Enterprise: Current Status and Future Directions

D. Szafron, J. Schaeffer, P. Iglinski,
I. Parsons, R. Kornelsen, C. Morrow

Department of Computing Science,
University of Alberta,
Edmonton, Alberta,
CANADA T6G 2H1

Abstract

Software development costs for parallel programs can be considerably higher than for sequential software. There are a variety of reasons for this, but two of the major ones are the programming model and the execution environment. Most parallel programming models are very different from sequential ones, so there is strong resistance to change. In most parallel programming systems, the user must be aware of the target hardware configuration and must tailor programs to match the execution environment. Ideally, programming a parallel application should be no more difficult than programming a sequential one.

Enterprise is a programming environment for designing, coding, debugging, testing, monitoring, profiling and executing parallel programs in a distributed hardware environment. The two major goals of the Enterprise project are to provide an easy-to-use, familiar programming model for writing parallel applications (the user's interface to the system) and to remove considerations of the target hardware from the user (the system's interface to the hardware). This paper discusses the current status and the future directions of the project. Features of particular note are sequential C for programming; an analogical model for representing the parallelism; transparent access to a variety of underlying communications subsystems; and visual tools for supporting program design, program animation, forced replay and message level debugging.

By providing a familiar programming model, utilizing a uniform graphical interface and abstracting away all considerations of the hardware, Enterprise can be used to reduce the costs of parallel software development.

The IBM contact for this paper is Jacob Slonim, Centre for Advanced Studies, IBM Canada Ltd., Dept. 21/894 844 Don Mills Road, North York, Ontario M3C 1V7.

1 Introduction

Writing programs to exploit concurrency in parallel and distributed architectures adds an extra dimension of complexity to software development. The programmer must deal with additional issues while writing code (such as communication, synchronization and process-processor mapping) and at run-time (such as load balancing and fault tolerance). Ideally, with the proper parallel programming system, many of these considerations can be abstracted from the user's view. Unfortunately, despite a decade of active research into tools for parallel and distributed program development, there still exists a large gap between what has been developed in the research laboratory and what is used in practice.

Researchers have taken several diverse approaches to extending the ideas of sequential programming to parallel programming:

1) Design a new language that either explicitly or implicitly supports parallel programming constructs (for example, Orca [1]). However, user communities are reluctant to change languages, given that they have a substantial investment in legacy software.

2) Extend an existing programming language with new programming constructs (for example, High Performance Fortran [6]). This approach is popular with the scientific community since the myriad of new features added to Fortran allows users to modify their code to maximize performance. Unfortunately, if an application does not use vectors, HPF has little to offer.

3) Provide a library of routines to augment a programming language (for example, PVM [3]). This approach is widely used, but it forces the programmer to deal with low-level issues. Users must write a substantial number of complex code fragments that

contain explicit messages, increasing the probability of programming errors.

4) Provide annotations for an existing programming language that extend its semantics to support concurrent execution of program components. The advantage is that parallel programming looks like sequential programming; the disadvantage is that the absence of user control over low-level details might restrict the level of performance.

While each of the above approaches might serve a particular community, the question is which approach offers the best chance of gaining wide-spread acceptance.

This paper describes the current status and future directions of Enterprise, a parallel programming system for developing applications that run concurrently on a distributed network of computers [10]. Enterprise embraces the fourth approach, enhancing the semantics of the C programming language to allow subroutines to be executed concurrently. Enterprise is publicly available (the *ftp* instructions are given at the end of the paper) and is currently used in Canada, the United States and Europe.

The Enterprise project has adopted the following design goals:

1) Support a commonly used sequential programming language (C in our case) with no syntactic extensions to the language or library calls to make.
2) Provide a programming environment that can be used for the complete software engineering life-cycle, including design, coding, compiling, executing, debugging, performance monitoring and performance tuning.
3) Provide a uniform view in accessing all tools in the environment. This reduces the learning curve for users and the time for context switches between tools.
4) Abstract as much of the run-time environment as possible from the user, including hardware, operating system, compiler and file system.

Clearly, these are ambitious goals. However, we feel they must be achieved if parallel/distributed computing is to move from the research laboratory to commercial use.

This paper discusses the current accomplishments and the future directions of the Enterprise project in two areas: the programming interface and the communications interface. The programming interface provides a uniform graphical interface to all tools in the system. Of particular note are the use of sequential C for programming, an analogical model used to represent the parallelism and the visual tools for supporting program design, program animation, forced replay and message level debugging. The communications interface provides transparent access to a variety of underlying communications subsystems (PVM or MPI [12] for example) so the user is shielded from most considerations of the hardware used. In effect, the computing environment may be treated as a black box by the user.

The combination of these two features allows users to write distributed programs without providing any explicit code for parallelism. The users concentrate their efforts on the important issue, identifying the potential parallelism in their applications, without being sidetracked by unnecessary details.

2 An Overview of Enterprise

In Enterprise, the interactions of processes in a parallel computation are described using an analogy based on the parallelism inherent in a business organization. Since business enterprises coordinate many asynchronous individuals and groups, the analogy is beneficial to understanding and reducing the complexity of parallel programs. Inconsistent parallel terminology (such as master-slave, pipelines or divide-and-conquer) is replaced with more familiar business terms (*assets* called *individuals*, *lines*, *departments*, *divisions* and *services*). Every sequential procedure that will execute concurrently is assigned an asset type that determines its parallel behavior. The user code for each of these procedures is sequential C, but a procedure call to such an asset is automatically translated into a message send by Enterprise.

Consider the following user C code, assuming that `func` is an asset in the program:

```
result = func( x, y );
/* other C code */
a = result;
```

When Enterprise translates this code to run on a network of workstations, the parameters `x` and `y` are packed into a message and sent to the process that executes the asset `func`. The caller continues executing and only waits for the function result when it accesses the result `(a =`

`result)`. Results that allow such concurrent execution are called *future*s [4].

Enterprise has three components: an object-oriented graphical interface, a pre-compiler and a run-time executive. The user specifies the application parallelism by drawing a hierarchical enterprise that consists of assets. At run-time, each asset corresponds to one or more processes. Sequential procedure calls in C are translated into message send/receives across a network by the pre-compiler. The execution of the program (process/processor assignment, establishing communication links and monitoring network load) is done by the run-time executive.

For example, consider a Simulation program that displays a group of fish swimming across a display screen (this problem was contributed by a research group in our Department and is more complex than portrayed by the following description). The main procedure, Model, consists of a loop that, for each frame in the simulation, performs some work on the frame and calls PolyConv. PolyConv manipulates the image received from Model and calls Split. Split polishes the frame and writes it to disk. There will be three assets: Model, PolyConv and Split.

The user could enter all of the code for Model, PolyConv and Split into one individual asset and run the program sequentially. However, there is no reason why Model should wait until PolyConv completes the first simulation frame to start processing the second frame. Similarly, PolyConv does not need to wait for Split. In the parallel processing community this type of parallelism is often called a pipeline. Using the Enterprise analogy, these three routines act like an assembly or production line and are represented by a line. Therefore, the user can transform the single individual into a line containing the components Model, PolyConv and Split.

One of the strengths of the Enterprise model is that it is easy to experiment with alternate parallelization techniques without changing C source code. Each asset represents at least one process. If a call is made to the individual Split, it is executed by a process; if a subsequent call is made to Split before the first call is complete, the second call must wait for the first call to finish. However, if the Split asset is replicated, multiple processes can be used to execute calls concurrently. When PolyConv calls Split, a process is activated, and if a subsequent call is made to Split before the first call is done, then a second process is activated if there is an available machine. Replication can be dynamic in Enterprise so that all available processors on the network can be used, subject to a lower and upper bound supplied by the user.

Figure 1 shows the structure of the Simulation program. The double line rectangle represents the enterprise. The dashed-line rectangle represents the line asset and each inner icon represents a component. The first component is a receptionist that shares the name, Model, with the line that contains it. All calls to a line are received by the receptionist. The other two components are subordinate individuals. The last individual in the line, Split, is replicated with one to five replicas.

When a user compiles a program, the Enterprise pre-compiler automatically inserts code to handle the distributed computation and compiles the program. When a user executes a program, the Enterprise run-time executive allocates the necessary number of processors to start the program, initiates processes on the processors and dynamically allocates work to processes.

3 Current Status

In this section the current state of the publicly available version of Enterprise is described.

3.1 Programming Model

The *gcc* compiler has been modified to support the Enterprise programming model. No syntax changes to the C programming language have been made and the user is not required to make any library calls. The compiler supports futures for both simple and complex variable types, including pointers.

3.2 Meta-Programming Model

As described in Section 2, Enterprise uses the analogy of a business organization to represent different parallelization techniques by assets. The user expresses the parallelism in an application by selecting an appropriate decomposition of assets. At any time, an asset can be replicated so that multiple identical copies are created. In this way, when a replicated asset is called, the first replica executes the call and if a second call is made to the replicated asset before the first call is completed, a second replica can immediately execute the second call. Currently, five asset types are supported:

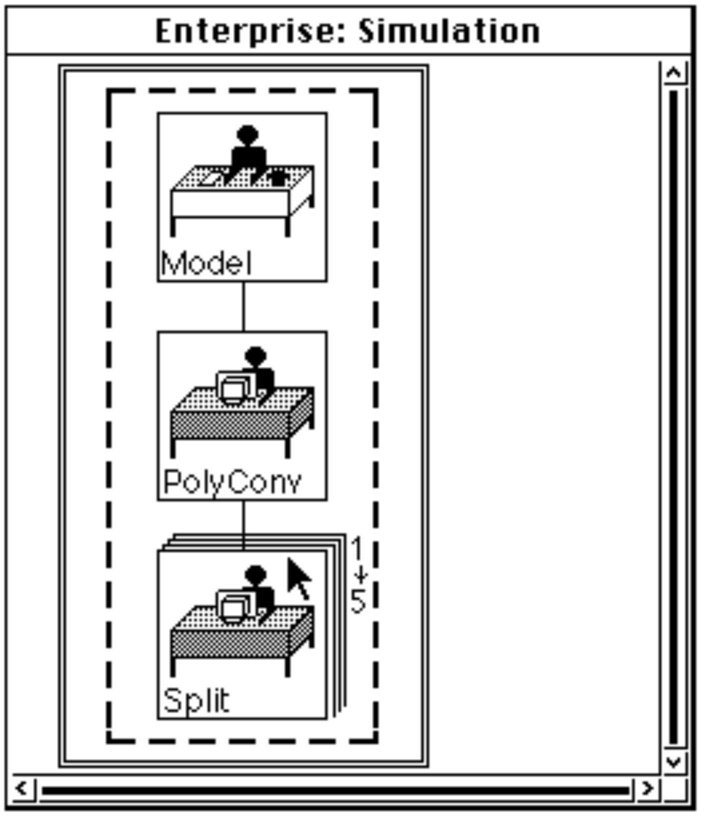

Figure 1: A program in the Enterprise programming environment.

- An individual asset is analogous to a person in an organization. When an individual asset is called, it executes its sequential code to completion.
- A line asset is analogous to an assembly line. It contains a fixed number of heterogeneous assets in a specified order. Each component asset contains a call to the next asset in the line.
- A department asset is analogous to a department in an organization where a receptionist is responsible for directing all incoming communications to the appropriate place. It contains a fixed number of heterogeneous assets.
- A division asset contains a hierarchical collection of identical assets where work is divided and distributed at each level. It can be used to parallelize divide-and-conquer computations. Divisions are the only recursive assets in Enterprise.
- A service is analogous to any asset in an organization that is not consumed by use and whose order of use is not significant. For example, a clock on the wall can be considered a service. A service can be called by any other asset except another service.

3.3 Programming Interface

The design view of the user interface is used to edit asset diagrams that specify the parallel structure of the program as shown in Figure 1 [5]. The user interface supports all asset types described in the meta-programming model. Program code can be edited by opening editor windows on the code of any assets. Enterprise programs are compiled and run from within the Enterprise environment simply by pressing buttons. An automatic make facility is used for compilation and the user has the option of generating event logs when programs are run. Event logs can be used to animate programs so that users can view the status of assets (idle, busy, blocked or dead), watch messages travel between assets and inspect message contents. Program animation is useful in the testing, monitoring and performance tuning of programs. Event logs can also be used to force re-execution of a program in the same event order to debug parallel programs. The debugger also allows users to set message level breakpoints during replay. These breakpoints can be based on the identity of sending or receiving assets or the values of logged parameters when messages are sent or received.

3.4 Communication Interface

The communication interface supports clusters of homogeneous machines running UNIX. Three architectures are currently supported (IBM, Sun and SGI workstations) with two more available soon (HP and DEC). Enterprise can generate code to interface with several communication kernels, including full support for NMP [7] and PVM and partial support for DCE [9] and ISIS [2].

4 Research Directions

Enterprise is being used as a test-bed for research in a number of areas in distributed computing and programming environments. This section summarizes some of the major efforts.

4.1 Programming Model

Currently two extensions to the programming model are being investigated. First, some of the restrictions on futures can be relaxed. For example, futures currently cannot be passed to other routines. A future used as a function parameter is treated as an access of that variable, implying that program execution stops until the future has returned. More concurrency in an application is possible if blocking on futures can be delayed as long as possible.

Second, using pointers to alias variables is a problem in parallel as well as sequential code. In the sequential world, the presence of pointers can inhibit some types of code optimization. However, in Enterprise, the use of pointers to alias futures is currently not detected by the compiler, possibly resulting in run-time errors. Solving this is a difficult problem, since the locations referenced by a pointer cannot be statically detected at compile time. A solution is to have all pointers checked at run-time, either through software (expensive) or preferably with hardware assistance (currently not supported by most operating systems).

4.2 Meta-Programming Model

Enterprise assets attempt to create an abstraction by which high-level process interactions can be easily specified. The current suite of assets is suitable for a wide range of applications. Using Enterprise service assets can model any communication structure, incurring only the overhead of the intermediate service processes. However, several high-level meta-programming structures occur often enough that new assets should be created to handle them.

The most important commonly occurring communication structure not supported by an Enterprise asset is peer-to-peer communication. Consider a problem such as parallel sorting. N processes each sort $1/N$th of the data. Each process then wants to send part of its sorted subset of the data to the other N-1 processes. In Enterprise, asset calls look like sequential subroutine calls, but in reality are message sends to concurrent processes. In the parallel sorting example we need to send the data to a particular instance of a collection of identical assets. One way to specify this is to add an extra parameter to each asset call that specifies the receiver. Another is to create an array of assets and then index into the array to identify the receiver. Clearly both solutions are workable without changing the syntax or semantics of the C programming language. The two disadvantages of these schemes are that the programmer must take a more active role than in other Enterprise assets and they introduce the possibility of deadlock.

We are also investigating the use of size-independent reference schemes. For example, references like up, down, left and right in mesh assets. In this way, the size of a mesh does not have to be determined at compile time. Instead, the mesh can be created when a computation is started, based on the available number of processors, so that user code can be independent of the size.

Interesting work has already begun on preprocessing existing C programs (legacy software) to identify the data and communication structure with the goal of automatically parallelizing the application or offering hints to the user on how this might be achieved [8]. Although automatic parallelization is an ambitious project, recent research has shown that the simplicity of the Enterprise model allows automatic parallelization of some applications.

4.3 The Programming Interface

Other than support for new asset types, few changes are required to the user interface for supporting design, coding, compiling or testing. In the area of monitoring, the animation view is being augmented to supply more performance information in the form of gauges, dials and graphs. In addition, we are studying the use of sound to represent the changing state of assets and their utilization.

However, the biggest problem in monitoring is neither a lack of information nor a mechanism to report the information. The real problem is how to filter and display selective information. Parallel programming systems should support user queries at a higher level of abstraction. For example, the user should be able to say: "show me a situation in which a single asset is executed for a long time". Enterprise should respond by searching through an event log and displaying 10 seconds of an animation in which only one asset is busy and all others are idle or blocked and indicating where in the user's code this situation occurs. Similarly the user should be able to say: "show me a situation where I have specified a replication factor that is too large so that some replicated assets are idle" or "show me a situation where I have specified a replication factor that is too small so that there is a large input queue for a replicated asset". In our first attempt to support such filtering the user will be given a menu which contains entries such as sequential execution, replication too large and replication too small.

4.4 Communications Interface

Support for heterogeneous computing environments is currently being developed. The problems of message passing between heterogeneous processors are well understood and have been dealt with by others. The current work is on developing a mechanism for automatic decomposition of arbitrary data structures into flat byte streams (and their recovery) that is independent of the communication system used. This is particularly important in languages like C that support untyped pointers.

The Enterprise compiler has been modified to allow any asset to be executed either sequentially or in parallel. This adds an interesting dimension to optimizing the run-time performance of a program, since Enterprise can dynamically monitor program execution and toggle processes between parallel and sequential modes as conditions change.

I/O in a heterogeneous computing environment is difficult to achieve in a transparent manner. Two of the major problems are concurrent read/write access to a shared file and data file formats (different vendors use different formats for data). Research has begun on creating a virtual file system for an Enterprise program that supports both parallel and sequential I/O semantics.

5 Conclusions

It took many years before the computing community finally moved from low-level assembly languages to high-level compiled languages; users were unwilling to sacrifice the performance gains possible in assembler for the reduced software design costs of using compiled languages. These higher level tools abstracted out the hardware considerations, making it possible to write machine-independent programs. With parallel/distributed computing, a similar scenario is occurring. Users are unwilling to give up the performance they can achieve using access to low-level primitives in return for the reduced software design costs that are possible using high-level development tools. These higher level tools abstract out many of the hardware considerations, making it possible to build portable parallel programs. Two considerations will cause the user community to evolve towards the higher level tools: retirement of the senior members of the work force and the continued reduction in the cost of parallel/distributed computing hardware reducing the need for high performance.

Enterprise attempts to provide the user with a familiar sequential programming model that hides most of the hardware details. Experiments have shown that this programming model increases programmer productivity, though possibly at the cost of reduced run-time performance [11]. Given the high cost of sequential software development, and the increased complexity of parallel software, the parallel/distributed computing community must eventually move towards higher level programming models.

Enterprise is publicly available and can be obtained through the World Wide Web (WWW) from http://web.cs.ualberta.ca/labs/enterprise.html or obtained be *ftp* from the directory pub/Enterprise at ftp.cs.ualberta.ca.

Acknowledgments

This research has been funded by NSERC grants OGP-8173, OGP-8191 and CRD-148317 and a grant from the Centre for Advanced Studies, IBM Canada Limited.

About the Authors

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.

Jonathan Schaeffer is a Professor of computing science at the University of Alberta. His research interests include parallel computing (programming environments and algorithms) and artificial intelligence (heuristic search). He received a Ph.D. and M.Math. from the University of Waterloo and a B.Sc. from the University of Toronto. His Internet address is jonathan@cs.ualberta.ca.

Paul Iglinski is finishing an M.Sc. at the University of Alberta. The focus of his research is distributed parallel debugging and object-oriented user interfaces for parallel computing. He received a B.A. (Math) from the University of South Florida, an M.A. (Chinese) from Stanford University, and a B.Sc. from the University of Alberta. His Internet address is iglinski@cs.ualberta.ca.

Ian Parsons is in the Ph.D. program at the University of Alberta. His main interest is in programming environments for distributed parallel applications. He received a B.Sc. and M.Sc. from the University of Alberta, and a B.Sc. (Chemistry) from the University of Western Ontario. His Internet address is ian@cs.ualberta.ca.

Randal Kornelsen is a programmer/analyst at the University of Alberta. His areas of interest include neural networks and distributed parallel applications. He received a B.Sc. from the University of Alberta. His Internet address is rand@cs.ualberta.ca.

Chris Morrow is a programmer/analyst at the University of Alberta. His research interests include parallel and distributed computing. He received a B.Sc. from the University of Alberta. His Internet address is morrow@cs.ualberta.ca.

References

[1] H. Bal, M. Kaashoek and A. Tanenbaum. "Orca: A Language for Parallel Programming of Distributed Systems". *IEEE Transactions on Software Engineering*, vol. 18, no. 3, pp. 190-205, 1992.

[2] K. Birman et al. "The ISIS System Manual, Version 2.1". ISIS Project, Computer Science Dept., Cornell University, 1991.

[3] G. Geist and V. Sunderam. "Network-Based Concurrent Computing on the PVM System". *Concurrency: Practice and Experience*, vol. 4, no. 4, pp. 293-311, 1992.

[4] R. Halstead. "Multilisp: A Language for Concurrent Symbolic Computation". *ACM Transactions of Programming Languages and Systems*, vol. 7, no. 4, pp. 501-538, 1985.

[5] G. Lobe, D. Szafron, and J. Schaeffer. "The Enterprise User Interface". TOOLS (Technology of Object-Oriented Languages and Systems) 11, R. Ege, M. Singh and B. Mayer (editors), pp. 215-229, 1993.

[6] D. Loveman. "High Performance Fortran". *IEEE Parallel and Distributed Technology*, vol. 1, no. 1, pp. 25-42, 1993.

[7] T.A. Marsland, T. Breitkreutz and S. Sutphen. "A Network Multi-processor for Experiments in Parallelism", *Concurrency: Practice and Experience*, vol. 3, no. 1, pp. 203-219, 1991.

[8] S.N. McIntosh-Smith, B.M. Brown and S. Hurly. "Intelligent Algorithm Decomposition For Parallelism with Afar". IFIP WG10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems, Monte Verità, Ascona, Switzerland, pp. 5.1-5.10, 1994 (to appear).

[9] Open Software Foundation. "OSF DCE Release 1.0 Application Development Guide". OSF, Cambridge Mass., 1992.

[10] J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons. "The Enterprise Model for Developing Distributed Applications". *IEEE Parallel & Distributed Technology*, vol. 1, no. 3, pp. 85-96, 1993.

[11] D. Szafron and J. Schaeffer. "Experimentally Assessing the Usability of Parallel Programming Systems". IFIP WG10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems, Monte Verità, Ascona, Switzerland, pp. 19.1-19.7, 1994 (to appear).

[12] D.W. Walker. "The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers". *Parallel Computing*, vol. 20, no. 4, pp. 657-673, 1994.