation of this boundary condition.

Other boundaries can be defined similarly. For example,

$Hydro[i][t].port.E$ send $\quad\leftrightarrow\quad$ $No\text{-}op()$; $\qquad [0 : \bar{r}\text{-}1]$

states that sending data to an unbound East port results in a no-op (This is the default). Once these functions have been defined, boundary values can be accessed through ports in the same manner that interior processes access values in neighboring sections. Thus, all processes can execute the same code.

Phase Definition. A phase is the composition of data ensembles, a code ensemble and a port ensemble. For the *Hydro* phase, for example, the latter two bindings have already been made explicitly. To incorporate the necessary data ensembles we specify

$Hydro.data \quad\leftrightarrow\quad x,\ u,\ a,\ rho,\ p,\ q,\ delta\_tau,\ e,\ J,\ S,\ bound\_p$

which conceptually declares the actual parameters that correspond to the formals in the process preamble. Any or all of these could have been specified at the call site if they varied from call to call in the problem level (Z level) program.

5 Experiments

Given a particular application and a particular machine, it's not always clear what choice of partitioning is best [21]. Two obvious choices are to partition the data into squares or into strips. For applications with nearest neighbor communication, square sections result in less data transmission but in more messages, since each interior section has four neighbors. With strips, each section has at most two neighbors, but more data is transmitted because each section has a larger perimeter to area ratio.

Pingali and Rogers [20] pose the question of whether squares are better than strips for SIMPLE. Data ensembles ease the task of changing data partitions and provide a mechanism

```
Eboundary(x, u)
     Vector   x[s+1][t+1];
     Vector   u[s+1][t+1];
{
     double   alpha, beta, omega;
     int      i, j;

     for (i=1; i<s+1; i++)
     {
          x[i][t+1].r = x[i][t].r + (x[i][t].r - x[i][t-1].r);
          x[i][t+1].z = x[i][t].z + (x[i][t].z - x[i][t-1].z);
          u[i][t+1].r = u[i][t-1].r;
          u[i][t+1].z = u[i][t-1].z;
     }
}
```

Figure 8: Function to Handle East Boundary Condition in *xHydro()*

for studying this question. The *Block* partitioning is the data ensemble described in the previous section. This will be compared against the *Strip* partitioning in which each section contains a vertical strip of the data arrays.

Recall that the data ensembles discussed earlier create $\bar{r} \times \bar{c}$ arrays of blocks. So with the Phase Abstractions, the *Strip* partitioning is easily derived from the *Block* partitioning by setting $\bar{r} = 1$ and $\bar{c} = Number\text{-}of\text{-}Processors$. 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.

Before presenting the results, we first explain our experimental methodology and hardware environment.

Methodology. Since the Phase Abstractions version of SIMPLE is portable, it can execute on several different machines and serve as a tool for studying the effects of various program characteristics. Using a single portable program has the advantage of controlling one important variable, namely, the application under study.

Since no Phase Abstractions compiler currently exists[3] we hand ported our program to run on the various machines. Our X level language was C, and we used machine specific routines (also written in C) to implement the other Phase Abstractions entities. In porting

[3]A Phase Abstraction compiler is under construction at the University of Washington.

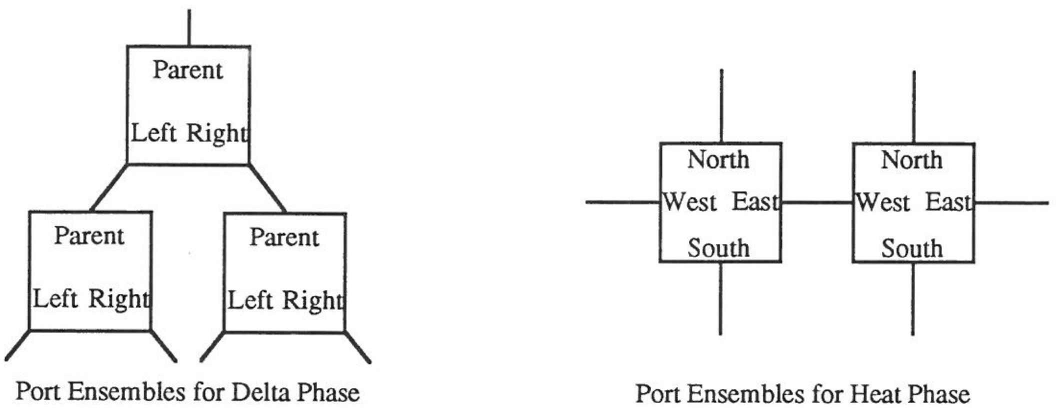


Figure 9: Examples of Port Ensembles

this program, only rudimentary source level changes dealing with the differences in message passing and process/node management were necessary.[4]

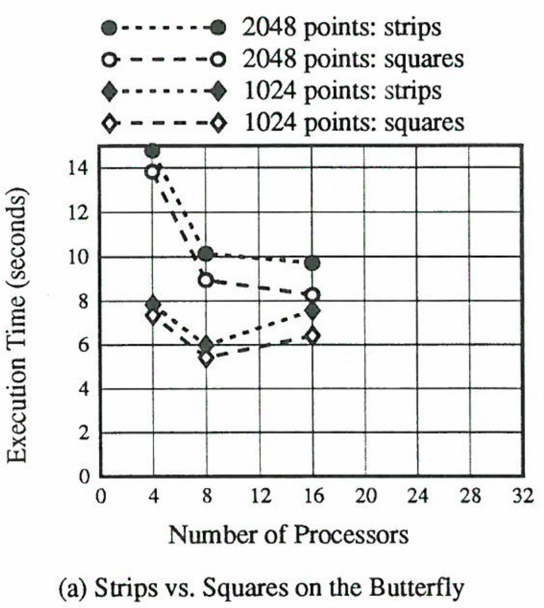
Hardware. We used four multiprocessors in our experiments. The first is a Sequent Symmetry Model A, which has 20 Intel 80386 processors connected by a shared bus to a 32 MB memory module. Each processor has a 64K cache (for both data and instructions) and an 80387 floating point accelerator [17].

A second machine is a 24 node BBN Butterfly GP1000. In addition to a Motorola 68020 processor, each node has 4 MB of local memory and a processor node controller which interacts with an omega network to make remote references when needed. Together, the 24 memory modules, the process node controllers, and the network form a single shared memory which all processors may access. Local memory access is about 12 times faster than remote access [1].

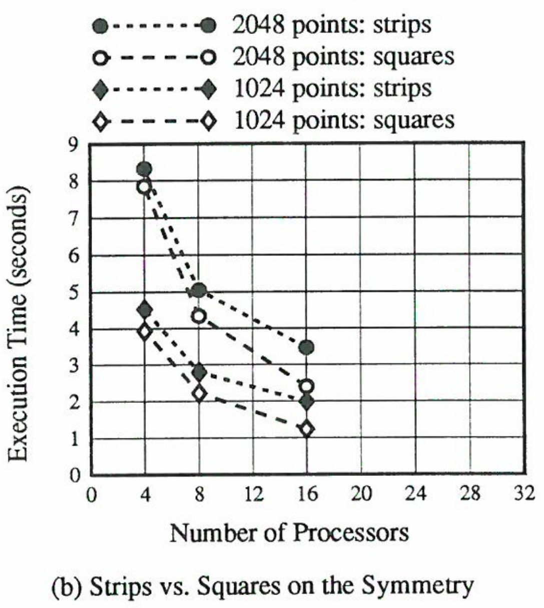
The remaining machines are hypercubes. On the 32 node Intel iPSC/2 each node contains an 80386 processor, an iPSC SX floating point accelerator, and 8 MB of memory. All inter-processor communication is through message passing [14]. On the 64 node NCUBE/7 each node has a custom main processor and 512 KB of memory. Like the iPSC/2, the NCUBE/7 is a nonshared memory machine [19].

Results. Figure 10 shows our results for problem sizes of 1K, 2K, and 4K points. 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

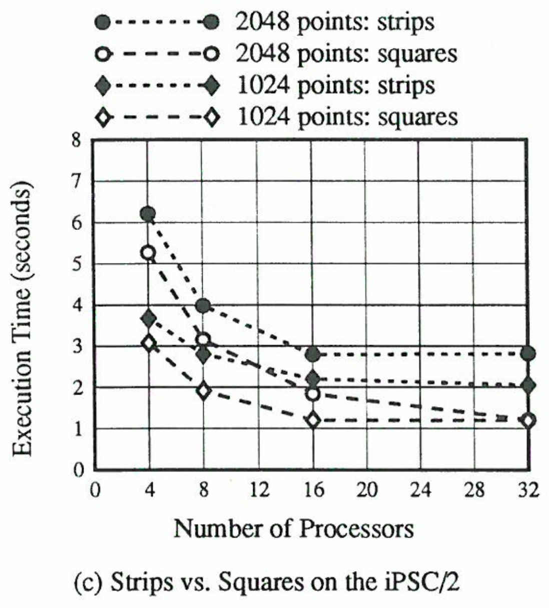
[4]For the shared memory machines, message passing routines were implemented using shared memory.



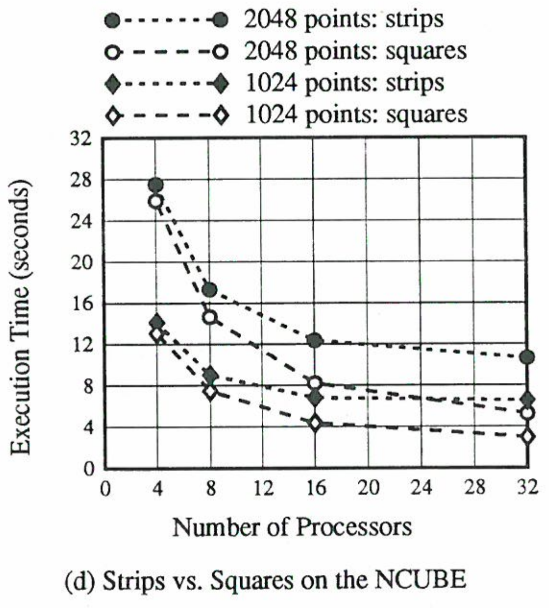
(a) Strips vs. Squares on the Butterfly



(b) Strips vs. Squares on the Symmetry



(c) Strips vs. Squares on the iPSC/2



(d) Strips vs. Squares on the NCUBE

Figure 10: Strips vs. Squares.

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 reasoning. We conclude that for SIMPLE, partitioning by blocks is superior to partitioning by strips.

As currently written, our program assumes equal-sized data sections for each processor. Note that when this assumption is relaxed, choosing the optimal data partitioning is further complicated by issues of load balance [23] and by an increase in the number of possible partitioning strategies.

6 Conclusion

We have presented the Phase Abstractions and shown how they can be used to write a parallel version of SIMPLE. Furthermore, we have demonstrated how the port and data ensembles facilitate the creation of alternate program implementations, and we have used this flexibility to study the issue of choosing the best data partitioning for SIMPLE.

Acknowledgments We wish to thank our colleagues on the Orca Project; Hans Mandt and the Advanced Systems Laboratory of Boeing Computing Services for their help in providing access to a Butterfly multiprocessor; and Walter Rudd and others at the Oregon Advanced Computing Institute for providing access to their iPSC/2 and NCUBE/7.

References

[1] Gail Alverson, *Abstractions for Effectively Portable Shared Memory Parallel Programs*, Ph.D. Thesis, University of Washington (1990).

[2] G. Alverson, W. Griswold, D. Notkin, and L. Snyder, A Flexible Communication Abstraction for Nonshared Memory Parallel Computing, *Proceedings of Supercomputing '90*, New York, New York, November 1990.

[3] T. S. Axelrod, P. F. Dubois, and P. G. Eltgroth, A Simulator for MIMD Performance Prediction – Application to the S-1 MkIIa Multiprocessor. *Proceedings of the International Conference on Parallel Processing* pp. 350-358 (1983).

[4] F. Berman, M. Goodrich, C. Koelbel, W.J. Robison III, and K. Showell. Prep-P: A Mapping Preprocessor for CHiP Computers, *Proceedings of the International Conference on Parallel Processing*, (August 1985), pp 731-733.

[5] W. Crowley, C. P. Hendrickson, and T. I. Luby, The Simple Code, Technical Report UCID-17715, Lawrence Livermore Laboratory (1978).

[6] David E. Culler and Arvind, Resource Requirements of Dataflow Programs. *Proceedings of the International Symposium on Computer Architecture*, pp. 141-150 (1988).

[7] Kattamuri Ekanadham and Arvind, SIMPLE: Part I, An Exercise in Future Scientific Programming. CSG Technical Report 273, MIT (1987).

[8] Dennis Gannon and Jiro Panetta, Simple on the CHiP. Technical Report 469, Computer Science Department, Purdue University (1984).

[9] Dennis Gannon and Jiro Panetta, Restructuring Simple for the CHiP Architecture. *Parallel Computing*, (1986) 3:305-326.

[10] Kevin Gates, Simple: An Exercise in Programming in Poker. Technical Report, Applied Mathematics, University of Washington (1989).

[11] R. E. Hiromoto, O. M. Lubeck and J. Moore, Experiences with the Denelcor HEP. *Parallel Computing*, 1:197-206 (1984).

[12] W. Griswold, G. Harrison, D. Notkin, and L. Snyder, Scalable Abstractions for Parallel Programming, *Proceedings of the Fifth Distributed Memory Computing Conference*, Charleston, South Carolina, (1990).

[13] Thomas John Holman, Processor Element Architecture for Non-Shared Memory Parallel Computers. Ph.D. Thesis, University of Washington (1988).

[14] Intel Corporation, *iPSC/2 User's Guide*, October, 1989.

[15] Jinling Lee, Extending the SIMPLE Program in Poker, Technical Report 89-11-07, Dept. of Computer Science and Engineering, University of Washington (1989).

[16] Calvin Lin and Lawrence Snyder, Portable Parallel Programming: Cross Machine Comparisons for SIMPLE. *Fifth SIAM Conference on Parallel Processing* (1991).

[17] Tom Lovett and Shreekant Thakkar. *The Symmetry Multiprocessor System,* Proceedings of the International Conference on Parallel Processing, (1988) pp. 303-310.

[18] J. M. Meyers, Analysis of the SIMPLE Code for Dataflow Computation. Technical Report MIT/LCS/TR-216, MIT (1979).

[19] NCUBE Corporation. *NCUBE Product Report*, Beaverton OR, 1986.

[20] Keshav Pingali and Anne Rogers, *Compiler Parallelization of SIMPLE for a Distributed Memory Machine*, Technical Report 90-1084, Cornell University (1990).

[21] J. Saltz, V. Naik, and D. Nicol. Reduction of the Effects of the Communication Delays in Scientific Algorithms on Message Passing MIMD Architectures," *SIAM J. Sci. Statist. Computing,* vol 8, number 1, (January 1987) s118-s134.

[22] Lawrence Snyder, *Applications of the "Phase Abstractions" for Portable and Scalable Parallel Programming,* to appear in Proceedings of the ICASE Workshop on Programming Distributed Memory Machines.

[23] David Socha, An Approach to Compiling Single-point Iterative Programs for Distributed Memory Computers, *Proceedings of the Fifth Distributed Memory Computing Conference* (1990).

[24] Mark R. Thistle and Burton J. Smith, A Processor Architecture for Horizon. *Proceedings of Supercomputing 88*, IEEE, pp. 35-41 (1988).