A pilot study to compare programming effort for two parallel programming models

Lorin Hochstein [a,*], Victor R. Basili [b], Uzi Vishkin [c], John Gilbert [d],

[a] *University of Nebraska, Lincoln, Department of Computer Science & Engineering*

[b] *University of Maryland, Computer Science Department*

[c] *University of Maryland, Institute for Advanced Computer Studies*

[d] *University of California, Santa Barbara, Computer Science Department*

Abstract

Context. Writing software for the current generation of parallel systems requires significant programmer effort, and the community is seeking alternatives that reduce effort while still achieving good performance.

Objective. Measure the effect of parallel programming models (message-passing vs. PRAM-like) on programmer effort.

Design, Setting, and Subjects. One group of subjects implemented sparse-matrix dense-vector multiplication using message-passing (MPI), and a second group solved the same problem using a PRAM-like model (XMTC). The subjects were students in two graduate-level classes: one class was taught MPI and the other was taught XMTC.

Main Outcome Measures. Development time, program correctness.

Results. Mean XMTC development time was 4.8 hours less than mean MPI development time (95% confidence interval, 2.0-7.7), a 46% reduction. XMTC programs were more likely to be correct, but the difference in correctness rates was not statistically significant (p=.16).

Conclusions. XMTC solutions for this particular problem required less effort than MPI equivalents, but further studies are necessary which examine different types of problems and different levels of programmer experience.

Key words: MPI, XMT, message-passing, PRAM, empirical study, parallel programming, effort

* Corresponding author.
Email addresses: lorin.hochstein@ieee.org (Lorin Hochstein),

1 Introduction

While desktop computers today are very powerful, there remain many computational tasks of interest that conventional computers cannot complete in a reasonable time. Such tasks are especially common in the domain of computational science, where physical phenomena (e.g., nuclear reactions, earthquakes, planetary weather and climate) are studied through computer simulation. For these problems, scientists must turn to high-performance computing (HPC) systems. These systems are able to provide more processing power than conventional systems through parallelism: by connecting many processing units together in parallel, such HPC systems are able to obtain much greater performance, at least in principle. In practice, it can be difficult to achieve performance gains on HPC systems because of the complexities involved in implementing efficient parallel programs. While the challenges of parallel programming have have traditionally been a concern for the HPC community alone, the rise of multicore architectures is making the parallel programming challenge increasingly relevant to all programmers[38].

Programmers must specify parallelism explicitly in their source code to take advantage of HPC machines. Researchers have proposed many different *parallel programming models* to express parallelism. It is through the programming model that the programmer specifies how the different processes in a parallel program coordinate to complete a task. Many models have been proposed, with corresponding implementations as libraries, extensions of sequential languages (e.g. C, Fortran), and new parallel languages. These models include: message-passing[16,37], threaded[15,31,28,34]), partitioned global address space (PGAS)[9,32,43], data-parallel[5,11], dataflow [17], bulk synchronous parallel (BSP)[22], tuple space[27] and parallel random access memory (PRAM)[41,26].

The pilot study in this paper addresses the following research question: would a PRAM-like system offer measurable benefits over alternative parallel systems? We conducted a study in an academic setting to compare the time required to solve a particular programming problem using the XMTC[2] extensions to the C language (which supports a PRAM-like model) versus using the MPI[16] library (which supports a message-passing model).

`basili@cs.umd.edu` (Victor R. Basili), `vishkin@umd.edu` (Uzi Vishkin), `gilbert@cs.ucsb.edu` (John Gilbert).

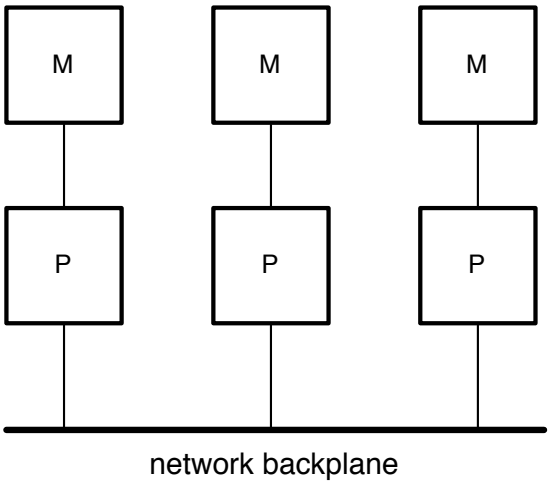

Fig. 1. Message-passing model of a parallel computer

1.1 Message-passing with MPI

In the message-passing model, the parallel machine is modeled as a set of processing elements that each have their own bank of addressable local memory. The processing elements are connected to each other over a network. Figure 1 depicts this model: boxes labeled P are processing elements and boxes labeled M are memory banks. Processing elements coordinate to complete tasks by exchanging messages over the network.

The MPI library is one implementation of the message-passing model with bindings to languages such as Fortran, C and C++. When an MPI program runs, a fixed number of processes are launched on the parallel machine, where each process is typically assigned to a separate processor. Each process has a unique ID, which can be retrieved with a function call. Programmers use *send* and *receive* function calls to communicate among the different processes. There are six basic function calls in MPI:

- *MPI_Init* - initialize MPI environment (called at beginning of program)
- *MPI_Finalize* - clean up MPI environment (called at end of program)
- *MPI_Comm_size* - returns total number of processes
- *MPI_Comm_rank* - returns ID of the current process
- *MPI_Send* - send a message to another process
- *MPI_Recv* - receive a message from another process

While these six calls are sufficient to implement any message-passing program in MPI, many other functions are provided for convenience and which may provide better performance than the basic send/receive calls. They include different types of send/receive calls (buffered vs. unbuffered, blocking vs. non-blocking), multipoint communications (e.g. broadcast, scatter, gather), reduc-

Listing 1. MPI code

```
#include <mpi.h>
#include <stdio.h>
#define N 3

int main (int argc, char *argv[]) {
  int my_id, num_procs;
  int data[N];
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&my_id);
  MPI_Comm_size(MPI_COMM_WORLD,&num_procs);
  printf("Hello from process %d of %d\n",
         my_id, num_procs);
  /* Send data from process 0 to process 1 */
  if(my_id==0) {
    data[0]=1;data[1]=3;data[2]=5;
    MPI_Send(data,N,MPI_INT,1,0,MPI_COMM_WORLD);
  } else if (my_id==1) {
    MPI_Recv(data,N,MPI_INT,0,0,
             MPI_COMM_WORLD,MPI_STATUS_IGNORE);
}
 MPI_Finalize();
 return 0;
}
```

tion operations (e.g. sum, product, maximum), barrier operations, and timing functions for performance analysis.

Listing 1 shows an example of a simple MPI program that prints out the process ID of each process and then sends an array of integers from process 0 to process 1.

The great strength of the MPI model is that it maps well to a broad range of parallel systems in use today. While there are some shared memory systems where the time to access any memory address is the same for all processors, most HPC systems are either distributed shared memory machines (where processors can directly access all memory, but some accesses are faster than others), clusters (where processors have their own local memory and are connected together over a local area network) or hybrids. Accessing local processor memory is typically much faster than accessing remote memory or communicating over the network. Because MPI gives the programmer low-level control of communication, it allows programmers to exploit locality: they can write programs that minimize communication overhead, thereby avoiding costly remote memory accesses or network communications. Because of its versatility,

MPI is currently the most widely used parallel programming method on HPC systems.

While MPI is the most popular parallel programming technology in terms of number of users, it is not well-liked. MPI is considered difficult to program compared to serial programming. In particular, MPI forces programmers to work at a very low-level of abstraction to deal with many of the communication details. Several reports commissioned by the U.S. government have pointed out the challenges of programming today's parallel systems with MPI [4,19,21,25].

1.2 PRAM-like with XMTC

The PRAM model [18] is a generalization of the Random Access Machine (RAM) model, the basic sequential computing model exposed to programmers in traditional programming languages. Figure 2 depicts this model; although only a fixed number of processors is shown, in the PRAM model, the PRAM theory permits assuming an unbounded collection of RAM processors in a PRAM algorithm, as this will be readily translated to a fixed number. The memory can also be assumed to have an unbounded collection of memory cells, which are accessible to all processors in unit time. The main difference between a sequential program and a parallel program using the PRAM model is the existence of parallel *for* loops, where each iteration of the loop is executed in parallel on a separate processor. This is typically referred to as a pardo-loop, short for *parallel do*.

When executing parallel loops, all processors execute the loop instructions *synchronously*. The synchronous execution of the processors distinguishes PRAM from other shared memory models such as POSIX threads[31] or OpenMP[15], and avoids most problems associated with race conditions. Any reference to the PRAM model is usually associated with assumptions on the outcome of having concurrent access to the same memory location. The arbitrary concurrent-read concurrent-write (CRCW) convention allows concurrent-reads to the same memory locations; in case, of multiple attempts to write to the same memory location simultaneously, an one among the attempting write will succeed, but it is not known in advance which one.

XMTC is an extension of the C programming language that adds parallel directives to provide a PRAM-like model to the programmer. The main addition is the *spawn* directive, which provides support for a PRAM-like pardo loop. The directive will spawn multiple virtual threads and execute the ensuing code black in parallel. Within these parallel blocks, each thread is assigned an ID which can be accessed using the $ symbol. There is also the *sspawn* directive, that can be nested, for launching a single additional thread.

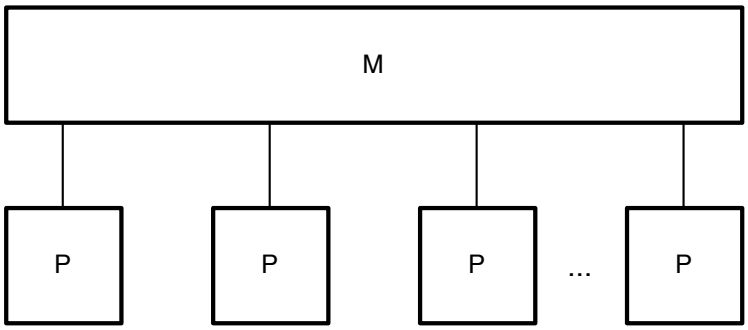

Fig. 2. PRAM model of a parallel computer

Listing 2. XMTC code

```
#include <xmtc.h>
#include <xmtio.h>
#define N 20

int main() {
  int i;
  int A[N],B[N],C[N];
  /* initialize A,B arrays  */
  ...

  spawn(0,N-1) {
    if($ % 2 ==0) {
      C[$] = A[$] + B[$];
    } else {
      C[$] = A[$] - B[$];
    }
  }
  for(i=0;i<N;i++)  {
    printf("%d ",C[i]);
  }
}
```

XMTC implements a CRCW PRAM: if multiple threads attempt to update the same memory location simultaneously, then an arbitrary one will succeed. XMTC also provides prefix-sum directives that implement concurrent writes, among other things.

Listing 2 shows XMTC code where the even elements of arrays B and A are added, and the odd elements of B are subtracted from odd elements of A.

The Parallel Random Access Machine (PRAM) model [18] has long been ad-

vocated as a model for designing parallel algorithms [24]. Historically, PRAM has been criticized because it assumes that processors run synchronously and interprocessor communication is free [14], assumptions which are severely violated on currently available HPC systems. It is not possible to program current systems using the PRAM model because modern architectures are not designed to support such a model efficiently. However, because of the recent trends in semiconductor technology towards multicore processors [38], it may soon be feasible to build large-scale, fine-grained uniform-memory-access parallel machines which could be modeled as PRAMs. For example, the XMT project at the University of Maryland is conducting research on how to build chips[42,1] that could efficiently support programs written using a PRAM-like model [41].

1.3 Context of the study

This study is one of a series of studies being carried out as part of the DARPA High Productivity Computing Systems [1] project (HPCS), which is investigating alternative parallel programming models to determine their impact on productivity relative to existing models such as MPI. All of these studies, including the one described in this paper, have been carried out by software engineering researchers (the first two authors of the paper). These researchers had full control over the results reported in this study (with the exception of XMTC performance analysis in Section 4.4) and have no vested interest in any one particular programming model or technology.

2 Related work

Several empirical studies have been previously done to evaluate the impact of parallel programming technologies on productivity, although we found no previous studies that focused specifically on PRAM. Szafron and Schaeffer ran on a study to evaluate the usability of a parallel programming environment compared to a message-passing library [39]. Browne et al. studied the effect of a parallel programming environment on defect rates [6]. Rodman and Brorsson [35] evaluated performance-effort trade-offs in porting a shared-memory program to use a hybrid shared-memory/message-passing model. Additionally, some studies have been done to evaluate the effect of a parallel language on effort by analyzing source code metrics [8,10,40].

[1] http://www.highproductivity.org

3 Description of the Study

This section first presents the goals and hypotheses, then gives a description of the study.

3.1 Goals

Stated in GQM form [3], the goal of this study is to analyze *message-passing and PRAM-like parallel programming models* for the purpose of *evaluation* with respect to:

- *program correctness*
- *development time*

from the viewpoint of the *researcher* in the context of

- *graduate-level parallel programming classes*
- *solving small programming problems*

Note that we use the term *development time* to refer to the time that the subjects spend implementing a solution to the programming problem. In the software engineering literature, this is sometimes referred to as effort [33], and we use the terms interchangeably.

3.2 Hypotheses

Proponents of the PRAM model claim that it is much simpler than the message-passing model for implementing parallel algorithms, since programmers do not have to deal with issues such as domain decomposition and explicit communication between processes. We use program correctness and development time as outcome variables to measure ease of use.

Based on the above, we consider the following two hypotheses in our study.

- *H1: Programs written in XMTC are more likely to be correct than programs written in MPI.*
- *H2: Writing XMTC requires less development time than writing MPI programs.*

3.3 Study Design

To conduct this study, we leveraged existing graduate-level parallel programming courses at two different universities. In one class, the students were given a parallel programming assignment to implement in MPI, and in the other class, the students were given the same parallel programming assignment in XMTC.

This study design is a nonequivalent control group design [7], which is technically a quasi-experiment since subjects were not randomly assigned to treatment groups.

3.4 Subjects and Groups

The subjects were students in graduate-level parallel programming courses at the University of California, Santa Barbara (UCSB) and the University of Maryland (UMD). The focus of the UCSB class was on developing parallel programs to run on the current generation of architectures, and the course covered MPI as well as other models (OpenMP[15], Matlab*P[12]). The focus of the UMD class was parallel algorithms in the PRAM model, and the students solved parallel programming programs in XMTC.

The students were assigned to treatment groups by class. UCSB students were given a problem to solve using MPI, and UMD students were given the identical problem to solve using XMTC. UCSB students could choose to solve the problem in either C/C++ or Fortran.

3.5 Procedure

The students in each class were given a parallel programming assignment which they were required to complete as part of their course. This assignment was one of several assignments in the classes. Students were not required to participate in the study.

Subjects were given a description of the task to be completed, as well as a C header file which contained some data structures necessary to complete the assignment. They were given a deadline of approximately two weeks, and worked on the assignment in their own time. In each class, the students had login accounts on a machine which they were to use for compiling and running their code.

The professors who taught the course did not have a direct role in either carrying out the study or in analyzing the data.

3.6 Study Task

The task was to write a function to multiply a sparse matrix with a dense vector. The subjects were provided with the data structures for representing the sparse matrix, and these data structures were identical for the MPI and XMTC groups. The professors tried to ensure that the students were exposed to the same type of information about the problem.

3.7 Apparatus

Development time data was collected using two different methods: self-reported and automatically collected. Subjects kept track of their development time with a self-reported time log. In the XMTC group, subjects used a web-based form to enter their development time data, and in the MPI group, subjects used papers forms. We also collected automatic development time data by instrumenting the compilers. These instrumented compilers recorded a set of data (including timestamps) at each compile. In both groups, subjects were required to compile and run their code on a remote machine. In the case of MPI, this was a departmental Linux cluster. In the case of XMTC, this was the prototype compiler simulator software that was available on the class server. From these two sources of data, we were able to come up with three separate estimates of development time: one based entirely on self-reported data, one based entirely on data from the instrumented compiler, and one based on a combination of the two approaches (more details about our algorithm for estimating development time based on the instrumented compiles and on combining the development time measures can be found in [23]).

Performance data for the MPI programs was measured by running the programs on a parallel machine and calculating the time spent doing matrix-multiply using the MPI timing functions. Performance data for the XMTC programs was measured using clock-cycle counts from the simulator.

Background information was collected from the subjects using on-line and paper-based questionnaires.

Table 1
Subject participation

	Total	Consented	Completed
UCSB (MPI)	26	21	16
UMD (XMTC)	16	15	14

Table 2
Subject major

	CS	CE	EE	ME	CS/M	Mgmt	?
MPI	13	0	0	2	0	0	1
XMTC	4	4	1	1	1	1	2

4 Data analysis

This section presents a data analysis of the results of the studies. We use a p-value of .05 in all statistical tests (or, equivalently, a confidence interval of 95%). All statistical tests were performed using the R software environment, [2] version 2.0. Power analyses were performed using Lenth's Java applets for power and size [29].

4.1 Characterization of groups

Table 1 shows the number of students in each class, the number of students who consented to participate in the study, and the number of consenting students who completed the assignment and submitted a solution (the other students dropped the class). Table 2 shows a breakdown of the subjects by major (CS: computer science, CE: computer engineering, EE: electrical engineering, ME: mechanical engineering, CS/M: computer science & math, Mgmt: management science, ?: did not specify background).

4.2 Correctness

- *H*1*: Programs written in XMTC are more likely to be correct than programs written in MPI.*

The programs were checked for correctness by running them against a known input and checking if the program output matched the expected output. A program that crashed during execution was counted as being incorrect. Table 3 provides a summary of the correctness across classes.

[2] http://www.r-project.org

Table 3
Correctness

Model	Number of correct submissions
MPI	7/13 (54%)
XMTC-1	12/14 (86%)
XMTC-2	11/14 (79%)

	χ^2	p-value	$p < .05$
MPI vs. XMTC-1	1.93	0.16	no
MPI vs. XMTC-2	0.91	0.34	no

Table 4
χ^2 test of correctness rates

For MPI correctness, we were only able to evaluate 13 of the 16 submitted programs. Of the remaining three, two were implemented in Fortran (which we could not evaluate because of technical reasons), and for the remaining one the subject had not conformed to the programming interface as given in the task description.

We wish to investigate whether there is a difference in the probability of implementing a correct program in MPI vs. XMTC. We use Pearson's χ^2 test [20] with Yates' continuity correction to check if there is a statistically significant difference in the frequency of correct solutions. The results of the tests are shown in Table 4. While the differences appear large in Table 3 (86% vs. 54%), the results are not statistically significant, and therefore we cannot claim that $H1$ is supported by the data.

4.3 Development time

- *H2: Writing XMTC requires less development time than writing MPI programs.*

We employ three methods for measuring development time in our analysis: self-reported, instrumented, and combined. Self-reported measures are based entirely on time logs, instrumented measures are based entirely on timestamps from the instrumented compilers, and combined measures are based on compiler timestamps when the subject is working on the instrumented machine, and self-reported time when the subject is working off the instrumented machine. Figure 3 shows the distribution of development time for the two classes using our three different development time measures.

We compute 95% confidence intervals for the differences in development time means to provide some notion of effect size rather than applying a t-test[13].

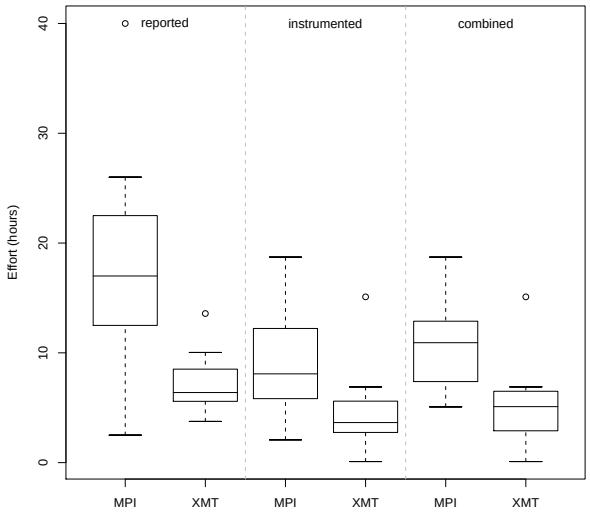

Fig. 3. Distribution of development time (effort)

Table 5
Limits of development time confidence intervals

	Lower limit	Upper limit
Reported	$4.9h$	$15.7h$
Instrumented	$0.7h$	$7.2h$
Combined	$2.0h$	$7.7h$

The confidence intervals for the difference in development time means are summarized in Table 5 and depicted in Figure 4. Note that for each measure, the confidence intervals does not include 0, so we conclude there is a statistically significant difference between treatment groups at the $p < .05$ level. Thus, $H2$ **is supported by all three development time measures**.

If we consider MPI to be our reference, we can compute a reduction in mean development time in using XMTC over MPI, which we define as

$$R = 1 - \frac{\overline{E}_{xmt}}{\overline{E}_{mpi}}$$

where $\overline{E}_{xmt}$ is mean time to implement the problem in XMTC, and $\overline{E}_{mpi}$ is mean time to implement the problem in MPI. Reduction in mean effort was 59% for reported time, 44% for instrumented time, and 46% for combined time.

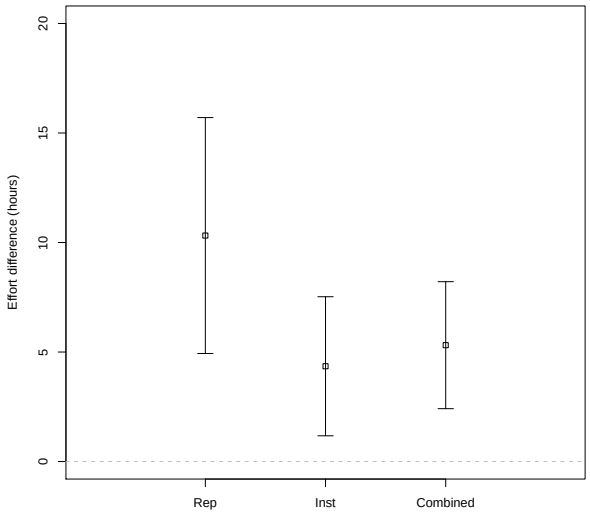

Fig. 4. Confidence intervals for development time differences

To evaluate if subject backgrounds had an effect on the development time score, we ran an analysis of variance based on the responses on the background questionnaire. We asked subjects about their current major as well as their experience in various areas, including: general software development, parallel programming, multithreaded programming, C programming, C++ programming, and sparse matrix methods.

We employed a one-way analysis of variance (ANOVA) to check if these variables had a statistically significant effect on the combined development time scores. Table 4.3 shows the ANOVA results. The only factor that exhibited a statistically significant effect at the $p < .05$ level was the parallel programming model.

4.4 A note about performance

A comparison of parallel programming models would not be complete without some consideration of the performance of the resulting codes. In this case, direct performance comparisons are not possible because MPI is a mature implementation that runs on existing systems, and XMTC is a prototype which runs only on a simulator. In addition, the two models exploit parallelism differently. XMTC uses a spawn/join model of parallelism, where the number of active threads varies over the lifetime of the process. In MPI, the number of processes is fixed over the lifetime of the program. Therefore, the performance of an MPI program can actually worsen as the number of processes is added if there is not enough work to distribute efficiently across the processors, so

Table 6
Analysis of variance

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Model	1	158.10	158.10	15.18	0.0025
Current Major	5	126.48	25.30	2.43	0.1022
Software Development Experience	1	0.36	0.36	0.03	0.8557
Parallel Programming Experience	1	0.86	0.86	0.08	0.7795
Thread Programming Experience	1	9.44	9.44	0.91	0.3614
Sparse Matrix Experience	1	2.07	2.07	0.20	0.6643
C Development Experience	1	26.99	26.99	2.59	0.1358
C++ Development Experience	1	14.79	14.79	1.42	0.2586
Residuals	11	114.58	10.42		

an MPI program must be evaluated at different processor counts to determine its peak performance.

Nevertheless, we felt that the paper would be incomplete without some discussion of performance. We use speedup versus a reference serial implementation (similar to "real speedup" [36]) as a measure of performance, where speedup is defined as

$$S = \frac{T_{ser}}{T_{par}}$$

where T_{ser} is reference serial execution time and T_{par} is parallel execution time. Speedup allows us to make comparisons across different machines. In the XMTC case, we had an XMTC expert code our reference serial implementation. For the MPI case, we did not have an implementation from an expert, so we used the fastest single-processor MPI implementation as the reference serial implementation.

MPI programs were run on a 24-processor Sun SunFire system (a shared memory machine), and XMTC programs were run on a simulator with 1024 thread-control units. Although the MPI subjects originally developed their code for a commodity Linux cluster, we felt that it would be a fairer comparison to measure the MPI performance on a shared memory machine, where there is less of a performance penalty due to communication among processes.

The MPI programs were timed when multiplying a 50180×50180 sparse-matrix containing 1185124 non-zero elements with a dense vector containing 50180 elements. The XMTC programs were timed when multiplying a 30000×100 sparse-matrix containing 60130 non-zero elements with a dense vector

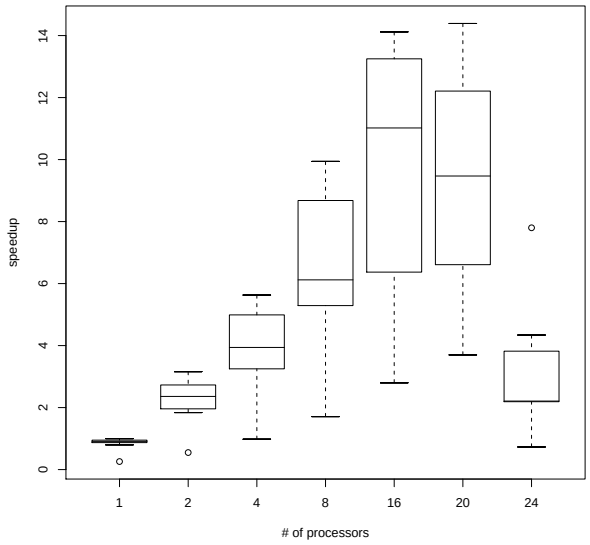

Fig. 5. MPI speedups

containing 100 elements.

Figure 5 shows the distribution of MPI speedups for a range of different processors, up to the limit of 24 processors. The MPI programs scale up to 16 processors, with a median speedup of 11x. However, as processors increase the performance worsens, and when 24 processors are used, the median speedup of only 2.2x. (Note that some super-linear speedup occurs at 2,4, and 8 processors, most likely due to cache effects).

Figure 6 shows the distribution of XMTC speedups for the implementations submitted by the subjects. The simulation-based empirical framework for XMT speedups is taken from [30]. The results were obtained through a cycle-accurate simulator that was derived from a synthesizable Verilog description of the XMT architecture. The median speedup for the subjects was 157x, and the speedup achievable by an expert was 206x.

Note that for the XMTC implementations, the median speedup for the "tuned" implementations is lower than the one for the more straightforward implementation. The outcome of a paired t-test [20] (p=.007) confirms that there is a statistically significant difference between the two implementations, although in the opposite of what was originally expected.

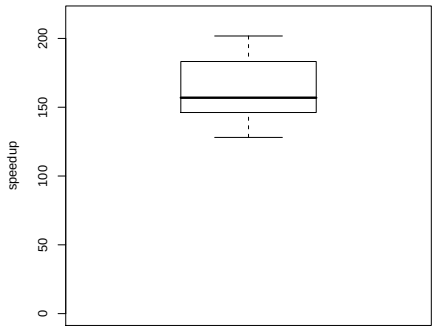

Fig. 6. XMTC speedups

5 Threats to Validity

5.1 Internal

Selection. Since we were not able to randomly assign subjects to treatment groups, the outcome of the study may have been affected by some difference across treatment groups other than the programming model that was used. Unfortunately, we did not have the opportunity to administer a pre-test in this study.

Selection-history interaction. The subjects received different amounts of training or experience in problem type, programming model, etc. The professors endeavored to provide the subjects with the same amount of information about the specific problem. However, the subjects had different amounts of experience using the programming models within their class. In the MPI class, the subjects had three previous MPI assignments before the one involved in the study, whereas in the XMTC class, the subjects had only one previous assignment.

The assignments were very similar, but not exactly the same. The MPI subjects were also asked to implement a conjugate-gradient algorithm in Matlab*P[12] which calls their sparse-matrix multiply function. We did not collect development time data on this part of the assignment. The XMTC subjects had to implement the algorithm twice: once using the same sparse-matrix data structures as the MPI subjects, and a second time using a different sparse-matrix data structure. We did collect development time data on this part of the assignment also, so our development time measures for XMTC may overestimate the total development time.

The motivations of the students in the two classes may be different, based on how they expected the assignment to be graded. In the MPI assignment, the students were told that 50% of the grade would be based on performance. In the XMTC assignment, the students were not told how performance would affect their grade.

The general emphasis of the two courses themselves are quite different. The MPI class focused on existing high performance computing architectures and practical issues (such as memory hierarchy) that programmers must deal with to achieve good performance. The XMTC class focused more on the theory of parallel algorithms in the PRAM model.

5.2 Construct

Mono-operation bias. Solving a small parallel programming problem in a classroom environment is qualitatively different from implementing a complete application. For example, in larger programs, more of the code will be inherently serial, and so the effect of the parallel programming model will not be as pronounced. On the other hand, in this problem the subjects in the MPI group were given the domain decomposition for the problem. In a real application, MPI programmers would have to come up with this decomposition on their own, whereas XMTC programmers do not have to deal with this issue.

Even for small-scale problems, this single study is not representative of all of the different types of parallelizable problems. This study focused on one particular problem: implementing a function to do sparse-matrix dense-vector multiply. This type of problem was easy to solve in parallel using XMTC (possibly even as easy or easier than the equivalent serial implementation,although this was not explicitly investigated here). Other types of problems will be easier or harder to parallelize using a message-passing model based on their communication patterns (e.g. "embarrassingly parallel" problems such as Monte Carlo simulations, or "nearest-neighbor" problems such as cellular automata simulations).

5.3 External

Interaction of selection bias and experimental variables. The results of this study only apply to novice parallel programmers in MPI and XMTC. These results cannot be generalized to more experienced parallel programmers working outside of a classroom environment, and the study may also be capturing learning effects. However, given that XMTC is currently a research language, there are no experienced XMTC programmers yet!

Table 7
Power analysis

	$\beta > .5$	$\beta > .8$
MPI vs. XMTC-1	22	37
MPI vs. XMTC-2	35	63

6 Discussion

6.1 Correctness

We did not find a statistically significant difference in the correctness rates between the different models. However, we can use the results obtained in this study to help plan the size of future studies through the use of power analysis [29]. Using the correctness rates we obtained in this study (MPI:54%, XMTC-1:86%, XMTC-2: 79%), we can estimate the number of subjects we would need to detect an effect with power (β) of 50% or 80%. (If the expected effect size is known in advance, there is little sense in conducting a study with power less than 50%, since the probability of a statistically significant result is less than a random coin flip, assuming the effect is real). Table 7 summarizes the number of subjects required in each group to achieve the desired power. Note that we would need at least 22 subjects in each group to achieve a power of 50% in comparing MPI to XMTC-1 (recall that in our study we had 16 subjects in the MPI group and 14 in the XMTC group).

6.2 Effort

To try to understand the differences in effort between MPI and XMTC, we examined the source code submitted by the subjects. The MPI programs are much larger than their XMTC counterparts (roughly 7 times larger than XMTC-1 and 2 times larger than XMTC-2 implementations). This difference in size is because of the additional source code necessary to handle the communication between the processors. The particular problem of sparse-matrix dense-vector multiply requires an "all-to-all" pattern of communication among processes: each process may potentially need data from all of the other processes to complete the computation.

MPI supports both point-to-point (send, receive) and collective communication (e.g. broadcast, scatter, gather, reduce, all-to-all, barrier) operations. While any MPI program can be written using only the point-to-point functions, use of the collective communication functions may improve performance, depending on the architecture. There was substantial variation across subjects

in their use of MPI functions. Only three subject used strictly point-to-point calls: the other students used at least one collective communication function. However, the particular collective communication function varied from one subject to the next. Half of the students used the vector variants of the collective communications calls (Allgatherv, Alltoallv), which are used when the size of the data being exchanged varies from one process to the other (e.g. when the number of data elements does not divide evenly by the number of processes). These calls are more complex than their non-vector counterparts, and their use may account for increases in effort.

By contrast, the XMTC code requires no explicit communication. For most of the submissions, the only substantial difference between the XMTC-1 implementation and the equivalent serial implementation is the use of the XMTC *spawn* function to create one thread per matrix row instead of an outer *for* loop.

The XMTC-2 implementations are larger than the XMTC-1 implementations but smaller than the MPI implementations. The extra code in these implementations is devoted to dividing up the work among a fixed number of threads.

7 Conclusions and future work

We evaluated the claim that a PRAM-like parallel programming model (XMTC) requires less effort than a message-passing model (MPI), through a quasi-experiment conducted with students in graduate-level parallel programming courses.

Our main result is that XMTC programs required about 45% less effort than MPI programs. There was insufficient power to detect a statistically significant difference in the rate of correctness between the two models. These results suggest that if architectures continue to evolve towards fine-grained uniform-memory access parallel machines, XMTC-like languages are worth pursuing. However, further studies are necessary to evaluate this claim with respect to different types of problems, as well as to larger programs.

While the sample size of this study was smaller than we would have liked, obtaining subjects for such studies is difficult. The population of programmers who have training in parallel programming is small, so we rely on available parallel programming courses for novice subjects. Nevertheless, we feel that this type of study is a good first step in the continued empirical research of parallel programming issues, and provides a basis of comparison for future studies which may involve more experienced programmers and different programming tasks.

In fact, this study is one of a series of studies we are involved in to explore the effect of parallel programming model on productivity. We are also investigating other parallel programming models (e.g. OpenMP [15], UPC [9], Matlab*P[12]), as well as other types of parallel programming problems. We are also conducting case studies of existing, larger-scale parallel programming projects to understand the differences between phenomena that we observe in the classroom studies and those that occur in actual development projects. While any single individual study can only provide a small amount of insight, we hope that by conducting several studies across multiple programming models, problem types, and problem sizes, we can gain a clearer picture of how these variables affect productivity.

8 Acknowledgments

This research was supported in part by Department of Energy grant awards DE-FG02-04ER25633 and DE-CFC02-01ER25489, and Air Force Rome Labs grant award FA8750-05-1-0100. We would also like to acknowledge Aydin Balkan, George Caragea, and Viral Shah for their help in collecting and analyzing the data.

A Raw data

Table A.1 shows the raw effort data, in hours, using the three measures: reported effort, instrumented effort and combined effort. Combined effort was computed by adding the instrumented effort to the fraction of reported effort that corresponded to work done off the instrumented machine. (When filling out the effort log, subjects indicated whether or not they were working on the instrumented machine).

Note that for one of the subjects (subject 3 in XMTC), the subject consented to participate but did not turn in any reported effort, and only two compiles were logged for that subject, so no effort data exists. Also note that the subject numbers are not necessarily sequential, because students who consented to participate but did not turn in a solution were not considered as part of the study.

Table A.1
Development time (effort) data

Model	ID	Reported	Instrumented	Combined
mpi	1	13.0	5.7	11.7
mpi	3		6.7	6.7
mpi	4	18.0	6.0	8.0
mpi	6	12.0	2.1	5.1
mpi	7	26.0	18.7	18.7
mpi	8	15.0	11.2	11.2
mpi	10	2.5	7.5	7.5
mpi	12	25.0	8.7	10.7
mpi	13	25.0	13.9	13.9
mpi	14	4.0	8.8	12.8
mpi	15	18.0	4.2	7.2
mpi	16	11.0	12.9	12.9
mpi	17	20.0	6.9	6.9
mpi	18	16.5	13.4	13.4
mpi	20	40.0	11.5	11.5
mpi	21	17.0	5.1	10.1
xmt-c	1		1.4	1.4
xmt-c	2	5.6	5.6	5.6
xmt-c	4	6.4	2.9	3.2
xmt-c	6	3.8	2.6	2.6
xmt-c	7	10.0	5.6	5.8
xmt-c	8	6.3	3.9	6.2
xmt-c	10	6.2	3.2	4.2
xmt-c	11	7.0	4.6	4.6
xmt-c	12	4.8	3.4	6.8
xmt-c	13	13.6	15.1	15.1
xmt-c	14	8.5	6.9	6.9

References

[1] A. Balkan, M. Horak, G. Qu, and U. Vishkin. Layout-accurate design and implementation of a high-throughput interconnection network for single-chip parallel processing. In *Proceedings of the 16th Annual IEEE Symposium on High-Performance Interconnects (Hot Interconnects)*, August 2007.

[2] A. Balki and U. Vishkin. Programmer's manual for XMTC language, XMTC compiler and XMT simulator. Technical Report UMIACS-TR 2005-45, University of Maryland, February 2006.

[3] V. Basili and H. Rombach. The TAME project: Towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, 14(6):758–773, June 1988.

[4] M. R. Benioff and E. D. Lazowska. PITAC report to the President on computational science: Ensuring America's competitiveness, June 2005.

[5] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M.-Y. Wu. Compiling Fortran 90D/HPF for distributed memory MIMD computers. *Journal of Parallel and Distributed Computing*, 21(1):15–26, 1994.

[6] J. Browne, T. Lee, and J. Werth. Experimental evaluation of a reusability-oriented parallel programming environment. *IEEE Transactions on Software Engineering*, 16(2):111–120, February 1990.

[7] D. T. Campbell and J. C. Stanley. *Experimental and quasi-experimental designs for research.* Houghton Mifflin, 1966.

[8] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi. Productivity analysis of the UPC language. In *IPDPS 2004 PMEO workshop*, 2004.

[9] W. Carlson, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, Center for Computing Sciences, May 1999.

[10] B. Chamberlain, S. Dietz, and L. Snyder. A comparative study of the NAS MG benchmark across parallel languages and architectures. In *2000 ACM/IEEE Supercomputing Conference on High Performance Networking and Computing (SC2000)*, pages 297–310, 2000.

[11] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, L. Snyder, W. D. Weathersby, and C. Lin. The case for high-level parallel programming in ZPL. *IEEE Computational Science & Engineering*, 5(3):76–86, 1998.

[12] R. Choy, A. Edelman, J. R. Gilbert, V. Shah, and D. Cheng. Star-P: High productivity parallel computing. In *8th Annual Workshop on High-Performance Embedded Computing (HPEC 04)*, 2004.

[13] J. Cohen. The earth is round ($p < .05$). *American Psychologist*, 49(12):997–1003, December 1994.

[14] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[15] L. Dagum and R. Menon. OpenMP: An industry-standard api for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46, 55 1998.

[16] J. Dongarra, S. Otto, M. Snir, and D. Walker. A message passing standard for MPP and workstations. *Communications of the ACM*, 39:84–90, July 1996.

[17] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, 1990.

[18] S. Fortune and J. Wylie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[19] S. Graham, M. Snir, and C. A. Patterson. Getting up to speed: The future of supercomputing. Technical report, National Research Council, 2004.

[20] W. L. Hays. *Statistics*. Wadsworth, Belmont CA, fifth edition, 1994.

[21] Federal plan for high-end computing: Report of the high-end computing revitalization task force (HECRTF), May 2004.

[22] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, 1998.

[23] L. Hochstein, V. R. Basili, M. Zelkowitz, J. Hollingsworth, and J. Carver. Combining self-reported and automatic data to improve programming effort measurement. In *Fifth joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE'05)*, September 2005.

[24] J. JaJa. *An Introduction to Parallel Algorithms*. Addison-Wesley Professional, March 1992.

[25] E. Joseph, A. Snell., and C. G. Willard. Countil on competitiveness study of U.S. industrial HPC users. Technical report, Council on Competitiveness, July 2004.

[26] C. Kessler and H. Seidl. The Fork95 parallel programming language: Design, implementation, application. *International Journal on Parallel Programming*, 25(1):17–50, 1997.

[27] J. S. Leichter and R. A. Whiteside. Implementing Linda for distributed and parallel processing. In *ICS '89: Proceedings of the 3rd International Conference on Supercomputing*, pages 41–49, New York, NY, USA, 1989. ACM.

[28] Leiserson and Plaat. Programming parallel applications in Cilk. *SINEWS: SIAM News*, 31, 1998.

[29] R. V. Lenth. Some practical guidelines for effective sample size determination. *The American Statistician*, 55(3):187–193, August 2001.

[30] D. Naishlos, J. Nuzman, C.-W. Tseng, and U. Vishkin. Towards a first vertical prototyping of an extremely fine-grained parallel programming approach. In *Special Issue for the 13th ACM Symposium on Parallel Algorithms and Architectures (SPAA-01)*, pages 521–552. Springer-Verlag, 2003.

[31] G. J. Narlikar and G. Blelloch. Pthreads for dynamic and irregular parallelism. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, 1998.

[32] R. Numrich and J. Reid. Co-Array Fortran for parallel programming, 1998.

[33] R. S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 2004.

[34] J. Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[35] A. Rodman and M. Brorsson. Programming effort vs. performance with a hybrid programming model for distributed memory parallel architectures. In P. Amestoy, P. Berger, M. Daydé, I. Duff, V. Frayssé, L. Giraud, and D. Ruiz, editors, *Euro-Par'99 parallel processing : 5th International Euro-Par Conference*, volume 1685 / 1999, pages 888–898. Springer-Verlag GmbH, September 1999.

[36] S. Sahni and V. Thanvantri. Performance metrics: Keeping the focus on runtime. *IEEE Parallel & Distributed Technology*, 4(1):43–56, Spring 1996.

[37] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20(4):531–545, 1994.

[38] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30, March 2005.

[39] D. Szafron and J. Schaeffer. An experiment to measure the usability of parallel programming systems. *Concurrency: Practice and Experience*, 8(2):147–166, March 1996.

[40] S. VanderWiel, D. Nathanson, and D. Lija. Complexity and performance in parallel programming languages. In *2nd International Workshop on High Level Programming*, April 1997.

[41] U. Vishkin, S. Dascal, E. Berkovich, and J. Nuzman. Explicit multi-threading (XMT) bridging models for instruction parallelism. In *10th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '98)*, 1998.

[42] X. Wen and U. Vishkin. PRAM-on-chip: First commitment to silicon. In *Proceedings of the 19th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, June 2007.

[43] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In ACM, editor, *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press.