Views on Template-Based Parallel Programming

Ajit Singh,[1] Jonathan Schaeffer,[2] Duane Szafron,[2]
asingh@etude.uwaterloo.ca, jonathan@cs.ualberta.ca, duane@cs.ualberta.ca

[1] University of Waterloo,
Dept. of Electrical and Computer Eng.,
Waterloo, Ontario,
Canada N2L 3G1

[2] University of Alberta,
Dept. of Computing Science,
Edmonton, Alberta,
Canada T6G 2H1

Abstract

For almost a decade we have been working at developing and using template-based models for coarse-grained parallel computing. Our initial system, FrameWorks, was positively received but had a number of shortcomings. The Enterprise parallel programming environment evolved out of this work, and now, after several years of experience with the system, its shortcomings are becoming evident. This paper outlines our experiences in developing and using the two parallel programming systems. Many of our observations are relevant to other parallel programming systems, even though they may be based on different assumptions. Although template-base models have the potential for simplifying the complexities of parallel programming, they have yet to realize these expectations for high-performance applications.

1 Introduction

Along with the growing interest in parallel and distributed computing, there has been a corresponding increase in the development of models, tools and systems for parallel programming. Consequently, practitioners in the area are now faced with a somewhat difficult challenge: how to select parallel programming tools that will be appropriate for their applications. There is no easy answer. The decision is a function of many parameters, including some that are specific to the user and the computing environment.

As is evident from the formation of user groups such as the Parallel Tools Consortium, there is a concern in the community about the lack of post-development analysis and evaluation of the various tools and technologies that are being proposed. Typically, researchers envision a new tool or technology, develop it and, depending on their initial experiences, report it in the literature. With few exceptions, long-term experiences with parallel programming systems and their relationships with similar systems are hardly ever reported.

Many different approaches have been taken towards the development of parallel programming models. These include developing a new programming language, building parallel computing features on top of existing common sequential languages, building libraries for parallelization, and extending a sequential language with compiler directives or keywords. A relatively new alternative has begun to emerge that allows a programmer to benefit from the existing code and knowledge of a sequential program, while minimizing the modifications that are required for parallelization. The programmer provides a specification of the parallel structuring aspects of the application, in the form of code annotations. One interesting approach to code annotation is to recognize that there are commonly occurring parallel techniques. A parallel programming tool can support these techniques by providing code skeletons [14], or templates, that capture the parallel behavior. The user simply supplies the sequential application code (such as in PIE [32] and HeNCE [6]), and selects the templates to be used from the collection of templates provided by the system. The system then generates all the parallel code for the application. Template-based models separate the specification of the parallel structuring aspects — such as synchronization, communication and process-to-processor mapping — from the application code that is to be parallelized. The goal here is to provide an easy approach for the initial development and restructuring of parallel applications.

This paper discusses our long-term experiences with two template-based parallel programming systems for coarse-grained parallelism. Our research began in 1986, when we used templates[1] to experiment with different parallel structures for a computer animation application [19]. We quickly realized that the approach was more general, and could be used to build a larger class of parallel applications. Building on this success, the FrameWorks parallel programming tool was developed [34, 35, 36]. Our initial experience with FrameWorks was encouraging. However, for a number of reasons it was not possible to refine the system beyond a certain point. Consequently, an entirely new project, Enterprise, was initiated. Enterprise is a template-based parallel programming environment that offers a much wider range of related tools for parallel program design, coding, debugging and performance tuning [21, 25, 29, 30]. It has been publicly available since 1993 (http://web.cs.ualberta.ca/~enter).

Several other parallel programming systems have relied on techniques that are similar to the approach we used, such as [4, 6, 9, 10, 11, 31, 32]. Many of our results and experiences are applicable to such systems, as well as to other types of high-level parallel programming systems.

In this paper, we look at template-based parallel programming models from two viewpoints. First, as the designers, we can address the difficulties in the design and implementation of these tools. Second, we have had considerable interaction with users developing template-based parallel applications. Controlled experiments, which compared Enterprise with a number of tools including PVM, give insights into the strengths and weaknesses of the template approach. The result is that, although template-based models have tremendous potential for bridging the gap between sequential and parallel code, there still remain a number of shortcomings that must be addressed before the technology will be widely used.

Section 2 presents the motives for using the template-based approach. Section 3 outlines the objectives for a template-based parallel programming tool, and discusses how well these objectives were met in FrameWorks and Enterprise. Section 4 describes the requirements for future template-based tools. Finally, Section 5 presents our conclusions.

2 Template-based Programming

In the context of parallel programming, a template represents a prepackaged set of characteristics that can fully or partially specify the nature of scheduling, communication, synchronization and processor bindings of an entity. Templates implement various types of interactions found in parallel systems, but with the key components — the application-specific procedures — unspecified. A user provides the application-specific procedures, and the tool provides the glue to bind it all together. The templates abstract commonly occurring structures and characteristics of parallel applications, allowing users to develop parallel applications in a quickly and easily.

Consider developing a parallel application on a network of workstations. Parallel program development would require a significant amount of time and effort if a low-level tool were used (for example, Unix sockets [23] or

[1]It should be noted that we have used the term "template" (since 1986) to mean "prepackaged set of application-independent characteristics for parallel programming". This has no intended relationship with the C++ templates that are used to build generic sequential subprograms.

a message-passing library such as PVM [18]). Further, the parallelism would be explicit in the user's code, increasing the complexity of the application code. Each time the programmer wanted 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.

Template-based parallel programming systems have attempted to address this situation. These systems provide skeletons (templates) of commonly occurring parallel structures. A user simply provides sequential modules of code and selects the appropriate skeletons for structuring their parallel application. The tool 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, all the coding is sequential; all the parallel aspects are provided by the system.

In the object-oriented world, there has been a push towards cataloging commonly occurring program structures, called *design patterns* [1]. In effect, these patterns are templates. In the parallel world, popular parallel programming techniques, such as master/slave and pipelines, have been known for years, and have been the basis of a number of tools for automatically constructing the structure of programs. There are several parallel programming systems that are based on exploiting these recurrent patterns, such as [4, 6, 9, 10, 11, 31, 32]. All these systems can be viewed as template-based.

There are important differences between the template-based approach and other well-known, high-level techniques for building parallel applications. A template encapsulates certain behavior in a parallel environment. A programmer using a template is concerned only with its specified behavior. The actual implementation may vary from environment to environment depending on, among other things, the architecture and the operating system. In some ways, this is analogous to programming with abstract data types, which provide well-defined means for manipulating data structures while hiding all the underlying implementation details from the user.

Although other software engineering techniques, such as macros and code libraries, also provide high-level abstractions, the *separation* of application code and parallelization code is a key difference between templates and these techniques. For example, to use macros or library functions, the programmer must insert macro or function calls in the application code. The use of templates, on the other hand, is *non-intrusive*. The sequential code of the application need not have any reference to the templates it is attached to. This has important implications for initial program development as well as for the restructuring of parallel applications.

New programming languages are another commonly discussed technique for supporting high-level abstractions for parallel programming [2, 8]. Although the approach has some advantages, a serious disadvantage is that a programmer cannot make use of the existing code for the sequential version of the application. Some argue that parallel applications should be written from scratch. This argument is not consistent with the way complex tasks are solved. Initially, the emphasis is on finding a (sequential) solution to the task. It is only when the solution begins to take a significant amount of execution time that people start thinking about parallelizing the application. However, by this time, a large investment has been made in the writing of the sequential solution. In a template-based system, the programmer can often reuse the existing sequential legacy code.

3 A Brief Outline of the Enterprise Model

Enterprise refined and extended the template-based model used by FrameWorks. In Enterprise, an application can be viewed as a network of modules. Each module consists of a set of sequential procedures that interact with each other via remote procedure calls. A remote procedure call looks the same as a local procedure call. Furthermore, the system handles marshaling and unmarshaling of parameters. Also, a remote call that receives a reply from the called process need not block the calling process. Instead, *futures* [20] are used to delay blocking until the calling program attempts to use the reply variable. To further organize these modules into high-level parallel structures, the system provides a library of templates such as pipeline, recursive master-slave, and divide-and-conquer. The system allows composition as well as hierarchical refinement to build applications that use several different templates. In this way, the user simply defines the application graph of sequential modules and assigns templates to these modules (by using icons). The system then generates all the parallel code and allocates processes to processors. The Enterprise environment provides tools for designing, coding, debugging, performance monitoring, and tuning of a parallel application [21, 25, 29, 30].

4 Desirable Characteristics of Template-Based Models

As we gain more insight into how programmers develop parallel applications, and how different template-based systems can be built, we get a better understanding of characteristics that should be (or could be) present in a template-based system. In this section, we outline what we feel are the important characteristics of the ideal template-based model. No tool presently exists that supports all of these features. The list is used in this paper to serve as a benchmark for analyzing FrameWorks, Enterprise and other systems, and as a specification for future systems.

4.1 Structuring the Parallelism

Template-based systems should allow the minimum possible restrictions on how the user can structure the parallelism in their application. This includes having properties such as:

1. **Separation of Specification (*Separation*):** This is the central feature of a template-based system. It means that it should be possible to specify the templates (that is, the parallelization aspects of the application) separately from the application code. This characteristic is crucial for rapid prototyping and performance tuning of a parallel application. It also allows for the application code and its parallelization structures evolve in a relatively independent manner.
2. **Hierarchical Resolution of Parallelism (*Hierarchy*):** This allows the refinement of a component in a parallel application graph by expanding it using the same model. That is, templates can include other templates. Therefore, there is no need to have separate models for "programming-in-the-large" and "programming-in-the-small".
3. **Reuse via Composition (*Reuse*):** It is not sufficient to define some templates that can be used in other templates. The meanings of all templates should be context-insensitive so that they can be used in other templates.

The significance of separating sequential application program components from the ways in which these components interact has long been recognized. In early systems, component interaction was specified in separate text files [12]. The advent of workstation technology and graphical user interfaces (GUI) greatly enhanced the ease, efficiency and effectiveness of specifying parallel structures [6, 10, 24]. Many of the systems that employ a separation of specifications and code are based on the data-flow model. Example systems are CODE [10], DGL [22], LGDF [15] and Paralex [3]. Some of these models also provide hierarchical resolution of parallelism [10, 24]; others don't [3, 15, 22].

Several models based on control-flow that address the *separation* objective have emerged. Example systems include CAPER [38], PIE [32], and Parallel Utilities Library (PUL) [13].

However, separate specification-based parallel computation models are also not limited to procedural programming languages. For example, Cole's algorithmic skeletons [14] and P3L [5] are designed using the functional programming model. Similarly, Strand uses logic programming to design its templates [16].

Both FrameWorks and Enterprise attempted to achieve *separation* by separating the application-specific sequential programming (*programming model*) from the specification of the parallelism (*meta-programming model*). Both tools allowed the user to express the parallelism graphically, and to annotate the resulting graph with sequential procedures. However, it is often not possible to achieve perfect independence; there are still inherent dependencies between the two components. For example, pointers and global variables present in a sequential program can cause problems when restructuring the program for distributed-memory-based systems.

The above points illustrate that there are flaws in the Enterprise model. Similar weaknesses exist in other template-based models. The ideal orthogonal relationship between sequential code and parallel specifications is hard to achieve since the needs of the programming model and those of the meta-programming model are sometimes conflicting.

4.2 Templates

To be useful, template-based systems must provide a powerful set of building blocks for constructing parallel applications. Some of the desirable properties include:

1. **Mutually Independent Templates** (*Independence*)**:** This relates to reuse via composition. It should be possible to combine various templates with few or no exceptions.

2. **Extendible Repertoire of Templates** (*Extendible*)**:** It should be possible to integrate more templates into the library of templates.

3. **Large Collection of Useful Templates** (*Utility*)**:** The system should be useful over a wide range of applications.

4. **Open Systems** (*Open*)**:** It should be possible for the programmer to use templates, or a lower-level mechanism, such as message passing, for developing an application. The absence of such a feature results in a closed system in which the only applications that can be developed are those whose required parallel structures match the templates. This is a very difficult requirement as it has significant implications for application development, debugging and tuning.

FrameWorks and Enterprise provided a small set of templates. These include (using the Enterprise terminology): *lines* (pipelines), *departments* (one process distributing work to a heterogeneous collection of workers), *divisions* (recursive divide-and-conquer), and *services* (resource processes that are accessible to all). The components of these structures can be replicated, with automatic distribution of parallel work to the next available process. Enterprise extended FrameWorks to allow these templates to be hierarchically combined, allowing the user to create complex parallel structures quite quickly. Although these templates are sufficient to build a number of interesting parallel applications, many important real-world problems are more amenable to parallel structures not directly supported in FrameWorks or Enterprise.

Neither FrameWorks nor Enterprise (nor any other template-based model) allow users to create their own templates. The lack of extendibility forces users either to use a possibly inappropriate parallel structure, or to abandon the tool altogether.

4.3 Programming

Templates may impose constraints on how users write sequential code.

1. **Program Correctness** (*Correctness*)**:** The system should offer some guaranteed

properties of correctness. For example, deadlock-free, deterministic execution and fault-tolerance are some desirable features.

2. **Programming Language (*Language*):** The system should build on an existing commonly-used language. Ideally, there should be no changes to the syntax or the semantics of the language. In addition to facilitating reuse of existing sequential code, this feature also makes it possible to take advantage of existing expertise in sequential programming.

3. **Language Non-Intrusiveness (*Non-Intrusiveness*):** A system may satisfy the *language* objective, but force the user to change sequential code to accommodate limitations in the parallel programming model. For example, to develop a parallel application using a message-passing library, the user may have to appropriately restructure the code and insert calls to the message-passing library in the code. The only way to eliminate this problem properly and satisfy the *language* constraint is to have a compiler that automatically parallelizes the code. Unfortunately, for coarse-grained applications, the required compiler technology does not yet exist.

FrameWorks extended the C programming language to include new keywords to allow communication and synchronization among processes. Enterprise used compiler support to do it automatically using *futures*. Consider a call from a module A() to a module B():

```
Result = B( Param1, Param2, ..., ParamN );
/* some other code */
Value = Result + 1;
```

The sequential semantics of such a call is that A() calls B(), passing it N parameters, and then blocks waiting for the return value(s) from B() before resuming execution. Enterprise preserves the effects of the sequential semantics but allows A() and B() to execute concurrently. When A() calls B(), the parameters to B() are packaged into a message (marshaled) and sent to the process that executes B(). After calling B(), A() continues with its execution until it tries to access Result to calculate Value. If B() has yet not completed execution, then A() blocks until B() returns the Result. These so-called futures significantly increase the concurrency without requiring any additional specification from the user.

Although the idea of futures is attractive, it causes some subtle changes to the semantics of the programming language. For example, to increase the parallelism in the application, the user may need to make additional calls to parallel functions, possibly resulting in code that looks inefficient if executed sequentially. As well, the user needs to understand the blocking semantics of futures, so that sufficient computational work can be done between creating and accessing the future. Again, this runs counter to familiar sequential programming.

Since Enterprise and FrameWorks assume a distributed memory environment, data structures containing pointers cause problems. When a user passes a pointer to the invocation of a parallel procedure, how much data should be passed? In the sequential world, this is not an issue; in the parallel world it is an important performance issue. FrameWorks requires the user to write additional code to package all the parameters to a parallel function into a single structure to be passed. Enterprise uses compiler support to automatically package most parameters, but requires all pointers to include an additional size argument. Again, this is a significant departure from the familiar sequential model.

Unfortunately, by forcing as much of the C semantics as possible on the Enterprise code, the system gives up correctness. For example, it is possible to alias a memory location containing a future. Any access to a future should cause the appropriate future semantics. However, aliases may not be properly detected by the compiler, creating an incorrect program. In general, it is impossible to solve the alias problem in C without sacrificing something (such as efficiency, adding new keywords, or restricting feature usage).

4.4 User Satisfaction

The system must satisfy a number of performance constraints, both at program development time and at run time. These include:

1. **Execution Performance (*Performance*):** The maximum performance possible, subject to the combination of templates chosen by the user, should be achievable. There will always be limitations to the achievable performance. The complexity and interdependence of components external to the system (communication subsystem, operating system, network, and so on) make it very difficult to abstract and still attain the highest possible performance. Often, a solution generated by a high-level tool may not achieve the same performance as a solution handcrafted by an expert. The tradeoff is better software engineering and shorter development time in exchange for possibly slower execution performance.

2. **Application Portability (*Portability*):** The tool should allow the user to port an application to a number of different architectures. Some performance losses may be expected for a poorly-chosen architecture, but the program should still run.

3. **Support Tools (*Support*):** The system should provide a complete set of design, coding, debugging and monitoring tools that support the template-based model. These tools must support the same level of abstraction as does the programming model.

4. **Tool Usability (*Usability*):** The ideal tool should have a high degree of usability. It should be easy to learn and easy to use. Usability assessments have been neglected in the literature [40].

Enterprise has a simple interface that allows it to use a variety of communication packages such as PVM, ISIS and NMP. Enterprise can be viewed as a software layer on top of, for example, PVM. The question arises as to what the user gains and loses by moving to a higher level of abstraction.

There are two main goals of the Enterprise system: to create a high-level programming environment that is easy to use, and to promote *code reuse* by encapsulating parallel programming code into templates. For example, Enterprise's model allows the user to achieve *separation* of specification. The use of a pre-compiler allows the Enterprise system to automatically insert communication, parameter packing and synchronization code into the user's application. In contrast with PVM, for example, the user must explicitly address these issues by inserting PVM library calls into the code (thereby violating the *non-intrusiveness* objective). It is the user's responsibility to structure the code so that a compiler flag can be used to include or exclude the parallel code.

Enterprise offers the user additional benefits. For example, the model allows for the *hierarchical* use of the templates, guaranteeing a deadlock-free application. Also, the user has the assurance that the generated code for the specified structures is correct. Both points contribute to the *correctness* objective.

In moving to a higher-level model such as Enterprise, the user has lost something. Most noticeable is the possible decrease in *performance*. Message-passing libraries, such as PVM, allow much more flexibility; users can easily tune their systems to maximize performance. Furthermore, PVM has a large support infrastructure that has resulted in the system being made available on most major platforms (excellent *portability*).

The choice between PVM and a higher-level tool is not easy. The choice can be simplified to a tradeoff between execution performance and software engineering. High-level parallel programming tools have the potential to enable users to build parallel applications more quickly and reliably. In return, they may have to accept slightly worse performance.

The metric most often used and abused in the parallel computing literature is program execution speedup. However, with the availability of relatively inexpensive multiprocessor machines and the widespread use of networked single-processor workstations, more and more people are turning towards parallel computing. For such users, a shorter learning curve, ease of program design, development and debugging are just as important as speedup. A tool that quickly achieves a performance improvement, even if it stops short of achieving the peak performance, may be quite acceptable.

Two controlled experiments were conducted to assess the usability of Enterprise system. These experiments compared the usability of Enterprise with two communication libraries (PVM and NMP [26]) and one other high-level parallel programming systems [39, 40]. Some of the conclusions from the experiments included the following:

1. Users were able to complete all the programming tasks using all the systems under examination.

2. Users wrote significantly less code (66 percent) with Enterprise, compared with using message-passing libraries.

3. The PVM and NMP solutions each had better performance (about 25 percent faster).

4. Users were able to develop their first prototype quickly using Enterprise. However, they found it difficult to tune their solutions for better performance.

5. The fact that users were constrained to developing their entire solution using templates of Enterprise was considered a significant weakness as far as tuning the solution was concerned.

Although template-based systems showed a lot of promise in the experiments, users found that writing PVM code, although cumbersome, was straightforward after a bit of practice. They found that tools like PVM provided them with complete control over the application's parallelism so they could achieve maximal performance.

5 A Next-Generation Tool

Templates represent a powerful abstraction mechanism. We believe templates have the potential to make as strong an impact on the art of parallel programming as macros and code libraries have. However, from our experiences with FrameWorks and Enterprise, we have learned a number of lessons that must be remembered while developing new template-based tools:

1. *Open Systems*: Enterprise provides a high-level parallel programming model that the user *must* use. There are no facilities allowing the user to step back from the model to access lower-level primitives to achieve better performance, or to accommodate an application for which a suitable template is not available. For example, even though Enterprise generates PVM code, this code is hidden from the user. There is no easy way to use Enterprise to generate a correct PVM program, and then to incrementally tune this program to achieve better performance. A high-level template-based tool must allow the user the possibility of accessing lower-level primitives. Also, it should be possible to develop an application partially with the use of templates and partially by using low-level communication primitives [33].

2. *Extendibility*: FrameWorks and Enterprise support a fixed number of templates. There is no easy way for the programmers to add templates to the system. An important step towards enhancing the *utility* of a template-based model would be to design a system that provides a standard interface for attaching templates to the user code. In such a system, it may be possible for the user to develop new templates. As long as the templates are mutually independent, it should be possible to integrate them into the rest of the system. This would result in a system that is *extendible* and can support a large number of templates [37].

3. *Portability*: It is imperative to continue building on top of existing, established

technology. Some *de facto* standards seem to be emerging. For example, PVM (and possibly MPI soon) is currently adequate as the lowest-level building block. PICL seems to be a popular choice for parallel program instrumentation [17]. Given the significant effort required to build a parallel programming system, it seems foolhardy to continue to invent, when one can reuse.

4. *Language*: Many parallel programming tools make subtle changes to the semantics of an existing sequential language. We believe this is a mistake. Changing a programming language's semantics can increase the user's learning curve and result in difficulties in understanding and debugging parallel code.

5. *Importance of Compiler Technology*: Our research would greatly benefit from better compiler technology. Following are some reasons:

   (a) Some of the semantic confusion in Enterprise could be eliminated.

   (b) Static analysis of the code can do a better automatic job of code reorganization to improve concurrency and delay synchronization, thereby improving performance.

   (c) Compilers can uncover data dependencies, possibly uncovering programming errors at compile time rather than at run time.

   (d) Flow control analysis can identify communication patterns that can assist in the initial process-processor mapping. (Orca, for example, uses compile-time analysis to help distribute the data [7].)

6. *Utility*: There are commonly occurring patterns in other areas of parallel programming such as parallel I/O, shared memory access, data distribution and alignment. Work is in progress to provide templates to allow code reuse in some of these areas [28, 27].

7. *Tradeoffs*: Should we build a tool for the inexperienced user or the experienced user? For example, it is conceivable to build an *open* and *extendible* system such as the one outlined in item 1 and 2 above. However, in such a system it may no longer be possible to give the correctness guarantees that Enterprise offers. The requirements of users vary with their skill and experience levels. For the former, simplicity of the model and ease of use are the most important considerations. For the latter, performance is often the only metric that matters.

6 Conclusions

Who are the potential users of parallel computing technology? There will always be a user community that uses parallel computing to squeeze every last nanosecond of performance out of a machine. We believe this group to be a very small percentage of the potential user community. Local area networks of workstations are commonplace and the popularity of low-cost multiprocessor shared-memory machines is rapidly growing. However, few people take advantage of the parallelism in these architectures. Many people want their programs to run faster but are unwilling to invest the time necessary to achieve this.

For most users, sequential program improvement stops at the compiler level. Ideally, the same should be true for coarse-grained parallel program development (such as is seen with vectorizing compilers). Given that compilation techniques are still in their infancy for coarse-grained applications, the next logical step is to provide a tool that allows users to parallelize their application with minimal effort. Template-based models offer real prospects of making this a reality.

Rather than putting forward yet another model for building parallel applications, this paper was aimed at consolidating an existing approach to parallel programming. Usability experiments of Enterprise have added a new dimension to our understanding of how programmers with little or no experience in parallel computing build their parallel applications. We hope our experience in developing two such models into working systems as well as results of our experiments in estimating the usability of parallel programming systems would be useful to researchers and practitioners in this area.

We have identified several areas where effort is necessary to enhance the usability of the template-based systems. Work on several of these issues is in progress [27, 28, 33, 37]. Template-based techniques alone may not be enough to provide an easy-to-use, high-level parallel programming system that supports code reuse and quick prototyping and restructuring of parallel applications. However, we believe that template-based techniques would play a significant role in building the ideal parallel programming systems of the future.

Acknowledgments

The constructive comments from Ian Parsons, Greg Wilson, and Stephen Siu are appreciated. This research was conducted using grants from the Natural Sciences and Engineering Research Council of Canada (OGP8173 and OGP0155467) and IBM Canada Ltd.

About the Authors

Ajit Singh is an Assistant Professor of Electrical and Computer Engineering at the University of Waterloo. His research interests include parallel and distributed computing, and database systems.

Jonathan Schaeffer is a Professor of Computing Science at the University of Alberta. His research interests include parallel programming systems and artificial intelligence.

Duane Szafron is an Associate Professor of Computing Science at the University of Alberta. His research interests include object-oriented computing, programming environments and user interfaces. He received a Ph.D. from the University of Waterloo and a B.Sc. and M.Sc. from the University of Regina. His Internet address is duane@cs.ualberta.ca.

References

[1] E. Gamma anad R. Helm, R. Johnson, and J. Vlissides. "Design Patterns: Abstraction and Reuse of Object-Oriented Design". *Addison-Wesley*, 1995.

[2] G. Andrews, R.A. Olsson, M.A. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. "An Overview of the SR Language and Implementation". *ACM Trans. on Prog. Languages and Systems*, 10(1):51–86, 1988.

[3] O. Babaoglu, L. Alvisi, A. Amoroso, and R. Davoli. "Paralex: An Environment for Parallel Programming in Distributed Systems". *Technical Report UB-LCS-91-01, Department of Mathematics, University of Bologna, Italy*, 1991.

[4] 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.

[5] B. Bacci, M. Danelutto, and S. Pelagatti. "Resource Optimization via Structured Parallel Programming". In *Programming Environments for Massively Parallel Distributed Systems*, pages 13–26, Birkhauser Verlag, Basel, Switzerland, 1994.

[6] 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.

[7] H. Bal and M. Kaashoek. "Object Distribution in Orca using Compile-Time and Run-Time Techniques". In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 162–177, 1993.

[8] H. Bal, M. Kaashoek, and A. Tannenbaum. "Orca: A Language for Parallel Programming of Distributed Systems". *IEEE Transactions on Software Engineering*, 18(3):190–205, 1992.

[9] 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.

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

[11] 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.

[12] J.C. Browne, A. Tripathi, S. Fedak, A. Adiga, and R. Kapur. "A Language for Specification and Programming of Reconfigurable Parallel Structures". In *International Conference on Parallel Processing*, pages 142–149, 1982.

[13] L. Clarke, R. Fletcher, S. Trevin, R. Bruce, and S. Chapple. "Reuse, Portability and Parallel Libraries". In *Programming Environments for Massively Parallel Distributed Systems*, pages 171–182, Birkhauser Verlag, Basel, Switzerland, 1994.

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

[15] D.C. DiNucci and R.G. Babb II. "LGDF Parallel Programming Model". In *IEEE COMPCON*, pages 102–107, 1989.

[16] I. Foster and S. Taylor. "Strand: A Practical Parallel Programming Tool". In *North American Conference on Logic Programming*, Cambridge, Mass., 1989. MIT Press.

[17] G. Geist, M. Heath, B. Peyton, and P. Worley. "PICL: A Portable Instrumented Communication Library". *Technical report ORNL/TM-11130, Mathematical Sciences Section, Oak Ridge National Laboratory*, 1990.

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

[19] M. Green and J. Schaeffer. "Frameworks: A Distributed Computer Animation System". In *Canadian Information Processing Society, Edmonton*, pages 305–310, 1987.

[20] A.R. Halstead. "MultiLisp: A Language for Concurrent Symbolic Computation". *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[21] P. Iglinski, S. MacDonald, D. Novillo, I. Parsons, J. Schaeffer, D. Szafron, and D. Woloschuk. "Enterprise User Manual, Version 2.4". *Department of Computing Science Technical Report*, No. 95-02, 1995.

[22] 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.

[23] 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.

[24] T.G. Lewis and Rudd W.G. "Architecture of the Parallel Programming Support Environment". In *IEEE COMPCON*, pages 589–594, 1990.

[25] G. Lobe, D. Szafron, and J. Schaeffer. "The Enterprise User Interface". In *TOOLS 11 (Technology of Object-Oriented Languages and Systems)*, pages 215–229, 1994.

[26] T. Marsland, T. Breitkreutz, and S. Sutphen. "A Network Multiprocessor for Experiments in Parallelism". *Concurrency: Practice and Experience*, 3(1):203–219, 1991.

[27] D. Novillo. "High-level Representations for Distributed Shared Memory". *Department of Computing Science, University of Alberta*, 1995. Internal report.

[28] I. Parsons. "Templates for Parallel I/O". *Department of Computing Science, University of Alberta*, 1995. Internal report.

[29] J. Schaeffer and D. Szafron. "Software Engineering Considerations in the Construction of Parallel Programs". In *High Performance Computing: Technology and Applications*, pages 271–289, 1996.

[30] 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.

[31] L. Schafers, C. Scheidler, and O. Kamer-Fuhrmann. "TRAPPER: A Graphical Programming Environment for Industrial High-Performance Applications". In *Parallel Architectures and Languages Europe*, pages 403–413, 1993.

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

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

[34] A. Singh, J. Schaeffer, and M. Green. "Structuring Distributed Algorithms in a Workstation Environment: The FrameWorks Approach". In *International Conference on Parallel Processing*, volume II, pages 89–97, 1989.

[35] 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.

[36] 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.

[37] S. Siu. "An Extendible Template-Based System for Parallel Programming". *Electrical and Computer Engineering Dept., University of Waterloo*, 1995. Internal report.

[38] B. Sugla, J. Edmark, and B. Robinson. "An Introduction to the CAPER Application Programming Environment". In *International Conference on Parallel Processing*, pages 107–111, Illinois, U.S.A., August 1989.

[39] D. Szafron and J. Schaeffer. "Experimentally Assessing the Usability of Parallel Programming Systems". In *Programming Environments for Massively Parallel Distributed Systems*, pages 195–201, Birkhauser Verlag, Basel, Switzerland, 1994.

[40] D. Szafron and J. Schaeffer. "An Experiment to Measure the Usability of Parallel Programming Systems". *Concurrency: Practice and Experience*, 8(2):147–166, 1996.