Abstractions for Portable, Scalable Parallel Programming

Gail A. Alverson
William G. Griswold
Calvin Lin
David Notkin
Lawrence Snyder

January 26, 1994

Abstract

In parallel programming, the need to manage communication costs, load imbalance, and irregularities in the computation puts substantial demands on the programmer. Key properties of the architecture, such as the number of processors and the costs of communication, must be exploited to achieve good performance. Coding these properties directly into a program compromises the portability and flexibility of the code because significant changes are usually needed to port or enhance the program. We describe a parallel programming model that supports the concise, independent description of key aspects of a parallel program—such as data distribution, communication, and boundary conditions—without reference to machine idiosyncrasies. The independence of such components improves portability by allowing the components of a program to be tuned independently, and encourages reuse by supporting the composition of existing components. The architecture-sensitive aspects of a computation are isolated from the rest of the program, reducing the need to make extensive changes to port a program. This model is effective in exploiting both data parallelism and functional parallelism. This paper provides programming examples, compares this work to related languages, and presents performance results.

1 Introduction

The diversity of parallel architectures puts the goals of performance and portability in conflict. Programmers are tempted to exploit machine details—such as the interconnection structure and the granularity of parallelism—to maximize performance. Yet software portability is needed to reduce the high cost of software development, so programmers are advised to avoid making machine-specific assumptions. The challenge, then, is to provide a parallel language that minimizes the tradeoff between performance and portability.[1] Such a language must allow a programmer to write code that assumes no particular architecture, allow a compiler to optimize the resulting code in a machine-specific manner, and allow a programmer to perform architecture-specific performance tuning without making extensive modifications to the source code.

[1]We define a program to be portable with respect to a given machine if its performance is competitive with machine-specific programs solving the same problem [2].

In recent years, a parallel programming style has evolved that might be termed *aggregate data-parallel* computing. This style of programming is characterized by:

- *Data parallelism.* The program's parallelism comes from executing the same function on many elements of a collection. Data parallelism is attractive because it allows parallelism to grow automatically—or *scale*—with the number of data elements and processors. SIMD architectures exploit this parallelism at a very fine grain.
- *Aggregate execution.* Multiple elements are placed on a processor and manipulated sequentially because, in practice, the number of data elements far exceeds the number of processors. This is attractive because it places groups of interacting elements onto the same processor, vastly reducing communication costs. Moreover, this approach uses good sequential algorithms locally, which is often more efficient than simply multiplexing parallel algorithms. Another benefit is that data can be passed between processors in batches to amortize communication overhead. Finally, when a computation on one data element is delayed waiting for communication, other elements may be processed.
- *Loose synchrony.* Although strict data parallelism depends on the "same" function being executed on every element, local variations in the nature or positioning of some elements can require different implementations of the same conceptual function. For instance, data elements on the boundary of a space have no neighbors with which to communicate, but data parallelism normally assumes that interior and exterior elements be treated the same. By executing a slightly different function on the boundaries, these exceptional cases are easily handled.

These features make the aggregate data-parallel style of parallel programming attractive because it can yield efficient programs when executed on typical MIMD architectures. However, without linguistic support this style of programming promotes inflexible programs through the embedding of performance-critical features as constants, such as the number of processors, the number of data elements, boundary conditions, the processor interconnection, and system-specific communication syntax. If the machine, its size, or the problem size changes, significant program changes to the fixed quantities are generally required. As a consequence, several languages have been introduced to support key aspects of this style. However, unless all aspects of this style are supported, performance, scalability, portability, or development cost can suffer.

For instance, good locality of reference is an important aspect of this programming style. Low-level approaches [25] allow programmers to hand-code data placement. The resulting code typically assumes one particular data decomposition, so if the program is ported to a platform that favors some other decomposition, extensive changes must be made or performance suffers. Other languages [4, 5, 15] give the programmer no control over data decomposition, leaving these issues to the compiler or hardware. But because the best choice of data decomposition depends on characteristics of the application, compilers can make poor data placement decisions. Many recent languages [6, 22] provide support for data decompositions, but hide communication operations from the programmer and thus do not encourage locality at the algorithmic level. Consequently, there is a reliance on automated means of hiding latency. Unfortunately, these techniques—multithreaded hardware, multiple lightweight threads, caches, and compiler optimizations that overlap communication and computation—cannot always hide all latency. The trend towards relatively faster processors and relatively slower memory access speeds exacerbates the situation.

Other languages provide inadequate control over the granularity of parallelism, requiring either one data point per process [21, 40], assuming some larger fixed granularity [14, 29], or including no notion of granularity at all, forcing the compiler or runtime system to choose the best granularity [15]. Given the diversity of parallel computers, no particular granularity can be best for all machines. Computers such as the CM-5 prefer coarse granularities; those such as the J Machine prefer finer granularity; and those such as the MIT Alewife and Tera computer benefit from having multiple threads per process. Also, few languages provide sufficient control over the algorithm that is applied on aggregate data, preferring to multiplex the parallel algorithm when there are multiple data points on a processor [40, 41].

Many language models do not adequately support loose synchrony. The boundaries of parallel computations often introduce irregularities that require significant coding effort. Languages with SIMD semantics force all processes to execute the branches of a conditional in lock-step. Because all processes execute the same code, this solution leads to programs with many conditionals, increasing code size and making them difficult to understand and modify, as well as potentially inefficient. Programming in a typical MIMD-style language is not much cleaner. For instance, writing a slightly different function for each type of boundary process is problematic because a change to the algorithm is likely to require all versions to be changed.

In this paper we describe language abstractions—a programming model—that fully support the aggregate data-parallel programming style. This model can serve as a foundation for portable, scalable MIMD languages that preserve the performance available in the underlying machine. Our belief is that for many tasks, programmers—and not compilers or runtime systems—can best handle the performance-sensitive aspects of a parallel program. This belief leads to three principles for designing our language abstractions.

First, we provide abstractions that are efficiently implementable on all MIMD architectures, along with specific mechanisms to handle the common types of parallelism, data distribution, and boundary conditions. Our model is based on a practical MIMD computing model called the Candidate Type Architecture (CTA) [42].

Second, the insignificant but diverse aspects of computer architectures are hidden. If exposed to the programmer, assumptions based on these characteristics can be sprinkled throughout a program, making portability difficult. Examples of characteristics that are hidden include the details of the machine's communication style and the processor (or memory) interconnection topology. For instance, one machine might provide shared memory and another message passing, but either can be implemented with the other in software.

Third, architectural features that are essential to performance are exposed and parameterized in an architecture-independent fashion. A key characteristic is the speed, latency, and per-message overhead of communication relative to computation. As the cost of communication increases relative to computation, communication costs must be reduced by aggregating more processing onto a smaller number of processors, or by finding ways to increase the overlap of communication and computation.

The result is the *Phase Abstractions* parallel programming model, which provides control over granularity of parallelism, control over data partitioning, and a hybrid data and function parallel construct that supports concise description of boundary conditions. The core of our solution is the *ensemble* construct that allows a global data structure to be defined and distributed over processes, and allows the granularity—and the location of data elements—to be controlled by load-time parameters. The ensemble also has a code form for describing what operations to execute on which subarrays and for handling boundary conditions. Likewise, interprocessor connections are described with a port ensemble, providing similar flexibility. By using ensembles for all three components of a global operation—data, code and communication—they can be scaled together with the same load time parameters. Because the three parts of an ensemble and the boundary conditions are specified independently, reusability is enhanced.

The remainder of this paper is organized as follows. We first present our solution to the problem by describing our architectural model and the basic language model—the CTA and the Phase Abstractions. Section 3 then gives a detailed illustration of our abstractions, using the Jacobi Iteration as an example. To demonstrate the expressiveness and programmability of our abstractions, Section 4 shows how simple array language primitives can be built on top of our model. In Section 5 we discuss the advantages of our programming model with respect to performance and portability, and in Section 6 we present experimental evidence that the Phase Abstractions support portable parallel programming. Finally, we compare Phase Abstractions with related languages and models, and close with a summary.

2 Phase Abstractions

In sequential computing, languages such as C, Pascal and Fortran have successfully combined efficiency with portability. What do these languages have in common that make them successful? All are based on a model in which the execution of a program is a sequence of operations that manipulate some infinite random-access memory. This programming model succeeds because it preserves the characteristics of the von Neumann machine model, which itself is a faithful representation of sequential computers. While these models are never literally implemented—unit-cost access to infinite memory is only an illusion provided by virtual memory, caches and backing store—the model is accurate for the vast majority of programs. There are only rare cases, such as programs that perform extreme amounts of disk I/O, where the deviations from the model are costly the to the programmer. It is critical that the von Neumann model capture machine features that are relevant to performance. If some essential machine features were ignored, better algorithms could be developed using a more accurate machine model. Together, the von Neumann machine model and its accompanying programming model allow languages such as C and Fortran to be both portable and efficient.

In the parallel world, the Candidate Type Architecture (CTA) plays the role of the von Neumann model,[2] and the Phase Abstractions the role of the programming model. Finally, the sequential languages are replaced by languages based on the Phase Abstractions, of which Orca C is an example [31, 33].

The CTA. The CTA [42] is an asynchronous MIMD model. It consists of P von Neumann processors that execute independently. Each processor has its own local memory, and the processors communicate through some sparse but otherwise unspecified communication network. Here "sparse" means that the network has a constant degree of connectivity. The network topology is intentionally left unbound to provide maximum generality. Finally, the model includes a global controller that can communicate with all processors through a low bandwidth network. Logically,

[2]The more recent BSP [44] and LogP [8] models present a similar view of a parallel machine and for the most part suggest a similar way of programming parallel computers.

the controller provides synchronization and low bandwidth communication such as a broadcast of a single value.

Although it is premature to claim that the CTA is as effective a model as the von Neumann model, it does appear to have the requisite characteristics: It is simple, makes minimal architectural assumptions, but captures enough significant features that it is useful for developing efficient algorithms. For example, the CTA's unbound topology does not bias the model towards any particular machine, and the topologies of existing parallel computers are typically not significant to performance. However, the distinction between global and local memory references is key, and this distinction is clear in the CTA model. Finally, the assumption of a sparse topology is realistic for all existing medium and large scale parallel computers.

The Phase Abstractions extend the CTA in the same way that the sequential imperative programming model extends the von Neumann model. The main components of the Phase Abstractions are the *XYZ levels of programming* and *ensembles* [1, 19, 43].

2.1 XYZ Programming Levels

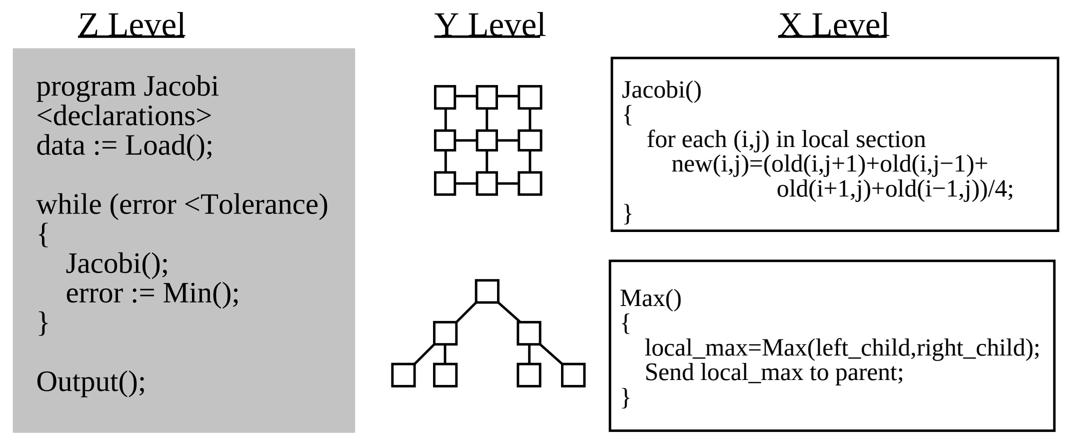
A programmer's problem-solving abilities can be improved by dividing a problem into small, manageable pieces—assuming the pieces are sufficiently independent to be considered separately. Additionally, these pieces can often be reused in other programs, saving time on future problems. One way to build a parallel program from smaller reusable pieces is to compose a sequence of independently implemented phases, each executing some parallel algorithm that contributes to the overall solution. At the next conceptual level, each such phase is comprised of a set of cooperating sequential processes that implements the desired parallel algorithm. Each sequential process may be developed separately. These levels of problem solving—program, phase, and sequential process, also called the Z, Y, and X levels—have direct analogies in the CTA.

The *X level* corresponds to the individual von Neumann processors of the CTA, and an X level program specifies the sequential code that executes in one process. Each process can communicate with other processes by passing messages. Because the model is MIMD, each process can execute different code.

The *Y level* is analogous to the set of von Neumann processors cooperating to compute a parallel algorithm, forming a phase. The Y-level may specify how the X-level programs are connected to each other for message passing. Examples of phases include parallel implementations of the FFT, matrix multiplication, matrix transposition, sort, and global maximum. A phase has a characteristic communication structure induced by the data dependencies among the processes. For example, the FFT induces a butterfly, while Batcher's sort induces a hypercube [1].

Finally, the *Z level* corresponds to the actions of the CTA's global controller, where sequences of parallel phases are invoked and synchronized. A Z level program gives the high level logic of the computation by specifying the sequential invocation of phases (although their *execution* may overlap) that are needed to solve complex problems. For example, the Car-Parrinello molecular dynamics code simulates the behavior of a collection of atoms by iteratively invoking a series of phases that perform FFT's, matrix products, and other computations [45]. In Z-Y-X order, these three levels give a top-down view of a parallel program.

Example: XYZ Levels of the Jacobi Iteration. Figure 1 illustrates the XYZ levels of programming for the Jacobi Iteration. The Z level consists of a loop that invokes two phases, one



```
program Jacobi
<declarations>
data := Load();

while (error <Tolerance)
{
   Jacobi();
   error := Min();
}

Output();
```

```
Jacobi()
{
   for each (i,j) in local section
      new(i,j)=(old(i,j+1)+old(i,j−1)+
                old(i+1,j)+old(i−1,j))/4;
}
```

```
Max()
{
   local_max=Max(left_child,right_child);
   Send local_max to parent;
}
```

Figure 1: XYZ Illustration of the Jacobi Iteration

called Jacobi(), which performs the over-relaxation, the other called Max(), which computes the maximum difference that is used to test for termination.

Each Y level phase is of a collection of processes executing concurrently. Here, the two phases are graphically depicted with squares representing processes and arcs representing communication between processes. The Jacobi phase uses a mesh interconnection topology, and the Max phase uses a binary tree. Other details of the Y level, such as the distribution of data, are not shown in this figure but will be explained in the next subsection.

Finally, a sketch of the X level program for the two phases is shown at the right of Figure 1. The X level code for the Jacobi phase assigns to each data point the average of its four neighbors. The Max phase finds, for all data points, the largest difference between the current iteration and the previous iteration. □

A Z level program is basically a sequential program that provides control flow for the overall computation. An X level program, in its most primitive form, can also be viewed as a sequential program with additional communication operations that allow it to interact with other processes. Although parallelism is not explicitly specified at the X and Z levels, these two levels may still contain useful parallelism. For example, phase execution may be pipelined, and the X level processes can execute on superscalar architectures to achieve instruction-level parallelism.

It is the Y level that specifies scalable parallelism and most clearly departs from a sequential program. *Ensembles* support the definition and manipulation of this parallelism.

2.2 Ensembles

The Phase Abstractions use the *ensemble* structure to describe data structures and their partitioning, process placement, and process interconnection. In particular, an ensemble is a partitioning of a set of elements—data, codes, or port connections—into disjoint *sections*. Each section represents a thread of execution, so the section is a unit of concurrency, and the degree of parallelism is modulated by increasing or decreasing the number of sections. These sections are bound to processors for execution of a phase: Each process computes on its local section of data and exchanges non-local data with neighbors. Because all three aspects of parallel computation—data, code and communication—are unified in the ensemble structure, all three components can be reconfigured and scaled in a coherent, concise fashion to provide flexibility and portability.

The data-partitioning aspect of ensembles is analogous to the data-partitioning features supplied in languages such as HPF [22], but since ensembles also incorporate communication and code components, ensembles provide greater control over performance. For instance, the *code ensemble* may logically have identical procedures in each section, but they may be different, effectively yielding MIMD execution with the convenience of SPMD programming. Code ensembles also support the handling of boundary conditions, which frequently arise in otherwise regular computations and can be difficult to handle efficiently without compromising development costs or portability.

A *data ensemble* is a data structure with a partitioning. At the Z level the data ensemble provides a logically global view of the data structure. At the X level each process sees only the portions of the ensembles mapped to its section. Such a portion is seen as a locally defined data structure with local indexing. For example, the 6×6 data ensemble in Figure 2 has a global view with indices $[0:5] \times [0:5]$, and a local view of 3×3 subarrays with indices $[0:2] \times [0:2]$. The mapping of the global view to the local view is performed at the Y level and will be described in Section 3. The use of local indexing schemes allows an X level process to refer to generic array bounds rather than to global locations in the data space. Thus, the same X level source code can be used for multiple processes.

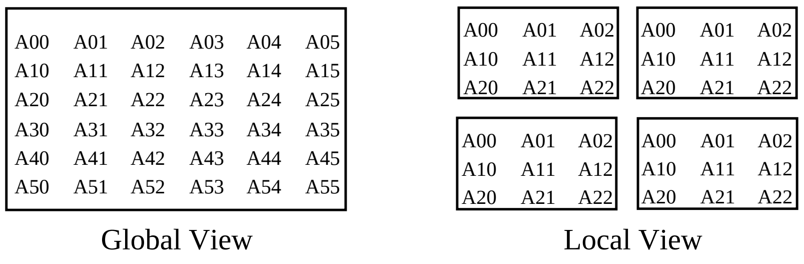


Figure 2: A 6×6 Array (left) and its corresponding Data Ensemble for a 2×2 array of sections.

A *code ensemble* is a collection of procedures with a partitioning. The code ensemble gives a global view of the processes performing the parallel computation. When the procedures in the ensemble differ the model is MIMD; when the procedures are identical the model is SPMD. Figure 3 shows a code ensemble for the Jacobi phase in which all processes execute the Jacobi() function.

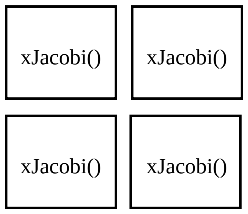


Figure 3: Illustration of a Code Ensemble

Finally, a *port ensemble* defines a logical communication structure by specifying a collection of port name pairs. Each pair of names represents a logical communication channel between two sections, and each of these port names is bound to a local port name of the X level. Hence, the port ensemble specifies the overall communication structure of a phase. Figure 4 depicts a port ensemble for the Jacobi phase. For example, the north port (**N**) of one process is bound to the south port (**S**) of its neighboring process.

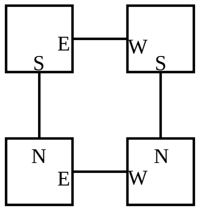


Figure 4: Illustration of a Port Ensemble

A Y level phase is composed of three primary components: a code ensemble, a port ensemble that connects the code ensembles processes, and data ensembles that provide arguments to the processes of the code ensemble. The sections of each ensemble are ordered numerically so that the i^{th} section of a code ensemble is bound to the i^{th} section of each data and port ensemble. This correspondence allows each section to be allocated to a processor for normal sequential execution: The process executes on that processor, the data can be stored in memory local to that processor, and the ports define connections for interprocessor communication. Consequently, the i^{th} sections of all ensembles are assigned to the same processor to maintain locality for any phase invocation. If two phases share a data ensemble but require different partitionings for best performance, a separate phase may be needed to move the data.

The Z level logically stores ensembles in Z level variables, composes them into phases and stores their results. The phase invocation interface between the Z and X levels encourages modularity because the same X level code can be invoked with different ensemble parameters in the same way that procedures are reused in sequential imperative languages.

The ensemble abstraction helps hide the diversity of parallel architectures. However, to map well to individual architectures the abstraction must be parameterized, for example, by the number of processors and the size of the problem. This parameterization is illustrated in the next section.

3 Ensemble Example: Jacobi

To provide a better understanding of the ensembles and the Phase Abstractions, we now complete the description of the Jacobi program. We adopt notation from the proposed Orca C language [30, 31], but other languages based on the Phase Abstractions are possible (see Section 4).

3.1 Overall Program Structure

As shown in Figure 5, a Phase Abstractions program consists of X, Y, and Z descriptions, plus a list of machine configuration parameters that are used by the program to adapt to different execution environments. In this case, two runtime parameters are accepted: `Processors` and `shape`. The

```
#define Rows 1                           /* Constants to define the shape */
#define Cols 2                           /* of the logical processor array */
#define TwoD 3

program Jacobian (shape, Processors)
    switch (shape){                      /* Configuration Computation */
    case Rows: rows = Processors;
               cols = 1;
               break;
    case Cols: rows = 1;
               cols = Processors;
               break;
    case TwoD: Partition2D(&rows, &cols, Processors);
               break;
    }
(rows, cols, Processors)                 /* Configuration Parameter List */

    <data ensemble definitions>;         /* Y Level */
    <port ensemble definitions>;
    <code ensemble definitions>;
    <process definitions>;               /* X Level */

begin                                    /* Z Level */
    Input();
    while (tolerance > delta)
    {
        Jacobi(values);
        tolerance = Max(p, newP);
    }
    Output();
end
```

Figure 5: Overall Phase Abstraction Program Structure

first parameter is the number of processors, while the second specifies the shape of the processor array. As will be discussed later, the program uses a 2D data decomposition, so by setting `shape` to `Rows` (`Cols`) we are choosing a horizontal strips (vertical strips) decomposition. (The function `Partition2D()` computes values of `rows` and `cols` such that (`rows * cols`) = `Processors` and the difference between `rows` and `cols` is minimized.) With this particular configuration computation this program, through the use of different load time parameters, can adapt to different numbers of processors and can assume three different data decompositions. The configuration computation is executed once at load time.

3.2 Z Level of Jacobi

After the program is configured, the Z level program is executed, which initializes program variables, reads the input data, and then iteratively invokes the Jacobi and Max phases until convergence is reached, at which point an output phase is invoked. The data, processing, and communication components of the Jacobi and Max phases are specified by defining and composing code, data and port ensembles as described below.

3.3 Y Level: Data Ensembles

This Jacobi iteration uses a single array to store floating point values at each point of a 2D grid. Parallelism is achieved by partitioning this array into contiguous 2D blocks:

```
    partition   block[r][c]          float p[rows][cols];
```

The array `p` has dimensions (`rows * cols`) and is partitioned onto a section array (process array) of size (`r * c`). The keyword `partition` identifies `p` as an ensemble array, and `block` names this partitioning so that it can be reused to define other ensembles. This partitioning corresponds to the one in Figure 2 when `rows=6`, `cols=6`, `r = 2` and `c = 2`, and this ensemble declaration belongs in the <*data ensembles*> meta-code of Figure 5.

(Section 5 shows how an alternate decomposition is declared.)

The values of `r` and `c` are assumed to be specified in the program's configuration parameter list. Each section is implicitly defined to be of size (`s * t`), where $\texttt{s} = \frac{\texttt{rows}}{\texttt{r}}$ and $\texttt{t} = \frac{\texttt{cols}}{\texttt{c}}$. (If `r` does not divide `rows` evenly, some sections will have $\texttt{s} = \lceil\frac{\texttt{rows}}{\texttt{r}}\rceil$ while others will have $\texttt{s} = \lfloor\frac{\texttt{rows}}{\texttt{r}}\rfloor$.) Consequently, X level processes contain no assumptions about the data decomposition except the dimension of the subarrays so the program can scale in both the number of logical processors and in the problem size.

3.4 Jacobi Phase

Port Ensemble. The Jacobi phase computes for each point the average of its four nearest neighbors, implying that each section will communicate with its four nearest neighbor sections, as shown in Figure 4. The following Y level ensemble declaration defines the appropriate port ensemble:

```
    Jacobi.portnames    <-->   N, E, W, S      /* North, East, West, South */

    Jacobi[i][j].port.N <--> Jacobi[i-1][j].port.S where 1 <= i < r, 0 <= j < c
    Jacobi[i][j].port.W <--> Jacobi[i][j-1].port.E where 0 <= i < r, 1 <= j < c
```

The first line declares the phase's port names so the following bindings can be specified. The second and third lines define a mesh connectivity between Y level port names. This port ensemble declaration does not specify connections for the ports that lie on the boundaries. In this case these unbound ports are bound to *derivative functions*, which compute boundary conditions using data local to the section. The following binds derivative functions to ports on the edges of Jacobi.

```
Jacobi[0][i]   .port.N   receive  <-->  RowZero, 0 <= i < c
Jacobi[i][c-1] .port.E   receive  <-->  ColZero, 0 <= i < r
Jacobi[i][0]   .port.W   receive  <-->  ColZero, 0 <= i < r
Jacobi[r-1][i] .port.S   receive  <-->  RowZero, 0 <= i < c
```

`RowZero` and `ColZero` are defined as:

```
double RowZero()
{
    static double row[1:s]              /* default initialized to 0's */
    return row;
}

double ColZero()
{
    static double col[0][1:t]           /* default initialized to 0's */
    return col[0];
}
```

The values of `s` and `t` are determined by the process' X level function—in this case `xJacobi()`.

In the absence of derivative functions, X level programs could check for the existence of neighbors, but such tests complicate the source code, increasing the chance of introducing errors. Also, as Section 5 shows, even modestly more complicated boundary conditions can lead to a proliferation of special case code.

Code Ensemble. To define the code ensemble for Jacobi, each of the `r * c` sections is assigned an instance of the `xJacobi()` code:

```
Jacobi[i][j].code  <-->  xJacobi();    where 0 <= i < r, 0 <= j < c
```

Because Jacobi contains heterogeneity only on the boundaries, which in this program is handled by derivative functions, all the functions are the same. In general, however, the only restriction is that the types of the functions must conform in the argument types and return type specified in a phase invocation.

X Level. The X level code for Jacobi is shown in Figure 6. It first sends edge values to its four neighbors, it then receives boundary values from its neighbors, and finally uses the five point stencil to compute the average of each interior point. Several features of the X level code are noteworthy:

- *parameters*—The arguments to the X level code establish a correspondence between local variables and the sections of the ensembles. In this case, the local *value* array is bound to a block of ensemble *values*.

```
xJacobi(value[1:s][1:t])
    double value[0:s+1][0:t+1];           /* extra storage on all four sides */
    port   North, East, West, South;
{
    double new_value[0:s+1][0:t+1];
    int    i, j;

    /* Send neighbor values */
    North <== value [1][1:t];             /* 1:t is an array slice */
    East  <== value[1:s][t];
    West  <== value[1:s][1];
    South <== value[s][1:t];

    /* Receive neighbor values */
    value[s+1][1:t]    <== South;
    value[1:s][0]    <== West;
    value[1:s][t+1]  <== East;
    value[0][1:t]    <== North;

    for (i=1; i<=s; i++)
    {
        for (j=1; i<=t; i++)
        {
            new_value[i][j] = (value[i][j+1] + value[i][j-1] +
                               value[i+1][j] + value[i-1][j]) / 4;
        }
    }

    for (i=1; i<=s; i++)
    {
        for (j=1; i<=t; i++)
        {
            value[i][j] = new_value[i][j];
        }
    }
}
```

Figure 6: X Level Code for the Jacobi Phase

- *communication*—Communication is specified using the transmit operator (`<==`), for which a port name on the left specifies a send of the righthand side, and a port on the right indicates a receive into the variable on the lefthand side. The semantics are that receive operations block, but sends do not.

- *uniformity*—Because derivative functions are used, the `xJacobi()` function contains no tests for boundary conditions when sending or receiving neighbor values.

- *border values*—The values `s` and `t`, used to define the bounds of the *value* array, are parameters derived from the size of the sections of the data ensemble. Also, *value* is declared to be one element wider on each side than the incoming array argument to hold values from neighboring sections. This extra storage is explicitly specified by the difference between the local declaration, `x[0:s+1][0:t+1]`, and the formal declaration, `x[1:s][1:t]`, where the upper bounds of these array declarations are inclusive.

- *array slices*—Slices provide a concise way to refer to an entire row (or in general, a d-dimensional block) of data. When slices are used in conjunction with the transmit operator (`<==`), the entire block is sent as a single message, thus reducing communication overhead.

The Complete Phase. To summarize, the data ensemble, the port ensemble, and the code ensemble collectively define the Jacobi phase. Upon execution the sections declared by the configuration parameters are logically connected in a nearest-neighbor mesh, and each section of data is manipulated by one `xJacobi()` process. The end result is a parallel algorithm that computes the Jacobi iteration.

3.5 Max Phase

The Max phase finds the maximum change of all grid points, and uses the same data ensemble as the Jacobi phase. The port ensemble is shown graphically in Figure 8 and is defined below.

```
Max.portnames       <-->   P, L, R         /* Parent, Left, Right */

Max[i].port.R       <--> Max[2*i].port.P       where 0 <= i < r*c/2 - 1
Max[i].port.L       <--> Max[2*i+1].port.P     where 0 <= i < r*c/2 - 1
```

The derivative functions for this phase are bound so that for each leaf section a "receive" from the `Left` port or `Right` port will return the value computed by the `Smallest_Value()` function, and a send from the root's unbound `Parent` port will be a no-op.

```
Max[i].port.L   receive  <-->  Smallest_Value() where r*c/2 -1 <= i < Processors
Max[i].port.R   receive  <-->  Smallest_Value() where r*c/2 -1 <= i < Processors
Max[i].port.P   send     <-->  No_Op()          where i = 0
```

The `Smallest_Value()` derivative function simply returns the smallest value that can be represented on the architecture. The code ensemble for this phase is similar to the Jacobi phase, except that `xMax()` replaces `xJacobi()`. (See Figure 7.)

With a more complicated application than Jacobi, the benefit of using ensembles increases and the cost of using them is amortized over a larger program. The cost of using ensembles will decrease

```
xMax(value[1:s][1:t], new_value[1:s][1:t])
    double value[1:s][1:t];
    double new_value[1:s][1:t];
    port   Parent, Left, Right;
{
    int    i, j;
    double local_max;
    double temp;

    /* Compute the local maximum */
    local_max = abs(value[0][0] - new_value[0][0]);
    for (i=1; i<=s; i++)
    {
        for (j=1; j<=t; j++)
        {
            temp = abs(value[i][j] - new_value[i][j]);
            local_max = Max(temp, local_max);
        }
    }

    /* Compute the global maximum */
    temp      <== Left;                     /* receive */
    local_max  =  Max(temp, local_max);
    temp      <== Right;                    /* receive */
    local_max  =  Max(temp, local_max);
    Parent    <== local_max;                /* send */

    /* Broadcast the result */
    local_max <== Parent;                   /* receive */

    Left      <== local_max;                /* send */
    Right     <== local_max;                /* send */
}
```

Figure 7: X Level Code for the Max Phase

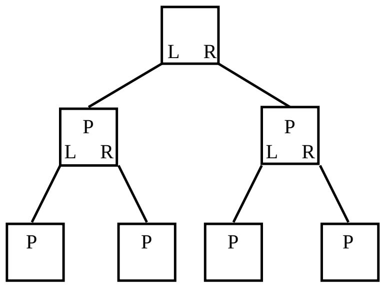


Figure 8: Illustration of a Tree Port Ensemble

as libraries of ensembles, phases, derivative functions and X level codes are built. For example, the Max phase of Jacobi is common to many computations and would not normally be defined by the programmer.

4 High Level Programming with the Phase Abstractions

Phase Abstractions are not a programming language, but rather a foundation for the development of parallel programming languages that support the coding of efficient, scalable, portable programs. Orca C, used in the previous section, is a literal, textual instantiation of the Phase Abstractions. It clearly shows the power of the Phase Abstractions, but some may find it too low-level and tedious.

Other languages faithful to the Phase Abstractions need not suffer from Orca C's drawbacks. By precisely understanding the uses for which a language is intended, high-level, easy-to-use languages can be built from the Phase Abstractions. Such languages might introduce entirely different syntax specific to a particular domain, thus providing simplicity at the expense of generality.

In fact, a departure from the literal Orca C language is not required to achieve an elegant programming style. By adopting certain conventions, it is possible to build reusable abstractions directly on top of Orca C. By staying within the Orca C framework, this solution has the advantage that different sublanguages can be used together for a single large problem that requires diverse abstractions for good performance. As an example, consider the design of an APL-like array sublanguage for Orca C.

Recall that an X level procedure receives two kinds of parameters—global data passed as arguments and port connections—that support two basic activities: computations on data and communication. However, it is possible to constrain X level functions to perform just one of these two tasks—a local computation or a communication operation. Furthermore, we can constrain a particular phase to consist of only computation functions or only communication functions. That is, there could be separate computation phases and communication phases. For example, there can be X level computation functions for adding integers, computing the minimum of some values, or sorting some elements. There can be X level communication functions for shifting data cyclically in a ring, for broadcasting or aggregating data, or for communicating up and down a tree structure. Reductions, which naturally combine both communication and computation, are notable exceptions where the separation of communication from computation is not desirable. For such operations it suffices to define a communication-oriented phase that takes an additional function parameter for combining the results of communications.

To illustrate, reconsider the Jacobi example. Rather than specify the entire Jacobi iteration in one X level process, each communication operation can be performed in a separate phase and the results can be combined by Z level add and divide phases. The test for convergence is computed at the Z level by subtracting the old array from the new one and performing a Max reduction on the differences. The program skeleton in Figure 9 illustrates this method, providing examples of X level functions for `+` (referred to as `operator+` in the syntactic style of C++), shift, and reduce; the Z level code shows how data ensembles are declared and how phase structures for add, left-shift and reduce are initialized. The divide and subtract phases are analogous to `add`, and the other shift functions are analogous to the left-shift.

There are three consequences of this approach. First, the interface to a phase is substantially simplified. Second, some problems are harder to describe because it is not possible to combine computation and communication within a single X level function. Finally, X level functions (and the phases that they comprise) are smaller and are more likely to perform just one task, increasing their composability and reusability.

Although the array sublanguage defined here is similar to APL, it has some salient differences. Most significantly, the Orca C functions operate on subarrays, rather than individual elements, which means that fast sequential algorithms can be applied to subarrays. So while this solution achieves some of the conciseness and reusability of APL, it does not sacrifice control over data decompositions or lose the ability to use separate global and local algorithms. This solution also has the advantage of embedding an array language in Orca C, allowing other programming styles to be used as they are needed.

5 Discussion

The power of the Phase Abstractions comes from the decomposition of parallel programs into X, Y and Z levels, the encoding of key architectural properties as simple parameters, and the concept of ensembles, which allows data, port and code decompositions to be specified and reused as individual components. The three types of ensembles work together to allow the problem and machine size to be scaled. In addition, derivative functions allow a single X level program to be used for multiple processes even in the presence of boundary conditions. This section discusses the Phase Abstractions with respect to performance and expressiveness.

Portability and Scalability. When programs are moved from one platform to another they must adapt to the characteristics of their host machine if they are to maintain good performance. If such adaptation is automatic or requires only minor effort, portability is achieved. The Phase Abstractions support portability and scalability by encoding key architectural characteristics as ensemble parameters and by separating phase definitions into several independent components.

Changes to either the problem size or the number of processors are encapsulated in the data ensemble declaration. As in Section 3, we relate the size of a section (`s * t`), the overall problem size (`rows * cols`), and the number of sections (`r * c`) as follows:

```
s = rows/r
t = cols/c
```

```
xproc TYPE[1:s][1:t] operator+(TYPE x[1:s][1:t], TYPE y[1:s][1:t])
  {
  TYPE result[1:s][1:t];
  int i, j;

  for (i=1; i<=s; i++)
    for (j=1; j<=t; i++)
      result[i][j] = x[i][j] + y[i][j];

  return result;
  }


xproc void shift(TYPE val[1:s][1:t])
  port write_neighbor,
       read_neighbor;
  {
  TYPE temp[1][1:t];
  int i;

  write_neighbor <== val[1];

  temp <== read_neighbor;
  for (i=2; i<=t; i++)
    val[i-1] = val[i];

  val[s] = temp;
  }

...

xproc int reduce(TYPE val[1:k], TYPE*() op)
  port Parent,
       Child[1:n];
  {
  int i;
  TYPE accum;

  accum = val[1];
  for (i=2; i<=k; i++)
    accum = op(accum,val[i]);

  for (i=1; i<=n; i++)
    accum = op(accum,Child[i]);

  Parent <== accum;
  }
```

```
begin Z

  double X[1:J][1:K], OldX[1:J][1:K];
  ...
  phase operator+;
  phase Left;
  phase Reduce;
  ...

  operator+.code = operator+;

  Left.code = shift;
  Left.port = WriteLeft(Zero);
  ...

  Reduce.code = reduce;
  Reduce.port = Tree(No_Op, Largest_Value, Largest_Value);

  do
    {
    OldX = X;
    X := (Left(X) + Right(X) + Up(X) + Down(X)) / 4;
    } while (Reduce(X - OldX, max) > tolerance);

end Z
```

Figure 9: Jacobi Written in an Array Style Using Orca C

The problem size scales by changing the values of `rows` and `cols`, the machine size scales by changing the values of `r` and `c`, and the granularity of parallelism is controlled by altering either the number of processors or the number of sections in the ensemble declaration. This flexibility is an important aspect of portability because different architectures favor different granularities.

While it is desirable to write programs without making assumptions about the underlying machine, knowledge of machine details can often be used to optimize program performance. Therefore, tuning may sometimes be necessary. For example, it may be beneficial for the logical communication graph to match the machine's communication structure. Consider embedding the binary tree of the Max phase onto a mesh architecture: Some logical edges must span multiple physical links. This edge dilation can be eliminated with a connectivity that allows comparisons along each row of processors and then along a single column (see Figure 10).

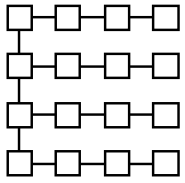


Figure 10: Rows and Columns to Compute the Global Maximum

To address the edge dilation problem the fixed binary tree presented in Section 3 can be replaced by a new port ensemble that uses a tree of variable degree. Such a solution is shown in Figure 11, where the child ports are represented by an array of ports. This new program can use either a binary tree or the "rows and columns" approach. The port ensemble declaration for the latter approach is shown below.

```
/* Rows and Columns communication structure */
Max[i][j].port.P <--> Max[i][j-1].port.C[0] 0 <= i < r, 1 <= j < c
Max[i][0].port.P <--> Max[i-1][0].port.C[1] 1 <= i < r
```

With the code suitably parameterized, this program can now execute efficiently on a variety of architectures by selecting the proper port ensemble.

Locality. The best data partitioning depends on factors such as the problem size, the machine size, the machine's communication and computation characteristics, and the application's communication patterns.

In the Phase Abstractions model, changes to the data partitioning are encapsulated by data ensembles. For example, to define a 2D block partitioning on P processors, the configuration code can define the number of sections to be `r =`$\sqrt{P}$, `c =`$\sqrt{P}$. If a 1D strip partitioning is desired, the number of sections can simply be defined to be `r = 1, c = P`. This strip decomposition requires that each process have only East-West neighbors instead of the four neighbors used in the block decomposition. By using the port ensembles to bind derivative functions to unused ports—in this case the North and South—the program can easily accommodate this change in the number of neighbors. No other source level changes are required.

```
xMax(value[1:s][1:t], new[1:s][1:t], numChildren)
    double  value[1:s][1:t];
    double  new_value[1:s][1:t];
    port    Parent, Child[numChildren];
{
    int     i, j;
    double  local_max;
    double  temp;

    /* Compute the local maximum */
    local_max = abs(value[0][0] - new_value[0][0]);
    for (i=1; i<=s; i++)
    {
        for (j=1; i<=t; i++)
        {
            temp = abs(value[i][j] - new_value[i][j]);
            local_max = Max(temp, local_max);
        }
    }

    /* Compute the global maximum */
    for (i=0; i<numChildren; i++)
    {
        temp     <== Child[i];               /* receive */
        local_max =  Max(temp, local_max);
    }
    Parent    <== local_max;                 /* send */

    /* Broadcast the result */
    local_max <== Parent;                    /* receive */
    for (i=0; i<numChildren; i++)
    {
        Child[i] <== local_max;              /* send */
    }
}
```

Figure 11: Parameterized X Level Code for the Max Phase

The explicit dichotomy between local and non-local access encourages the use of different algorithms locally and globally. Batcher's sort, for example, benefits from this approach (see Section 1). This contrasts with most approaches in which the programmer or compiler identifies as much fine-grained parallelism as possible and the compiler aggregates this fine-grained parallelism to a granularity appropriate for the target machine.

Boundary Conditions. Typically, processes on the edge of the problem space must be treated separately. In the Jacobi Iteration, for example, 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 in most realistic applications these lead to convoluted code. For example, SIMPLE can have up to nine different cases—depending on which portions of the boundaries are contained within a process—and these conditionals can lead to code that is dominated by the treatment of exceptional cases [18, 38].

For example, suppose a program with a block decomposition assumes in its conditional expression that a process is either a NorthEast, East, or SouthEast section, as shown below:

```
if (NorthEast)
{
    /* special case 1 */
}
else if (East)
{
    /* special case 2 */
}
else if (SouthEast)
{
    /* special case 3 */
}
```

A problem arises if the programmer then decides that a vertical strips decomposition would be more efficient. The above 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 the above boundary condition code.

Our model, however, attempts to insulate the port and code ensembles from changes in the data decomposition: Processes send and receive data through ports that in some cases involve interprocess communication and in other cases invoke derivative functions. The handling of boundary conditions has thus been decoupled from the X level source code. Instead of cluttering up the process code, special cases due to boundary conditions are handled at the problem level where they naturally belong.

Reusability. The same characteristics that provide flexibility in the Phase Abstractions also encourage reusability. For example, the Car-Parrinello molecular dynamics program [45] consists of several phases, one of which is computed using the Modified Gram-Schmidt (MGS) method of solving QR factorization. Empirical results have shown that the MGS method performs best with a 2D data decomposition [34]. However, other phases of the Car-Parrinello computation require a 1D decomposition, so in this case a 1D decomposition for MGS yields the best performance since it avoids data movement between phases. This illustrates that a reusable component is most effective if it is flexible enough to accommodate a variety of execution environments.

Irregular Problems. Until now this paper has only described statically defined ensembles that are array-based. However, this should not imply that Phase Abstractions are ill suited to dynamic or unstructured problems. In fact, to some extent LPAR [28], a set of language extensions for irregular scientific computations (see Section 7), can be described in terms of the Phase Abstractions. The key point is that an ensemble is a set with a partitioning; to support dynamic or irregular computations we can envision dynamic or irregular partitionings that are managed at runtime.

Consider first a statically defined irregular problem such as finite element analysis. The programmer begins by defining a logical data ensemble that will be replaced by a physical ensemble at runtime. This logical definition includes the proper record formats and an array of portnames, but not the actual data decomposition or the actual port ensemble. At runtime a phase is run to determine the partitioning and create the data and port ensembles: The size and contents of the data ensemble are defined, the interconnection structure is determined, and the sections are mapped to physical processors. We assume that the code ensemble is SPMD since this obviates the need to assign different codes to different processes dynamically. Once this partitioning phase has completed the ensembles behave the same as statically defined phases.

Dynamic computations could be generalized from the above idea. For example, a load balancing phase could move data between sections and also create revised data and port ensembles to represent the new partitioning. Technical difficulties remain before such dynamic ensembles can be supported, but the concepts do not change.

Limits of the Non-Shared Memory Model. The non-shared memory model encourages good locality of reference by exposing data movement to the programmer, but the performance advantage for this model will be small for applications that inherently have poor locality. For example, direct methods of performing sparse Cholesky factorization have poor locality of reference because of the sparse and irregular nature of the input data. For certain solutions to this problem, a shared memory model performs better because the single address space leads to better load balance through the use of a work queue model [35]. The shared memory model also provides notational convenience, especially when pointer-based structures are involved.

6 Portability Results

Experimental evidence suggests that the Phase Abstractions can provide portability across a diverse set of MIMD computers [31, 32]. This section summarizes these results for just one program, SIMPLE, but similar results were also achieved for QR factorization and matrix multiplication [30]. Here we briefly describe SIMPLE, the machines on which this program was run, the manner in which this portable program was implemented, and the significant results.

SIMPLE is a large computational fluid dynamics benchmark whose importance to high performance computing comes from the substantial body of literature already devoted to its study. It was introduced in 1977 as a sequential benchmark to evaluate new computers and Fortran

Machine	Sequent	Intel	Intel	nCUBE	BBN	Transputer
model	Symmetry A	iPSC/2 S	iPSC/2 F	nCUBE/7	Butterfly GP1000	simulator
nodes	20	32	32	64	24	64
processors	Intel 80386	Intel 80386	Intel 80386	custom	Motorola 68020	T800
memory	32MB	4 MB/node	8 MB/node	512 KB/node	4 MB/node	N/A
cache	64KB	64 KB	64KB		none	none
network	bus	hypercube	hypercube	hypercube	omega	mesh

Table 1: Machine Characteristics

compilers [7]. Since its creation it has been studied widely in both sequential and parallel forms [3, 9, 13, 16, 17, 23, 24, 37, 39].

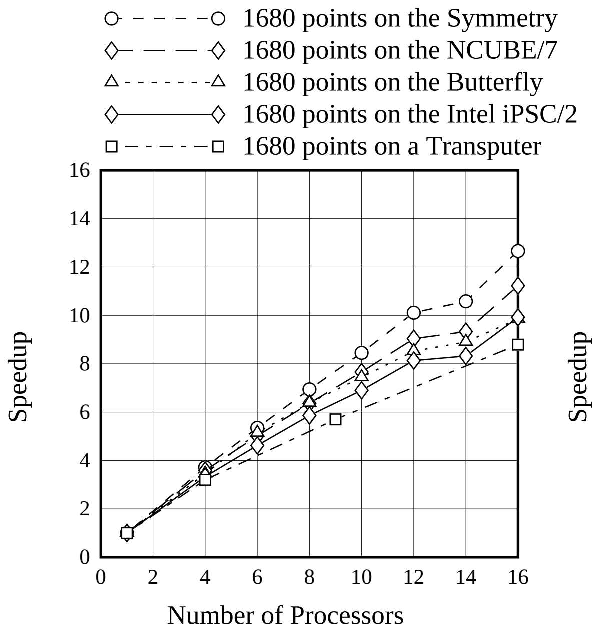
Hardware. The portability of our parallel SIMPLE was investigated on the iPSC/2 S, iPSC/2 F, nCUBE/7, Sequent Symmetry, BBN Butterfly GP1000, and a detailed Transputer simulator. These machines are summarized in Table 1. The two Intel machines differ in that one the iPSC/2 S has a slower Intel 80387 floating point coprocessor, while the other has the faster iPSC SX floating point accelerator. The simulator is a detailed Transputer-based non-shared memory machine. Using detailed information about arithmetic, logical and communication operators of the T800 [24], this simulator executes a program written in a Phase Abstraction language and produces time estimates for the program execution.

Implementation The SIMPLE program was written in Orca C. Since no compiler exists for any language based on the Phase Abstractions, the SIMPLE program was hand-compiled in a straightforward fashion to C code that uses a runtime substrate for supporting the Phase Abstractions. The resulting C code is machine-independent except for process creation, which is dependent on each operating system's method of spawning processes.

Figure 12(a) shows that similar speedups were achieved on all machines. Of course, many hardware characteristics can affect speedup, and these can explain the differences among the curves. In this discussion we concentrate on communication costs relative to computational speed, the feature 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 [11, 12]. This has the effect of reducing speedup because it decreases the percentage of time spent computing and increases the fraction of time spent on 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.

Figure 12(b) shows the SIMPLE results of Hiromoto *et al.* on a Denelcor HEP using 4096 data points [23], which indicate that our portable program is roughly 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, Figure 12(b) compares our results on the iPSC/2 S against those of Pingali and Rogers' parallelizing compiler for Id Nouveau, a functional language [39]. Both



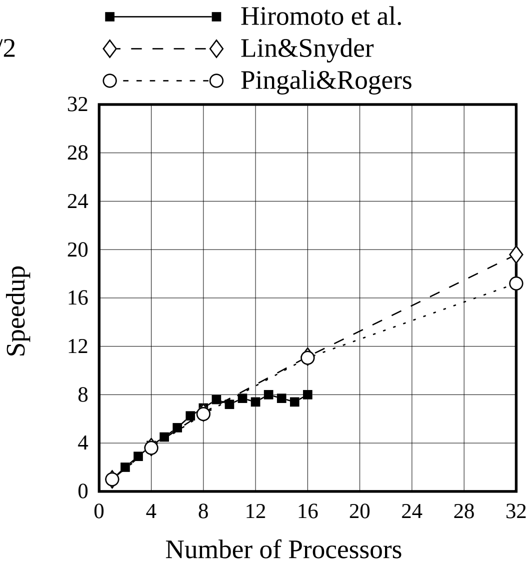


Figure 12: (a) SIMPLE Speedup on Various Machines (b) SIMPLE with 4096 points

experiments were run on iPSC/2's with 4MB of memory and 80387 floating point units. All other parameters appear to be identical. The largest potential difference lies in the performance of the sequential programs on which speedups are computed. Although these results are encouraging for proponents of functional languages, we point out that our results do not make use of a sophisticated compiler: The type of compiler technology developed by Pingali and Rogers can likely improve the performance of our programs as well. Moreover, our program requires only currently-available C compilers to achieve portability.

Even though the machines differ substantially—for example, in memory structure—the speedups fall roughly within the same range. Moreover, this version of SIMPLE compares favorably with machine-specific implementations. These results suggest, then, that portability has been achieved for this application and these machines.

7 Related Work

Many systems support some type of global view of parallel computation, SPMD execution, and data decomposition that are similar to various aspects of the Phase Abstractions. None, however, provide support for an X-level algorithm that is different from the Z-level parallel algorithm. Nor do any provide general support for handling boundary conditions or controlling granularity. This section discusses how some of these systems address scalability and portability in the aggregate data parallel programming style.

Dataparallel C. Dataparallel C [21] (DPC) is a portable shared-memory SIMD-style language that has similarities to C++. Unlike the Phase Abstractions, DPC supports only point-wise parallelism. DPC has point-wise processor (*poly*) variables that are distributed across the processors of the machine. Unlike its predecessor C* [40], DPC supports data decompositions of its data to improve performance on coarse-grained architectures. However, because DPC only supports point-wise communication, the compiler or runtime system must detect when several point sends on a processor are destined for the same processor and bundle them. Also, to maintain performance of the SIMD model on a MIMD machine, extra compiler analysis is required to detect when the per-instruction SIMD synchronizations are not necessary and remove them. Because each point-wise process is identical, meaning that edge effects must be coded as conditionals that determine which processes are on the edge of the computation. Reusing such code is harder because the boundary conditions may change from problem to problem. Constant and variable boundary conditions, however, can be supported by expanding the data space and leaving some processes idle.

Dino. Dino [41] is a C-like, SPMD language. Like C*, it constructs distributed data structures by replicating structures over processors and executing a single procedure over every element of the data set. Dino provides a shared address space, but remote communication is specified by annotating accesses to non-local objects by the special **#** symbol, and the default semantics is true message-passing. Parallel invocations of a procedure synchronize on exit of the procedure. Dino allows the mapping of data to processes to be specified by programmer-defined functions. To ensure fast reads to shared data, a partitioning can map an individual variable to multiple processors. Writes to such a variable are broadcast to all copies. Dino handles edge effects in the same fashion as C*. Because Dino only supports point-wise communication, as in C*, the compiler or runtime system must combine messages.

Mehrotra and Rosendale. A system described by Mehrotra and Rosendale [36] is much like Dino in that it supports a small set of data distributions. However, this system provides no way to control or determine precisely which points are local to each other, so it is not possible to control communication costs or algorithm choice based on locality. On the other hand, this system does not require explicit marking of external memory references as in Dino. Instead, their system infers, when possible, which references are global and which are not. In algorithms where processes dynamically choose their "neighbors," this simplifies programming. Also, programs are more portable than those written in Dino. The communication structure of the processor is not visible to the programmer, but the programmer can change the partitioning clauses on the data aggregates. SPMD processing is allowed, but there are no special facilities for handling edge effects.

Parallel Fortrans. Recent languages such as Kali [26], Vienna Fortran [6], and HPF [22] focus on data decomposition as the expression of parallelism. Their data decompositions are similar to the Phase Abstractions notion of data ensembles, but the overall approach differs fundamentally from Phase Abstractions. Phase Abstractions require more effort from the programmer, while this other approach relies on compiler technology to exploit loop level parallelism. This compiler-based approach has clean semantics because it can guarantee deterministic sequential semantics, but it has less potential for parallelism since there may be cases where compilers cannot transform a sequential algorithm into an optimal parallel one.

Kali, Vienna Fortran and HPF depart from sequential languages primarily in their support for data decomposition, although some of these languages do provide mechanisms for specifying parallel loops. Vienna Fortran provides no form of parallel loops. HPF has the `FORALL` statement that can be used to specify loops with no loop carried dependencies. To ensure deterministic semantics of updates to common variables by different loop iterations, values are deterministically merged at the end of the loop. This construct is optional in that the compiler will attempt to extract parallelism even where `FORALL` is not used. In contrast to HPF's optional `FORALL` loops, Kali requires `FORALL` loops with the same restriction that each loop can execute independently.

HPF and Vienna Fortran allow arrays to be aligned with respect to some abstract partitioning. These are very powerful constructs. For example, arrays can be dynamically remapped, and procedures can define their own data distribution. Together these features are potentially very expensive because although the programmer helps in specifying the data distribution at various points of the program, the compiler must determine how to move the data. In addition to data distribution directives, Kali allows the programmer to control the assignment of loop iterations to processors through the use of the `On clause`, which can help in maintaining locality.

LPAR. LPAR is a portable language extension that supports structured, irregular scientific parallel computations [28, 27]. In particular, LPAR provides mechanisms for describing non-rectangular distributed partitions of the data space to manage load-balancing and locality. These partitions are created through the union, intersection and set difference of arrays. Because support for irregular decompositions has a high performance cost, LPAR syntactically distinguishes irregular decompositions so that faster runtime support can be used for regular decompositions.[3] Computations are invoked on a group of arrays by the `foreach` operator, which executes its body in parallel on each array, thus yielding coarse-grained parallelism. LPAR uses the overlapping indices of distributed subarrays to support sharing of data elements. Overlapping domains also provide an elegant way of describing multilevel mesh algorithms and computations for boundary conditions. There is an operator for redistributing data elements, but LPAR depends on a routine written in the base language to compute what the new decomposition should be.

The Phase Abstraction's potential to support dynamic, irregular decompositions is discussed in Section 5. For multigrid decompositions, a sublanguage supporting scaled partitionings and communication between scaled ensembles would be useful. The Phase Abstractions' support for loose synchrony naturally supports the use of refined grids in conjunction with the base grid.

Split-C. Split-C is a shared-memory SPMD language with memory reference operations that support latency-hiding [10]. Split-C procedures are concurrently applied in an "owner-computes" fashion to the partitions of an aggregate data structure such as an array or pointer-based graph. A process reads data that it does not own with a global pointer (a Split-C data type). To hide latency, Split-C supports an asynchronous read—akin to an unsafe Multilisp future [20]—that initiates a read of a global pointer but does not wait for the data to arrive. The read is guaranteed to be complete only after a `sync()` operation has been called by the process, which blocks until all of the process's outstanding reads complete. There is a similar operation for global writes. These operations hide latency while providing a global namespace and reducing the copying of data in and out of message queues. (Copying may be necessary for bulk communication of non-contiguous

[3]Scott Baden, Personal Communication.

data, such as the column of an array.) However, these operations can lead to complex programming errors because a misplaced reference or synchronization operation can lead to incorrect output but no immediate failure.

Array distribution in Split-C is straightforward but somewhat limited; some number of higher order dimensions can be cyclically distributed while the remaining dimensions are distributed as blocks. Load balance, locality, and irregular decompositions may be difficult to achieve for some applications. Array distribution declarations are tied to a procedure's array parameter declarations, which can limit reusability and portability because a procedure's array declarations and the code that depends on those declarations may need to change to tune distributions for a particular architecture. This coupling can also incur a performance penalty because the benefit of an optimal array distribution for one procedure invocation may be offset by the cost of redistributing the array for other calculations that use the array. Split-C provides no special support for boundary conditions. The typical trick of creating an enlarged array is possible; otherwise, irregularities must be handled by conditional code in the body of the SPMD procedures.

8 Conclusion

The use of parallelism to satisfy the need for fast computing has been hampered by a lack of portability, scalability and ease of programming, unacceptably increasing the cost and time required to develop efficient programs. Support is required for quickly programming a solution and easily moving it to new machines as old ones become obsolete, or else the time saved by fast execution is squandered in programming time.

Rather than defining a new parallel programming paradigm, the Phase Abstractions language model supports well-known techniques for achieving high-performance—computing sequentially on local aggregates of data elements and communicating large groups of data elements as a unit—by allowing the programmer to partition the global data set across the parallel machine in a scalable manner. Additionally, by separating different aspects of a program into reusable parts—X level, Y level, Z-level, ensemble declarations, and boundary conditions—the creation of subsequent programs can be significantly simplified. This approach provides machine-independent, low-level control of parallelism and allows programmers to write in an SPMD manner without sacrificing the efficiency of MIMD processing.

Message passing languages have often been praised for their efficiency, but they have been condemned for being difficult to use. The contribution of the Phase Abstractions is a language model that focuses on efficiency while reducing the difficulty of non-shared memory programming. The programmability of the Phase Abstractions model is exemplified by the straight-forward solution of problems such as SIMPLE, as well as the ability to define specialized higher-level sublanguages such as an array language. Because the Phase Abstractions model is designed to be structurally similar to an MIMD architecture, it performs very well on a variety of MIMD processors. This claim is supported by tests on machines such as the Intel iPSC, the Sequent Symmetry and the BBN Butterfly.

Acknowledgments. We would like to thank Francine Berman for her insightful comments on a draft of this paper. We would also like to thank Scott Baden for his assistance in developing the discussion on LPAR.

References

[1] G. Alverson, W. Griswold, D. Notkin, and L. Snyder. A flexible communication abstraction for nonshared memory parallel computing. In *Proceedings of Supercomputing '90*, November 1990.

[2] G. Alverson and D. Notkin. Program structuring for effective parallel portability. *IEEE Transactions on Parallel and Distributed Systems*, 4(9), September 1993.

[3] T. S. Axelrod, P. F. Dubois, and P. G. Eltgroth. A simulator for MIMD performance prediction – application to the S-1 MkIIa multiprocessor. In *Proceedings of the International Conference on Parallel Processing*, pages 350–358, 1983.

[4] G. E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-92-103, School of Computer Science, Carnegie Mellon University, January 1992.

[5] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[6] B. Chapman, P. Mehrotra, and H. Zima. Vienna Fortran – a Fortran language extension for distributed memory multiprocessors. Technical Report No. 91-72, ICASE, September 1990.

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

[8] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eiken. Logp: Towards a realistic model of parallel computation. In *Proceedings of the Fourth Symposium on Principle and Practice of Parallel Programming*, pages 1–12, May 1993.

[9] D. E. Culler and Arvind. Resource requirements of dataflow programs. In *Proceedings of the International Symposium on Computer Architecture*, pages 141–150, 1988.

[10] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[11] T. Dunigan. Hypercube performance. In *Proceedings of the 2nd Conference on Hypercube Architectures*, pages 178–192, 1987.

[12] T. Dunigan. Performance of the Intel iPSC/860 and NCUBE 6400 hypercubes. Technical Report ONRL/TM-11790, Oak Ridge National Laboratory, 1991.

[13] K. Ekanadham and Arvind. SIMPLE: Part I, an exercise in future scientific programming. Technical Report CSG Technical Report 273, MIT, 1987.

[14] W. Fenton, B. Ramkumar, V. Saletore, A. Sinha, and L. Kale. Supporting machine independent programming on diverse parallel architectures. In *Proceedings of the International Conference on Parallel Processing*, pages II 193–201, 1991.

[15] J. Feo, D. C. Cann, and R. Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, 10:349–366, December 1990.

[16] D. Gannon and J. Panetta. SIMPLE on the CHiP. Technical Report 469, Computer Science Department, Purdue University, 1984.

[17] D. Gannon and J. Panetta. Restructuring Simple for the CHiP architecture. In *Parallel Computing*, pages 3:305–326, 1986.

[18] K. Gates. Simple: An exercise in programming in Poker. Technical report, Applied Mathematics Department, University of Washington, 1989.

[19] W. Griswold, G. Harrison, D. Notkin, and L. Snyder. Scalable abstractions for parallel programming. In *Proceedings of the Fifth Distributed Memory Computing Conference*, 1990. Charleston, South Carolina.

[20] R. H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[21] P. J. Hatcher, M. J. Quinn, R. J. Anderson, A. J. Lapadula, B. K. Seevers, and A. F. Bennett. Architecture-independent scientific programming in Dataparallel C: Three case studies. In *Proceedings of Supercomputing '91*, pages 208–217, 1991.

[22] High Performance Fortran Forum. *High Performance Fortran Specification.* January 1993.

[23] R. E. Hiromoto, O. M. Lubeck, and J. Moore. Experiences with the Denelcor HEP. In *Parallel Computing*, pages 1:197–206, 1984.

[24] T. J. Holman. *Processor Element Architecture for Non-Shared Memory Parallel Computers.* PhD thesis, University of Washington, Department of Computer Science, 1988.

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

[26] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[27] S. R. Kohn and S. B. Baden. Lattice parallelism: A parallel programming model for non-uniform, structured scientific computations. Technical Report CS92-261, University of California, San Diego, Dept. of Computer Science and Engineering, September 1992.

[28] S. R. Kohn and S. B. Baden. An implementation of the LPAR parallel programming model for scientific computations. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, March 1993.

[29] M. S. Lam and M. C. Rinard. Coarse-grain parallel programming in Jade. In *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, April 1991.

[30] C. Lin. *The Portability of Parallel Programs Across MIMD Computers.* PhD thesis, University of Washington, Department of Computer Science and Engineering, 1992.

[31] C. Lin and L. Snyder. A portable implementation of SIMPLE. *International Journal of Parallel Programming*, 20(5):363–401, 1991.

[32] C. Lin and L. Snyder. Portable parallel programming: Cross machine comparisons for SIMPLE. In *Fifth SIAM Conference on Parallel Processing*, 1991.

[33] C. Lin and L. Snyder. Data ensembles in Orca C. In *5th Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[34] C. Lin and L. Snyder. Accommodating polymorphic data decompositions in explicitly parallel programs. In *Proceedings of the 8th International Parallel Processing Symposium*, April 1994.

[35] C. Lin and W. D. Weathersby. Towards a machine-independent solution of sparse cholesky factorization. In *Proceedings of Parallel Computing '93*, (to appear) 1993.

[36] P. Mehrotra and J. Rosendale. Compiling high level constructs to distributed memory architectures. Technical Report ICASE Report No. 89-20, Institute for Computer Applications in Science and Engineering, March 1989.

[37] J. M. Meyers. Analysis of the SIMPLE code for dataflow computation. Technical Report MIT/LCS/TR–216, MIT, 1979.

[38] 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. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, July 1988.

[39] K. Pingali and A. Rogers. Compiler parallelization of SIMPLE for a distributed memory machine. Technical Report 90–1084, Cornell University, 1990.

[40] J. Rose and G. L. Steele Jr. C*: An extended C language for data parallel programming. In *2nd International Conference on Supercomputing*, March 1987.

[41] M. Rosing, R. Schnabel, and R. Weaver. The Dino parallel programming language. Technical Report CU-CS-457-90, Dept. of Computer Science, University of Colorado, April 1990.

[42] L. Snyder. Type architecture, shared memory and the corollary of modest potential. In *Annual Review of Computer Science*, pages I:289–318, 1986.

[43] L. Snyder. The XYZ abstraction levels of Poker-like languages. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 470–489. MIT Press, 1990.

[44] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[45] J. Wiggs. A parallel implementation of the Car-Parrinello method. Technical Report General Exam, Dept. of Chemistry, University of Washington, June 1993.