CHAPTER 81

Portable Parallel Programming: Cross Machine Comparisons for SIMPLE*

Calvin Lin†
Lawrence Snyder†

Abstract. Portability is essential if parallel programs are to amortize their costs over a long lifetime. But because parallel machines are so diverse, it is difficult to create parallel programs which are both portable *and* efficient. Recent advances in parallel programming abstractions promise a solution.

We present the first supporting evidence using Livermore's SIMPLE computation. Using the *Phase Abstractions*, a single program was written. This was then hand translated (compilers remain unavailable) for the Sequent Symmetry, the BBN Butterfly, the Intel iPSC/2, the NCUBE/7, and a transputer based nonshared memory machine.

Our results show similar speedups for the various machines, indicating that the abstractions properly structure the program for effective compilation.

1. Introduction. Because parallel machines are so diverse, it is difficult to create parallel programs which are both portable *and* efficient: A program written to exploit the capabilities of one machine may perform poorly on another machine. Portability is important because while hardware costs continue to decrease, the expense of writing parallel programs remains high. If such software can be made to execute across multiple machines with minimal effort, the high cost of software can be amortized over a long lifetime. Portability can further reduce software costs by encouraging sharing.

The Phase Abstractions [6, 7, 16] present a nonshared memory model of programming which aims to support portable, scalable code for all classes of MIMD multiprocessors. In this paper we compare the performance of a single program – written using Phase Abstractions – on several different machines. The program is a version of the SIMPLE fluid dynamics benchmark produced by Lawrence Livermore Labs [3], and the machines include the Intel iPSC/2, the NCUBE/7, the BBN Butterfly, and the Sequent Symmetry. Our results show that portability has been achieved for this program.

The next two sections give background concerning the SIMPLE program and the Phase Abstractions. Section 4 discusses issues of methodology and is followed by a description of our experimental setup. After presenting results in section 6, we evaluate the role of the Phase Abstractions in achieving portability, then give some concluding remarks.

2. The SIMPLE computation. The SIMPLE program represents a typical computational fluid dynamics application. We use the Gannon and Panetta version of SIMPLE [4] as revised by Gates [5] and Lee [10]. Logically, the state of the computation is represented

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

†Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, WA 98195.

by arrays containing real values for physical quantities such as pressure and density. The computation is divided into 5 phases – Delta, Hydro, Heat, Energy1, and Energy2 – which are applied iteratively. The overall logic is illustrated in Figure 1.

Each phase is logically applied to the entire state of the computation and is composed of a set of processes operating on separate rectangular subregions of the data arrays, ie. a process "owns" the corresponding elements of each array. A process performs the local operations for one application of a phase, and processes of the same phase communicate with each other by a characteristic pattern induced by the data dependencies of the computation. Figure 2 shows the patterns for the phases, where the squares represent processes and the edges indicate interprocess communication.

```
data := Load();
while (error > $\delta$)
{
    Delta(data);
    Hydro(data);
    Heat(data);
    Energy1(data);
    error := Energy2(data);
}
```

Figure 1: Z Level for SIMPLE

3. Background. *Phase Abstractions* were introduced [6, 7] to provide a means of structuring a parallel program to simplify scalability and portability. The reasoning is as follows: If a program has the "right form" the compiler can easily control those features that affect performance on different machines. There are two sets of abstractions collectively referred to as Phase Abstractions [16]: the *XYZ Programming Levels* and *Ensembles.* Because the above description of SIMPLE implicitly uses Phase Abstractions, it can serve as an example.

Phase Abstractions recognize that the instructions of a parallel program serve in one of three roles: There are processes (X level) that are the building blocks of the parallel program, there are phases (Y level) that are formed by the composition of processes into a concurrent computation, and there is the problem level (Z level) describing the overall logic of phase invocation. The Z level control logic for SIMPLE is shown in Figure 1. The phases themselves are composed of processes having the communication structures shown in Figure 2.

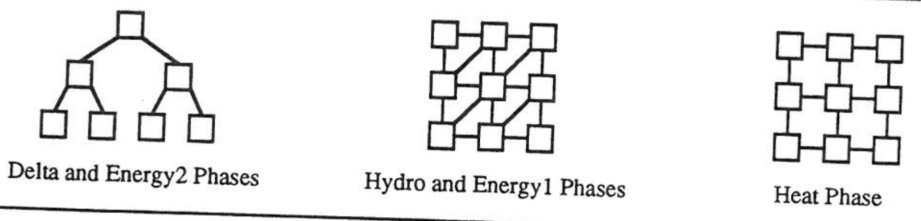

Figure 2: Communication Graphs for SIMPLE

Ensembles are used to define phases. An ensemble is a set with a partitioning; the partitions are called *sections.* Three types of ensembles are needed to define a phase: data

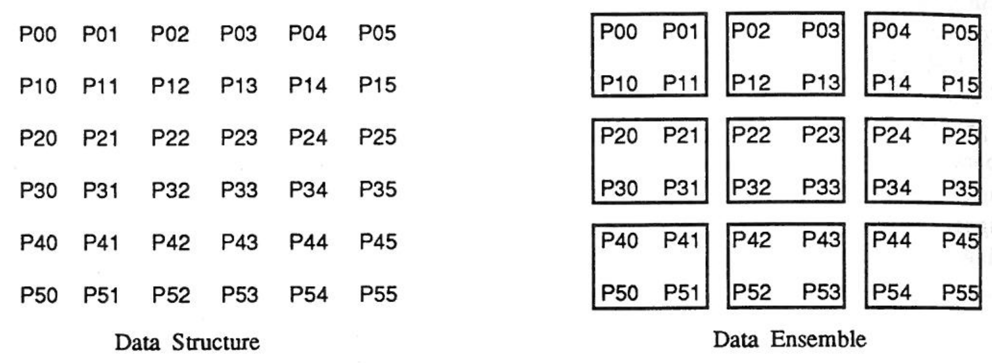

Figure 3: Data Ensemble for Pressure Array on 9 Processors

ensemble, code ensemble and port ensemble. A *data ensemble* is a data structure with a partitioning. For example, Figure 3 illustrates the data ensemble for the 2D **Pressure** array. Notice that the rectangular blocking of the data mentioned in the SIMPLE description is reflected in the partitioning. A *code ensemble* is a partitioning of a set of process instances, where each process is defined by an X level program written in some sequential language such as C or FORTRAN. A *port ensemble* is a set of port name pairs with a partitioning. In the Phase Abstraction model, messages are passed through named ports, so a port ensemble defines a phase's communication structure. A phase is constructed by composing data, code, and port ensembles with the same partitioning (See Figure 4). The semantics are that each section represents logical concurrency: A process assigned to a given section operates on the data in the corresponding section of the data ensembles, and communicates through the ports of the corresponding section of the port ensemble. Each section can be assigned to a processor which executes the process, stores the data of the section locally, and refers to its neighboring processors via the port names.

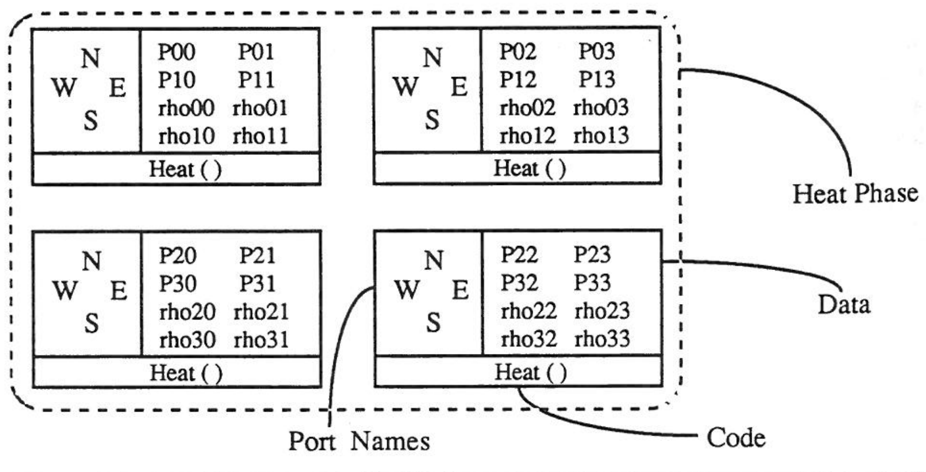
N W E S
P00 P01 P10 P11 rho00 rho01 rho10 rho11
Heat ( )
N W E S
P02 P03 P12 P13 rho02 rho03 rho12 rho13
Heat ( )
N W E S
P20 P21 P30 P31 rho20 rho21 rho30 rho31
Heat ( )
N W E S
P22 P23 P32 P33 rho22 rho23 rho32 rho33
Heat ( )
Heat Phase
Data
Port Names
Code

Figure 4: Data, Code, and Port Ensemble combine to form a Phase.

After presenting the experimental results, the role of the Phase Abstractions in portability will be discussed.

4. Methodology. To see whether portability is achieved, we observe the performance of a single program across a number of different machines. Where possible, we compare the results of the portable program with those published by others. We use speedup curves to compare performance because they help remove some machine differences such as clock speed and compiler quality.

Since no Phase Abstraction compilers yet exist,[1] we hand translated our single program to run on the four multiprocessors and one simulator. Only rudimentary source level changes were necessary, most involving the low level message passing interface. Our program used C to implement each of the X, Y and Z levels. All of the nonshared memory machines provide message passing primitives, while for the shared memory machines, Poker-style message passing [15] was simulated through shared memory.

5. Hardware Setting. The first of our machines 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 [12].

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].

We also used two 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 interprocessor communication is through message passing [9]. 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 [13].

Finally, we have a detailed Transputer-based nonshared memory simulator. Using detailed information about arithmetic, logical and communication operators, this simulator executes a Poker C program and produces time estimates for the program execution.

6. Results. Our results (see Figure 5) show similar shaped speedup curves[2] for all machines and support the claim that our program is portable. Figure 5 also shows the Hiromoto *et al.* results for the Denelcor HEP [8]. We include these results to show that our program is competitive with machine-specific code, but because of the many differences between the two experiments (different architectures, different problem sizes, and perhaps even different problem specifications) we caution the reader against drawing stronger conclusions.

Figure 6 compares our results with those of Pingali and Rogers [14]. Although both experiments were run on iPSC/2's, there may still be machine differences – such as different floating point coprocessors and different memory sizes – which can influence speedup. On the other hand, our problem sizes are identical, as are most, if not all, of the machine characteristics. We note that Pingali and Rogers wrote their program based on an Id program.

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

[2] Speedup was computed based on a sequential version of SIMPLE. The NCUBE nodes have so little memory that in order to keep the problem size constant for all machines we approximated the sequential execution time on the NCUBE by solving a smaller problem size and then extrapolating for the desired problem size. This gives a conservative estimate of speedup since it underestimates the sequential execution time.

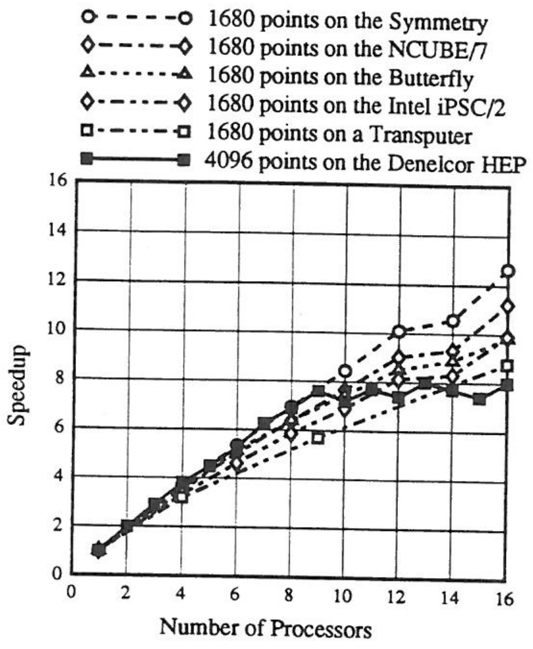

Figure 5: SIMPLE on Various Machines

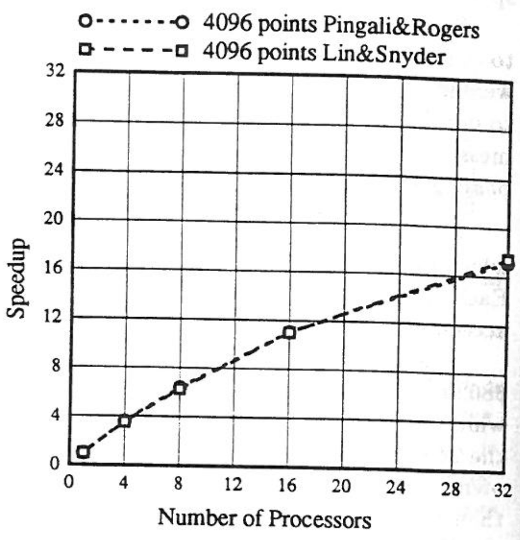

Figure 6: SIMPLE on the iPSC/2

7. The Role of the Phase Abstractions. With the experience of only one application running on five machines, it is premature to generalize the claim that Phase Abstractions support portability. Nevertheless, some advantages of the Phase Abstractions are clear.

It is obvious that the partitioning of the data ensembles provides critical information needed for allocating SIMPLE's arrays to the memory of nonshared memory machines such as the iPSC/2 and NCUBE/7. Since the data of a section will be local to the process instance of that section, it can be stored in the memory of one processor, minimizing the need for expensive message passing.

What is perhaps less obvious is the value of this partitioning information in the shared memory case. For example, the BBN Butterfly transparently provides both local and remote memory access to form a single global address space. Though the data ensembles could be referenced through remote accesses, the section structure – a process operating on local data – provides an alternative which exploits locality. Since local access is much faster than remote access, there is a performance advantage in using local data references. The result is that the program based on Phase Abstractions runs *well* on the Butterfly. Though we have no measurements to prove it, this same benefit of locality is almost certainly improving the use of the cache for the Sequent Symmetry [11].

Another property of the Phase Abstractions is the explicit availability of communication patterns. This information can aid process allocation for distributed memory machines. For machines like the NCUBE/7 where communication time depends on the number of "hops" a message must travel, there are performance advantages to minimizing the number of hops per message by finding good embeddings of the port ensemble graph in the hypercube. (The benefits of knowing the communication graph are more extensive than illustrated in these few experiments [2] and will likely become more significant as the "mesh" machines, with their more limited communication capabilities, become available.)

8. Conclusions. Using recently developed program abstractions we have demonstrated the portability of a single program across a variety of different machines. This is the first empirical study of portability across all classes of MIMD multiprocessors. Though the results are preliminary, we believe the data supports the following assertion: New application programs should be developed using these abstractions if portability is desired.

Acknowledgments. It is a pleasure to thank Hans Mandt, Stu Stern, and the Advanced Systems Laboratory of Boeing Computing Services for their help in providing access to a Butterfly multiprocessor; Walter Rudd, Michael Young, Michael Quinn and others at the Oregon Advanced Computing Institute for providing access to their iPSC/2 and NCUBE/7; and Tom Holman for producing the detailed Transputer simulator. We wish to thank Kevin Gates for creating the first Poker C version of SIMPLE, and most importantly, we thank Jinling Lee for rewriting SIMPLE according to the Phase Abstraction model.

REFERENCES

[1] G. 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 Non-shared Memory Parallel Computing,* Proceedings of Supercomputing '90, New York, New York, November 1990.

[3] W. Crowley, C. P. Hendrickson and T. I. Rudy, *The Simple Code,* Technical Report UCD-17715, Lawrence Livermore Laboratory (1978).

[4] D. Gannon and J. Panetta, *Restructuring SIMPLE for the CHiP Architecture,* Parallel Computing, (1986) 3:305-326.

[5] K. Gates, *Simple, An Exercise in Programming Poker,* Technical Report 88-2, Dept. of Applied Mathematics, University of Washington (1989).

[6] W. Griswold, G. Harrison, D. Notkin, and L. Snyder, *How Port Ensembles Aid the Efficient Retargeting of Reduction Algorithms,* Proceedings of the International Conference on Parallel Processing, St. Charles, Illinois, (1990) pp. II 286-287.

[7] 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).

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

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

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

[11] C. Lin and L. Snyder, *A Comparison of Programming Models for Shared Memory Multiprocessors,* Proceedings of the International Conference on Parallel Processing, St. Charles, Illinois, (1990) pp. II 163-170.

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

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

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

[15] L. Snyder, *Parallel Programming and the Poker Programming Environment,* Computer, (July 1984) pp. 27-36.

[16] L. 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.