Design Patterns for Parallel Programming

S. Siu, M. De Simone, D. Goswami, A. Singh
Dept. of Electrical & Computer Engineering
University of Waterloo,
Waterloo, Ontario, Canada N2L 3G1
email: asingh@etude.uwaterloo.ca
phone: (519)888-4567 X2805
fax: (519)746-3077

Abstract

This paper describes the concept of design patterns for the development of parallel applications. Such design patterns are implemented as reusable code skeletons for quick and reliable development of parallel applications. In the past, parallel programming systems have allowed fast prototyping of parallel applications based on commonly occurring communication and synchronization structures. The essential difference in this system is the use of a standard definition and interface for a design pattern. This has two important implications. First, design patterns can be defined and added to the system's library not only by the system developers but also by the users *(Extensibility)*. Second, customization of a parallel application is possible by mixing design patterns with low level code resulting in a flexible and efficient parallel programming tool *(Openness)*.

1 Introduction

Building software tools that ease the development of parallel applications is one of the primary concerns in the area of parallel computing. To reduce the complexity of software development, sequential as well as parallel programming models commonly employ abstractions such as macros, functions, abstract data types, and objects. In this paper, we introduce the concept of a design pattern for parallel programming that can potentially reduce the time and effort needed to develop a large variety of parallel applications. A parallel programming system, called DPnDP (Design Patterns and Distributed Processes), that employs such design patterns is described.

Design patterns describe a recurring problem and a reusable solution to that problem [5]. Design patterns in DPnDP provide a general mechanism to support high level synchronization and communication structures. The concept of a design pattern for parallel programming is based on the realization that a large number of parallel applications (especially medium- and coarse-grained applications) are built using commonly occurring parallel techniques such as a task-farm or a divide and conquer structure. Developing a program that employs such complex parallel structures would require a significant amount of time and effort if a low-level tool is used (for example, UNIX sockets [8] or a message-passing library such as PVM [6]). Further, because sequential computation code is not separated from parallel communication code, the parallelism would be explicit, thus increasing the complexity of the application code. Each time the programmer wants to experiment with a different parallel structure for the application, additional programming effort would be required. Moreover, such an effort would be replicated, knowingly or unknowingly, by other programmers while writing other applications.

DPnDP is a design pattern driven parallel programming system that separates the specification of the parallel structuring aspects —such as synchronization, communication and process-processor mapping— from the application code that is to be parallelized. A design pattern in DPnDP is a software abstraction that implements a certain commonly occurring parallel structure and behavior in the form of a reusable and application independent code skeleton. The system provides a collection of design patterns that are stored in a library. To use a design pattern, the user simply provides the necessary sequential code. The system generates extra code to instantiate the design pattern.

An important feature of design patterns in DPnDP is that they are context insensitive. This means that implementation or use of a design pattern makes no assumptions about implementation of any other design patterns. Each design pattern provides a standard interface that can be used with other design patterns to compose an application. By providing a standard interface, new design patterns can be added to the system incrementally, making the system extensible. Also, it allows a design pattern to be used in conjunction with low level communication and synchronization primitives (such as message passing) thus facilitating customization and tuning of an application.

2 The Model

Design patterns implement various types of common process structures and interactions found in parallel systems, but with the key components — the application-specific procedures — unspecified. A user provides the application-specific procedures and a high level description of the parallel application. Design patterns abstract commonly occurring structures and communication characteristics of parallel applications, allowing users to develop parallel applications in a rapid and easy manner. This approach potentially enhances the correctness of the parallel application by providing well tested communication code skeletons that otherwise would have to be written from scratch by the user. Examples of currently supported design patterns include task-farm, pipeline, fan-out structure, divide and conquer, and process replication.

In DPnDP, a parallel application is represented by a directed graph (Figure 1). Each node of the graph may be associated with a sequential computation code module written in a sequential language (C or C++ in our case). Each module also has one or more message handlers, each having its own message context, which receive messages from other nodes and invoke the sequential code inside the module to process these messages accordingly.

Alternatively, a node of the application graph may represent a design pattern. The interface of a design pattern is indistinguishable from that of a module i.e., in each case other nodes interact with it via message passing through a common interface. However, the internal structure of the design pattern may be a multi-node subgraph. Some of these nodes may, in

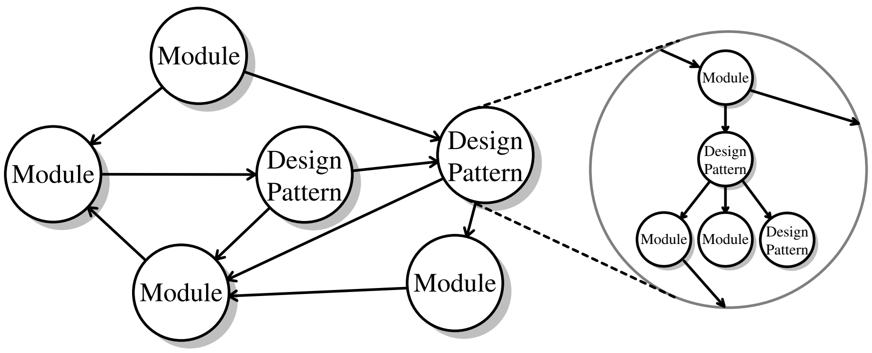


Figure 1: Overall Structure of a DPnDP Application

turn, be design patterns. Nodes interact with one another by sending and receiving messages.

When using a design pattern, a user only deals with communication that is application related. The generated code takes care of all other synchronization and communication needed for proper management of the processes in a design pattern. For example, in a task farm, the application code simply supplies work to the replicated processes and receives results. All the necessary process creations and interprocess communication for the management of replicated worker processes is handled by the code generated for the specified configuration of the task farm.

2.1 Sample Applications

2.1.1 Example 1: Parallel Quick Sort with Regular Sampling

Because of quick sort's divide and conquer nature, this algorithm fails to spread workload evenly to a large number of processors quickly. Performance of parallel quick sort, therefore, is usually limited by the time taken to perform this initial partitioning. Parallel Quick Sort with Regular Sampling (PQSRS) is a parallel algorithm developed by researchers in University of Alberta [9] to make quick sort efficient on distributed memory parallel computers by letting all processors sort its own data without waiting for the initial partitioning and then collaborate to generate the final solution. This parallel algorithm can be implemented by using a single master/slave parallel design pattern (Figure 2).

The following steps are required to generate this parallel application using DPnDP:

1. Select a master/slave pattern from the graphical user interface.

2. Fill in the structural parameter - Number of Slaves.

3. Select "Generate" to generate the code skeleton. The code skeleton is generated in a directory structure where each node has a separate directory containing all necessary files (source files and a makefile). A connection file specifying how nodes are connected is also included.

4. Edit the files to insert sequential entry procedures into the skeleton. In this case, one is needed for the master node (Figure 4) and another for the slave node (Figure 5).

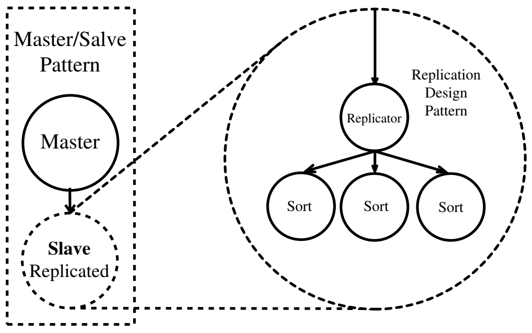
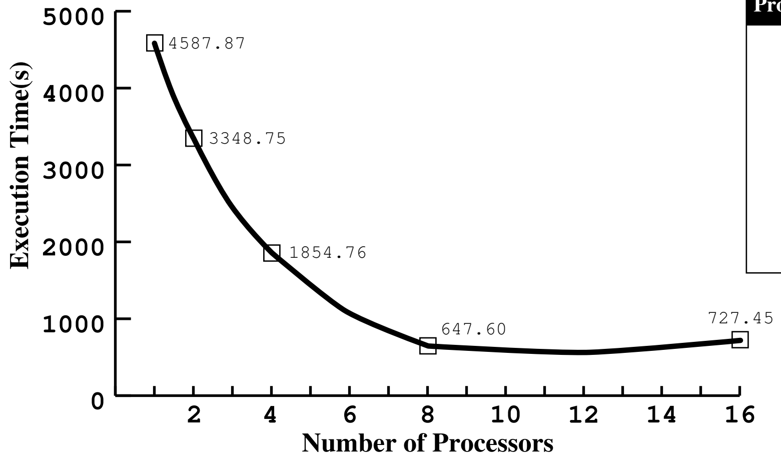


Figure 2: Structure of the PQSRS program

5. Issue the "make" command. All executables will be built and copied to the appropriate directory for execution.
6. Execute the parallel program.

Some experimental data was collected for this sample program (Figure 3). These experiments are run on a cluster of workstations connected by ethernet.



Number of Processor	Execution Time (s)	Speed-Up Ratio
1	4587.87	1
2	3348.75	1.4
4	1854.76	2.5
8	647.601	7.1
16	727.452	6.3

Figure 3: Experimental results of PQSRS

2.1.2 Example 2: Graphics Animation

Here is another more complex example that uses more than one design pattern. Consider a graphics animation program consisting of three modules Generate(), Geometry() and Display() where a sequence of graphical images, called frames, are to be generated. Depending on the subject of animation, Generate() computes the location and motion of each object for each frame. It then calls Geometry() to perform actions such as viewing transformations,

```
int NodeMain(NL_PortList& thePorts, int argc, char* argv[])
{
    // Cache output port handle from Port Array
    NL_Port& InPort = (*gPortList)["in0"];
    NL_Port& OutPort = (*gPortList)["out0"];
    ...

    // Step 1: Divide Array into NoOfProcess subarrays
    ...

    // Step 2: Distribute to slaves for sorting
    for (i = 0; i < NoOfProcess; i+=1) {
        OutPort.send(i);                       // Send Msg Tag
        OutPort.send(Size);                    // Send Size of array
        OutPort.send(Temp[i].ArrayPtr, Size);  // Send array of size Size
    } // for

    // Step 3: Collect Messages
    for (i = 0; i < NoOfProcess; i+=1) {
        int index, aSize;
        InPort.receive(index);                // Receive Msg Tag
        InPort.receive(aSize);                // Get Size of Msg
        InPort.receive(Temp[index].ArrayPtr, aSize); // Put Msg in array
    } // for

    A = Cat(Temp, NoOfProcess);

    // Step 4: Sample/Sort Samples, Find/Use Pivots to rearrange list
    // Step 5: Distribute to slaves for sorting (Same as Step 2)
    // Step 6: Collect Messages (Same as Step 3)
    ...

} // NodeMain
```

Figure 4: Entry procedure for the Master node

```
int ASP_M_NodeMain::MsgHandler(const NL_Port& thePort)
{
    // Cache output port handle from Port Array
    NL_Port& OutPort = (*gPortList)["output0"];
    int size, ID; int *buf;

    // Get the message from received port
    thePort.receive(ID);
    thePort.receive(size);
    buf = new int[size];
    thePort.receive(buf, size);

    QuickSort(buf, size);

    // Send the message back to where it comes form
    OutPort.send(ID);
    OutPort.send(size);
    OutPort.send(buf, size);

    delete [] buf;
    return GOOD;

} // MsgHandler
```

Figure 5: Entry procedure for the Slave node

projection and clipping. Finally, the frame is processed by Display() which performs hidden-surface removal and anti-aliasing. Then it stores the frame on the disk. After this, Generate() continues with the computation of the next frame and the whole process is repeated.

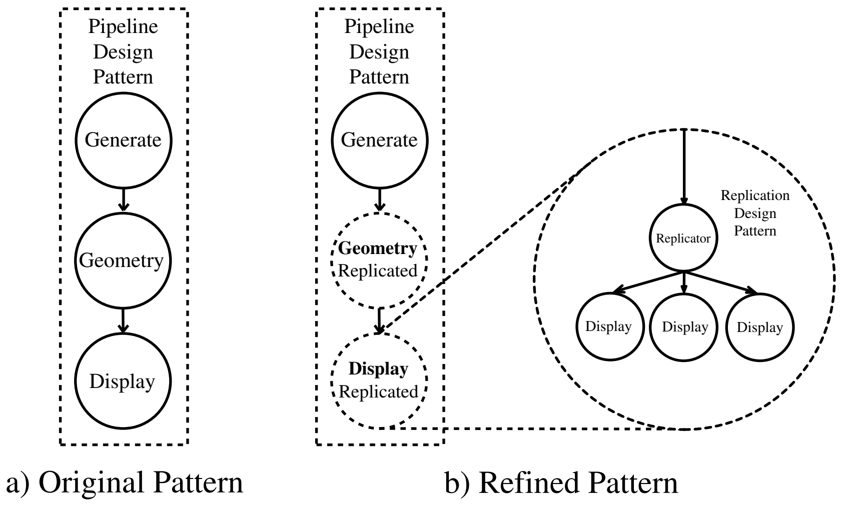


Figure 6: Structure of the Graphics Animation program

A simple way to parallelize this application would be to let the three modules work in a pipelined manner on different processors. After computing a frame, Generate() passes it to Geometry() for processing and starts working on the next frame. Similarly, Geometry() can pass its output to Display() and then receive its next frame from Generate(). Therefore, all three modules can work in parallel on different frames (Figure 6a). Now, if Display() takes much longer to do its processing as compared to Generate() and Geometry() (which is generally the case in reality; hidden-surface removal and anti-aliasing require much more time than the other components of the program), more than one instance of Display() can be initiated. This is possible because the processing of each frame is independent. Similarly, if we were to improve the performance of Geometry(), several instances of it may be initiated as well. This situation is shown in Figure 6b where Geometry() and Display() have several active instances.

Although this example uses only pipeline and replication design patterns, it illustrates several points:

1. Other than the minor changes required to convert the local procedure calls in Generate and Geometry, the sequential and the parallel version of the application code are the same.

2. The two parallel versions of the application differ only in terms of the design patterns they use. The application code is the same in each case.

3 Implementation

DPnDP has the following components: user interface, design pattern library, code skeleton generator and other supporting libraries (Figure 7). User specifies the directed graph through the user interface. The user interface uses functions in the design pattern library. The library passes the directed graph with all the structural and behavioral information of design patterns it contains to a code skeleton generator to generate a set of code skeletons that correspond to the directed graph specified by the user.

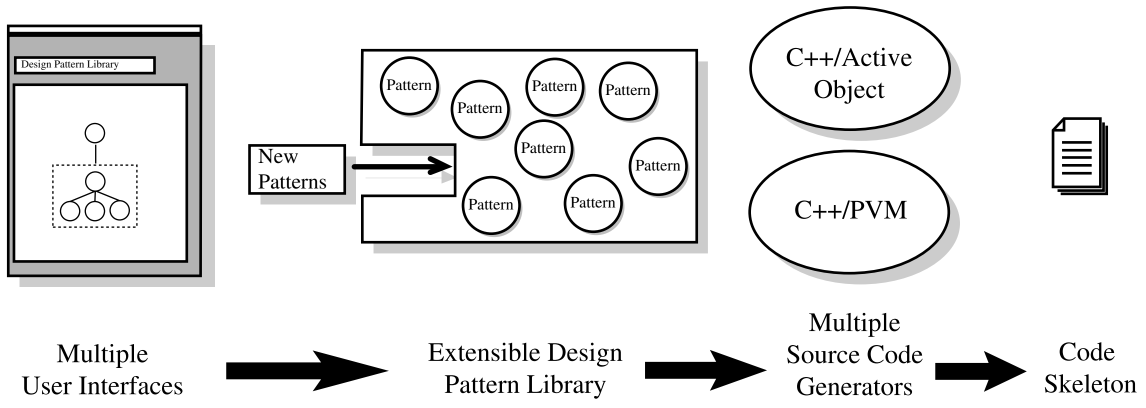


Figure 7: System Architecture of DPnDP

The design pattern library contains parallel design patterns written as C++ classes. These design patterns capture the structure, behavior and architectural information of a communication pattern. For example, an N-stage pipeline is a design pattern. A divide and conquer structure with variable width and depth is another example. A pattern can be instantiated into a communication skeleton when all necessary parameters are provided by the user. Parameters help capture generic behavior of design patterns (an N-stage pipeline) rather than specific instances of them (a 2-stage or a 3-stage pipeline).

All design patterns in DPnDP have a uniform way of definition and implementation. Each pattern inherits from a default design pattern class that provides standard behavior and common interface for all design patterns. The default pattern class contains some pure virtual functions — functions with interface but not implementation — that individual pattern can override to define its unique structure and behavior. This approach hides the code generation mechanism from pattern designers who need not know how code skeletons are generated. It also enforces a common interface so that all design patterns can be accessed the same way and thus can be used interchangeably. Each pattern is context insensitive meaning that it does not need to know the implementation of other patterns in order to work with them. Through this common interface, users can also develop new design patterns and add them to the library.

Design patterns not only can be used individually but also can be composed and refined to form more complex parallel structures. Composition lets users connect multiple design patterns in a directed graph so that more complex parallel programs can be expressed as a collection of interacting code modules and design patterns (Figure 1). Refinement lets users redefine the components of an existing pattern while retaining the structural and communication behavior. This allows users to reuse as much of the power of an existing pattern as possible to solve problems that differ slightly from that design pattern (Figure 6). Composition and refinement are not mutually exclusive.

Code Skeleton Generators translate the intermediate representation produced by the design pattern library into a set of source code skeletons completed with makefiles that are nicely packaged into directories. Separating code generation from the design pattern library allows DPnDP generate code skeleton for multiple programming languages, message passing libraries and operating systems without modifying the design patterns in the library. Code Skeleton Generators are written in perl [15], a powerful and cross platform text and file processing language.

Multiple user interfaces can be built on top of the design pattern library. Currently, a Motif interface is used to let users specify complex process graphs using functions provided in the library (Figure 8). This interface is designed for those who want to use all the features of the design pattern library. It visualizes nodes, design patterns and their interconnections graphically on a drawing canvas as process graphs. The interface is especially useful when the user needs to use the composition and refinement features of DPnDP because a graphical representation of a process graph is easier to visualize and modify. In the future, this interface will integrate more supporting tools for programming and executing parallel programs. A World Wide Web interface is under construction to let a user specify design patterns through the HTML forms and then retrieve the packaged and compressed code skeleton across the Internet. This interface allows casual users to experiment with DPnDP without installing and setting up the whole system.

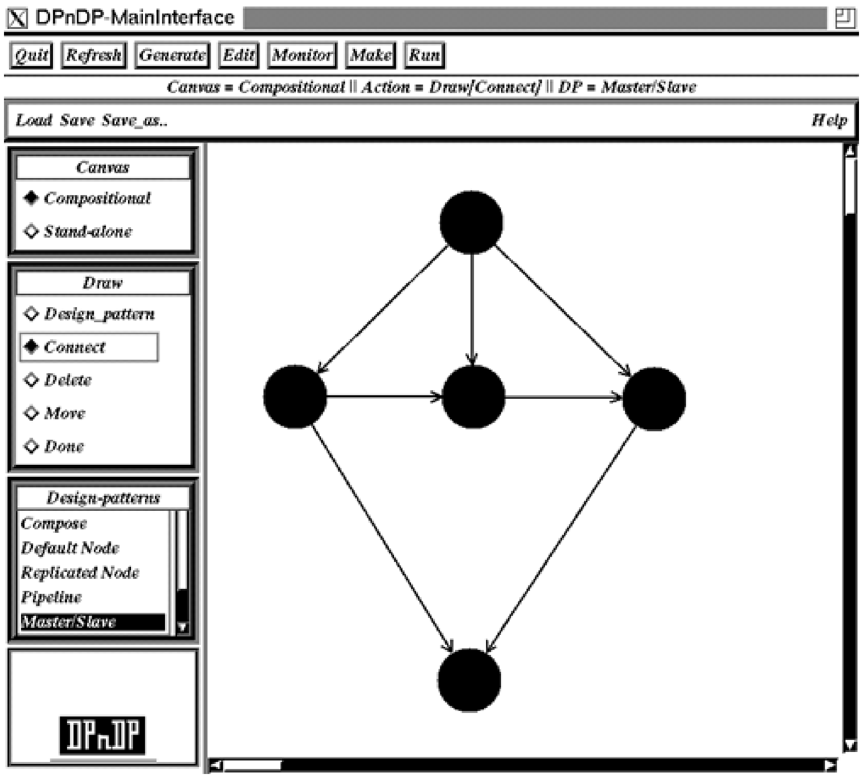

Canvas = Compositional || Action = Draw/Connect || DP = Master/Slave

Figure 8: Motif Interface for DPnDP

Another component of DPnDP is a thin message passing library called "*Node Layer*" [4] where particularities of different message passing libraries are abstracted by a common programming interface. It has been designed so that it can be ported atop most of the popular message passing libraries such as PVM, MPI, etc. It encapsulates complex message passing code into C++ objects and function calls so that the code skeleton generated by DPnDP would be easier for the user to understand and modify.

One powerful aspect of this model is that the programmer is not restricted to developing the complete application using only the high level design patterns in DPnDP. The system is open and the code skeletons generated can be easily understood and modified to tune for performance. Also, development of an application can combine high level design patterns with low level message passing features. For example, in our present implementation, the message passing layer is built using PVM. Therefore, nodes in an application that are not associated with any design pattern could use any message passing feature of PVM. This also allows design patterns to be used only as part of a bigger PVM program.

Another important feature is extensibility. By laying down a uniform way of defining and implementing design patterns, programmers can develop new patterns and add them to the library incrementally, thus making the system truly extensible. A large collection of commonly used patterns would enhance the utility of the system as well.

DPnDP has been implemented using workstation clusters that run under UNIX. Work is also in progress to port the system to tightly coupled message passing and shared memory multiprocessors.

4 Related Research Works

A number of researchers and system designers have recognized the significance of commonly occurring parallel structures. A number of parallel programming systems support such structures [3, 11, 1, 13, 10, 14, 12, 2]. The design of DPnDP has in part been inspired by this large body of work. However, DPnDP differs from most of the previous systems in three significant aspects.

Firstly, most of the previous systems support only interconnection of modules (possibly with special syntax) to perform replication, fan in or fan out. There is no encapsulation of higher level communication behavior as design patterns. Furthermore, for systems that support higher level design patterns, such as TRAC [2], their structure and behavior are not parameterized. This, for example, would require separate patterns for divide and conquer with different depths and widths. DPnDP tries to raise the level of abstraction to provide parameterized parallel structural and behavioral design patterns. They can be used in different context when supplied with different parameters.

Secondly, the support for high level parallel structures in most previous systems is built directly into the design (and implementation) of the system. This would mean that only system programmers can add new parallel structures to the system and it usually requires major modification. The users cannot add their own parallel structures for reusability. On the other hand, in DPnDP, the context insensitive nature of the design patterns allows gradual addition of new patterns to the system. For example, even the existing library of design patterns of DPnDP can support all the high level structures found in systems like FrameWorks [13], Enterprise [10], HeNCE [1], DGL [7], etc.

Thirdly, most often while developing applications with such systems, the user is restricted to using only the high level structures supported by the system. If the user's application requires certain structures that are not directly supported by the system, it may be very difficult or even impossible to develop the application using the system. In DPnDP, the use of design patterns can be combined with the use of primitives supported by the lower layers of the system.

To the best of our knowledge, TRAC [2] is the only other parallel programming system that has the goal of providing openness and extensibility while supporting high level parallel structures. However, there are two major differences between TRAC and DPnDP. In TRAC, the user can design certain multiprocess structure, save it in the library and use it later. The system does not use any standard interface for design patterns. This would adversely impact the capability to compose or refine an application using more than one design pattern. Also, the structural aspects of design patterns are not parameterized.

5 Summary

DPnDP is the first system that has demonstrated implementation and use of parameterized application independent design patterns as a library of code skeletons. The library is extensible. A parallel application can be composed using several design patterns. Also, design patterns can be combined with parts that may use low level communication and synchronization primitives.

6 Acknowledgments

This research was conducted using grants from the Natural Sciences and Engineering Research Council of Canada and IBM Canada Ltd.

References

[1] A. Baguelin, J. Dongarra, G. Giest, R. Manchek, and V. Sunderam. Graphical development tools for network-based concurrent computing. In *Supercomputing'91*, pages 435–444, 1991.

[2] A. Bartoli, P. Cosini, G. Dini, and C.A. Prete. Graphical design of distributed applications through reusable components. *IEEE Parallel and Distributed Technology*, 3(1):37–51, 1995.

[3] M. Cole. Algorithmic skeletons: Structured management of parallel programming. *MIT Press, Cambridge, Mass.*, 1989.

[4] M. De Simone. Openness and extendibility in high level parallel programming systems. *Electrical and Computer Engineering Dept., University of Waterloo*, 1995. Internal report.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, 1994.

[6] G. Geist and V. Sunderam. Network-based concurrent computing on the PVM system. *Concurrency: Practice and Experience*, 4(4):293–311, 1992.

[7] R. Jagannathan, A.R. Downing, W.T. Zaumen, and R.K.S. Lee. Dataflow based technology for coarse-grain multiprocessing on a network of workstations. In *International Conference on Parallel Processing*, pages 209–216, August 1989.

[8] S.J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman. The design and implementation of 4.3 BSD Unix operating system. *Addison-Wesley Publishing Company, Inc.*, 1990.

[9] X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi. On the versatility of parallel sorting by regular sampling. Technical report, University of Alberta, June 1991.

[10] J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons. The enterprise model for developing distributed applications. *IEEE Parallel and Distributed Technology*, 1(3):85–96, 1993.

[11] Z. Segall and L. Rudolph. PIE: A programming and instrumentation environment for parallel processing. *IEEE Software*, 2(6):22–37, 1985.

[12] A. Singh, J. Schaeffer, and M. Green. A template-based tool for building applications in a multicomputer network environment. In D. Evans, G. Joubert, and F. Peters, editors, *Parallel Computing 89*, pages 461–466. North-Holland, Amsterdam, 1989.

[13] A. Singh, J. Schaeffer, and M. Green. A template-based approach to the generation of distributed applications using a network of workstations. *IEEE Transactions of Parallel and Distributed Systems*, 2(1):52–67, January 1991.

[14] A. Singh, J. Schaeffer, and D. Szafron. Experience with template-based parallel programming. *Submitted to a Journal*, 1996.

[15] L. Wall and R. L. Schwartz. *Programming perl.* O'Reilly & Associates Inc., 1991.