An Experiment to Measure the Usability of Parallel Programming Systems[1]

Duane Szafron
Jonathan Schaeffer

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

SUMMARY

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 (PPS). However, little work has been done to evaluate their usability, or to develop criteria for such evaluations. As a result, the usability of a typical PPS is based on how easily a small set of trivially parallel algorithms can be implemented by its authors.

This paper discusses the design and results of an experiment to objectively compare the usability of two PPSs. Half of the students in a graduate parallel and distributed computing course solved a problem using the Enterprise PPS while the other half used a PVM-like library of message-passing routines. The objective was to measure usability. This experiment provided valuable feedback as to what features of PPSs are useful and the benefits they provide during the development of parallel programs. Although many usability experiments have been conducted for sequential programming languages and environments, they are rare in the parallel programming domain. Such experiments are necessary to help narrow the gap between what parallel programmers want, and what current PPSs provide.

1. INTRODUCTION

In the past decade, the technology of parallel and distributed computing (henceforth simply parallel computing) has moved from the laboratory to the commercial domain. A large infusion of research and development money has accompanied this transition and has accelerated the development of a diverse collection of innovative architectures such as hypercubes, massively parallel processor arrays, multicomputers and shared-memory multiprocessors. Although each architecture achieves high performance for some classes of applications, it does poorly for others, adding an extra dimension of complexity to selecting an architecture.

Unfortunately the development of software for these new hardware architectures has lagged behind. Parallel software developers must contend with problems not encountered during sequential program development. Non-determinism, communication, synchronization, fault-tolerance, heterogeneity, shared and/or distributed memory, deadlock and race conditions present new challenges. Further, if the parallelism in an application is not suited to the topology of a given parallel machine, the designer may have to contort the program to match the architecture. Underlying all of this is the implicit need for high performance.

---

[1] A summary of this work appeared as: D. Szafron and J. Schaeffer, "Experimentally Assessing the Usability of Parallel Programming Systems", *Programming Environments for Massively Parallel Distributed Systems*, Karsten Dekker and Rene Rehmann (Editors), Birkhauser Verlag, Basel, Switzerland, pp. 195 - 201, 1994.

This is a pre-print of a copyrighted paper that will appear in Concurrency Practice and Experience.

A number of software systems have been developed to simplify the task of developing parallel software. Each is suitable for one or more architectures ranging from tightly-coupled shared-memory multiprocessors to loosely-coupled networks of workstations with distributed memory. At one extreme, some of these systems support specialized programming models that allow programmers to quickly achieve high performance for selected applications. Unfortunately, this performance cannot be attained across all classes of applications. Other systems provide a set of low-level primitives that allow the programmer to achieve high performance for many applications, but at the expense of increased software development time. In this paper we use the term parallel programming system (PPS) to encompass all software systems that support the design, implementation and execution of concurrent program components.

There are many considerations that affect the assessment of PPSs, but the majority fall into three categories [1]:

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 PPSs and the performance they are (in)capable of achieving.
2) Applicability: What types of parallelism are easily expressed using the PPS? Is the PPS available on a variety of hardware platforms, and will it achieve high performance on each? Although a PPS might be available on a variety of machines, it may only perform well on a specific platform.
3) Usability: How easy is application design, development, coding, testing and debugging? Some PPSs address only one of these activities, without providing support for the rest of the software development cycle.

Within each of these categories, several issues must be considered. Table 1 lists some of the most important ones.

Recently, the parallel/distributed computing community has focused its attention on the development of benchmark test suites, consisting of a collection of programs, as a way of quantifying performance. 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 favorite machine or base language.

Of the aspects listed here, the least frequently measured is usability. Nevertheless, it may be the most important since it directly influences the productivity of programmers. Given the extra complexity of debugging and testing parallel and distributed software, it is essential that a PPS eliminate, reduce, or at least mask the complexity. There are many papers in the literature that compare different parallel programming systems based on their technical merits (for example [2]), but none of them attempt to assess quantitatively the effect of the programming system on the productivity of the users of the system.

CATEGORY	ASSESSMENT METRIC
Performance	benchmark results speed of generated code memory usage turnaround time
Applicability	portability hardware dependence programming languages supported types of parallelism 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.

This paper focuses on the usability of PPSs and introduces one way to measure it. A controlled experiment was conducted in which half of the graduate students in a parallel/distributed computing class solved a problem using the Enterprise PPS [3] while the rest used a PPS consisting of a PVM-like [4] library of message passing calls (NMP [5]). The specific PPSs used in this experiment are not the focus of this paper. Instead, we argue that controlled experiments must be conducted so that the PPS developers meet the usability of their user community. An objective evaluation during system development can give valuable feedback on the programming model, completeness of the programming environment, ease of use, learning curves, etc. Although controlled experiments have been performed to compare the usability of sequential programming languages and environments [6], it is surprising that except for one [7] we know of no other comparable work for PPSs. Results of objective experiments are necessary to help narrow the gap between what parallel programmers really want and what current PPSs provide.

Section 2 describes two types of proposed experiments for measuring usability, one for novices and one for experts. The experiment described in this paper is for novices. Section 3 describes the sample problem that was used in the experiment and outlines the Enterprise and NMP solutions. Section 4 describes the design of the experiment. Section 5 presents an analysis of the experimental results. Section 6 describes how this work should influence the design of future experiments. Section 7 describes the future directions of this work.

2. MEASURING USABILITY

Quantifying and measuring usability involves human-factors considerations that are often ignored in "main-stream" computing science. Several features of a PPS 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 PPS productively? Note that some PPSs specifically address the needs of experts, while others are targeted at novices; few are suitable for both.

2) Programming errors: Some systems restrict the use of parallelism to prevent errors (e.g. Enterprise). Other systems, such as NMP and PVM, allow the user to do anything, trading flexibility for a higher chance of programming errors. Usually the potential for errors is directly related to the number of lines of user code. Therefore, systems that require more user code may be more susceptible to errors.

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

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

5) Integration with other tools: A PPS should either come with, or provide access to, a complete suite of software development tools including facilities for debugging, monitoring and performance evaluation.

Although there have been many human-factors studies of the productivity of sequential programmers [8], we know of no comparable studies for programmers developing parallel software. We proposed two experiments to assess the productivity of PPSs [1]. The first measures the ease with which novices can learn the PPS 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 PPS, 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 time it takes to produce a correct program that achieves a specified level of performance on a given machine.

There are a number of interesting statistics that can be gathered during such an experiment to quantify some of the usability properties of a system. For example, the number of lines of code taken to solve a given problem is one measure of the power of the programming model. Login hours can be a good indication of the ease or difficulty programmers have in learning the model and using it to solve a problem using the tools provided by the PPS. Other statistics like the number of edits, compiles, program runs, particular tool invocations give a more detailed picture of how the users spend their development time. This can be used to infer the strengths and weaknesses of different PPS's.

3. THE SAMPLE PROBLEM AND ITS SOLUTION

The problem chosen for the experiment was the computation of a transitive closure. This problem was used in a graduate course in parallel computing at the University of Alberta during the previous year, but all the students used NMP. Although consideration was given to changing the problem (perhaps making it more challenging), we felt that a change might introduce a bias. As it turned out, although we do not regard this problem as hard, the students spent a large amount of time on it. Clearly a more challenging problem was not appropriate (with hindsight)! In future experiments, it would be better to choose a problem from a published test suite. However, the problem must be carefully selected to avoid a bias in favor of one tool over the other. The Salishan problems [2] are a good starting point, as are the Cowichan problems, a new test suite proposed by Wilson [9].

Essentially, the transitive closure program iterates until all values in a set have been assigned a value (a simplified version of [10] with an application-independent interface provided). Each iteration must traverse a graph using the information from the previous iteration to resolve additional data values. This problem has a straightforward solution where each processor is responsible for a subgraph, and the processes synchronize at the end of each iteration. It is possible to create a chaotic solution, where the processes do not synchronize, but this requires careful consideration of the termination conditions.

Two data sets were provided for the students. Data set 1 contained a problem with 100,000 nodes in the graph; data set two had 1,000,000 nodes. The first data set was used as a "simpler" problem to verify program correctness; the second was more compute-intensive and was used for timings. However, the second data set also had different execution properties so that a program tuned to perform well on data set 1 would not do well on data set 2 and visa-versa.

3.1 The Enterprise Solution

In Enterprise, the interactions of processes in a parallel computation are described using an analogy based on the parallelism in a business organization [3]. 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 *lines*, *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 by Enterprise. There are no library routines to call (as in NMP or PVM) and there are no language extensions or compiler directives. A brief description of Enterprise can be found in the Appendix.

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)`. A pending result that allows concurrent actions has been called a *future* [11].

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 by the pre-compiler into message send/receives across a network. The execution of the program (process/processor assignment, establishing communication links, monitoring network load) is done by the run-time executive.

In Enterprise, the transitive closure problem can be modeled as a line (pipeline) of two assets as shown in Figure 1. The double line rectangle represents the whole program (enterprise). The dashed-line rectangle represents the line asset and each inner icon represents a component. The first asset, called TC, is essentially a master process that divides the work into disjoint subsets of nodes to evaluate. The second asset, called Iterate, computes new values for each node in the subset it is assigned. Instead of having one Iterate process that sequentially traverses all the nodes in the graph, the asset is replicated eight times (as shown in Figure 1). At run-time, each of the eight replicas can traverse its subset of the problem concurrently using a separate process.

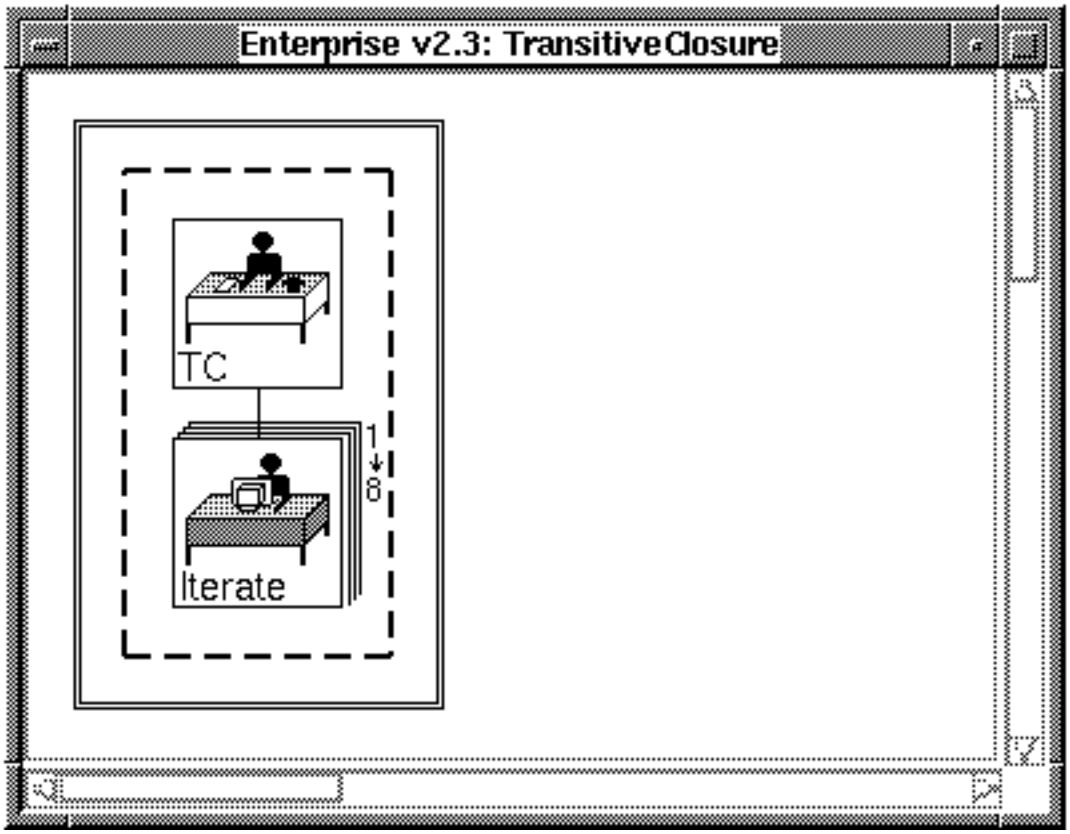

Figure 1. The Transitive Closure program in the Enterprise PPS.

When a user compiles a program, the Enterprise pre-compiler automatically inserts code to handle the distributed computation. 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.

3.2 The NMP Solution

The Network Multiprocessor Package (NMP) is a PVM-like message passing library built on top of sockets. Essentially it is a friendly interface to TCP and UDP. NMP provides the same basic facilities as PVM, except support for using heterogeneous processors and dynamic reconfiguration of processes is not provided.

Process/processor mappings are defined by a configuration file supplied by the user at run-time. Figure 2 shows a sample configuration file for the transitive closure problem. The user must explicitly map the processes to processors and indicate all communication paths. Processes are given ids, starting with zero for the first process in the configuration file, and sequentially thereafter. Connections are specified by a static Boolean connection matrix where each entry specifies whether a process can talk to another process or not. The user must make this available to the application prior to running it and must update it whenever the number and/or identity of available processors changes. In contrast, the Enterprise asset diagram need not change as run-time conditions vary.

```
# Processors, their executable, and in/out/err files
machine0; 0;
machine1; 0; Parallel/TC/a.out -slave ; ; out1; err1
machine2; 0; Parallel/TC/a.out -slave ; ; out2; err2
machine3; 0; Parallel/TC/a.out -slave ; ; out3; err3
machine4; 0; Parallel/TC/a.out -slave ; ; out4; err4
# Process connections:
# 1 indicates connection, 0 no connection.
0 1 1 1 1
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0
1 0 0 0 0
```

Figure 2. An NMP configuration file.

NMP provides a number of routines, of which the four most important are:

```
NodeInit(am_i_root,config_file);
SendNode(who,message_ptr,send_bytes);
bytes_received=ReceiveNode(who,message_ptr,rcv_bytes);
who_array=PollAll(block_or_not);
```

Each process calls `NodeInit` to initialize its connections, `SendNode` and `ReceiveNode` to communicate, and `PollAll` to query whether there are any messages waiting. There are a variety of other support routines.

Before the experiment began, consideration was given to whether PVM should be used in the experiment, instead of the locally developed NMP. The advantage of PVM is that it is widely known. However, NMP was chosen for several reasons:

1) NMP is a subset of PVM, and has less than 20 different library calls, most of which were not relevant for this experiment. PVM has more than 50 library calls which increases the learning overhead.

2) The documentation for NMP is less than 20 pages, simplifying the student's startup overhead. The manual for PVM is over 100 pages long.

3) NMP has been used in a graduate course for the past 5 years, with no known bugs. On the other hand, PVM is still under development and we have occasionally encountered bugs.

4. EXPERIMENTAL DESIGN

There are a number of considerations that must be taken into account in the design of a fair experiment to measure usability.

4.1 Prelude

In the preliminary planning stages of the experiment, we consulted a cognitive psychologist with expertise in designing experiments that involve human subjects. She provided us with important advice on how to conduct the experiment to avoid introducing biases. In particular, it was important that the students not know the exact nature of what was being measured in the experiment. This point had several important implications to the design of the experiment. In this case, the students were only told that they would provide us with a subjective evaluation of the tool they were using and were not told that any measurements were being taken.

4.2 Subjects

The students in the CMPUT 507 Parallel Programming graduate course at the University of Alberta were used as subjects. None of them had any previous parallel programming experience prior to taking this course. Before the experiment, the students completed assignments to program a vector processor machine and a distributed memory multiprocessor (with C/Fortran language extensions for doing loops in parallel - Myrias Parallel C/Fortran [12]).

In general, selecting subjects for an experiment can be a difficult task [8]. The experimenter would like the subjects chosen to be 1) *representative* so that the results from a small sample can be used to make a statement about a larger population and 2) relatively *uniform* in their abilities and experience. However, these characteristics become contradictory as the parent population becomes more heterogeneous. Using students in a graduate course and controlling their introduction to parallel computing helps reduce some of the concerns about uniformity. However, as evidenced by the results presented in Section 5, there was a wide range in the programming abilities of the students. This is not surprising since differences in the abilities of students in a programming course have been reported before (see, for example [13]). The advantage of a larger population size is that these differences tend to distort the results less. Unfortunately, although we would have liked to have more students in the experiment, the enrollment in the course dictated our sample size.

4.3 Partitioning

The class of 15 students was divided into 2 groups, one using NMP and the other using Enterprise. The assignment of students to groups was done randomly, with 7 students in the Enterprise group and 8 in the NMP group.

4.4 Instruction

Two fifty minute lectures were given to the entire class, one on NMP and one on Enterprise. Each lecture described how the programming model could be used to define processes, process interconnections and inter-process communication. Students were provided with documentation for both systems. They were then assigned to the test groups.

A lab demonstration of each PPS was also presented. A simple program was demonstrated that computes the squares and cubes of numbers in parallel. Although this problem is trivial and does not have the granularity to justify a parallel implementation, its simplicity allowed the students to concentrate on the parallel programming issues, rather than the sequential algorithm. Each system was demonstrated for 20 minutes, illustrating how to convert a sequential program into a parallel one. The Enterprise and NMP solutions were then made available to the students for reference.

4.5 Student Help

In addition to the instructor, a teaching assistant who was familiar with both NMP and Enterprise was available to answer student questions. All queries for help were logged with the name of the student, the date and the nature of the question asked. Students were encouraged to discuss the assignment with each other, but they were expected to do the assignment individually.

4.6 Environment

Students had access to 50 Sun 4 workstations that were shared with several hundred undergraduates. Students were not allowed to use the machines during prime time and were restricted to a single processor for program development and testing, and a maximum of 8 machines for timing runs. During the final three days before the assignment was due, the students were given dedicated access to the machines from midnight to 8 am so that they could obtain accurate timings of their executions.

Each student account was set up to use a modified *zsh* shell (*zsh* is a public domain shell which is compatible with the Bourne shell but has C-shell job control enhancements). The shell was modified to log all commands executed by the students (date, time, command, and exit status of the command). Enterprise was also instrumented to record all editing, compiling and execution actions. The students were not told about the instrumentation. This is an important point since subjects who know about instrumentation may consciously or subconsciously modify their behavior. For example, a subject may try to avoid multiple compilations if it is known that compiles are being counted. On the other hand, it is necessary to conduct these experiments in an ethical manner so that the privacy of the subjects is not violated. The instrumentation only counted statistics directly connected to the PPS as opposed to monitoring the contents of private files such as mail files. That is, we only gathered statistics similar to those obtained by the Unix *lastcomm* utility. If the *lastcomm* utility was enabled it would have logged all (several hundred) system users, not just the (15) users involved in the experiment. We could not do this due to disk-space limitations.

4.7 Time Frame

Students were given two weeks to complete the assignment.

4.8 Epilogue

At the experiment's conclusion, students were asked to submit a two-page write-up commenting on their respective PPS, and were encouraged to be blunt about what they liked and disliked.

5. EXPERIMENT RESULTS

Prior to the experiment, we tried to anticipate events that might influence the results. For example, it was known that the NMP documentation was superior to that for Enterprise. Also, since Enterprise was an evolving system, it was inevitable that students would find bugs. The question was: how much would these factors bias the results? Intuitively, one would expect that since Enterprise is a higher-level tool, the Enterprise students would write less code and develop a working program more quickly than the NMP students. However, since NMP is a low-level tool, the NMP students would have more implementation alternatives. Enterprise has certain run-time overheads (such as larger message sizes, hidden manager processes and additional messages). These points suggested that NMP should have better run-time performance.

How does one objectively measure usability? Intuitively, one PPS is more usable than another if it is easier to solve a problem. Our experiment measured five factors that seem to be indirect measures of usability as well as one factor (run-time performance) that may be traded off against increased usability. Figures 3 to 8 shows the six statistics that were analyzed. In the first five cases, a lower number should indicate higher usability, while in the sixth case, a lower number indicates better run-time performance.

1. The number of hours each student was logged in, actively working on the assignment (Figure 3). Idle periods of more than 15 minutes were not included.
2. The number of lines of code in the solution program, including blank lines and comments (Figure 4). Students were given the sequential program (128 lines of code) and were expected to parallelize it. They were also given a library containing the parts of the program that did not have to be altered (over 1000 lines of code). The figure shows the parallel code written less the 128 lines of sequential code.
3. The number of editing sessions (Figure 5).
4. The number of compiles that attempted to link the program together (Figure 6). Compiles which failed because of syntax errors were not included.
5. The number of times the students tested their parallel program by running it (Figure 7).
6. The execution times of their program on data set 2 (Figure 8).

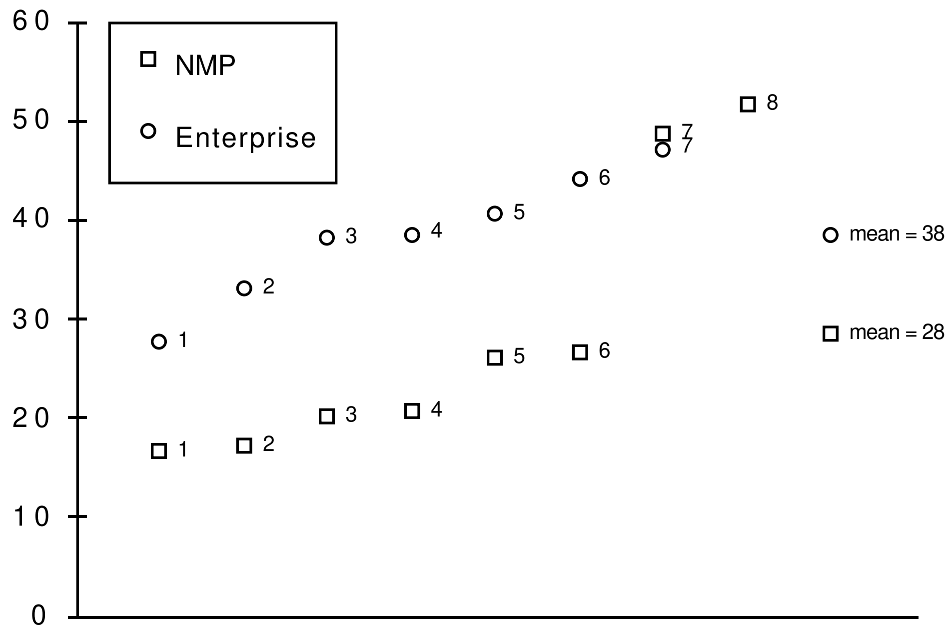

Figure 3. Login Hours

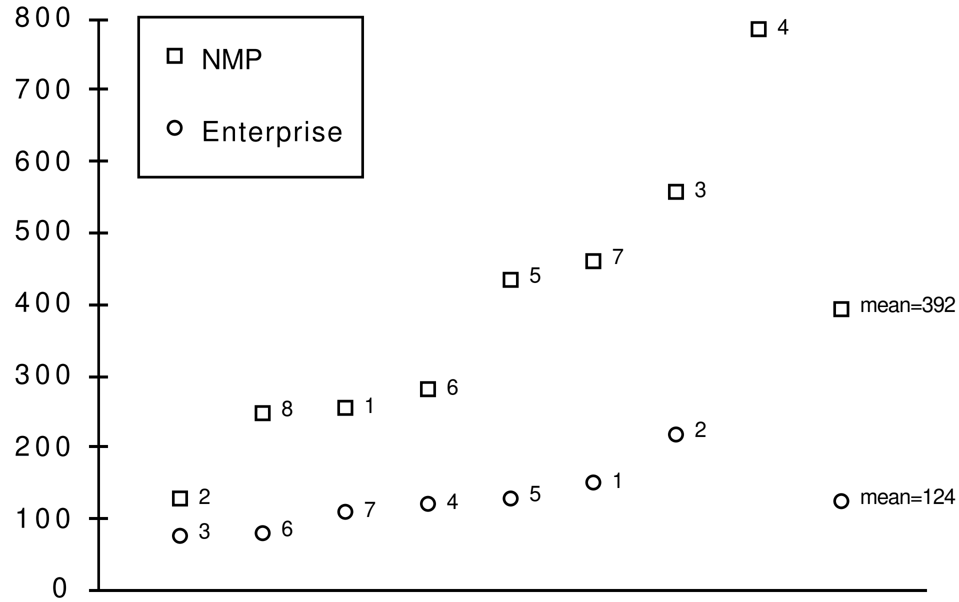

Figure 4. Lines of Code

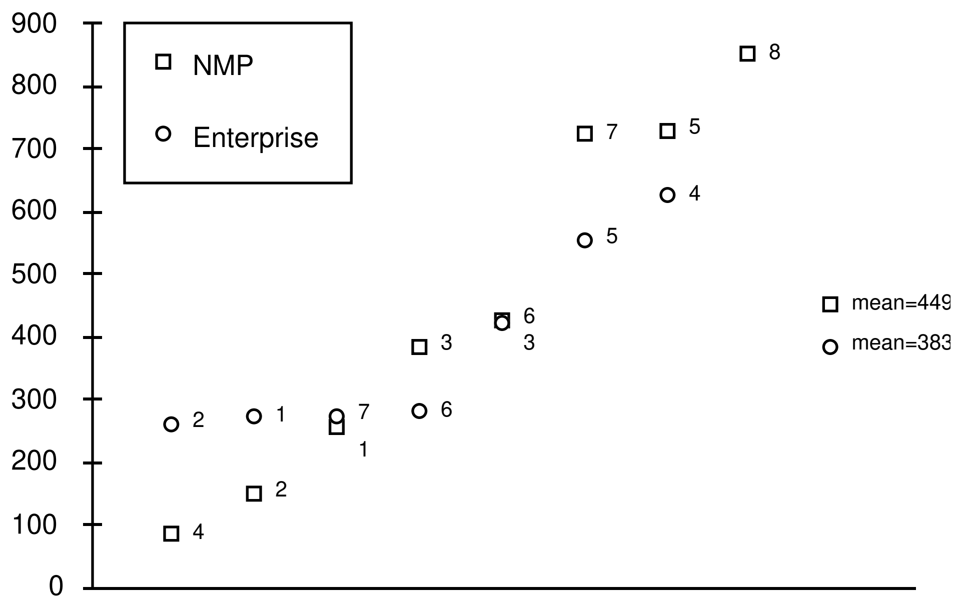

Figure 5. Number of Edits

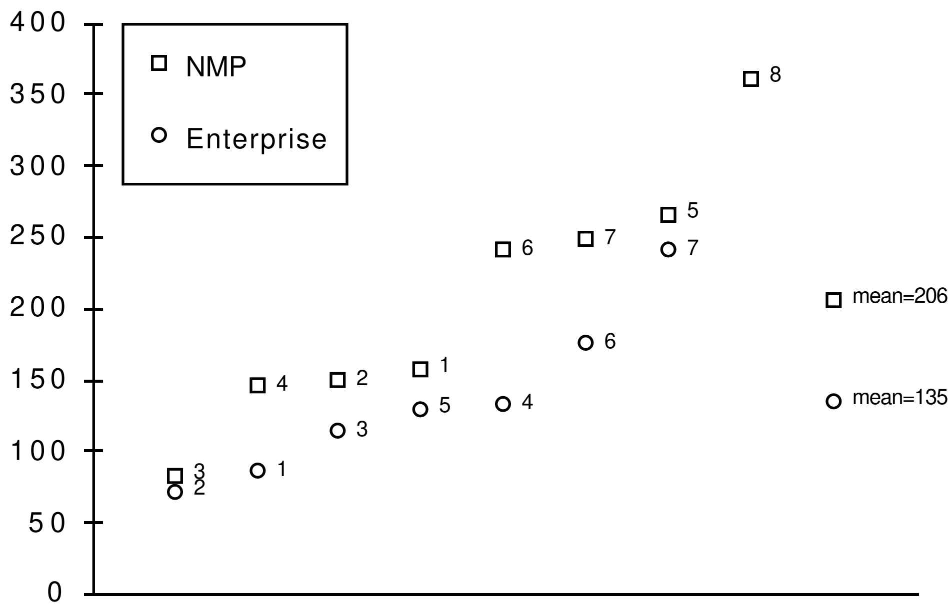

Figure 6. Number of Compiles

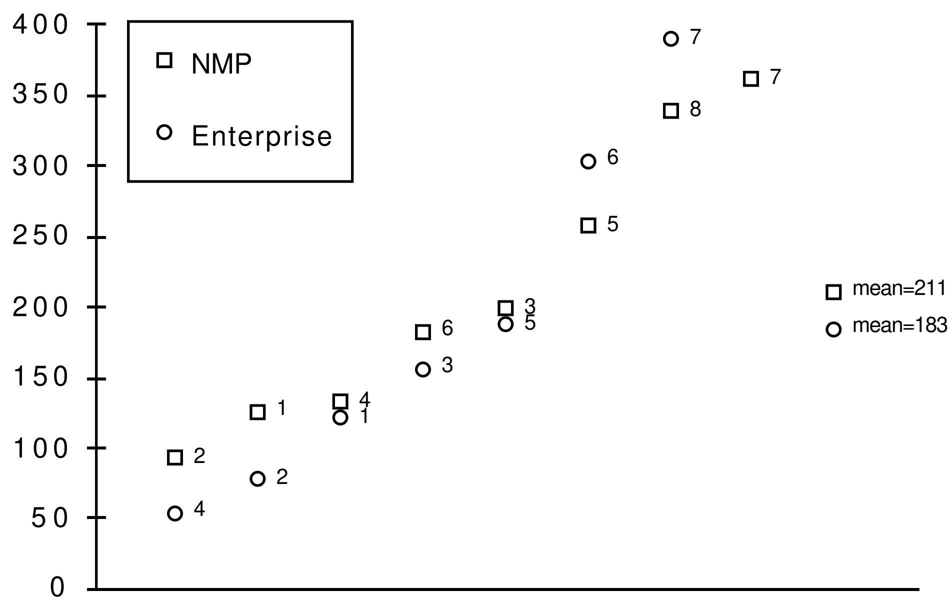

Figure 7. Number of Program Runs

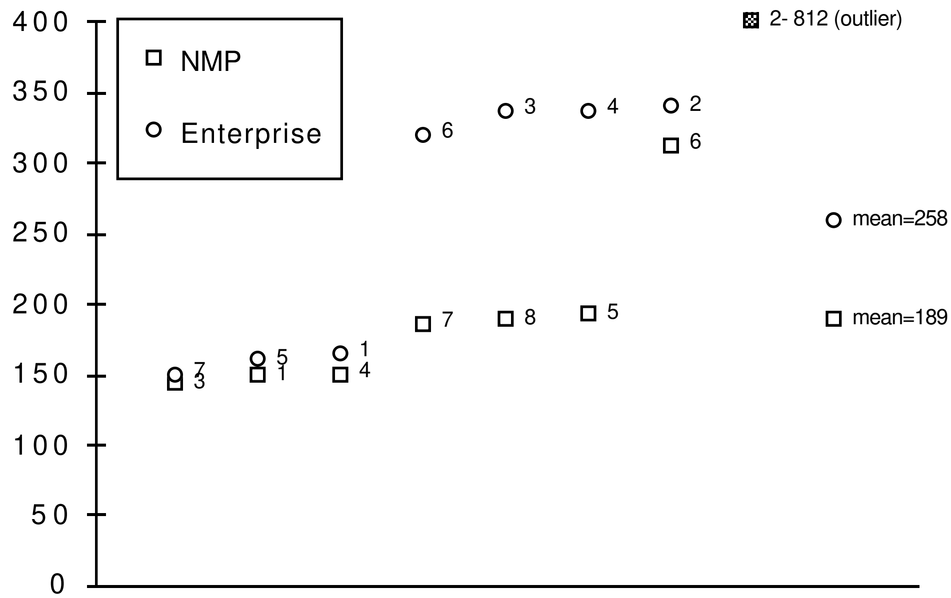

Figure 8. Parallel Performance in Seconds

In each figure, the squares represent NMP data points (8 students) and the circles represent Enterprise data points (7 students). Each student is given a number, so the reader can compare an individual's performance across graphs. These graphs are ordered with the best performer on the left and the worst on the right. For example, Figure 3 shows that Enterprise student 2 spent the second least amount of time working on the assignment, but Figure 4 shows that this student generated the largest program of any of the Enterprise students. The rightmost points in each figure shows the means of the NMP and Enterprise students. Note that all points are included in the means except for the NMP-2 student's performance in Figure 8 and Table 2.

The statistics support our initial expectations that students would do less work (higher usability) with Enterprise, but get better run-time performance with NMP. Table 2 summarizes the statistics that were computed from the data shown in Figures 3 to 8. The table contains the mean (μ) and standard deviation (σ) for each of the six values measured. It also contains the difference of the means and a 90% confidence interval for this difference. That is, we can state with 90% confidence that the difference in means lies in the interval given. However, if the interval contains the value zero then we cannot claim that there is a difference in the means for the NMP and Enterprise students. Since we have small sample sizes in this experiment, we expected large variances and have chosen 90% confidence intervals. If more students were involved we would have tried to make statements with 95% confidence.

The formula used to compute these confidence intervals is:

$$\overline{X}_1 - \overline{X}_2 \pm t_\alpha \sqrt{\frac{n_1 S_1^2 + n_2 S_2^2}{n_1 + n_2 - 2}\left(\frac{1}{n_1} + \frac{1}{n_2}\right)}$$

where the $\overline{X}_i$ are the sample means, the n_i's are the sample sizes, the S_i's are the sample variances and t_α is the percentile of the t-distribution with confidence value α and $(n_1 - n_2 - 2)$ degrees of freedom [14]. Note that $\alpha = 0.01$ for a 90% confidence interval.

	Login Hours	Lines of Code	Number of Edits	Number of Compiles	Number of Runs	Performance in seconds
μ_{NMP}	28	392	449	206	211	189
σ_{NMP}	14	210	288	89	100	58
μ_{Ent}	38	124	383	135	183	258
σ_{Ent}	7	48	152	58	122	94
$\mu_{NMP} - \mu_{Ent}$	-10 ± 8	268 ± 110	66 ± 164	71 ± 53	28 ± 77	-69 ± 54

Table 2. Statistics for the Experiment.

Nothing statistically significant can be stated about the number of edits and program runs done by Enterprise students and NMP students. However, NMP student wrote 158 to 378 (127% to 305%) more lines of code than Enterprise students. They also did from 18 to 124 (13% to 92% more compiles. Given that Enterprise students wrote less code and did fewer compiles, why did they use 2 to 18 more hours (6% to 65%) of login time? There are several reasons:

1) Enterprise compiles took roughly 4 times as long as the regular C compiles used by the NMP students. Enterprise preprocesses the user's code by making several passes over the input file before producing a file that is compiled by the C compiler. From Table 2, the average NMP user compiled 7.4 times per hour, while the average Enterprise user compiled only 3.6 times per hour.

2) Enterprise includes an option to replay a computation using animation, so that the user can examine the messages being sent and see the status of each process [15]. If the user watches an animation of the transitive closure program to completion using the default settings, it could take as long as 10 minutes. Each Enterprise user, on average, used this feature 25 times.

3) The students uncovered nine bugs in Enterprise; two of them serious errors that affected the student's progress. Although turnaround on bug fixes was rapid (less than 24 hours), most students assumed that the bug was in their program and not in Enterprise. We do not know how much time they devoted to solving these problems before they reported them.

4) Since the NMP performance was better, Enterprise students spent more time doing performance tuning to try to obtain better speed-ups.

5) The Enterprise students had to use a graphical user interface whose response time was very slow due to memory limitations.

Never having done an experiment like this before, we were quite unprepared for the variation and magnitude of the numbers. For example, the average NMP student performed 206 compiles (excluding compiles with syntax errors). To us, this was an astonishingly large number. We are not sure how to interpret this: was the assignment too difficult, is it a comment on our student's programming ability, or is it a comment on their difficulty in learning distributed computing?

Enterprise solutions required considerably fewer lines of code to be written than did their NMP counterparts. Superficially, this appears to be a strong endorsement for a higher-level programming tool. However, this conclusion must be qualified. The implication is that fewer lines of code implies fewer errors. It is possible that properly-designed and clearly-written code may take more lines, contain fewer errors, take less time to write and debug, and run more efficiently. An instance of this can be seen in Figures 4 and 8, where the smallest NMP program (126 lines of code) had the worst run-time performance (NMP-2), but the largest (783 lines) was one of the fastest (NMP-4)! The strength of a high-level programming tool, such as Enterprise, is not that the user writes fewer lines of code. Rather, it is that the time saved during coding can be used in the design of a parallel solution, without being distracted by unnecessary details.

We used lines of code in a program as a simple measurement of effort expended by the programmers, as is often done in many industrial settings. More complex measurements could have been used like Halstead's effort equation [16] or one of many other metrics [17]. However, automating these metrics will require some additional parsing support. We are considering this change for a future experiment.

As expected, the Enterprise solutions (excluding the anomalous NMP-2 data point) had worse run-time performance, taking from 15 to 123 seconds (8% to 65%) longer. For this problem, the Enterprise communication time could be as high as 30% of the execution time depending on how the problem was solved. Since Enterprise inserts hidden manager processes that forward messages to replicated assets, there could be twice as many messages as in a hand-coded NMP solution, where the master process communicates directly with its slaves. In addition, at least two of the Enterprise solutions had bugs in them whereby two futures overlapped, forcing sequential execution where concurrent execution was intended.

Enterprise claims that it lets users avoid common parallel programming mistakes such as deadlock, synchronization errors and failure to consume all messages. From the user feedback, it appears this claim is partially justified: one NMP student had problems with deadlock while two had problems coordinating messages. However, the Enterprise model introduced different parallel programming mistakes that the students might stumble on, such as the overlapping futures problem mentioned above. Superficially, it appears that Enterprise has traded one problem for another. However, it is important to note that the Enterprise problems only affected the *efficiency* of the code, while the NMP problems affected the *correctness* of the code.

Why are the numbers in Figures 3 to 7 so large? This assignment was a learning experience for the students, and as such they did a lot of experimenting. For example, log files show that some students did not parameterize their code for the number of processors used; they compiled separate versions for 1, 2, 3, ... 8 processors, increasing the number of edits and compiles. There was also a lot of experimenting with granularity. The programming problem intentionally had the property that the performance parameters that the user might tune for best results on data set 1 did not give good performance for data set 2.

All the results presented must be qualified by the sample size used for this experiment. Fifteen students yields large confidence intervals. Unfortunately, although we wanted more students in the experiment, this is the typical size of our graduate courses. It is not likely that any future experiments will have a larger sample size.

Table 1 identified a number of usability issues. A single experiment with novice parallel programmers was unable to address all these issues. The usability issue that received the most attention was the probability of reducing programming errors. As discussed earlier, the higher-level tool did a better job here since, obviously, Enterprise inserted all the communication and synchronization code into the application code for the user. Another issue which the students found important for debugging was deterministic execution. The Enterprise programmers found debugging their application code easier because of the deterministic execution and animation facility. One NMP programmer spent a great deal of their time (and the instructor's) chasing down a non-deterministic error in their program. This bug adversely affected all the students in the class for several days because of the large core dumps it generated.

One usability issue that we attempted to address in the experiment was to quantify the learning curve. Software development has at least three distinct phases: the learning phase (the time to familiarize yourself with the tool and its capabilities), the development phase (the time taken to get a correct working program) and the performance phase (the time taken to achieve the desired level of performance). Tool usability may be different for each of these phases. For example, just because a tool is easy to learn does not mean that it can be easily used to transform a correct program into a more efficient one. For this experiment, we defined milestones for the students and they were supposed to report in when a milestone was achieved (roughly corresponding to the completion of the three phases). It was our hope to quantify the time taken by the students to learn the tool, obtain a working solution and their performance tuning efforts. Unfortunately, more than half of the students in the class failed to report when they reached the specified milestones.

During the experiment, the students asked many questions and made many comments that highlighted the conceptual difficulty of several key concepts in parallel computing. The most important new non-sequential concepts are process startup, process termination and passing pointers between processes. What special operations, like variable initializations, need to be done when a process (or asset) is called the first time, versus when the process is called subsequently? How are termination conditions checked and how should the processes (assets) exit gracefully? Since distributed programs have separate address spaces on different processes, how should the source code be changed to pass pointer data structures between processes? These concepts are important and should be specifically highlighted in the documentation for all PPSs.

6. FUTURE EXPERIMENTS

The experiment revealed two important deficiencies in the experimental design itself, both of which are easily corrected. First, since usability involves the time it takes to obtain a solution with a certain performance, the experiment must define performance milestones and all measurements must be grouped based on these milestones. For example, statistics should be reported separately on the time taken to learn the tool, the time taken to obtain a working solution and the time spent tuning for performance. As mentioned previously, we were unsuccessful at gathering this information. The instrumentation must be changed to ensure these statistics are gathered.

Second, the experiment was conducted using inadequate hardware resources. The amount of main memory available made the Enterprise graphical user interface too slow. In addition, the workstations were shared with undergraduates so it was difficult to take accurate performance readings except at night. In general, hardware resources should be provided that adequately meet the requirements of all PPSs being evaluated so that no biases are introduced.

Ideally, a future experiment would be enlarged to include a larger student population, more than one programming problem and more PPSs (an experiment under consideration involves comparing PVM, Enterprise, Linda [18] and PAMS [12]). As well, it would be interesting to do these experiments using both novice and expert parallel programmers. It is not obvious which of the results obtained using novice parallel programmers carry over to experienced parallel programmers.

One interesting extension to the experiment that will improve the significance of the experimental results, would be to establish some control data points. In CMPUT 507, students did two parallel programming assignments (vector processing and Myrias) before the experiment. Data gathered from these two assignments would allow us to better understand the student's learning curve, and help us put each student's result in the proper perspective. It might provide some insight into learning effects, particularly if the experiment were redesigned so that we changed the order in which students were taught the PPSs. For example, for historical reasons, vector processing has always been taught first in CMPUT 507. Would the student's learning curve be different if they were taught PVM first?

Finally, having done one experiment, it may be difficult to conduct future experiments that are unbiased. We have acquired a "reputation" among the graduate students in our department. Since the student body in general now knows about the experiment and the factors that were measured, this may consciously or subconsciously bias participants in future experiments. For example, although we did not associate names with any of our data points, some students felt uncomfortable (and even embarrassed) when they were told of the experiment results. Perhaps these students were able to identify which data point were theirs knowing, for example, that they performed a lot of compiles. We suspect that future students taking this course will be conscious about the number of compiles and program executions they attempt.

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 measurements that flatter the performance of their machines (MIPS, SPECmarks, Whetstones, etc.), but neglect to quantify the usability of their software. The growing base of parallel computing users could significantly benefit from an objective assessment of the usability of PPSs.

One can question the results of a single experiment with a small sample size. Nevertheless, a number of conclusions can be drawn from this experiment:

1. **It demonstrated that the usability of PPSs can be measured objectively**. It was not difficult to set up a controlled experiment. Having invested the time in this first experiment (such as building an instrumented environment), future experiments will involve less overhead. Periodically obtaining objective feedback on tool usability can be of significant benefit to the design and implementation of a PPS.
2. **It identified some fundamental PPS independent concepts that are difficult for most novice parallel programmers to understand**. While these concepts are well understood by practitioners, they are not well understood by novices. These concepts should be integral considerations in the design of a PPS and a fundamental part of the PPS documentation.
3. **It supported the claim that higher-level tools can be more usable than low-level message passing libraries**. Although this point is not a surprise, it does serve to emphasize that there are significant productivity gains possible through the use of high-level tools. Further, since Enterprise is still under development and the bugs that may have affected the results in this experiment have been fixed, one can expect a subsequent experiment to reveal a more decisive advantage for the high-level tool. Of course, the question the parallel/distributed user community has to ask themselves is, why are the majority of system developers still using tools like PVM? This is probably not a good reflection on the state-of-the-art of PPSs.

In addition, the experiment provided valuable feedback to the tool developers. In other words, this experiment was more than just an academic exercise; it forced us to address issues that were important to the user community, but of lesser importance to our systems development community.

4 **It produced several direct benefits to the Enterprise PPS**. This issue is important for any designer of a PPS. User feedback helps identify potential problem areas in the programming model, user interface and run-time performance. For Enterprise, a number of major problems were identified. We must consider the possibility of adding a textual interface since many students wanted to work at home and did not have an X-windows interface. The statistics indicate that the speed of the Enterprise pre-compiler must be improved. Student comments caused us to increase our priority on debugging tools. Program solutions revealed some subtle flaws/omissions in the programming model. Finally, the students identified major weaknesses in the Enterprise documentation. Except for the documentation, we were not aware of any of the other problems. We have used this information to alter some of our implementation directions.

High-level parallel programming systems cannot provide an all-encompassing general-purpose solution for all parallel applications. However, they usually allow an important class of problems to be solved more quickly. The time required to solve a particular problem is not just a function of the high-level programming model, it is also a function of the usability of the tools provided by the PPS. The programmer time saved by using the abstractions of a PPS can be more than offset by the time spent fighting with a poorly designed tool. 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 a high-level tool has a high degree of usability. However, until scientific experiments are done to compare PPSs, anyone's claim is as valid as any other.

Although this is only a first attempt at measuring the usability of PPSs, the experiment nevertheless highlights the human factors issues that have been neglected to date. We propose that the above experiment (or variations on it) should be an integral part of the development cycle for parallel software tools. Given the diversity of programming systems available, researchers need more feedback as to 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 a low-usability PPS, will certainly be much larger.

ACKNOWLEDGMENTS

We would like to thank the students in CMPUT 507 for their cooperation. Renee Elio, Randal Kornelsen, Ian Parsons, Paul Iglinski, Robert Lake, Carol Smith and Bob Beck helped make this experiment possible. Many of the ideas in this paper originated from discussions with Greg Wilson. Paul Lu and Greg Wilson gave valuable feedback on earlier drafts of this paper. Some of this research was done while one of the authors (JS) was a Visiting Professor at the Department of Computer Science, University of Limburg, Maastricht, The Netherlands. This research has been funded in part, by NSERC grant OGP-8173, a grant from IBM Canada Limited's Centre for Advanced Studies, and the Netherlands Organization for Scientific Research (NWO).

REFERENCES

[1] G. Wilson, J. Schaeffer and D. Szafron. Enterprise in Context: Assessing the Usability of Parallel Programming Environments. *IBM CASCON*, Toronto, pp. 999-1010, 1993.

[2] J. Feo (editor). *Comparative Study of Parallel Programming Languages: The Salishan Problems*. North-Holland, Amsterdam, 1993.

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

[4] 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.

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

[6] E. Soloway and I. Sitharama (editors). *Empirical Studies of Programmers*. Ablex, Norwood N.J., 1986.

[7] M. Rao, Z. Segall and D. Vrsalovic. Implementation Machine Paradigm for Parallel Processing. *Supercomputing '90*, ACM Press, New York, pp. 594-603, 1990.

[8] R. Brooks. Studying Programmer Behavior Experimentally: The Problems of Proper Methodology. *Communications of the ACM,* vol. 23, no. 4, pp. 207-213, 1980.

[9] G. Wilson. Assessing the Usability of Parallel Programming Systems: The Cowichan Problems. *Programming Environments for Massively Parallel Distributed Systems*, Karsten Dekker and Rene Rehmann (Editors), Birkhauser Verlag, Basel, Switzerland, pp. 183-193, 1994.

[10] R. Lake, J. Schaeffer and P. Lu. Solving Large Retrograde Analysis Problems Using a Network of Workstations. *Advances in Computer Chess 7*, H.J. van den Herik, I.S. Herschberg and J.H.W.M. Uiterwijk (editors), University of Limburg, Maastricht, The Netherlands, pp. 135-162, 1994.

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

[12] M. Beltrametti, K. Bobey, R. Manson, M. Walker and D. Wilson. PAMS/SPS-2 System Overview. Supercomputing Symposium, pp. 63-71, 1989.

[13] B. Schneiderman, R. Mayer, D. McKay and P. Heller. Experimental Investigation of the Utility of Detailed Flowcharts in Programming. *Communications of the ACM*, vol. 20, no. 6, pp. 373-381, 1977.

[14] B. W. Lindgren. *Statistical Theory*, Macmillan, 1968.

[15] 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.

[16] M. Halstead. *Elements of Software Science*, North-Holland, 1977.

[17] N. Fenton. *Software Metrics: A Rigorous Approach*. Chapman and Hall, London 1991.

[18] N. Carriero, D. Gelernter, T. Mattson and A. Sherman. The Linda Alternative to Message-passing Systems. *Parallel Computing*, vol. 20, no. 4, pp. 633-655, 1994.

[19] 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.

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

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

[22] Z. Segall and L. Rudolph. Pie (A Programming and Instrumentation Environment for Parallel Processing), *IEEE Software*, vol. 2, no. 6, pp. 22-37, 1985.

[23] A. Beguelin, J. Dongarra, G. Geist, R. Manchek and V. Sunderam. Graphical Development Tools for Network-Based Concurrent Supercomputing. *Supercomputing '91*, pp. 435-444, 1991.

APPENDIX: A Brief Description of Enterprise[2]

Enterprise is a programming environment for designing, coding, debugging, testing, monitoring, profiling, and executing programs for distributed hardware. Developers using Enterprise do not deal with low-level programming details such as marshalling data, sending/receiving messages, and synchronization. Instead, they write their programs in C, augmented by new semantics that allow procedure calls to be executed in parallel. Enterprise automatically inserts the necessary code for communication and synchronization. There are four common approaches for expressing parallelism: designing a new language (such as ORCA [19]), extending an existing language with parallel constructs (such as HPF [20]), providing library calls (such as PVM [4]) or using templates [21]. Enterprise is based on templates, uses standard C and has no language extensions or library calls.

Enterprise does not choose the type of parallelism to apply. The developer is often the best judge of how parallelism can be exploited in a particular application. Enterprise lets the programmer draw a diagram of the parallelism using a familiar analogy that is inherently parallel: a business organization, or enterprise, which divides large tasks into smaller tasks and allocates *assets* to perform those tasks. These assets correspond to techniques used in most large-grained parallel programs — pipelines, master/slave processes, divide-and-conquer, and so on — and the number and kinds of assets used determine the amount of parallelism (similar to Pie [22]). For example, an individual performs a task sequentially, but four individuals can perform four similar tasks concurrently. A department can subdivide a task among components that can then perform the subtasks concurrently. An assembly or processing line can start a second task as soon as the first component of that line has passed the first task to the second component.

Most parallel/distributed computing tools require the developer to draw communication graphs in which nodes (processes) are connected by arcs (communications paths) (such as Hence [23]). In Enterprise, a programmer constructs an organization chart from the top down, expanding assets to explore the application's hierarchical structure, and transforming assets from one type to another to specify how they communicate with their neighbors.

Based on the organization chart, Enterprise inserts parallel constructs and communications protocols into the code, compiles the routines, assigns processes to processors, establishes the necessary connections, launches the processes, executes the program, and (when desired) logs events for performance monitoring and debugging. Because the (sequential) code that calls the parallel procedures is independent of those procedures (although the code generated by Enterprise certainly is not), programmers can adapt applications to varying numbers and types of processors without rewriting their code. Thus, they can experiment with different kinds of parallelism, construct recurring parallel structures more easily, generate code more quickly, and reduce errors.

---

[2] This appendix is a summary of a more complete description of Enterprise which can be found in [3].