Composing Parallel Applications Using Design Patterns

Stephen Siu and Ajit Singh (Contact Author)

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

Building software tools that ease the development of parallel applications is one of the primary concerns in the area of parallel computing. To deal with the complexity of software development, abstractions such as macros, functions, abstract data types, and objects are commonly employed by sequential as well as parallel programming models. This paper describes the concept of a design pattern for the development of parallel applications. A design pattern in our case describes a recurring parallel programming problem and a reusable solution to that problem. A design pattern is implemented as a reusable code skeleton 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 uniqueness of our approach is in the use of a standard structure and interface for a design pattern. This has several important implications: First, design patterns can be defined and added to the system's library in an incremental manner without requiring any major modification of the system (Extensibility). Second, customization of a parallel application is possible by mixing design patterns with low level parallel code resulting in a flexible and efficient parallel programming tool (Flexibility). Also, a parallel design pattern can be parameterized to provide some variations in terms of structure and behavior. A parallel programming system, called DPnDP (Design Patterns and Distributed Processes), that employs such design patterns is described.

1 Introduction

Designing software tools that simplify the task of development of parallel applications is a major concern in the area of parallel computing. Even though networks of single and multiprocessor workstations are now commonplace, most of the the programs are still being written for single processor computers. This is often attributed to the complexities involved in developing parallel applications.

To reduce the complexity of software development, abstractions such as macros, functions, abstract data types, and objects are commonly employed by sequential as well as parallel programming models. In this paper we introduce the concept of design patterns that are meant specifically for providing solution strategies for parallel applications. These design patterns 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.

A design pattern describes a recurring problem and the essence of a reusable solution to that problem. 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.

DPnDP is a design pattern driven parallel programming system that, to a large extent, 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. The present collection of design patterns in DPnDP is especially suited to developing parallel applications using a network of processors. DPnDP is part of our ongoing effort in structuring design patterns for parallel programming. Earlier versions of this work are described in [22, 23].

The paper is organized as follows: Section 2 presents an example to highlight the issues addressed in this work. Section 3 presents the DPnDP parallel programming model. It also briefly describes the design patterns currently present in DPnDP's library and provides an example illustrating the process of application development using the programming model. Implementation of the DPnDP system is described in section 4. A critical assessment of the DPnDP system and its performance are considered in section 5. The research work presented here is discussed in the context of other related works in section 6. Section 7 presents some future directions for the project and also provides the concluding remarks.

2 Design Patterns for Parallel Programming: An Example

In the context of parallel programming, a design pattern represents a prepackaged set of characteristics which can fully or partially specify the nature of scheduling, communication, and synchronization of an entity. Design patterns implement various types of interactions found in parallel systems, but with the key components – the application-specific structures and procedures – unspecified. A user provides the application-specific structures and procedures and the tool provides the glue to bind it all together. The design patterns abstract commonly occurring structures and characteristics of parallel applications, allowing users to develop parallel applications in a rapid and easy manner.

To understand the nature of issues involved here, 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, 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.

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 1a). Now, if Display() takes much longer to do its processing as compared to

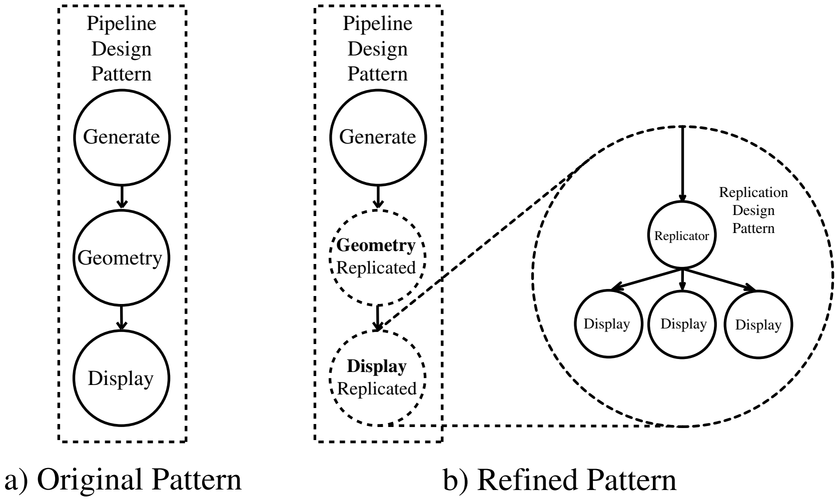

Figure 1: Structure of the Graphics Animation program

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 the performance of Geometry() is to be improved, several instances of it may be initiated as well. This situation is shown in Figure 1b where Geometry() and Display() have several active instances.

The parallel versions of this graphics example contain a few of the commonly-used structures for parallel computing, such as a pipeline and a replication pattern. Consider parallelizing this application on, for example, a network of workstations. Parallel program development would require a significant amount of time and effort if a low-level tool is used (for example, UNIX sockets [14] or a message-passing library such as PVM [12] or MPI [25]). Further, the parallelism would be explicit, 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 to rewrite the code. Moreover, such an effort would be replicated, knowingly or unknowingly, by other programmers while writing other applications.

A design pattern based system could be used here to address this situation. Such a system should provide skeletons [9] (also sometimes called templates [16, 20, 21]) of implementations of parallel structures. A user simply provides sequential modules of code and selects appropriate skeletons for structuring their parallel application. The procedural relationships in the diagram indicate that the three modules interact in a pipeline manner and that *Geometry()* and *Display()* can have multiple instances that execute independently from each other. The choice of patterns indicates the communication pattern that should be generated by the system. The resulting parallel program automatically spawns the processes on available processors, establishes the communication links and ensures the proper communication and synchronization. From the user's point of view, most of the coding is sequential; the parallel aspects are provided by the system. By separating the application-specific code from the parallel implementation, a design pattern-based development tool has the potential to decrease both the chances of program error due to parallelization and program development time.

There are several high level parallel programming systems that directly support the high level patterns used in the above example (for example [2, 3, 4, 6, 7, 18]). The essential difference between the existing systems and DPnDP is in having a generic definition of a design pattern that provides a standard structure and interface for a pattern. This allows new design patterns to be integrated into the system in a gradual manner. These patterns can also be parameterized to capture certain variations in structure or behavior. Also, unlike existing systems, the user is not restricted to working only with the design patterns. Instead, the use of the design patterns can be combined with the use of low layer communication primitives. These distinctions between existing high level parallel programming systems and DPnDP are further elaborated in section 6 where we discuss DPnDP in the context of existing research works.

3 The DPnDP Programming Model

The DPnDP parallel programming model assumes a MIMD processor architecture and an operating system that supports process creation and message passing among processes. In this model, a parallel program is represented by a directed graph (Figure 2). Each node of the graph may be a single-process design pattern, called a singleton, or a multi-process design pattern. Nodes in the graph communicate and synchronize by message passing. Each node in the graph has a set of input and output ports that are used for receiving and sending messages respectively. Output port of a node is connected to an input port of another node. When a node sends a message to one of its output ports it reaches the input port of the connected node which can receive this message. We have found the abstraction of nodes and ports quite valuable in designing and using our design-pattern based system. It has allowed the model to remain independent of the specifics of the underlying message passing models such as Sockets [14], PVM [12] or MPI [25]. Similar abstractions have previously been used by other researchers [1].

A node in the application graph (Figure 2) could be a multi-process structure that abstracts a design pattern. Such a node has one or more processes that act as interface processes for the node. Only interface processes of a multi-process node directly communicate with other nodes in the application. The interface of a node is indistinguishable from that of a single process, i.e., in each case other nodes interact with it via message passing through a common interface. The interface processes of a multi-process design pattern hide the inner details of the pattern's structure and implementation, and present the interface of a single process to other nodes. Interpretation of messages received and sent by interface processes is specific to the design-pattern encapsulated by the node. This interpretation forms part of the documentation required for a design pattern.

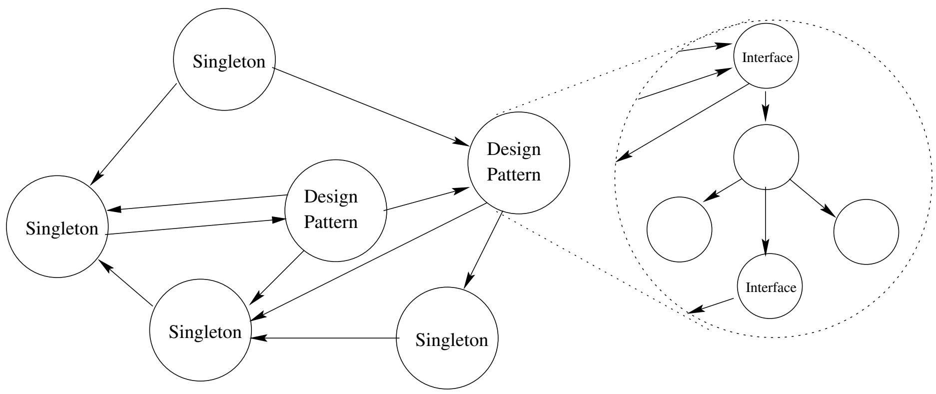

Figure 2: Structure of a DPnDP Application

Each process (or node) in a DPnDP application operates in a service-loop whereby it waits for incoming messages on any of its ports from other processes (Figure 3). When an asynchronous message arrives at a port of a process, the process notifies an appropriate user-provided message handler to process the message. The message handler receives the message from the port, processes it, and sends out messages, through ports in the process, to other processes if appropriate. The message-handler can also call any local function in the process. Thus, every process in the application may operate in the dual role of a client as well as a server: it provides services to other processes and in doing so, it may send requests for service to other processes connected through its ports. At any point within the user-code, the process can perform a *non-blocking send* to any port of the process or a *blocking receive* from any port of the process to wait for a particular incoming message.

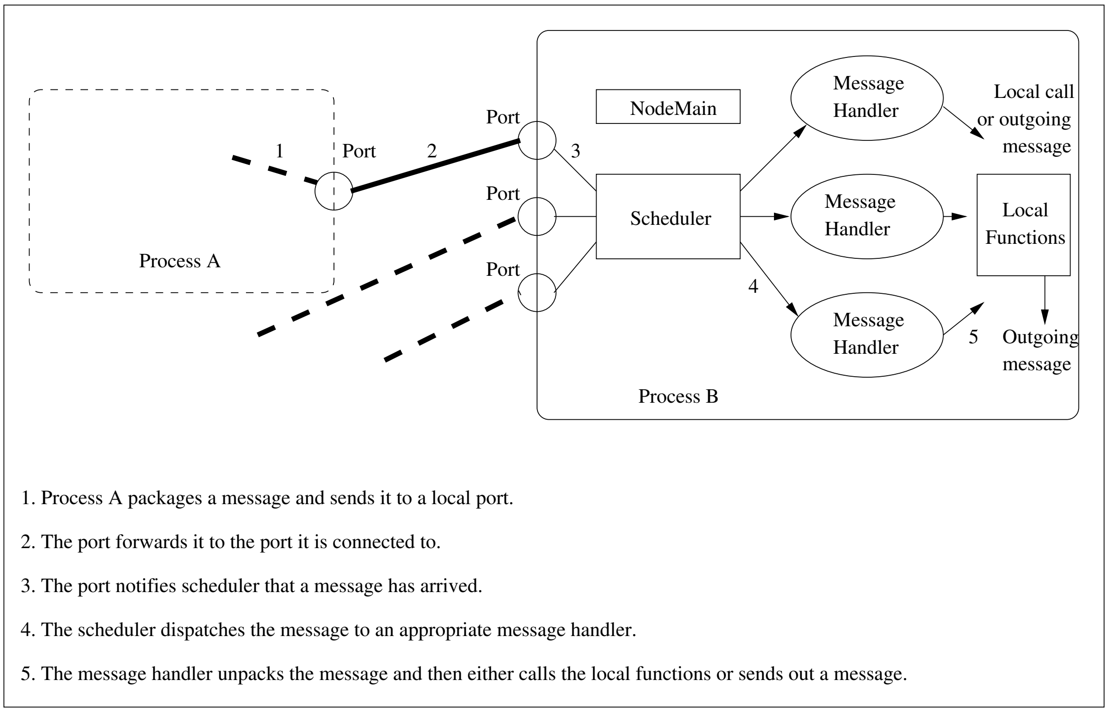

Figure 3: The Message Passing Model of a DPnDP Process

As shown in Figure 3, each process also has a special system-generated function called “NodeMain” which is used to perform initialization before the process waits for incoming messages. DPnDP code skeleton generates all communication code except the contents of the application-specific message handlers. Users can also provide their own “NodeMain” to perform application-specific initialization.

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 the minimum amount of communication code necessary to send application specific data and results among nodes. The design patterns abstract parallel program structuring techniques, allowing users to develop parallel applications in a rapid and easy manner. The approach also 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 pipeline, master-slave, divide and conquer, and process replication. Simple parallel applications use only one design pattern. Complex parallel programs can use a combination of design patterns. DPnDP generates communication code skeletons according to the specific graph. Developers can then insert application specific code into these skeletons.

When using a design pattern, a user only deals with communication that is application related. All other synchronization and communication needed for proper management of the processes in a design pattern is taken care by the generated code. Every design pattern is designed in a similar way in the sense that it implements certain process structure and behavior. The behavior of the pattern is manifested by the types of messages received by the design pattern and their processing. Since a design pattern may implement an arbitrarily complex parallel solution strategy, proper documentation is necessary to explain the operation and usage of each design pattern. Documentation of a design pattern describes the types of messages exchanged by the design pattern and their interpretation. For example, in a replication pattern (described later in this section), the pattern simply executes a number of copies of the user supplied application module. The interface process for this pattern, called coordinator, receives work from other nodes. The coordinator simply passes this work to a currently available replica. If necessary, the application code in replica sends back the result via a message to the coordinator which then forwards the message to the appropriate node. All the necessary process creations and interprocess communication for the management of replicated worker processes is handled by the code generated for the specified number of replicas. The exact number of replicas may be specified statically or varied at execution time.

3.1 Present Collection of Design Patterns

The utility of a design-pattern-based parallel programming system depends on the design patterns in the system. Here are a collection of design patterns currently supported by DPnDP (Figure 4). New patterns can be added to the library. The existing patterns can also be building blocks for new design patterns.

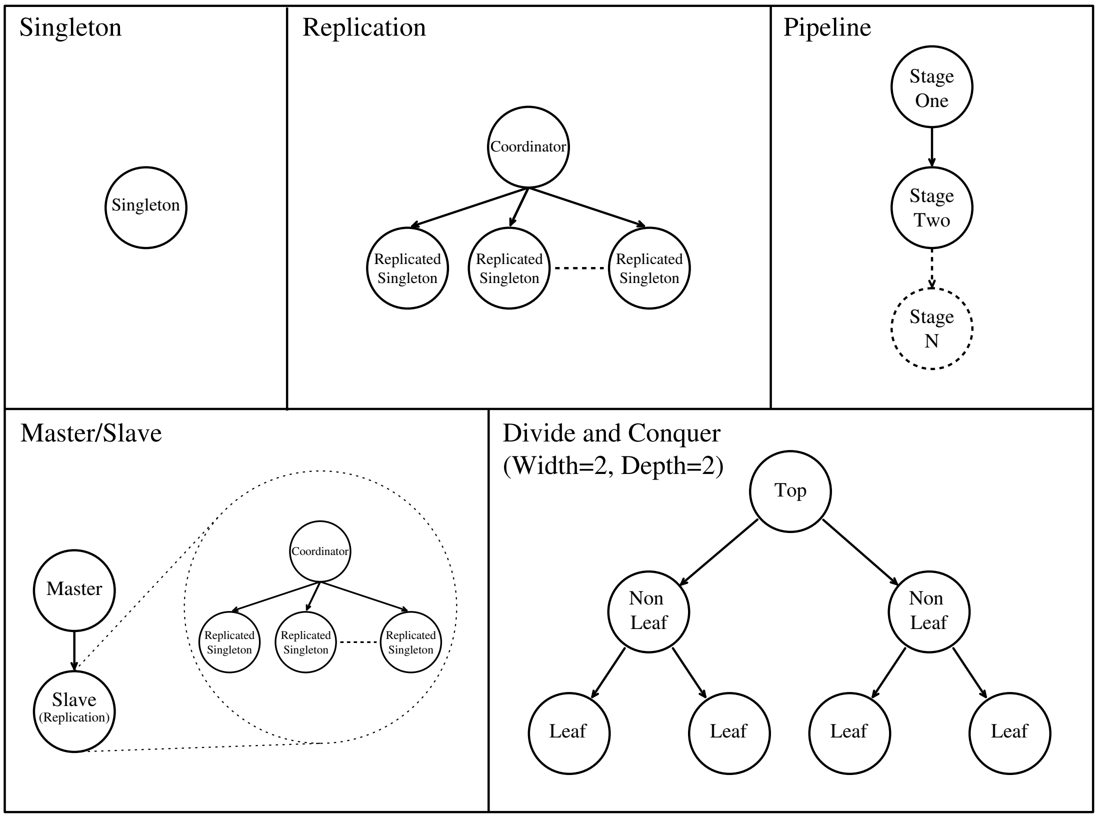

Figure 4: Design Patterns in DPnDP

- Singleton:

  A *singleton* is a single-process design pattern. In conventional terms, it represents a sequential process that directly interfaces with other nodes and communicates with them via message passing through its input and output ports. It also serves as a basic building block for other patterns.

- Replication:

  Externally, a *replication* appears like a singleton, but internally it replicates multiple copies of a singleton. Therefore, it can service multiple messages at a time. As shown in Figure 4, a replication contains a *coordinator* that acts as its input and output interface. The coordinator accepts external input. It finds a *replicated singleton* that is selected based on a load balancing algorithm and forwards the input to it for processing. After a replicated singleton finishes processing the message, it sends the result back to the coordinator which, in turn, forwards the message appropriately. Replication has one parameter for developers to specify: the number of required replicas.

- Pipeline:

  A *pipeline* pattern consists of an ordered set of stages in which the output of each stage is the input of its successor. A pipeline has multiple singletons connected in a chain-like fashion where the output of a singleton is forwarded to the next singleton in the chain. The first stage of the pipeline acts as the input interface and the last stage of the pipeline acts as the output interface of the design pattern. A pipeline has one parameter for developers to specify: the number of required stages.

- Master/Slave:

  A *master/slave*, also called a task farm, consists of a singleton, the Master, and a replication, the Slave. The master generates requests and passes them to the slaves. A slave services the request and sends the results back to the master. Since the slave is replicated, multiple requests can be serviced at a time. Master/Slave has one parameter for developers to specify: the number of required slaves. Master/Slave is an example of compositional pattern. It is created by combining the Singleton and the Replication patterns.

- Divide and Conquer:

  A *divide and conquer* represents a tree of singletons where parent processes recursively pass work down to its children processes. When a child finishes the work, it returns the result back to its parent. Divide and Conquer has two parameters: one specifying the width of the tree and the other specifying the height of the tree (the level of recursion). The root or top process of the tree is the interface process for this design pattern. The behavior of divide and conquer is provided by supporting the functions for sending the work to children processes and collecting the results from them.

The design patterns presented above are mostly based on medium and coarse grain parallel structures. Such patterns are specifically suited to parallel computing over a processor-network.

3.2 An Example Application

Parallel implementation of the regular quick sort 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 [15] 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. A slightly modified version of this parallel algorithm can be implemented by using a single master/slave parallel design pattern (Figure 5).

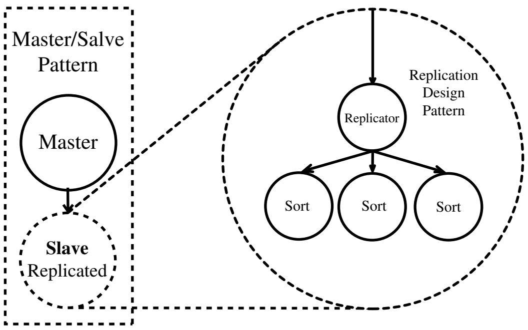

Figure 5: Structure of the PQSRS program

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 6) and another for the slave node (Figure 7).
5. Issue the "make" command. All executables will be built and copied to the appropriate directory for execution.
6. Execute the parallel program.

In our implementation of the PQSRS algorithm, the master process partitions the list into N sublists where N is the number of slaves used by the master-slave pattern. The sublists are distributed to slaves that sort the sublists and send the results back to the master. The master then picks some sample data from these sublists, sorts the sample data and selects some pivot values from the sorted list of regular samples. The pivot values are used to rearrange the sublists according to the scheme specified by the algorithm [15]. The new sublists are sent to the slaves again for sorting. The results are collected to form the final sorted list. Experimental results showing the performance of the application on a network of Unix workstations connected over an ethernet are given in section 5.

4 Implementation of DPnDP System

DPnDP has been implemented using a network of workstations that run under the Solaris operating system. The system has the following components: user interface, design pattern library, code skeleton generator and other supporting libraries (Figure 8). A user specifies the directed graph through the user interface. A Motif interface is used to let users specify arbitrary process graphs using functions

```
// NodeMain is the startup procedure used in the Master node to send
// work to the Slaves.  It takes a global port list as it first
// argument.  argc and argv are arguments passed into the program at
// the command line.
//------------------------------------------------------------------
int NodeMain(NL_PortList& thePorts, int argc, char* argv[])
{
    // Cache output port handle from Port Array
    NL_Port& InPort = (*gPortList)["in0"];    // The Port that receives results
    NL_Port& OutPort = (*gPortList)["out0"];  // The Port it sends work to
    ...                                        // Slaves

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

    // Step 2: Distribute to slaves for sorting
    for (i = 0; i < NoOfProcess; i+=1)
        // Port << Message ID << Array Size << Array << send message;
        OutPort << i << Size << array(Temp[i].ArrayPtr, Size) << flush;

    // Step 3: Collect Messages
    for (i = 0; i < NoOfProcess; i+=1) {
        int index, aSize;
        // Port >> receive message >> Message ID >> Array Size >> Array;
        InPort >> fetch >> index >> aSize >> array(Temp[index].ArrayPtr, aSize);
    } // 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 6: Entry procedure for the Master node

```
// It is a message handler that processes messages coming from the Master
// node.  Since each message handler can handle messages from multiple ports,
// its first parameter contains the port where a message is waiting for
// processing.  A Scheduling Primitive in a node dispatches messages
// to appropriate message handlers.
//----------------------------------------------------------------------
int ASP_M_NodeMain::MsgHandler(const NL_Port& thePort)
{
    // Cache output port handle from Global Port Array
    NL_Port& OutPort = (*gPortList)["output0"]; // OutPort is where it sends
                                                // back the result to the
                                                // Master
    int size, ID; int *buf;

    // Get the message from received port
    thePort >> fetch >> ID;                     // Receive message ID
    thePort >> fetch >> size;                   // Receive array size
    buf = new int[size];                        // Allocate Memory for Message
    thePort >> array(buf, size);                // Receive Message

    QuickSort(buf, size);

    // Send the message back to where it comes form
    OutPort << ID << size << array(buf, size) << flush;  // Send ID, size and array

    delete [] buf;                              // Free Memory used
    return GOOD;
} // MsgHandler
```

Figure 7: Message handler for the Slave node

provided in the interface (Figure 9). It provides features to 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 compose an application using several design patterns as a graphical representation of a process graph is easier to visualize and modify.

The back-end of the user interface uses components from the design pattern library. The library implements intermediate level code for parallel design patterns. These design patterns capture the structural and behavioral information of a parallel solution strategy. The design patterns are generally parameterized. For example, a 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 (a N-stage pipeline) rather than specific instances of them (a 2-stage or a 3-stage pipeline). Thus by instantiating specified design patterns with appropriate parameters, the user interface passes all the structural information provided by the user into an intermediate design file.

The Code Skeleton Generator translates the intermediate representation produced by the design pattern library into a set of source code skeletons along with makefiles that are packaged into directories. Separating code generation from the design pattern library allows DPnDP to generate code skeletons for multiple programming languages, message passing libraries and operating systems without modifying the design patterns in the library. Currently, the Code Skeleton Generator produces C++/PVM code for the Solaris operating system. Code Skeleton Generators is written in perl [26], a cross platform text and file processing language. User provided application code is added to the generated code skeletons and the code is compiled to produce executable modules for the parallel application (Figure 8b).

Figure 10 shows the layered architecture of DPnDP's message passing subsystem. The MPL layer presently consists of PVM. The "*NodeLayer*" [10] forms a thin layer on top of MPL. NodeLayer abstracts the peculiarities of different message passing libraries and provides the programmer a common interface for message passing that is based on nodes and ports. Other than using the abstraction of nodes and ports, NodeLayer provides message passing capabilities similar to those found in a message passing system like PVM.

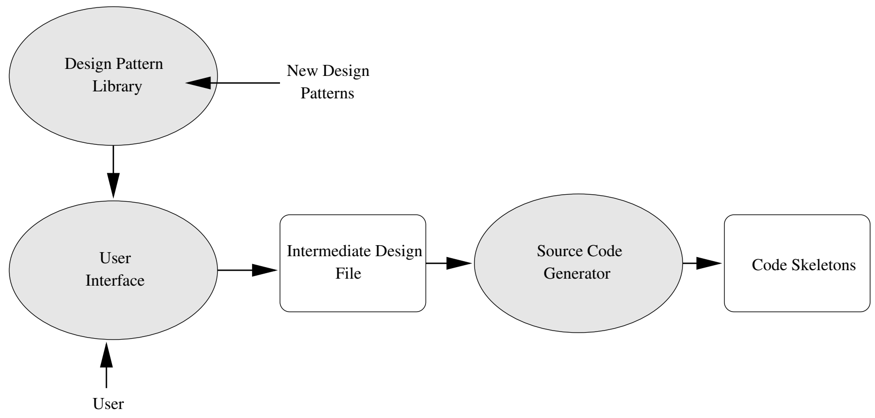

Figure 8a: Operational Architecture (Part I)

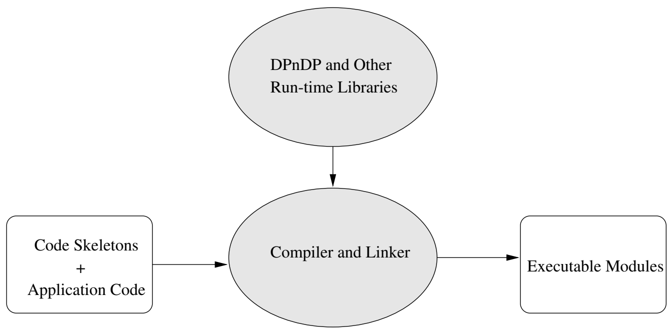

Figure 8b: Operational Architecture (Part II)

Figure 8: Operational Architecture of DPnDP System

5 A Critical Assessment of DPnDP

After having described the DPnDP model and the system, we turn our attention, in this section, towards reviewing the features of extensibility and flexibility in DPnDP. Results of experiments conducted to assess the performance of the system are also described.

DPnDP model is essentially based on the message passing paradigm. The model has been implemented using the layered message passing architecture as discussed earlier (Figure 10). An important implication of this approach is that the programmer is not restricted to developing an entire application using design patterns. For example, an application can be developed using singletons that use the message passing features of NodeLayer. Similarly, an application may be composed using one or more

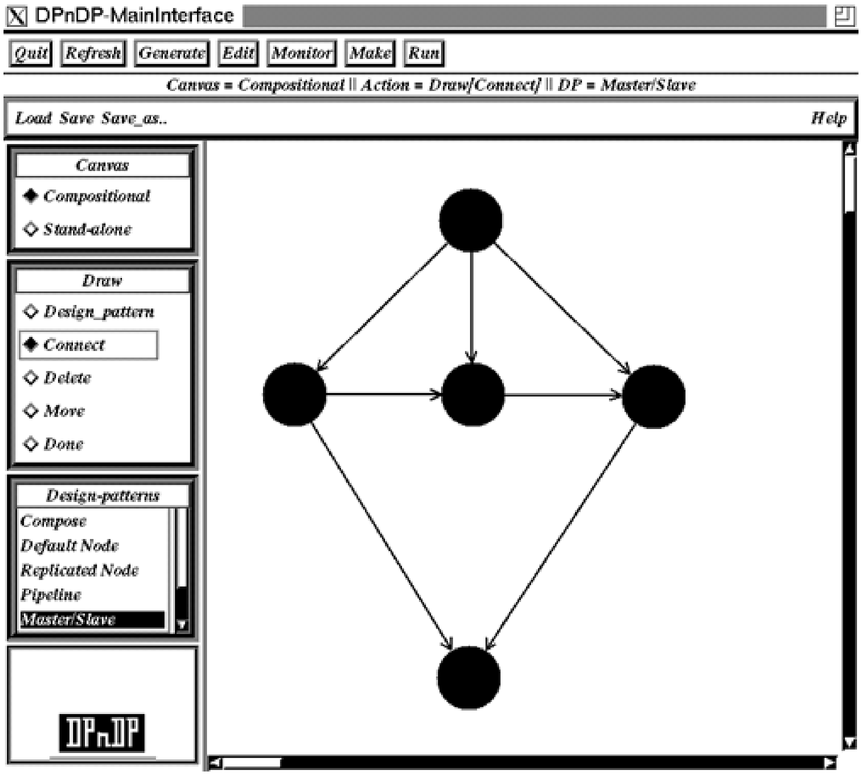

Figure 9: Motif Interface for DPnDP

singletons and some multi-process design patterns. Thus a programmer can mix the high level abstractions provided by the design patterns with the flexibility of using low level message passing primitives.

By laying down a uniform way of defining and implementing design patterns, programmers can develop new patterns and add them to the library incrementally making the system truly extensible. All design patterns in DPnDP have a uniform way of definition and implementation. Patterns have been packaged in a C++ class library. 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. The approach enforces a common interface so that all design patterns can be accessed the same way and thus can be used interchangeably. From the viewpoint of interaction among patterns, each pattern is context insensitive meaning that it does not need to know the implementation of any other patterns in order to work with them.

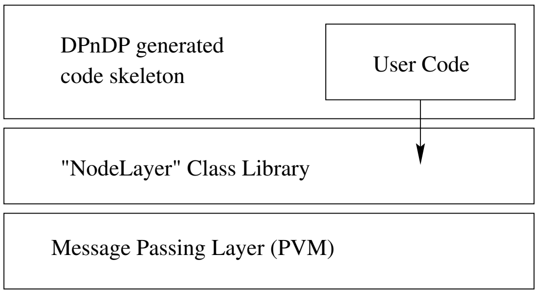

Figure 10: Layered Architecture of DPnDP's Message Passing Subsystem

5.1 Performance of DPnDP System

Performance is another primary concern for any parallel programming system. In the context of DPnDP, it is important to ensure that the implementation of flexibility and extensibility in DPnDP does not severely compromise the performance of programs it generates.

This section reports performance of programs generated using DPnDP. Two types of programs were used to evaluate the performance of DPnDP. The first type consists of application independent programs. These are benchmark programs to evaluate a particular aspect of the system such as its message passing efficiency. The second type of experiments deal with the performance of the PQSRS application described earlier.

5.2 Application Independent Performance

5.2.1 Message Passing Performance

The current version of NodeLayer is built on top of PVM. Therefore, tests were done to measure its message passing performance with respect to PVM. This benchmark program measures how long it takes one process to send 1000 messages to another process. Two versions of this program were developed: one using PVM and the other using NodeLayer. To ensure fairness, these tests were all run on the same workstations connected by ethernet. Messages of different sizes are sent using both NodeLayer and PVM. Each program was run ten times and the results are presented in Table 1.

Message Size (int)	NodeLayer (s/1000 msg)			PVM (s/1000 msg)			% Differences
	Avg.	Max	Min	Avg.	Max	Min	
1	28.64	29.67	27.57	28.14	28.81	27.38	1.71%
10	28.50	30.41	26.23	28.68	32.05	27.64	-0.63%
100	31.30	32.52	29.17	28.88	29.59	28.15	8.37%
1000	44.43	56.69	39.86	40.71	52.18	37.92	9.13%
10000	243.85	322.40	218.21	233.22	288.58	216.96	4.56%

Table 1: Message Passing Performance of NodeLayer

The time taken for NodeLayer to send 1000 messages of various length is all within 10% of the time taken for PVM to send the same 1000 messages. The result is expected because NodeLayer is only a very thin encapsulation on top of PVM with overhead of only a few function calls. The anomalous result with messages of size ten, where NodeLayer appears faster than PVM is most likely due to the variations in the load of the communication network that connects the workstations. Overall, these results show that NodeLayer's performance is somewhat slower but still comparable to PVM independent of the message size.

5.2.2 Code Skeleton Performance

Tests were also done to compare the performance of a code skeleton generated by DPnDP design patterns. Two Master/Slave programs were developed, one using the DPnDP generated code skeletons, the other using PVM message passing primitives. To ensure fairness, the PVM program and the DPnDP program have identical computation code, identical process structure (identical process graph) and are executed on the same workstation cluster. The PVM version is hand-crafted with no error-checking to obtain maximum performance. The DPnDP version is used “as is” without any modification. Each program was run ten times and their execution time is shown in Table 2.

The performance penalty of the code skeleton generated by DPnDP design patterns is on average around 10% compared to the PVM program. These overheads are partially due to extra error-checking costs in the NodeLayer library that are not in PVM. The replication pattern also performs load balancing

No. of Stages	NodeLayer			PVM			% Difference
	Avg.	Max	Min	Avg.	Max	Min	
2	14.19	16.17	13.31	15.01	17.26	13.35	8.37%
4	19.53	20.22	17.89	20.36	24.35	16.04	4.22%
8	25.04	31.13	21.73	27.89	28.60	26.89	11.41%
16	35.81	37.53	35.08	38.14	39.67	37.41	6.49%

Table 2: Performance of DPnDP code skeleton

while the PVM program does not. Therefore, the results show that code skeleton generated by design pattern is not as efficient as the hand crafted PVM version. It is understandable since DPnDP is built on top of PVM. Such performance penalty is expected in high-level systems which provide more robust code. Other design-pattern-based systems report similar or worse loss of performance[24].

5.3 Application Specific Performance

The PQSRS application described in section 3 was implemented and experiments were conducted to ascertain its performance on a cluster of Sun Sparc workstations connected by ethernet. The application was used to sort randomly generated integers. The Table 3 and Figure 11 show the results of the Parallel Quicksort with Regular Sampling (PQSRS) program. It exhibits good speed-up because it distributes its workload evenly among available processors. It fails to achieve better speed-up with more than eight processors when the grain size becomes too small for a processor cluster connected by a 10-megabit ethernet.

6 Related Research Works

The concept of design patterns has previously been discussed in the area of software engineering [11]. Design patterns are abstract descriptions of the commonly used strategies for software design. Concrete realizations of design patterns in terms of code and associated documentation has been referred to as frameworks. Also, a family of patterns that cover a particular domain is called a pattern system or a pattern language [17]. From this viewpoint, DPnDP deals with a pattern system for parallel

Number of Processor	Time (s) (avg. over 10 runs)	Speed-Up Ratio
1	4587.87	1.0
2	3348.75	1.4
4	1854.76	2.5
8	647.601	7.1
16	727.452	6.3

Table 3: Performance of Parallel Quicksort with Regular Sampling

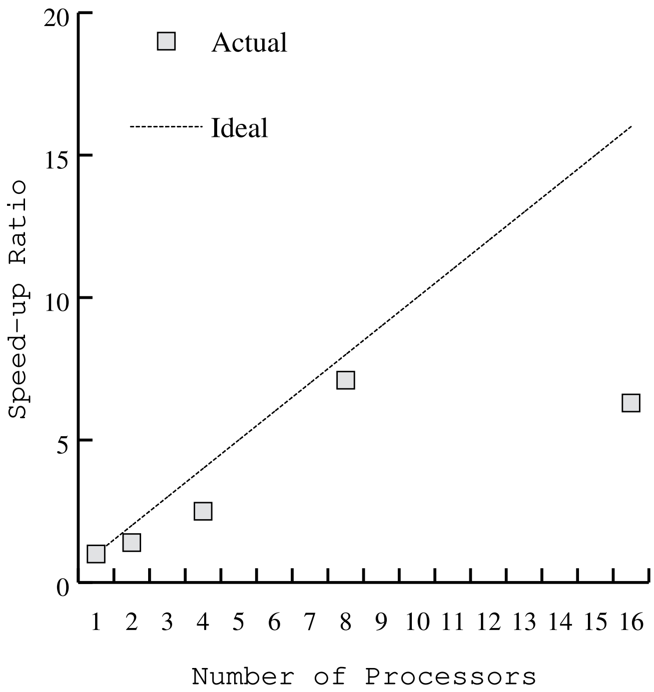

Figure 11: Speed-up of Parallel Quicksort with Regular Sampling

programming with an important distinction that the number of patterns belonging to the pattern system is not static. Patterns can be gradually added to the system.

The idea of using commonly occurring parallel structures for parallel programming is also not new. Its significance has previously been recognized by a number of researchers and system designers. A number of parallel programming systems support such structures [3, 4, 9, 16, 18, 20]. All these systems employ the technique of separation of specifications which means that the parallelization aspects of the application are specified separately from the application code. Although other software engineering techniques, such as macros and code libraries, also provide high-level abstractions, the *separation* of specifications requirement is a key difference between design patterns and these methods. For example, to use macros or library functions, the programmer must insert macro or function calls in the application code. The use of patterns on the other hand is much less intrusive in the sense that the sequential code of the application need not have any reference to the patterns it is attached to. This has important implications for initial program development as well as the restructuring of parallel applications.

The design of DPnDP has in part been inspired by this large body of above-mentioned previous work on supporting common parallel structures in high level parallel programming tools. However, DPnDP differs from most of the previous systems in three significant aspects.

First, 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 [4], the patterns 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 by providing parameterized parallel design patterns.

Second, the support for high level parallel structures in most previous systems is built directly into the design (and implementation) of the system. This means that adding a new parallel design pattern to the system usually requires major modifications to the system. On the other hand, in DPnDP the context insensitive nature of design patterns allows gradual addition of the 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 [20], Enterprise [16], HeNCE [3], DGL [13], etc. Each pattern has been developed in a context insensitive manner, i.e., its definition and implementation is unaware of the existence of other design patterns in the library.

Third, most often while developing applications with existing parallel programming 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, 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 [4] is the only other parallel programming system that has the goal of providing flexibility and extensibility while supporting high level parallel structures. However, there are two major differences between TRAC and DPnDP. In TRAC, a user can design certain multi-process 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 an application using more than one design pattern. Also, the design patterns in TRAC are not parameterized.

Recently, some researchers have turned their attention towards investigating application- independent solution strategies from the viewpoints of understanding and classifying them. In [5], a programming paradigm for parallel computing is defined as a class of algorithms that solve different problems but have the same control structure. The term archetype is used in [8] to denote a program design strategy for a class of parallel problems along with the associated program designs and example implementations. Emphasis here is on enhancing a developer's understanding of common classes of parallel problems via documentation and example implementations.

7 Future Directions and Conclusion

Supporting the abstraction of high level design patterns as reusable implementations of commonly occurring parallel parallel structures is a challenging goal. To the best of our knowledge, DPnDP is the first model and system that aims to provide such patterns in the form of parameterized and application independent library of code skeletons. The context insensitive nature of a design pattern facilitates extensibility of the design pattern library. Also, a parallel application can be composed using several design patterns as well as combining them with parts that may use low level communication and synchronization primitives.

So far, the thrust of the DPnDP system has been towards parallel structures that are suitable for message passing MIMD environments. Incorporation of new patterns for such environments is an ongoing research activity. We are presently also investigating patterns for algorithms that require peer-to-peer interactions. The goal of this project is to support patterns that can be used for parallel programming on a cluster of computers some of which may be shared memory multiprocessors. For this, we need to also look at patterns that are suited to shared memory based algorithms. Also, the code skeletons for the existing design patterns are continually being refined to enhance their usability.

8 Acknowledgments

We would like to thank M. De Simone and D. Goswami for their help in developing the DPnDP system. This research was conducted using grants from the Natural Sciences and Engineering Research Council of Canada and IBM Canada Ltd.

References

[1] G.R. Andrews and Fred B. Schneider. "Concepts and Notations for Concurrent Programming". *ACM Computing Surveys*, 15(1):3–43, 1983.

[2] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. "P3L: A Structured High Level Parallel Programming Language and its Structured Support". *Technical Report HPL-PSC 93-55, Pisa Science Centre, Italy*, 1993.

[3] 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.

[4] 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.

[5] P. Brinch Hansen. "Search for Simplicity: Essays in Parallel Programming". *IEEE Computer Society Press*, pages 422–446 (Chapter 22), 1996.

[6] J.C. Browne, M. Azam, and S. Sobek. "CODE: A Unified Approach to Parallel Programming". *IEEE Software*, pages 10–18, July 1989.

[7] J.C. Browne, S. Hyder, J. Dongarra, K. Moore, and P. Newton. "Visual Programming and Debugging for Parallel Computing". *IEEE Parallel and Distributed Technology*, 3(1):75–83, 1995.

[8] K. M. Chandy. "The Caltech Archetype/eText Project (Keynote Address)". In *International Parallel Programming Symposium*, Septaember 1996. http://www.etext.caltech.edu.

[9] M. Cole. "Algorithmic Skeletons: Structured Management of Parallel Programming". *MIT Press, Cambridge, Mass.*, 1989.

[10] M. De Simone. "Openness and Extendibility in High Level Parallel Programming Systems". *Electrical and Computer Engineering Dept., University of Waterloo*, 1995. Internal report.

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

[12] G. Geist and V. Sunderam. "Network-Based Concurrent Computing on the PVM System". *Concurrency: Practice and Experience*, 4(4):293–311, 1992.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] D.C. Schmidt. "Experience Using Design Patterns to Develop Reusable Object-Oriented Communication Software". *CACAM*, 38(10), October 1995.

[18] Z. Segall and L. Rudolph. "PIE: A Programming and Instrumentation Environment for Parallel Processing". *IEEE Software*, 2(6):22–37, 1985.

[19] 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.

[20] 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.

[21] Ajit Singh, Jonathan Schaeffer, and Duane Szafron. "Experience with Parallel Programming Using Code Templates". *Concurrency: Practice and Experience*, 1997. Accepted for Publication.

[22] S. Siu, M. De Simone, D. Goswami, and A. Singh. "Design Patterns for Parallel Programming". In *Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications*, pages 230–240, California, U.S.A., August 1996.

[23] S. Siu and A. Singh. "Design Patterns for Parallel Processing Using a Network of Processors". In *Sixth IEEE International Symposium on High Performance Distributed Computing*, Oregon, U.S.A., August 1997.

[24] D. Szafron and J. Schaeffer. "An Experiment to Measure the Usability of Parallel Programming Systems". *Concurrency: Practice and Experience*, 8(2):146–166, 1996.

[25] D. Walker. "The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers". *Parallel Computing*, 20(4):657–673, 1994.

[26] Larry Wall and Randal L. Schwartz. *Programming perl*. O'Reilly & Associates Inc., 1991.