Enterprise in Context: Assessing the Usability of Parallel Programming Environments

Gregory V. Wilson
Jonathan Schaeffer
Duane Szafron

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

Abstract

The growth of commercial and academic interest in parallel and distributed computing during the past fifteen years has been accompanied by a corresponding increase in the number of available parallel programming systems, and in the variety of approaches to parallel programming being taken. However, little or no work has been done to compare or evaluate different systems, or to develop criteria by which such comparisons could be made. As a result, a typical parallel programming system is usually evaluated by the ease or difficulty with which its author(s) can implement a small set of trivially-parallel algorithms.

This paper is a step toward rectifying this situation. We present several criteria by which parallel programming systems might be quantitatively evaluated, and assess the importance and measurability of each. Of these criteria, we feel that usability is the most important, but also the least frequently quantified. For illustration, we compare the approach taken in the Enterprise parallel programming environment with several other systems and their approaches. We also predict the results we expect from these comparisons. Finally, we argue that while the cost of performing quantitative measurements of usability might seem large, the cost of not performing them, as borne by a group which selects an inappropriate or low-performing programming system, is likely to be much larger.

1 Introduction

In the past decade, parallel and distributed computing (henceforth simply "parallel computing") has become a viable commercial technology. A large infusion of research and development money has accompanied this transition, and has accelerated the development of a diverse collection of new and innovative architectures, including hypercubes, massively-parallel processor arrays, networks of workstations and shared memory machines. Each architecture can achieve high performance for some classes of applications but does poorly on others. This adds an extra dimension of complexity to any purchase decision.

While hardware technology has advanced rapidly, the same cannot be said of the software provided to program these machines. Parallel software development has to contend with problems not found in a sequential environment, such as non-determinism, communication, synchronization, fault-tolerance, heterogeneity, shared and/or distributed memory, deadlock and race conditions. Further, if the parallelism in an application is not suited to the topology of a given parallel architecture, the designer may have to contort the program to match the machine.

The IBM contact for this paper is Patrick Finnegan, Centre for Advanced Studies, IBM Canada Ltd., Dept. 81/894 895 Don Mills Road, North York, Ontario M3C 1W3.

This paper is available as Department of Computing Science, University of Alberta, Technical Report TR93-09.

This is a pre-print of a copyrighted article in Proceedings of CASCON '93, Toronto, Canada, 1993 pp. 999-1010.

Underlying all of this is the implicit need for high performance.

A large number of parallel programming systems have been developed to simplify the task of developing parallel software. They support a variety of architectures ranging from tightly-coupled shared-memory multiprocessors to loosely-coupled networks of workstations with distributed memory. In this paper we use the term parallel programming system to encompass all architectures that support concurrent execution of program components.

At one extreme, some of these systems support specialized programming models that allow the user to quickly achieve high performance for selected applications, but poor performance on others. At the other extreme, some systems provide a set of low-level primitives that allow the programmer to achieve high performance by custom application crafting, but drastically increase software development time. In the literature, parallel programming systems are often illustrated and compared using trivial programs (such as matrix multiplication, prime number generation or Mandelbrot set), usually with conflicting results. There are no established guidelines for comparing these systems based on their run-time performance, ease of use, applicability and the time it takes to develop correct programs. This paper is a step toward rectifying this situation.

Enterprise is a programming environment currently under development at the University of Alberta [20]. It supports the development of distributed applications that execute on a network of workstations. Like other parallel programming systems, it would benefit from an objective, scientific evaluation. Early feedback from such an evaluation can also be used to identify components that can be improved during the development of the system.

Section 2 presents a taxonomy which groups parallel programming systems according to how they present parallelism to the user. In Section 3, we give a brief description of the Enterprise system which is used in Sections 4 and 6 as a point of comparison with other systems. In Section 4, we present several criteria according to which parallel programming systems might be evaluated, and assess the importance and measurability of each. Since performance issues are described in other places in the literature, we focus our attention on usability; Section 5 discusses experiments to measure it. In Section 6, a representative system from each taxonomic class is analyzed, and we make some predictions on how the systems will fare. In Section 7, we make some concluding remarks about the future of parallel programming systems.

2 A Taxonomy for Parallel Programming Systems

From the point of view of "traditional" programmers, trained in sequential imperative languages such as Fortran and C, parallel programming systems may be classified using two criteria: the extent to which they deviate from traditional imperative sequential systems and how the parallelism is presented to the user. Other considerations, such as message-passing versus shared-memory architectures, are implementation issues, not user issues.

One criteria is whether the parallel system is an evolutionary advance on an existing sequential imperative system, or a revolutionary break with the imperative tradition. Vectorizing compilers such as Parafrase-2 [19], which accept sequential code as-is and find fine-grained parallelism automatically, are at the evolutionary end of this scale. Systems such as PCN [11], in which serial Fortran fragments are nested in a parallel "harness" constructed in a system based on dataflow and logic programming, represent a radical departure from the mainstream programming tradition and are at the revolutionary end of the scale. Many systems fall in between these two extremes, with evolutionary systems and their variants predominating. For example, Fortran-D [17] and Vienna Fortran [4], in which users provide directives or "hints" to control data distribution, are evolutionary, but do require some knowledge of how parallelism can be exploited to be used effectively. Thinking Machines Corporation's CM-Fortran [22], which adds new data-parallel operators and semantics to Fortran, is a significant enhancement to its sequential predecessor, yet from the user's point of view, the changes are not a revolutionary departure from the conventional programming view.

A complementary classification scheme describes how a system presents parallelism to users. We begin by distinguishing between systems in which parallelism is implicit and systems in which it is explicit, as shown in Figure 1.

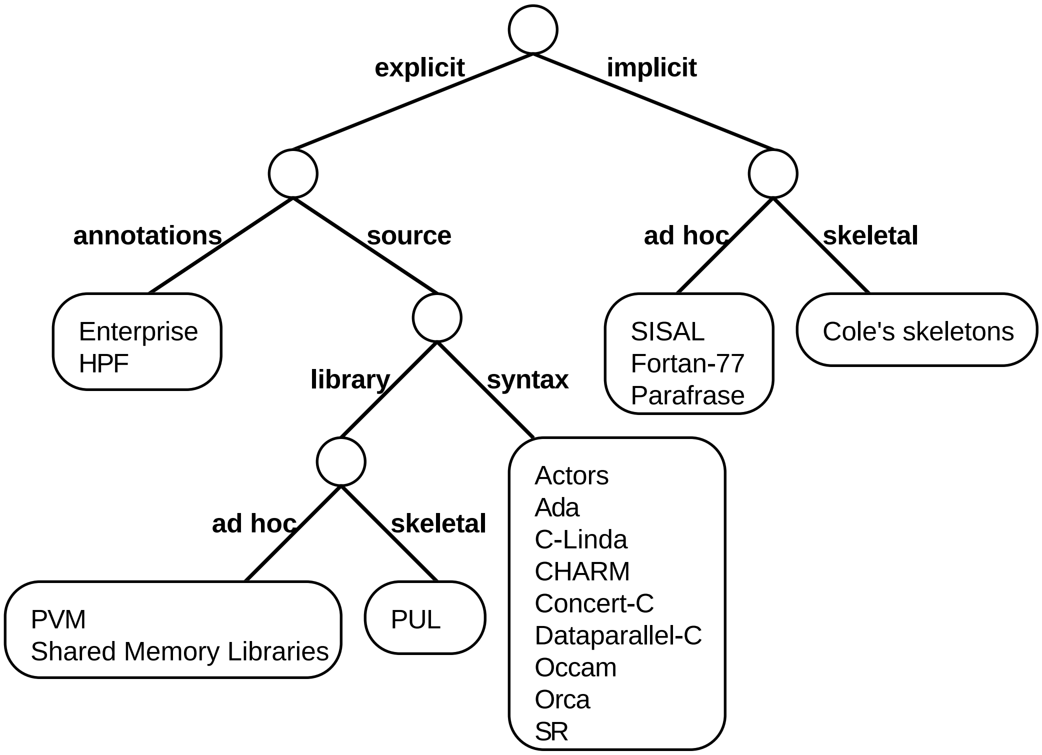

Figure 1: A taxonomy for parallel programming systems.

In the former, users cannot force program units to execute concurrently or serially, and are not responsible for protecting data against race conditions; these issues are handled automatically by the compiler or run-time system. In the latter, users have at least partial responsibility for execution order and data management.

Most parallel programming systems in use today are explicit. Such systems may be differentiated according to how users describe parallelism; either by adding annotations to serial code, or by modifying the source of their programs. Annotation-based systems include the Fortran-D and Vienna Fortran systems mentioned previously. They also include the Enterprise system described in Section 3, in which the annotations are denoted graphically by selecting one of a restricted (but expressive) set of options.

Systems in which parallelism is expressed in a program's source fall into two categories: those in which new syntax is introduced, and those which encapsulate parallelism in a set of libraries, so that the syntax of the original serial language is unaltered. There are many examples of syntax-based systems. Four based on C are Dataparallel-C [16], Concert/C [13], CHARM [21] (originally called the Chare Kernel), and C-Linda [8, 9]. The first of these allows users to specify domains, which are collections of identical records on which operations may be performed concurrently. Concert/C allows the user to create typed ports and use them to communicate between processes. These ports can be passed like data from process to process. CHARM is based on active messages; messages which can turn into processes, or which activate an existing process. Finally, C-Linda provides users with four new operations on a virtual shared memory. Three of these move data in and out of the shared memory, while the fourth causes new processes to be created.

The PVM message-passing system [14] and libraries for shared-memory programming (for example, those provided by Sequent for its Symmetry product range) are two well-known examples of the library approach. The former provides routines for sending a message from one process to another, for broadcasting a message to all other processes, and for receiving a block of bytes from an arbitrary or a selected process. The latter provides an interface to semaphores, barriers, and fork-style process creation.

Both of the above library approaches allow unstructured (or ad hoc) programming. An alternative is to provide libraries tailored for particular situations. The Parallel Utility Libraries (PUL) [3] are an example of such a system. The PUL packages are a set of skeletons, or templates, within which users may nest their own code fragments. There are, for example, packages to support task farming, divide-and-conquer algorithms, and regular and irregular mesh decomposition. These packages manage the mechanical details of parallelism; the user simply supplies the routines which (for example) generate, process, and consume work packages in a task farm. These systems act as "skeletons" for structured parallel programs.

This same distinction between ad hoc and skeletal systems can be made for implicit parallel programming systems as well. Ad hoc implicit systems include automatically vectorized or parallelized Fortran, and parallel functional and dataflow languages such as Sisal [12], in which it is the compiler's responsibility to find and take advantage of concurrency. However, one can also construct a parallel functional language in which parallelism is expressed through the use of higher-order functions, such as divide-and-conquer, for which efficient parallelization techniques are known. Cole's "algorithmic skeletons" [10] is a representative of this category.

The characteristics in this taxonomic scheme have been chosen in part to allow predictions to be made about the power and usability of parallel programming systems based on their classification. We would expect, for example, that implicit systems would be easier to use than explicit systems. However, it is generally (though not invariably) the case that implicit systems deliver lower performance, since they are completely dependent on the intelligence of the compiler being used. This is analogous to the issues raised when high-level language compilers were first introduced. Similarly, we would expect that users of skeletal systems would make fewer errors than users of ad hoc systems, for the same reason that someone using a form-based editor is less likely to omit important information than someone editing freehand. However, if the user's application is not matched by one of the skeletons provided by the system, the performance delivered by the skeletal system will be poor. We also expect that annotation-based systems will be better suited to recycling pre-existing serial programs than systems which require modification of program source, and that keeping annotations outside the source, as Enterprise does, will allow faster code development.

One result which we are unable to predict is the relative strengths and weaknesses of adding new syntax versus providing a library using conventional syntax. The former is often more concise, but high-quality compilers take a long time to produce, and it is not yet clear whether any of the syntactic supersets which have been proposed for serial languages such as C and Fortran (or any of the Algol descendant such as Orca [5] and SR [1, 2]) actually cover all the interesting cases. It is far easier to extend a library than to add yet another bit of syntax to a language. However, it is often clumsy to express simple programs in a library-based system, and users must often accept responsibility for low-level operations such as marshalling and un-marshalling linked data structures.

Figure 1 shows the taxonomy and illustrates each data point with a number of representative programming systems that have been developed. It is interesting to observe the apparent research emphasis on explicit/syntax/source parallel programming systems. However, in practice, systems that currently have (or will have) wide use (HPF and PVM) are often drawn from other places in the taxonomy. This indicates that there is a large gap between where the computer scientists and the user community perceive the research efforts should go.

3 An Overview of Enterprise

In Enterprise, the interactions of processes in a parallel computation are described using an analogy based on the parallelism in a business organization [20]. 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 *departments*, *receptionists*, *individuals*, *divisions*, and *representatives*). 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 blocks and waits for the function result when it accesses the result (`a = result`). Allowing concurrent actions until the result of a previous computation is required is called a *future* [15].

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 a process. 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, 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 this individual 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 individual into a line containing the components Model, PolyConv and Split.

Enterprise programs are not edited by free-hand drawing in which users connect assets in arbitrary topologies. Instead, the programming model limits users to four powerful operations: transforming the kind of an asset, expanding and collapsing an asset to reveal its components and replicating an asset. This approach is designed to eliminate programming errors.

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 then 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. Several other asset kinds are supported by Enterprise and can be combined in arbitrary hierarchies.

Figure 2 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.

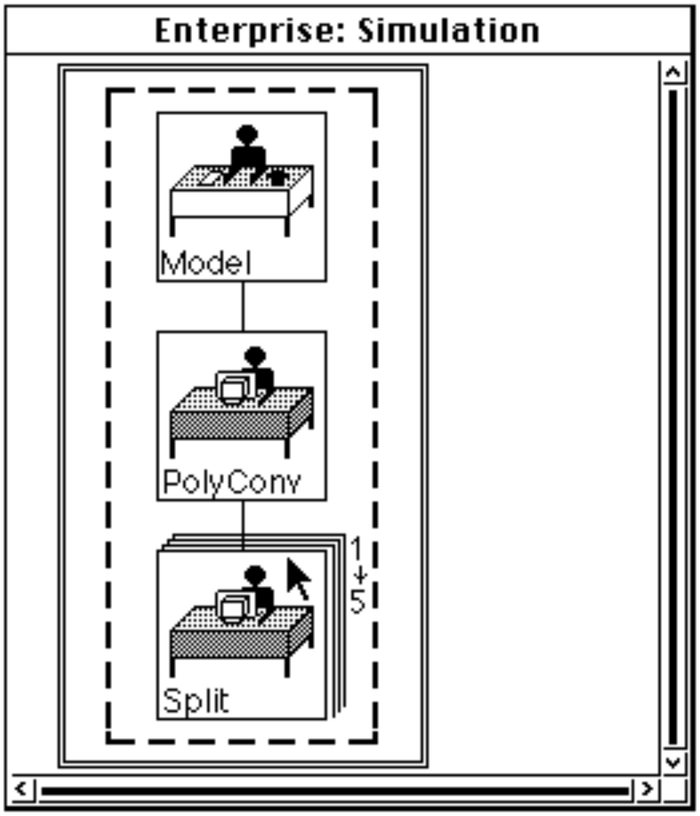

Figure 2: A program in the Enterprise programming environment.

Because the diagrams are at a high level, many common types of parallel programming errors are not possible in Enterprise, and correct applications can be generated more quickly. The disadvantage is that not every application is well-suited to the Enterprise approach.

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, ensuring that the work is evenly distributed.

4 Assessment Factors

There are many considerations that can go into the assessment of parallel programming systems, but the majority fall into three categories:

1) Performance: For the applications of interest, what kind of run-time performance will be achieved? In many organizations, performance is often considered to be a feature of hardware alone, without proper consideration of programming systems and the performance they are (in)capable of achieving.
2) Applicability: Is the programming system available on a variety of hardware platforms, and will it achieve high performance on each? Although a system might be available on a variety of machines, it may only achieve high performance on a specific platform.
3) Usability: How easy is application design, development, coding, testing and debugging? Some programming systems address only one of these activities, without providing support for the rest of the software development cycle.

Within each of these categories, there are a number of issues that are worth evaluating. Table 1 lists some of the notable ones.

CATEGORY	ASSESSMENT METRIC
Performance	benchmark results speed of code generated memory usage turnaround time
Applicability	portability hardware dependence programming languages supported
Usability	learning curve probability of programming errors functionality integration with other systems deterministic performance compatibility with existing software suitability for large-scale software engineering power in the hands of an expert ability to do incremental tuning

Table 1: Assessment factors.

Recently, the parallel/distributed computing community has focussed its attention on the development of benchmark test suites, consisting of a diverse collection of programs. Given the diversity of algorithmic techniques and communication patterns in these test suites, it is difficult for any system to provide uniformly high performance across all tests. Usually, a system does very well on a handful of test programs and has poor or mediocre performance on the rest. Since the performance issue is addressed in many other papers, we do not elaborate on it further.

There are a number of ways to assess the applicability of a programming system. Portability is an important issue in the sequential world, but has an extra dimension of complexity in the parallel world. Any programming system can be ported to a variety of hardware platforms. However, its performance may be low if its special hardware needs are not met. Availability can be assessed either globally (over a wide range of machines) or locally (meeting an individual organization's needs). Clearly, high availability of a system is meaningless if it does not support your machine or base language.

Of the aspects listed in this paper, the least frequently measured is usability. Nevertheless, it may be the most important dimension since it influences the productivity of programmers, which has a big impact on time-to-market and other factors that directly affect corporate profits. Given the extra complexity of debugging and testing parallel and distributed software, it is essential that a programming system eliminate, simplify, or at least mask the complexity.

5 Measuring Usability

Quantifying and measuring usability involves human-factors considerations that are often ignored in "main-stream" computing science. Several features of a parallel programming system determine its usability. Among these are:

1) Learning curve: How long does it take an expert or an inexperienced parallel programmer to be able to use the programming system productively? Note that some systems specifically address the needs of experts, while others are targeted at novices; few do both.

2) Programming errors: Some systems restrict the kinds of parallelism to prevent errors (e.g. Enterprise). Other systems, such as PVM, allow the user to do anything, trading flexibility for a greater chance of introducing

logic errors. Usually the potential for errors is directly related to the number of lines of user code. Therefore, systems that require more user code are more susceptible to errors.

3) Deterministic performance: Non-determinism, common in the implementation of some algorithms and inherent in some programming systems, can significantly increase the overhead in application debugging.

4) Compatibility with existing software: Legacy software cannot be ignored. Ideally, the programming system must support the integration of existing software with minimal effort.

5) Integration with other tools: A programming system should either come with, or provide access to, debugging, monitoring and performance-evaluation tools.

Although there have been many human-factors studies of the productivity of sequential programmers [6], we know of no comparable studies for programmers developing parallel software. We propose two experiments to assess the productivity of parallel programming systems. The first measures the ease with which novices can learn the programming system and produce correct, but not necessarily efficient, programs. The second measures the productivity of the system in the hands of an expert. The mechanics of these experiments are quite simple: put a group of programmers in a room, give them instructions for a programming system, give them some problems to solve, and measure what they do. For novices, we are interested in measuring how quickly they can learn the system and produce correct programs. For experts, we want to know the value of $p_{1/2}$, the time it takes to produce a correct program that achieves at least half the performance of the machine. (This is analogous with Hockney's $n_{1/2}$, which is the vector length on which a pipeline delivers half its peak performance [18]).

There are a number of considerations that must be taken into account in the design of the experiment to obtain fair results:

1) The subjects used in an experiment must be chosen with care. The novices should have no previous experience in parallel computing. To eliminate the cumulative effects of learning, a subject can only be used in one experiment. On the other hand, the experts should have full knowledge of the programming systems they are using. For each experiment, the test groups must be balanced to ensure the subjects have equivalent abilities and experience.

2) Given that all the programming systems will not run on the same hardware, the execution times of the system and the application must be normalized across all hardware platforms. For example, if machine A is ten times faster than machine B, then system response on A (such as compilation) and application execution times must be presented to the user ten times slower than real time. This allows the hardware speeds to be factored out of the experiment.

3) Ideally, the test problems should be chosen in a way that does not introduce significant biases into the experiment. For example, the problems should not favour one system over another. As this is impossible, one solution is to partition typical "real world" applications into a number of classes (for example: matrix manipulation, combinatorial search, Monte-Carlo simulations) and include a representative problem from each of the classes. The arguments for and against this solution are the same as the arguments for and against any performance evaluation suite

4) A uniform way of introducing the programming system to novices must be developed. We do not want the experiment adversely affected by the quality of instruction or documentation provided for each system. Novices will be taught by a neutral party. Instruction will be constrained by time and the number and type of sample programs presented.

Some of the important items to measure during each experiment include the following: time taken to solve each problem, number of lines of code written, number of compilations required, number of program executions, number and type of run-time errors uncovered, and program's performance achieved over time.

Although the details outlined here are sketchy, they nevertheless convey a sense of the human factors issues in parallel programing systems that have been neglected to date. We propose that the above experiment (or variations on it) should be a priority. Given the diversity of programming systems available, as evident in Section 2, researchers need more feedback into what works well and why. We recognize that the cost of performing such quantitative measurements will be large. However, the cost of not performing them, as borne by a group which selects an inappropriate or low-performing programming system, will certainly be much larger.

6 Anticipating the Results

As discussed in the previous section, we intend to compare parallel programming systems by measuring the time required by novices to produce correct programs, and the rate at which experienced programmers can produce programs with good performance. In Table 2, we outline the results we expect from these tests for several of the programming systems introduced in Section 2, and justify our expectations. In this, we assume that a *novice* programmer is one with experience of sequential programming in either C or Fortran (or, where noted, with functional programming), while an *expert* is someone who has worked with a system extensively. The rating column is an assessment of parallel programming system's usability, measured by the speed of the learning curve for novices (fast = high rating), and the speed by which experienced programmers can achieve $p_{1/2}$ (fast = high rating).

The revolutionary systems are exemplified by functional languages like Sisal and by the higher-order skeletons of Cole. To date, these have only delivered poor to moderate performance, although it is improving steadily [7]. We predict that novices will find such systems either very difficult to learn (if they have no previous experience with functional programming), or very easy (if higher-order function application, streams, and similar concepts are already part of their repertoires).

The first evolutionary system we consider is C-Linda. This is a "conservative" extension of C in that it introduces very little new syntax, and a small (but powerful) set of new concepts. The authors' experience is that novices can learn and use C-Linda quickly. Typically, a half-hour of instruction is all that is required. Similarly, expert users are often able to build prototypes of parallel systems in C-Linda quickly. However, it is no easier to improve the performance of C-Linda programs than those based on any other system. This is primarily because C-Linda allows less of the underlying hardware to show through than, for example, PVM, but it is exactly this underlying hardware which must be taken into account when programs are tuned.

Rating	**Novice**	**Expert**
High	C-Linda Enterprise (right problem) Sisal (functional prog.) Skeletons (functional prog.)	HPF (right problem) Enterprise (right problem)
Medium	Concert/C HPF PUL (right problem)	Concert/C C-Linda PUL (right problem) PVM Sisal
Low	Enterprise (wrong problem) PVM Sisal (imperative prog.) Skeletons (imperative prog.)	Enterprise (wrong problem) Skeletons
Zero	PUL (wrong problem)	PUL (wrong problem)

Table 2: Expected assessments.

Concert/C is a less conservative set of extensions to C. While its basic RPC model is easy to understand, the language contains a great deal of extra syntax. Thus, we expect that novices would require more time to produce working programs in Concert/C than in C-Linda. However, we expect that expert users would find program tuning equally easy (or hard) in the two systems.

PVM is a much lower-level system than either of these. It is presented as a set of libraries rather than as syntactic extensions, and leaves the responsibility for such things as data marshalling and unmarshalling entirely in the users' hands. The model it implements is also lower level than those used in C-Linda and Concert/C. In essence, PVM is just a way of moving bytes from one processor to another. Our experience is that such message-passing systems are significantly more difficult to learn than higher-level systems. In part, this is because the number of initialization and parameter-packing calls required by such systems means that there is no such thing as a short message-passing program, and that there are therefore many more lines of code in which novices might make errors. However, expert users can produce most of this code quickly and correctly. In addition, since PVM more accurately reflects the hardware on which it runs, it sometimes allows for easier tuning of programs. Thus, while C-Linda and Concert/C are rated less highly for experts than for novices, we expect that PVM's rating would be pulled up as its users' experience grew.

High Performance Fortran (HPF) combines Fortran 90's data-parallel operators with mechanisms for decomposing and distributing array structures. We predict that novices will find writing parallel programs in HPF as easy, or as difficult, as writing serial programs in Fortran, and so place it with Concert/C. When used on the right sorts of problems (i.e. large arithmetic calculations), we expect that HPF will allow experts to rapidly obtain high performance. On the wrong sorts of problems (for example, game-tree search or database manipulations), we would expect HPF to do poorly since its features were not designed with such problems in mind. We return to this topic in the final section.

Since PUL hides the details of parallelism from its users, it is relatively simple to use, if the problem being implemented is one which PUL supports. Thus, PUL's rating in the hands of novices is either medium (medium, rather than high, because there are a lot of little details about parameter passing and the like which novices have to master), or zero. Similarly, PUL is rated highly for expert use on the right sorts of problems, since those cliches which it supports are supported very efficiently.

Our predicted ratings for Enterprise are similar, as are their justifications. Enterprise's graphical interface makes it even easier to use than PUL, provided the framework required is one which Enterprise offers, so it is highly-rated for novice use. In expert hands, Enterprise is also highly rated for the right sorts of problems, but rated above PUL for other problems, since service assets can always be used to model general message passing. We note that a combination of Enterprise and PUL, i.e. a system in which users provided code fragments to be nested in pre-arranged configurations of assets, would probably be more powerful than either of its halves.

7 Conclusions

This paper has identified an area where the parallel/distributed computing community has been negligent in providing quantitative data. Hardware vendors are quick to cite measures that flatter the performance of their machines (MIPS, SPECmarks, Whetstones, etc.), but neglect to quantify the quality of their software. The growing parallel computing user base could significantly benefit from an objective assessment of parallel programming systems.

Although there have been numerous research and industrial efforts at developing parallel programming tools, few will gain wide acceptance. Two systems in particular will have a big impact on parallel computing in the 1990s. Because of the large vector-processing community and the legacy of Fortran, High Performance Fortran (HPF) will emerge as the dominant system in this area. In the area of distributed computing, PVM is rapidly becoming a *de facto* standard, largely because the system is free, and because there is a large effort to support and enhance the system. However, HPF is only targeted to a small part of the entire user community, while PVM is a low-level system that needs a high-level interface.

Much of the success of data parallel systems like HPF is because they hide the low-level details of fine-grained parallelism. This success can be attributed to the completeness of its data-parallel model. Unfortunately, no similar model exists yet for coarse-grained parallelism. To date, the best we can do is provide a tool-kit that supports the most commonly used techniques like pipelines, task farms, and divide-and-conquer. Using Enterprise's graphical annotations to hide the details of a lower-level message-passing kernel like PVM seems to be the best approach in the medium term.

The Enterprise effort is committed to the approach of using sequential code with parallel annotations expressed graphically. This approach is not an all-encompassing general-purpose solution for all parallel applications, but allows an important class of problems to be solved more quickly. We believe these high-level approaches to parallel computing will become more prevalent as their obvious software engineering advantages become recognized. It is easy to make comparisons on paper, emphasizing our perception that Enterprise has a high degree of usability. However, until scientific experiments are done to compare programming systems, anyone's claim is as valid as any other.

Acknowledgements

This research has been funded by NSERC grant OGP-8173. The Enterprise project is funded in part by IBM Canada Limited. Financial assistance for Greg Wilson's stay at the University of Alberta was provided the University of Alberta's Central Research Fund.

References

[1] G. Andrews. Concurrent Programming: Principles and Practice. Benjamin/Cummings, 1991.

[2] G. Andrews, R. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin and G. Townsend. An Overview of the SR Language and Implementation. *ACM Transactions of Programming Languages and Systems*, Vol. 10, No. 1, January 1988.

[3] R. Bruce, S. Chapple, N. MacDonald, A. Trew. CHIMP and PUL: Support for Portable Parallel Computing. Fourth Annual Conference of the Meiko User Society Proceedings, 1993.

[4] S. Benkner, B. Chapman and H. Zima. Vienna Fortran 90. *Scalable High-Performance Computing Conference Proceedings*, R. Voigt and J. Saltz (eds), IEEE Computer Society Press, 1992.

[5] 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, March 1992.

[6] R. Brooks. Studying Programmer Behavior Experimentally: The Problems of Proper Methodology. *CACM*, Vol. 23, No. 4, 1980.

[7] D. Cann. Retire Fortran? A Debate Rekindled. *CACM*, Vol. 35, No. 8, August 1992.

[8] N. Carriero and D. Gelernter. Linda in Context. *CACM*, Vol. 32, No. 4, April 1989.

[9] N. Carriero and D. Gelernter. *How to Write Parallel Programs*. MIT Press, Cambridge, Mass., 1990.

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

[11] K. Chandy and S. Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett, 1992.

[12] J. Feo, D. Cann and R. Oldehoeft. A Report on the Sisal Language Project. *Journal of Parallel and Distributed Computing*, Vol. 10, 1990.

[13] A. Goldberg. Concert/C: A Language for Distributed C Programming. Internal report, IBM T.J. Watson Research Center, Yorktown Heights, New York, 1993.

[14] G. Geist and V. Sunderam. Network-Based Concurrent Computing on the PVM System. *Concurrency: Practice and Experience*, Vol. 4, No. 4, June 1992.

[15] R. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions of Programming Languages and Systems*, October 1984.

[16] P. Hatcher and M. Quinn. *Data-Parallel Programming on MIMD Computers*. MIT Press, Cambridge, Mass., 1991.

[17] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer and C. Tseng. An Overview of the Fortran D Programming System. Technical Report CRPC-TR91121, Center for Research on Parallel Computation, Rice University, 1991.

[18] R. Hockney. Performance Parameters and Benchmarking of Supercomputers. *Parallel Computing*, Vol. 17, 1991.

[19] C. Polychronopoulos, M. Girkar, M.Haghighat, C. Lee, B. Leung and D. Schouten. The Structure of Parafrase-2: an Advanced Parallelizing Compiler for C and Fortran. *Languages and Compilers for Parallel Computing*, D. Gelernter (ed), A. Nicolau and D. Padua, Pitman, London, 1990.

[20] J. Schaeffer, D. Szafron, G. Lobe and I. Parsons, The Enterprise Model for Developing Distributed Applications. *IEEE Parallel and Distributed Technology*, to appear Fall 1993.

[21] W. Shu and L. Kalé. Chare Kernal: a Runtime Support System for Parallel Computations. *Journal of Parallel and Distributed Computing,* Vol. 11, 1991.

[22] Thinking Machines Corporation. CM-Fortran Users' Manual. 1991.

Biographies

Gregory V. Wilson received a B.Sc. in Mathematics and Engineering from Queen's University, Kingston, in 1984, an M.Sc. in Artificial Intelligence from the University of Edinburgh in 1986, and a Ph.D. in Computer Science, also from the University of Edinburgh, in 1993. His main interest is the development and assessment of parallel programming systems. He regularly writes popular science articles for various newspapers and magazines. His Internet address is gvw@epcc.ed.ac.uk.

Jonathan Schaeffer is an Associate Professor of computer 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.

Duane Szafron is an Assistant Professor of computer 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.