A preliminary empirical study to compare MPI and OpenMP
ISI-TR-676

Lorin Hochstein, Victor R. Basili

December 2011

Abstract

Context: The rise of multicore is bringing shared-memory parallelism to the masses. The community is struggling to identify which parallel models are most productive.

Objective: Measure the effect of MPI and OpenMP models on programmer productivity.

Design: One group of programmers solved the sharks and fishes problem using MPI and a second group solved the same problem using OpenMP, then each programmer switched models and solved the same problem again. The participants were graduate students in an HPC course.

Measures: Development effort (hours), program correctness (grades), program performance (speedup versus serial implementation).

Results: Mean OpenMP development time was 9.6 hours less than MPI (95% CI, $0.37 - 19$ hours), a 43% reduction. No statistically significant difference was observed in assignment grades. MPI performance was better than OpenMP performance for 4 out of the 5 students that submitted correct implementations for both models.

Conclusions: OpenMP solutions for this problem required less effort than MPI, but insufficient power to measure the effect on correctness. The performance data was insufficient to draw strong conclusions but suggests that unoptimized MPI programs perform better than unoptimized OpenMP programs, even with a similar parallelization strategy. Further studies are necessary to examine different programming problems, models, and levels of programmer experience.

Chapter 1

INTRODUCTION

In the high-performance computing community, the dominant parallel programming model today is MPI, with OpenMP as a distant but clear second place [1, 2]. MPI's advantage over OpenMP on distributed memory systems is well-known, and consequently MPI usage dominates in large-scale HPC systems.

By contrast, in shared-memory systems, there is no consensus within the community about when a programmer should choose MPI or OpenMP. As software engineering researchers who have studied software development from HPC, we have heard statements along the lines of "*Of course, OpenMP-style shared-memory programming is easier than MPI-style message passing. This is the reason why so much research effort has gone into the development of shared-memory programming models.*" We have also heard practitioners say that, "*Anybody who has actually worked in the field will tell you that it's easier to get better program performance using MPI, even on a shared memory system, because of data locality issues*". These two elements of community folklore, while not explicitly contradictory, are difficult to reconcile. The central question remains: When is it more productive to for a scientist to use MPI vs. OpenMP when developing on a shared-memory system?

With the recent ascension of multicore architectures, shared memory programming has taken on a new level of importance [3]. The rise of multicore has also put parallel computers within the reach of a much larger community of computational scientists than ever before. These scientists could benefit from parallelizing their codes, but are novices from a parallel programming prospective.

While the HPC community collectively has a wealth of experience with both technologies, the community lacks a body of knowledge based on systematic research through experimentation with human subjects. As a recent article in Communications of the ACM article on parallel computing put it: *"As humans write programs, we wonder whether human psychology and human-subject experiments shouldn't be allowed to play a larger role in this revolution"* [4].

However, given the challenge of designing and running such studies, and

the fact that the methodology involved is very different from the performance-oriented studies of traditional HPC research, it is understandable that few studies exist in the literature, with virtually none comparing the two most widely used models.

In this paper, we describe a pilot human-subjects experiment to compare the productivity of MPI and OpenMP, using students from a graduate-level computer science course on high-performance. Because of the human subjects involved, the methodology of the study derives from social-science research methods [5]. In particular we use null hypothesis significance testing (NHST), a method commonly used in psychology research to test hypotheses.

The purpose of this study is *not* to present a definitive evaluation of these two languages. Rather, it represents a first step in the development of a body of evidence about how the choice of MPI vs. OpenMP affects programmer effort and program performance. In addition to the flaws associated with this particular study, no one study can ever be definitive given the enormous number of context variables associated with software development. One of the major goals of this paper is to inspire researchers to run their own studies, addressing the flaws pointed out in this one.

Chapter 2

RELATED WORK

Because parallel programming has been confined largely to the scientific computing community until recently, there has been relatively little empirical research on parallel programming languages that directly involves human subjects. Notable exceptions include a study by Szafron and Schaeffer to evaluate the usability of a parallel programming environment compared to a message-passing library [6], and a study by Browne et al. on the effect of a parallel programming environment on defect rates [7]. In addition, there have been some studies of parallel programming languages that use source code metrics a proxy for programmer effort [8, 9, 10].

More recently, Pankratius et al. conducted a case study in a software engineering course to compare OpenMP to POSIX threads on multicore [11]. As a precursor to this study, the authors have previously performed some preliminary studies comparing MPI and OpenMP [12], as well as a controlled experiment that compared MPI to the XMTC language [13]. Ebcioglu et al. ran a study to compare MPI, UPC, and X10 [14], and Luff ran a study to compare three different models: the Actor model, transactional memory, and traditional shared-memory locking [15].

Chapter 3

BRIEF INTRODUCTION TO NHST

Because some readers may not be familiar with the null hypothesis significance testing (NHST) approach used in this paper, we provide a brief introduction here. Readers familiar with NHST can skip to the next section. For a full treatment, see any reference on experimental design for behavioral sciences (e.g., [16]).

3.1 Overview

The NHST approach to hypothesis testing is a kind of inductive version of proof by contradiction. As an example consider that we we are interested in knowing is whether it is easier to write parallel programs in MPI or OpenMP. We pick a specific programming problem, and a population of programmers of interest (in this case, graduate students), and we have them solve the problem and measure the amount of time it takes to solve the problem. Using this method, we assume initially that solving the programming problem requires, on average, the exact same amount of effort in MPI as it does in OpenMP. This assumption is called the *null hypothesis*. We then calculate the likelihood of obtaining the observed data under this hypothesis. This likelihood is known as the *p-value*.

If the p-value is below a certain predetermined threshold, then we say that we *reject the null hypothesis* and, consequentially, observed a *statistically significant* difference between the two groups. If the p-value is not below the threshold, we say that we *failed to reject the null hypothesis* and therefore did not observe a statistically significant difference between the two groups.

3.2 Calculating the p-value

We model the effort required to solve the problem in MPI and effort required to solve the problem in OpenMP as random variables (X, Y). Typically, we assume that X and Y have the same probability distribution, and the same (unknown variance). We have n samples from each $x_1 \ldots x_n, y_1 \ldots y_n$. Under the null hypothesis, X and Y have the same means ($\mu_x = \mu_y$).

One common approach is to assume that the sample means $\bar{x}_i$ and $\bar{y}_i$ are approximately Gaussian random variables by appealing to the central limit theorem. In this case, we calculate what is known as a t-statistic, which is a function of the difference between the sample means, the sample variances, and the number of samples, n. The t-statistic is itself a random variable with a t-distribution. Using this distribution, we can calculate the p-value: the likelihood that we would have obtained the observed t-statistic

3.3 Errors and power

When performing an NHST experiment, two types of errors are possible

- Type I errors. Incorrectly rejecting the null hypothesis: The experimenter concludes that there is a statistically significant difference between X and Y, when in fact $\mu_x = \mu_y$.
- Type II errors. Incorrectly failing to reject the null hypothesis: When $\mu_x \neq \mu_y$, but the experiment fails to detect a statistically significant difference between X and Y.

By convention, the probability of a Type I error is referred to as α, and the probability of a Type II error is referred to as β. The experimenter chooses a likelihood threshold by setting a minimum value for α in advance of running the experiment. Typical values are $\alpha = .05$ or $\alpha = .01$. The p-value must fall below this threshold to reject the null hypothesis.

The *power* of the study refers to the probability of detecting a statistically significant difference given that $\mu_x \neq \mu_y$, and is defined as $1 - \beta$. Power is a function of the α threshold, the true difference between the means, the true variance, and the number of sample points. By convention, we generally desire a power of at least 80%. If the difference between means and variance can be estimated, then we can calculate the number of sample points needed to obtain a desired power. Typically, pilot studies (such as the one described in this paper) are used to obtain estimates for the difference between means and the variance.

3.4 Uncertainty and effect sizes

The use of p-values have been criticized because they do not capture the uncertainty in the result of the experiment or the estimated size of the effect [17].

To communicate the uncertainty, we can report a *confidence interval*. Instead of reporting the likelihood that the null hypothesis is true, we report a range of values that is likely to include the true difference between the means: $\mu_x - \mu_y$. Testing that a p-value less than $\alpha = .05$ is mathematically equivalent to testing that a 95% confidence interval includes 0, and the size of the confidence interval also captures the magnitude of the uncertainty in the experimental result.

Typically, we are also interested in the *effect size*; we don't simply want to know if OpenMP is easier to program in than MPI, we want to know how much easier it is to program in. Several effect size measures have been proposed. One popular effect size measure is the difference in means divided by the standard deviation:

$$\frac{\mu_x - \mu_y}{\sigma}$$

Cohen's d is a popular effect size measure which uses sample means and an estimate of the standard deviation to estimate this effect size:

$$d = \frac{\hat{x}_i - \hat{y}_i}{\hat{\sigma}}$$

Chapter 4

DESCRIPTION OF THE STUDY

In this section we describes the goals and hypotheses of the study, and describe how it was carried out.

4.1 Goals

We use Basili's Goal-Question-Metric (GQM) template to define the goals of this study [18]:

The goal of this study is to analyze *message-passing and shared-memory parallel programming models* for the purpose of *evaluation* with respect to:

- *development effort*
- *correctness*
- *performance*

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

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

4.2 Hypotheses

Proponents of OpenMP model claim that it is much simpler than the message-passing model for implementing parallel algorithms, because of the additional work involved in partitioning and exchanging data in a message-passing model. We use development time and program correctness as outcome variables to measure ease of use.

Table 4.1: Question: What is your major?

Computer science	7
Computer engineering	3
Electrical engineering	2
Unspecified	2

In terms of performance, the message-passing model should incur performance overhead in communicating messages, which is not necessary in a shared-memory model. Therefore, we expect the performance of OpenMP programs to better than MPI programs on shared memory machines.

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

- *H1: Writing OpenMP requires less development time than writing MPI programs.*
- *H2: OpenMP programs are more likely to be correct than MPI programs.*
- *H3: OpenMP programs are more likely to run faster than MPI programs.*

4.3 Study Design

The study is a two-treatment, two-period crossover design: each participant was exposed to both treatments (MPI, OpenMP). Half of the participants were exposed to the treatments in one order (MPI first, then OpenMP), and half were exposed to the treatment in the other order (OpenMP first, then MPI). In all cases, we used the same parallel programming problem: *sharks and fishes* (see Section 4.5).

4.4 Participants and groups

To conduct this study, we leveraged an existing graduate-level course on high-performance computing in the Department of Computer Science at the University of Maryland. We gave them an initial questionnaire to collect information about their relevant background experience. 14 of the students who consented to participate responded to the questionnaire. As shown in Tables 4.4, 4.4 and 4.4, the students were a mix of computer science, computer engineering, and electrical engineering majors, who were reasonably experienced in programming. Almost all of them had experience programming in C in a Unix environment, which was the programming environment for the assignment. Most of the students had not used MPI or OpenMP before, but almost all had at least some exposure to programming with threads or synchronization.

Table 4.2: Question: Please rate your experience in the following activities

	Professional	Classroom only	None
Parallel programming	1	5	8
Tuning code for performance	4	3	7
Tuning code for parallel performance	0	4	10
Using MPI	0	3	11
Using OpenMP	0	3	11
Developing software in C	8	5	1
Developing software in C++	5	6	3
Programming with threads or synchronization	3	9	2
Developing software on a unix platform	6	7	1
Developing on a cluster	1	5	8
Experience using tools (e.g. debuggers, profilers, etc.)	6	6	2
Developing on a parallel shared-memory machine	0	4	10

Table 4.3: What is your previous experience with software development in practice?

I have never developed software	0
I have developed software on my own, as part of a course as a hobby	0
I have developed software as part of a team, as part of a course	4
I have developed software as part of a team **one time** in industry	3
I have developed software as part of a team **more than one time** in industry	7

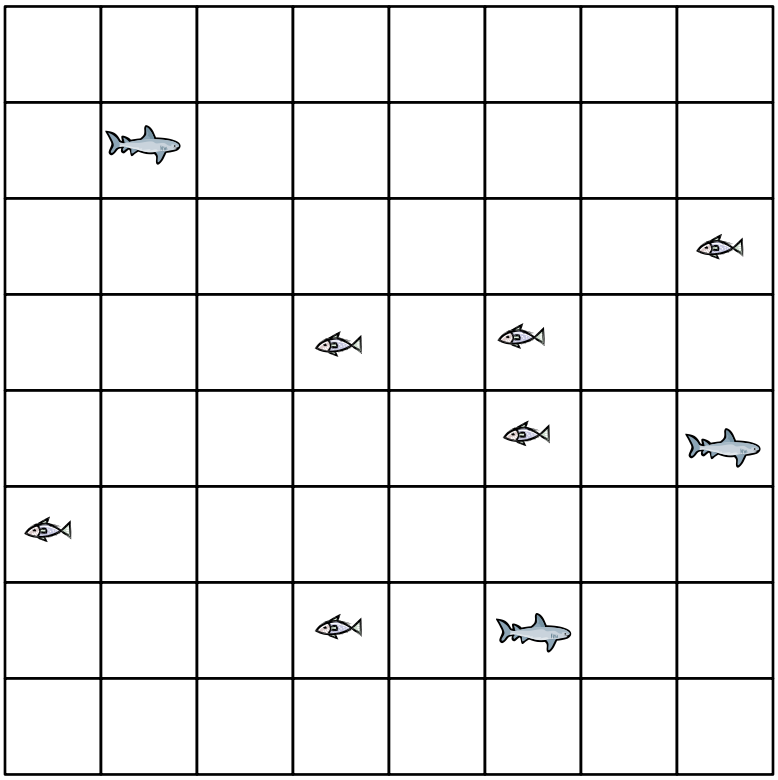

Figure 4.1: Example of a sharks and fishes board: a cellular automaton implementation of a simple predator-prey model

4.5 Study Task

The programming task of the study is a problem called *sharks and fishes*. It is an example of a population dynamics model from computational ecology, which simulates how the population of two species, predator (sharks) and prey (fish) change over time. The simulation is performed on a square grid containing cells. A cell may contain a shark, a fish, or be unoccupied, as shown in Figure 4.1. The model computes successive generations of the grid. In each time step, sharks and fishes move, breed, and die as a function of the contents of neighbor cells and how much time has elapsed. The original model, called *WaTor*, was first described by Dewdney[1] [19]. A full description of the assignment description appears in the appendix. Sharks and fishes is an appealing problem because many computational physics problems are simulated on a regular grid and are amenable to similar strategies for parallelization.

[1]The interested reader can view an implementation by Leinweber at `http://www.leinweb.com/snackbar/wator`

Table 4.4: Assignment schedule

			Deliverables	
	Assignment released	Assignment due	Group A	Group B
Assignment 1	Sep 27, 2005	Oct 11, 2005	serial, MPI	serial, OpenMP
Assignment 2	Oct 11, 2005	Oct 20, 2005	OpenMP	MPI

4.6 Procedure

The study participants were required to solve the sharks and fishes problem as part of the high-performance computing course. All students in the class were required to complete the assignment, but were not required to participate in the study. The assignment schedule is shown in Table 4.6. For the first assignment, students were required to submit a sequential version of the program and either an MPI or OpenMP version (determined randomly by the professor), along with a report describing their parallelization strategy and performance analysis. For the second assignment, students were required to submit an MPI version if they had previously submitted OpenMP, or an OpenMP version if they had previously submitted MPI, along with the report.

For MPI development, students were given accounts on a cluster with two types of nodes:

- Dual Pentium II 450 MHz processors, 1 GB RAM, Gigabit ethernet interface
- Dual Pentium III 550 MHZ processors, 1GB RAM, Gigabit ethernet interface

Students had access to two MPI implementations, LAM[20] and MPICH[21], and could use either one. For OpenMP development, students were given access to a Solaris-based Sun Fire 6800 symmetric multiprocessor (SMP) server with 24 UltraSparc III 750 MHz processors and 72GB RAM, installed with the Sun Studio C OpenMP compiler.

In addition to the task description, students were given source code files with utility functions for parsing the command-line arguments, makefiles for building MPI and OpenMP programs on the provided hardware, and a sample input and corresponding output file.

4.7 Apparatus

4.7.1 Effort

In software engineering, the term *effort* refers specifically to programmer labor, and is typically measured in person-months. In the context of the study, since

the programming task took hours rather than months, and since individuals worked alone on the programming tasks, we measured effort in hours.

We instrumented both the Linux cluster and the Sun server with UMDInst[2] and Hackystat[22] sensors, which collect data from the MPI/OpenMP compilers, emacs and vi editors, and the tcsh shell. The sensors captured timestamps during compiles, shell command invocations, and edit events, which we used to estimate programmer effort as described in previous work [23]. The processing of the captured data to compute the resulting effort scores was done by Mirhosseini-Zamani's EffortTool[24], and we used the Experiment Manager environment to manage the data captured from the study [25].

4.7.2 Correctness

Ascertaining the correctness of a particular software program is, in general, a difficult task. Indeed, it is one of the central challenges of software engineering research. Nevertheless, we need some (likely imperfect) measure of correctness to run this kind of study.

We used expert judgment as our measure of the correctness of the software. Specifically, we used the assignment grades determined by the professor of the course. Note that these grades take into account the quality of the written report in addition to program correctness.

4.7.3 Performance

To collect performance data, we compiled and ran all working implementations on a reference multicore machine, running all implementations against the same input. We ran the programs on a dual-processor quad-core 2GHz Intel Xeon, using only one of the quad-core processors for testing. We took a reference board of 1000×1000 and ran the sequential, MPI, and OpenMP versions. We used two MPI implementations: LAM/MPI 7.1.4 and MPICH2 1.0.8. We used a single OpenMP implementation: gcc 4.3.2 with the CPU affinity feature enabled so that each thread was bound to a particular core.

[2] http://hpcs.cs.umd.edu/index.php?id=18

Chapter 5

DATA ANALYSIS

All analysis was done with the R statistical package, version 2.8.0. For hypothesis testing, we set $\alpha = .05$ (or, equivalently, a a confidence interval of 95%).

Out of 18 students enrolled in the class, a total of 11 consented to participate in the study and produced data that we could use in the analysis. The other students either did not consent, failed to enable their instrumentation, or worked on uninstrumented machines.

Unfortunately, because of the missing students, the resulting data set is not fully counter-balanced: 8 students solved the problem first in OpenMP, and 3 students solved the problem first in MPI. This gives a potential advantage to the MPI model in the study, an issue we discuss in Section 6.

Note that we did not count development time for the sequential implementation in our effort calculations. Counting the sequential implementation would unfairly increase the measurement of the first assignment. In addition, we knew that students were much more likely to do their sequential implementations on an uninstrumented machine. By contrast, the students largely did not have access to other parallel computers at the time this study was run.

5.1 A note about boxplots

This paper makes extensive use of boxplots, a type of plot introduced by the statistician John Tukey [26]. We use boxplots here to give the reader a sense of the distribution of a set of data points, and to compare distributions. Figure 5.1 is an example of a boxplot, that shows the distribution of the size of the different serial, MPI, and OpenMP programs submitted by the subjects for this experiment.

The box in a boxplot contains half of the data points: the bottom of the box is the first quartile (25th percentile), and the top of the box is the third quartile (75th percentile). The width of the box is sometimes referred to as the interquartile range (IQR). The dark line inside the box is the second quartile, also known as the median (50th percentile).

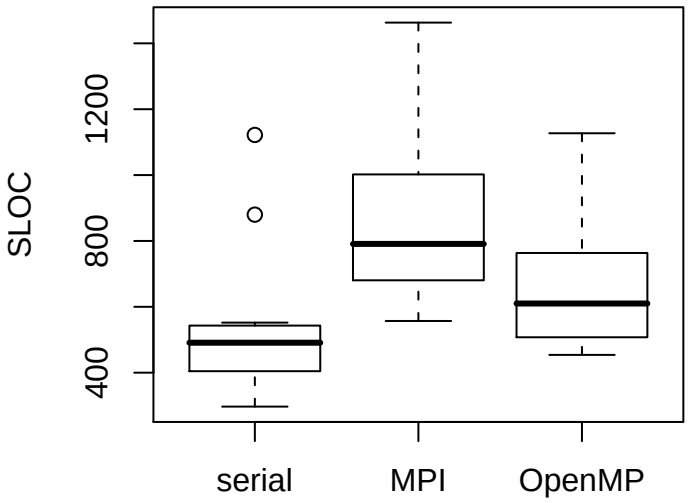

Figure 5.1: Boxplot that illustrates the distribution of program sizes across subjects, in source (non-commented, non-blank) lines of code, or SLOC

The "whiskers" are drawn to the furthest point that it is less than $1.5 \times IQR$ away from the edges of the box. All data points outside of this range are drawn as circles. For example, in the case of the serial submissions, the first quartile is 404.5, and the third quartile is 543, giving an IRQ of 138.5. In this case, the whisker extends upward to the furthest point that is less than $543 + 1.5 \times 138.5 = 750.75$, which is 552. The other two points, 880 and 1122, are drawn as circles.

From this plot, we can see at a glance that MPI programs tend to be larger than OpenMP programs, which tend to be larger than serial programs. We can also see how much variation there is in program size across the participant submissions.

5.2 Effort

- *H1: Writing OpenMP requires less development time than writing MPI programs.*

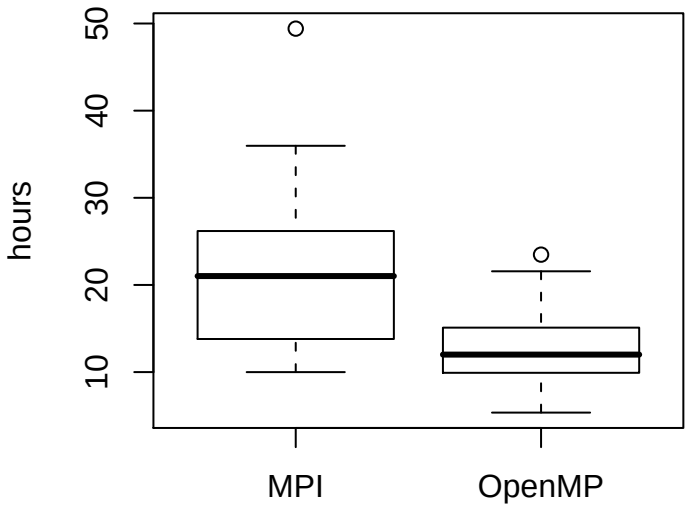

Figure 5.2: Boxplot that illustrates the distribution of effort data for MPI and OpenMP tasks.

Figure 5.2 shows the distribution of the effort for the two programming models. Figure 5.3 shows the same data, with line segments connecting the MPI and OpenMP efforts for each subject.

Because this is a within-subjects study, we apply a two-sided, paired t-test. We also show the effect size with Cohen's d, using the pooled standard deviation. Table 5.2 summarizes these results.

The test shows a statistically significant result. The effect size calculation indicates an effect size of 1.1, roughly one standard deviation of the sample population. Unfortunately, the software engineering community has not yet developed conventions about whether a particular effect size is large. Given the large variation across individual programmers in this data (see Figure 5.2), we consider this effect size to be substantial. Comparing the means of the groups, using OpenMP represents a mean effort savings of approximately 43%.

In addition, we asked the students on a post-test questionnaire to compare the difficulty of MPI and OpenMP to serial development over different types of development activity. Unfortunately, we had a very low response rate: only 4 students filled out the questionnaire. Their responses appear in Table 5.2. The

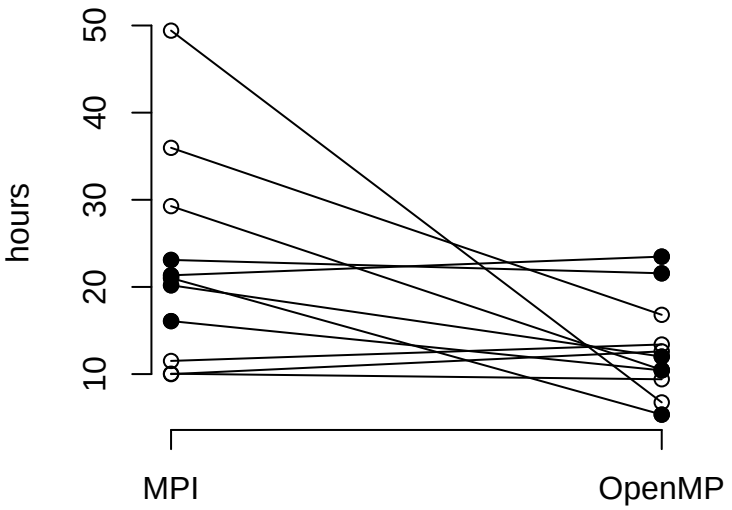

Figure 5.3: Dot plot of effort distribution, with MPI and OpenMP efforts for each subject connected by line segments. Closed circles indicate data points where we have performance data for both MPI and OpenMP, as discussed in Section 5.4

Table 5.1: Summary of effort data

group means	MPI:22.5 hours, OpenMP: 12.9 hours
mean of differences	9.6 hours
p-value	.042
confidence interval	$(0.37, 18)$ hours
effect size (Cohen's d):	1.1

Table 5.2: Post-test questionnaire: difficulty relative to serial

	MPI			OpenMP		
	as easy as	harder	much harder	as easy as	harder	much harder
Pre-program thinking	1	2	1	1	3	
Development time	1	2	1	2	2	
Learning	1	3		2	2	
Tuning for performance		2	2	1	2	1
Debugging		1	3		3	1
Overall difficulty		1	3	1	3	

data are consistent with the effort scores: OpenMP is perceived as easier than MPI across all different types of programming activity.

Because the students solved each problem twice, we are concerned about ordering effects[27]: it is reasonable to assume that parallelizing a program in, say, MPI, is easier if you have already parallelized it before in OpenMP. Because mortality left us with data that was not counter-balanced, this was a particular concern in our case.

Figure 5.4 shows an interaction diagram, which can be useful for identifying interactions among independent variables through visual inspection. In this case, the two variables are programming model (MPI, OpenMP) and the ordering (MPI-then-OpenMP, OpenMP-then-MPI). The diagram shows the mean effort for four groups:

- MPI effort when solving MPI first (3 data points)
- OpenMP effort when solving OpenMP first (8 data points)
- MPI effort when solving MPI second (8 data points)
- OpenMP effort when solving OpenMP second (3 data points)

If there was no interaction among variables, then the lines would be parallel. The diagram suggests that OpenMP requires less effort than MPI regardless of whether the programmer has experience solving the problem previously using a different model. It also suggests an interaction effect: it appears that it solving the problem previously in MPI made it easier to solve in OpenMP, but solving the problem previously in OpenMP made it harder to solve in MPI!

Note that we are extremely wary of drawing any significant conclusions based on the small sample size, especially given the variation in the data as seen in Figure 5.2.

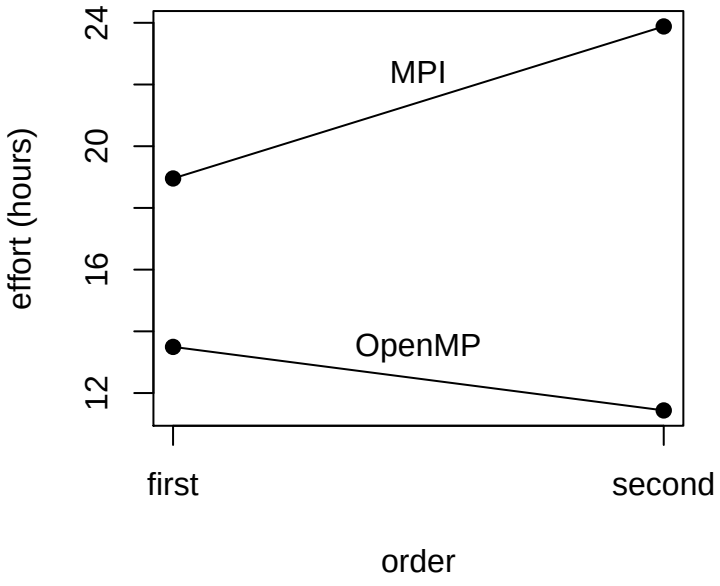

Figure 5.4: Interaction plot that shows the mean effort for four groups (MPI:first, MPI:second, OpenMP:first, OpenMP:second) which illustrates the interaction between programming model and ordering. If there was no interaction effect, the lines would be parallel.

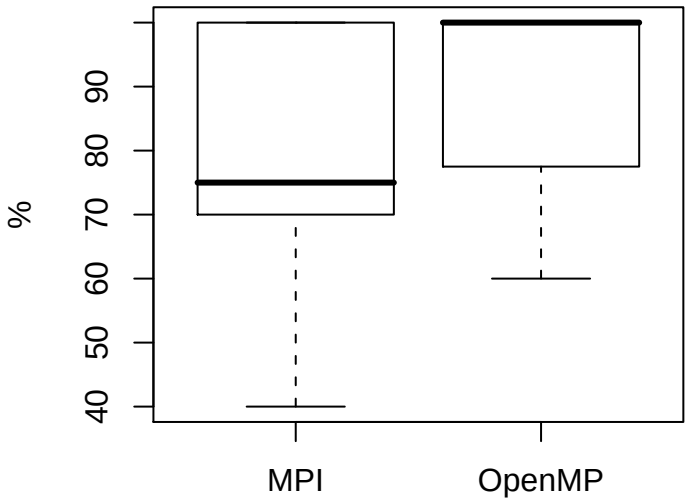

Figure 5.5: Boxplot that illustrates the distribution of assignment grades for MPI and OpenMP tasks.

5.3 Correctness

- *H2: OpenMP programs are more likely to be correct than writing MPI programs.*

Figure 5.5 shows the distribution of the grades for the two programming models. As previously, we apply a paired t-test. We also show the effect size with Cohen's *d*, using a pooled standard deviation.

The test shows no statistically significant result in correctness across the two groups. Although there is a reasonably large effect size (0.53), there is insufficient power to detect a difference, as suggested by the large confidence interval.

5.4 Performance

- *H3: OpenMP programs are more likely to run faster than MPI programs.*

Table 5.3: Correctness analysis (grade)

group means	MPI:80.5%, OpenMP: 89.1%
mean of differences	-8.6%
p-value	0.094
confidence interval	$(-19\%, 1.8\%)$
effect size (Cohen's d)	0.53

Table 5.4: Execution time

ID	Sequential time	LAM/MPI	MPICH2	OpenMP
33	28.6*s*	N/A	N/A	15.0*s*
34	22.4*s*	8.1*s*	12.9*s*	31.5*s*
36	37.8*s*	18.7*s*	18.9*s*	34.0*s*
38	23.2*s*	N/A	9.0*s*	12.6*s* / 24.3*s*
39	47.3*s*	N/A	63.3*s*	46.0*s*
108	34.6*s*	16.5*s*	15.4*s*	27.0*s*

To measure the performance of the parallel programs, we compute the speedup achieved by the parallel implementation on four cores versus the serial implementation on one core. As described in Section 4.7, we measured program execution time by executing all programs on four cores of a multicore machine using a reference serial input and recording the execution time.

When evaluating performance, we considered only programs that were considered correct by the assignment grader. Our rationale is that a defect in the code may result in result in a reduced execution time (e.g., failure to use locking correctly in OpenMP, failure to check that non-blocking I/O has completed in MPI). Only five students submitted both MPI and OpenMP implementations that were considered correct (participants 34,36,38,39,108) and one student submitted a correct OpenMP implementation but an incorrect MPI implementation (participant 33).

Execution times are shown in Table 5.4. Speedups for MPI and OpenMP implementations are shown in Table 5.4. Participant 38 submitted two separate OpenMP implementations, one using loop-level parallelism and another using parallel sections, the performance of both implementations is reported: the implementation using loop-level parallelism executed in less time. The strategies are discussed in more detail in Section7.

Two of the MPI implementations worked properly on MPICH2, but not LAM/MPI. For one of the implementations, all processes seemed to hang, taking up 100% of the CPU. The problem is what Nakamuara et al. refer to as a potential deadlock[28]. The correct behavior depends upon the behavior of the MPI_Send function call, which may or may not block. A description of this

Table 5.5: Speedups on 4 cores

	speedup			*strategy*	
ID	LAM/MPI	MPICH2	OpenMP	MPI	OpenMP
33	*	*	1.9×	1D decomp	1D decomp
34	2.8×	1.7×	0.7×	1D decomp	2D decomp
36	2.0×	2.0×	1.1×	1D decomp	1D decomp
38	*	2.6×	1.8× / 1.0×	1D decomp	loop-level/2D decomp
39	*	0.7×	1.0×	1D decomp	loop-level
108	2.1×	2.2×	1.3×	1D decomp	1D decomp
Mean	1.95×		1.30×		

Table 5.6: Summary of performance data

group means	MPI:1.95×, OpenMP: 1.18×
mean of differences	0.78×
p-value	.0.063
confidence interval	$(-0.069\times, 1.6\times)$
effect size (Cohen's d):	1.5

problem can be found in Nakamura's HpcBugBase[1]. For the other improperly functioning LAM/MPI implementation, the program appeared to complete the simulation and produce output, but hung before final termination: we were not able to determine why it behaved properly with MPICH2 but not LAM/MPI.

Since we are doing a within subject comparison, we must discard the data from participant 33, leaving us with only 5 pairs of data points for statistical analysis. Where we have data from two MPI implementations, we use the median of the two. For participant 38, we use the larger speedup for OpenMP. Table 5.4 shows the results of the t-test, which are not statistically significant at the $p < .05$ level.

Despite the lack of statistical significance for this test, we feel that the large effect size estimate of 1.5 suggest that for this particular computing configuration, novice programmers are more likely to get better performance with MPI than OpenMP. However, because of the small number of data points, and because program performance is sensitive to many different variables, we cannot draw any firm conclusions.

[1] `http://www.hpcbugbase.org/index.php/Potential_Deadlock`

Chapter 6

THREATS TO VALIDITY

No individual study is perfectly designed, and individual flaws in the design can raise concerns about the validity of the conclusions. Similarly, issues can arise during the execution of the study that also threaten the validity of the conclusions. In this section, we discuss such issues.

6.1 Internal

In a controlled experiment, the goal is to test whether a causal relationship exists between the independent variable (selection of MPI or OpenMP as parallel programming model) on the dependent variables (developer effort, correctness, and program performance) by systematically varying the independent variable, while attempting keep all other context variables constant.

However, factors other than the systematic varying of the independent variable can sometimes be the source of variation in the dependent variables. In the literature, these are referred to as *internal threats to validity*.

Mortality. One of the challenges of running human-subject experiments is ensuring that the study participants in each of the treatment groups are equivalent. Ideally, this is done by random assignment into treatment groups to avoid any systematic bias in the two groups.[1]. The problem of non-equivalent groups due to nonrandom assignment to treatments is known as the *selection* problem.

As we used random assignment to treatment groups, we did not have this threat to validity in this study. However, we ran into a different problems. Several of the students that consented to the study did not enable instrumentation on the machines or worked on a non-instrumented machine. This problem of *mortality*, where participants initially sign up for the study but fail to complete it, was particularly problematic in our case because it affected the treatment

[1]Note that the importance of equivalent groups makes it very difficult to run such studies with experts, because it requires a pool of participants that is equally experienced in both MPI and OpenMP, to ensure that the two groups have equal levels of experience once the random assignment has been done

groups unevenly: eight students started with OpenMP and only three started with MPI.

Since this is a within-subjects study, we were still able to compare OpenMP with MPI for all participants. However, the study was no longer fully counterbalanced, which means that the results may also reflect the difference in ordering in which the problems were solved, in addition to the effect due to programming model.

History / Repeated testing. It is possible that the students would learn from the experience of having solved it the first time with the different model. We tried to control for this effect by using counter-balancing: half of the students solved the problem with MPI first, and half solved it with OpenMP first. Unfortunately, due to *mortality* effects (specifically, students who did not enable the instrumentation or worked on uninstrumented machines), our final data set was not fully counterbalanced: 3 students solved the problem first with OpenMP and 8 solved the problem first with MPI. This suggests that the effort savings of using OpenMP may be even more pronounced than is suggested by the results of the study.

Confounding variables. The development machines used by the study participants for the MPI and OpenMP machines were different. Differences in development effort may be partially attributable to the environment differences. In particular, the mechanism for executing parallel programs was different on the two machines. The MPI machine was a batch-scheduled cluster: to run a parallel program, the programmer must write a job script which invokes the program, and submit the script to a scheduler. The scheduler executes the program when sufficient resources become available on the cluster. In contrast, the OpenMP machine was an interactive server and could invoke the parallel program immediately after it was successfully compiled. This confounding may explain some of the observed difference in effort across models, as MPI programmers may have spent additional development time writing job scripts and waiting for their jobs to run.

6.2 Construct

To conduct a controlled experiment, we must *operationalize* the dependent variables. That is, we must take fairly abstract concepts like "correctness" and "performance" and define suitable metrics that we can collect during the experiment and use for statistical analysis.

If the chosen metrics do not properly capture the underling concepts, then the study has problems that are commonly referred to as *construct threats to validity*.

Mono-measure bias. The underlying constructs we analyze in this study (effort, correctness, performance) are difficult to measure. In this study, we used only a single measure for each concept, which runs a risk of not capturing them properly.

For the correctness metric, we took advantage of the environment in which

the study was run: the professor was already grading the assignments, so we leveraged these grades. However, because this measure is subjective, it is difficult to identify how accurate it is, and whether there is any bias. For a future study, we would probably recommend either defining a set of test suites to use as a basis for correctness, which would provide a simple, objective measure (number of tests passed). If expert judgment was used in a follow-on study, we would recommend using multiple judges and measure the inter-rater agreement to check for consistent ratings across experts.

For the effort metric, the participants may have spent considerable amounts of effort on trying to solve the problem that were not captured by the instrumentation (e.g. working out an algorithm on paper). We originally attempted to mitigate this by asking the students to record their effort in a paper log. Unfortunately, the response rates were too low for the self-reported effort data to be usable: not enough students turned in the forms.

Capturing program performance is even more difficult, because the performance of a program is sensitive to many different factors, such as computer architecture, cache, compiler implementation, compiler optimization flags and library implementation. In addition, the relative performance of serial, MPI and OpenMP implementations can vary based on the input files. We expect that speedups will decrease for smaller input sizes because the ratio of parallelizable work to multi-thread/process overhead will decrease. In addition, the distribution of the computational load may vary based on the input.

6.3 External

This study was conducted in a specific context: solving a small parallel programming problem in the context of a graduate-level course. This is qualitatively different from implementing a complete application in a commercial or research environment. 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. In addition, the students did their development on remote machines, since they did not have access to parallel computers at the time that the study was conducted. While this was advantageous for the internal validity of the study, it threatens external validity because we expect development habits to change when developing and running on a local machine.

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: a *nearest-neighbor* problem that is common in many physical simulations. 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, *all-to-all* problems).

We also might expect different results if the assignment goals were stated differently[29]. For example, if the students were told that their grades depended on achieving a certain level of program performance, they may have

spent more time trying to optimize their code. It may be the case that OpenMP requires more effort to achieve the same level of performance as MPI on a certain platform because of the time required to tune various parameters that affect performance.

Finally, this study was performed with graduate students who were just learning MPI and OpenMP during the course. We do not know how the results of the study would change if the students had more experience in parallel programming.

Chapter 7

DISCUSSION

7.1 Examining student solutions

In addition to the statistical analysis, we also examined the strategies that the students used to achieve parallelism. For the MPI case, a natural approach is to use *domain decomposition*. The programmer subdivides the board into contiguous regions, and each process is assigned to simulate the behavior of the sharks and the fishes in one region. Because the MPI processes run in a separate address space, the borders of these regions need to be exchanged among processes, hence the term *nearest-neighbor* to describe this type of parallelism. In general, parallel programmers seek to minimize the amount of data exchanged among processes, although for multicore systems this is less of an issue than in clusters. In theory, the optimal region shape is a square, which minimizes perimeter given a fixed area. This is known as a 2D decomposition, as each process has to exchange information with neighbors in the left-right directions and in the up-down directions. Conversely, a simpler solution is to use rectangular regions that span entire rows, because each process only has to exchange information with neighbors in the up-down direction. This is known as a 1D decomposition. Both types of decompositions are shown in Figure 7.1.

To obtain optimal performance, we want to divide the computational work up as evenly as possible among the processors: otherwise processors with less work will have to wait for other processors to catch up. Figure 7.1 shows equally-sized regions being assigned to each process. However, work is not directly proportional to the number of grid cells. Instead, it is proportional to the number of *occupied* grid cells. Because the number of occupied grid cells in a region changes over time, ensuring that the amount of work remains evenly divided across processors, a task known as *load balancing*, increases the complexity of the implementation.

Out of the 11 MPI solutions examined, 9 of them used domain decomposition with communication among neighbors. The other two used a less efficient *master-worker* scheme[30], where communication was centralized through

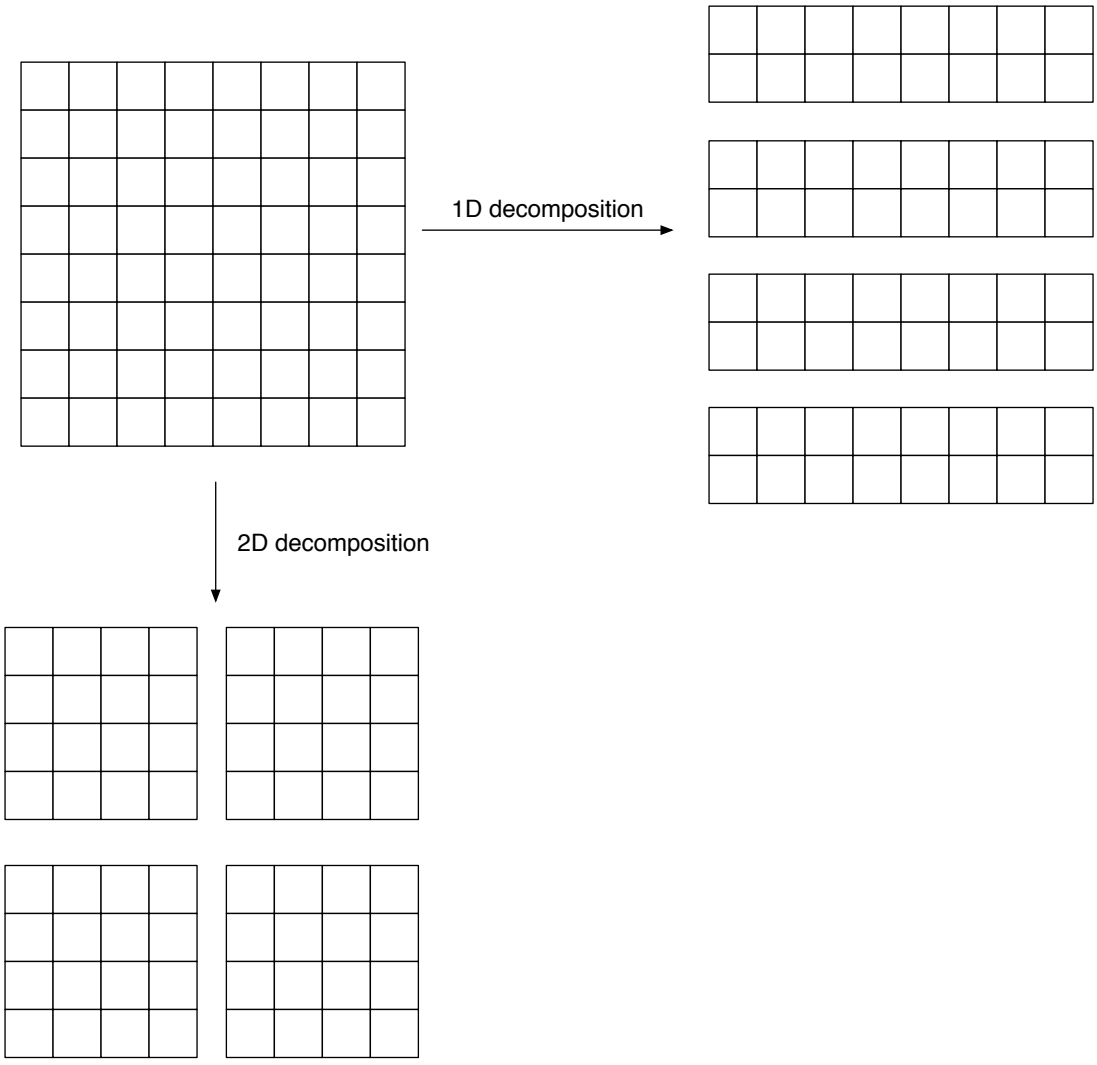

Figure 7.1: Two domain decomposition strategies used by students for parallelizing the solution

a single master process rather than between members. Only 1 of the 9 domain decomposition solutions used a 2D decomposition, the rest used a 1D decomposition.

For the OpenMP case, there are two general strategies available: loop-level parallelization, or using parallel sections to implement domain decomposition. Because the amount of computational work within loop iterations can be quite small, this parallel strategy is sometimes referred to as *fine-grained parallelism*. By contrast, the domain decomposition approach is referred to as *coarse-grained parallelism*, because there are relatively large chunks of computational work to be processed in parallel. MPI is generally only used for coarse-grained parallelism, where OpenMP supports both, in principle.

As written, the assignment description implies that students should use the domain decomposition approach for their OpenMP solution, and that is what 7 out of the 11 participants did (2 used 2D decomposition, and the other 5 used 1D decomposition). Two students used loop-level parallelism, and one student did both approaches and submitted two separate implementations. The professor did not deduct points for the students who used loop-level parallelism. Because of the small size of the data sample, it is difficult to draw firm conclusions. Nevertheless, the data suggest that a novice is more likely to achieve speedup with MPI than with OpenMP. This contradicts our *H*3 hypothesis: we assumed that there would be some overhead incurred in the MPI implementations because of the interprocess communication involved. One possible explanation for the overall poor OpenMP performance is that it is difficult to handle locking correctly and still obtain good performance.

7.2 Performance-effort tradeoffs in productivity

The study results suggest that implementing a parallel program using OpenMP requires substantially less effort than implementing the same program in MPI. In particular, taking a serial program and converting it to an MPI program requires more substantial modifications to the source code than converting it to an OpenMP program. This result is consistent with the folklore from the high-performance computing community which suggests that OpenMP is easier to write parallel programs in than MPI.

These performance results reinforce the importance of considering both programmer effort and program performance when considering the productivity of a parallel programming paradigm [31, 32]. If MPI programs tend to get better performance than OpenMP programs, then programmers need to understand the performance-effort tradeoffs when choosing the appropriate parallel programming model for their project.

Advocates of the OpenMP model may argue that the OpenMP programs in this experiment likely ran slower because of poor implementations that were too conservative in locking strategies to avoid race conditions. Indeed, it is conceivable that with modifications, the OpenMP programs would run even faster than their MPI counterparts. However, we do not know how much ef-

fort would be involved in this activity of improving performance. In particular, since the performance problems may be related to locking, then modifying this code may lead to race conditions, which are notorious for being difficult to debug. An interesting alternative study design would be to give the students a specific performance requirement, so that they had to keep working until the code met that requirement. Such a study would be more useful for practitioners who are developing under stringent performance requirements.

7.3 Using a parallel programming course for running a study

We felt that a graduate-level course with a parallel computing component was a natural fit for a parallel programming study. It would have been difficult to get prospective participants to commit to solving a parallel programming problem that may take up to 50 hours of effort, unless they were required to do so for some other purpose.

Conducting this type of a study in a classroom environment is surprisingly challenging. Professors who teach parallel programming issues are seldom versed in experimental design, and so running such a study requires a collaboration among empiricists and parallel computing experts. The programming assignment must be tailored to serve the purposes of the study, keeping in mind that the primary goal of the assignment is to serve the pedagogical needs of the course. Because of the length of time required to solve this program, we could not simply lock the participants in a room and time their efforts with a stopwatch. Having them work on their own time creates formidable measurement challenges. While these programming problems seem small ($\sim$ 1 KLOC), they are probably near the upper limit of the kind of problem that can be used in a controlled experiment if direct effort measurement is desired. For larger problems, obtaining sufficient number of participants and collecting accurate data would become much more complicated: such problems are more well-suited to a case study approach [33, 11].

7.4 Recommendations for future studies

The participants were not given any specific quality or performance objective in this study: they completed it as they would any other programming assignment in a course. We saw, as a result, wide variation in terms of both correctness and performance. An alternative study design would be to set some level of correctness that the code must achieve (e.g., provide participants with a comprehensive suite of test cases), and/or to provide them with a minimum level of speedup that they should achieve.

We expect different results for two types of parallel programming tasks. It would be interesting to see how results vary by task, and if it is, indeed, possi-

Table 7.1: Power analysis calculation

observed mean of differences	9.6 hours
pooled standard deviation	8.93 hours
minimum power	$\geq$ .0.8
alpha	$\leq$ .05
Total subjects required	30

ble to classify the relative productivity of MPI vs. OpenMP based on different problem types (e.g., embarrassingly parallel, nearest neighbor).

In this study, we used a crossover design with a single programming task. The results of our analysis suggest that there may be an interaction effect between programming model (MPI vs. OpenMP) and ordering (MPI-then-OpenMP vs. OpenMP-then-MPI). This means that even if we did not have the mortality issues described in Section 6, the effect of ordering would not cancel out across the groups. For a future study, we recommend not using such a crossover design. Other options are to use a simple design where participants solve only one problem, or a fully counterbalanced design where the participants solve two problems, rather than solving the same problem twice.

We can use a statistical technique known as power analysis to estimate the number of subjects required in a future study to observe a statistically significant difference in effort between MPI and OpenMP, using the observed difference of means and the observed variances from this study. For simplicity, we will assume that the future study will be a basic design with two groups (MPI, OpenMP), and we will assume that the variance will be the same across the two groups. We use our sample difference between the means as an estimate of the true difference between the means, and we used the pooled standard deviation as an estimate of the true standard deviation.

Table 7.4 shows a summary of the data involved in the power analysis calculation, which was computed using Lenth's power analysis tool [34]. A follow-on study would require a minimum of 30 participants: 15 in the MPI group, and 15 in the OpenMP group, to have better than 80% probability of detecting a difference in effort between MPI and OpenMP.

Chapter 8

CONCLUSION

Programmers will no longer be able to achieve performance improvements on the next generation of processor architectures without substantially modifying their programs to take advantage of parallelism. The study described in this paper suggests that it requires significantly less effort to make such changes with the OpenMP parallel programming model than the MPI parallel programming model. The performance data is less conclusive, but suggests that this reduction in effort may come at a cost of reduced performance. Even in the context of this single study, it is unclear which model would be more productive for developers without considering the relative importance of the performance of the final implementation.

Additional studies are required to determine how performance and effort vary for different programming problems, different populations of programmers, and for different software development contexts. We have seen very different programming tools used in the computational science community versus the IT community[35].

In addition, MPI and OpenMP are both moving targets. In this study, participants used features from the MPI-1 and OpenMP 2.5 standards. At the time of this writing, implementations that support MPI-2 and OpenMP 3.0 are available.

We hope that this study will become part of a larger body of empirical research to support the software community as it struggles to come to grips with the challenges of multicore software engineering. We are already seeing technologies hyped by proponents without empirical evidence supporting claims of improved productivity. As no clear market winner has yet emerged, we as researchers have an opportunity to help guide the course of adoption of multicore software technologies.

Chapter 9

ACKNOWLEDGEMENTS

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 like to acknowledge Alan Sussman for accommodating this research in his course, Taiga Nakamura for data cleaning, Nabi Zamani for building the tools to calculate effort, Thiago Craverio and Nico Zazwarka for building the experiment management software, and Sima Asgari for managing the IRB process. Finally, we would like to thank Marv Zelkowitz, Jeff Hollingsworth, Forrest Shull and Jeff Carver for their advice in the design of the study and artifacts.

Bibliography

[1] Graham SL, Snir M, Patterson CA ((eds.)). *Getting up to Speed: The Future of Supercomputing*. National Academies Press, 2004.

[2] Bronis RdS, Jeffrey KH, Shirley M, Patrick HW. Results of the PERI survey of SciDAC applications. *Journal of Physics: Conference Series* 2007; :012 027.

[3] Sutter H. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal* March 2005; **30**(3).

[4] Asanovic K, Bodik R, Keaveny JDT, Keutzer K, Kubiatowicz J, Morgan N, Patterson D, Sen K, Wawrzynek J, Wessel D, *et al.*. A view of the parallel computing landscape. *Communications of the ACM* October 2009; **52**(10).

[5] Shadish WR, Cook TD, Campbell DT. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Second edn., Wadsworth Publishing, 2001.

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

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

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

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

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

[11] Pankratius V, Jannesari A, Tichy WF. Parallelizing bzip2: A case study in multicore software engineering. *IEEE Software* 2009; **26**(6):70–77, doi: http://dx.doi.org/10.1109/MS.2009.183.

[12] Hochstein L, Carver J, Shull F, Asgari S, Basili VR, Hollingsworth J, Zelkowitz M. Parallel programmer productivity: A case study of novice HPC programmers. *SC '05: Proceedings of the ACM/IEEE Conference on Supercomputing*, 2005, doi:10.1109/SC.2005.53.

[13] Hochstein L, Basili VR, Vishkin U. A pilot study to compare programming effort for two parallel programming models. *Journal of Systems and Software* November 2008; **81**(11):1920–1930, doi:10.1016/j.jss.2007.12.798.

[14] glu KE, Sarkar V, El-Ghazawi T, Urbanic J. An experiment in measuring the productivity of three parallel programming languages. *Proceedings of the Third Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, 2006.

[15] Luff M. Empirically investigating parallel programming paradigms: A null result. *Proceedings of the PLATEAU 2009 Workshop on Evaluation and Usability of Programming Languages and Tools*, 2009.

[16] Kirk RE. *Experimental Design: Procedures for Behavioral Sciences*. Wadsworth Publishing, 1994.

[17] Cohen J. The earth is round (p¡.05). *American Psychologist* 1994; **49**(12).

[18] Basili VR, Caldiera G, Rombach HD. Goal question metric approach. *Encyclopedia of Software Engineering*. John Wiley & Sons, 1994; 528–532.

[19] Dewdney A. Sharks and fish wage an ecological war on the toroidal planet Wa-Tor. *Scientific American* December 1984; .

[20] Burns G, Daoud R, Vaigl J. LAM: An open cluster environment for MPI. *Proceedings of Supercomputing Symposium '94, University of Toronto*, 1994; 379–386.

[21] Gropp W, Lusk E, Doss N, Skjellum A. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing* September 1996; **22**(6):789–828.

[22] Johnson PM, Hongbing K, Agustin J, Chan C, Moore C, Miglani J, Shenyan Z, Doane WEJ. Beyond the personal software process: Metrics collection and analysis for the differently disciplined. *25th International Conference on Software Engineering*, 2003; 641–646.

[23] Hochstein L, Basili VR, Zelkowitz M, Hollingsworth J, Carver J. Combining self-reported and automatic data to improve programming effort measurement. *Fifth joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE'05)*, 2005, doi:10.1145/1081706.1081762.

[24] Mirhosseini-Zamani SN. High productivity computer systems: Realization of data cleaning and human effort measurement tools. Master's Thesis, University of Maryland, College Park 2007.

[25] Hochstein L, Nakamura T, Shull F, Zazworka N, Basili VR, Zelkowitz MV. *An Environment for Conducting Families of Software Engineering Experiments*, *Advances in Computers*, vol. 74. Elsevier, 2008; 175–200.

[26] TUkey JW. *Exploratory Data Analysis*. Addison Wesley, 1977.

[27] Kitchenham B, Fry J, Linkman S. The case against cross-over designs in software engineering. *Software Technology and Engineering Practice, 2003. Eleventh Annual International Workshop on* September 2003; :65–67.

[28] Nakamura T, Hochstein L, Basili VR. Identifying domain-specific defect classes using inspections and change history. *Proceedings of the 5th International Symposium on Empirical Software Engineering (ISESE'06)*, 2006; 346–355, doi:10.1145/1159733.1159785.

[29] Weinberg GM, Schulman EL. Goals and performance in computer programming. *Human Factors* 1974; **16**(1):70–77.

[30] Mattson TG, Sanders BA, Massingill BL. *Patterns for Parallel Programming*. Software Patterns Series, Addison-Wesley Professional, 2004.

[31] Kepner J. HPC productivity model synthesis. *The International Journal of High Performance Computing Applications* 2004; **18**(4):505–516. Other link at http://www.highproductivity.org/IJHPCA/10f-Kepner-Synthesis.pdf.

[32] Zelkowitz M, Basili VR, Asgari S, Hochstein L, Hollingsworth J, Nakamura T. Measuring productivity on high performance computers. *Proceedings of the 11th International Symposium on Software Metrics*, 2005.

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

[34] Lenth R. Java applets for power and sample size.

[35] Basili VR, Cruzes D, Carver JC, Hochstein LM, Hollingsworth JK, Zelkowitz MV, Shull F. Understanding the high-performance-computing community: A software engineer's perspective. *IEEE Software* July 2008; **25**(4):29–36, doi:10.1109/MS.2008.103.

Appendix A

Assignment

CMSC 714- High Performance Computing

Fall 2005 - Programming Assignments 1 and 2

Due October 11 and October 20, 2005 @ 6:00PM

The purpose of this programming assignment is to gain experience in parallel programming on an SMP and a cluster, using OpenMP and MPI, respectively. For this assignment you are to write a sequential and two parallel implementations of a program to simulate a game called Sharks and Fishes.

The game is a variation of a simple cellular automata. The game is played on a square grid containing cells. At the start, some of the cells are occupied, the rest are empty. The game consists of constructing successive generations of the grid. To understand how the game works, imagine the ocean is divided into a square 2D-grid, where:

- each grid cell can be empty or have a fish or a shark,
- the grid is initially populated with fishes and sharks in a random manner, and
- the population evolves over discrete time steps (generations) according to certain rules.

Rules for Fish

- At each time step, a fish tries to move to a neighboring empty cell, picking according to the method described below if there are multiple empty neighbors. If no neighboring cell is empty, it stays. Fish move up or down, left or right, but not diagonally (like Rooks not Bishops).
- If a fish reaches a breeding age, when it moves it breeds, leaving behind a fish of age 0, and its age is reset to 0. A fish cannot breed if it doesn't move (and its age doesn't get reset until it actually breeds).
- Fish never starve.

Rules for Sharks

- At each time step, if one of the neighboring cells has a fish, the shark moves to that cell eating the fish. If multiple neighboring cells have fish, then one is chosen using the method described below. If no neighbors have fish and if one of the neighboring cells is empty, the shark moves there (picking using the method described below from empty neighbors if there is more than one). Otherwise, it stays. Sharks move up or down, left or right, but not diagonally (like Rooks not Bishops).
- If a shark reaches a breeding age, when it moves it breeds, leaving behind a shark of age 0 (both for breeding and eating), and its breeding age is reset to 0. A shark cannot breed if it doesn't move (and its age doesn't get reset until it actually breeds).
- Sharks eat only fish, not other sharks. If a shark reaches a starvation age (time steps since last eaten), it dies (if it doesn't eat in the current generation).

More on Rules for Fish and Sharks

The specification above is not specific about when ages for starving or breeding are incremented, and whether starving or breeding takes precedence. The right way to think about it is that an animal's age is incremented between generations (and a generation consists of a red and a black sub-generation, as described in the next section). So if a shark reaches its starvation age in a generation, it dies, since the starvation age did not get reset by the end of the previous generation. As for breeding, an animal is eligible to breed in the generation after its current breeding age reaches the animal's minimum breeding age (supplied on the command line).

Traversal Order Independence

Since we want the order that the individual cells are processed to not matter in how the game evolves, you should implement the game using a so-called red-black scheme (as in a checkerboard) for updating the board for each generation. That means that a generation consists of 2 sub-generations. In the first sub-generation, only the red cells are processed, and in the second sub-generation the black cells are processed. In an even numbered row red cells are the ones with an even column number, and in an odd numbered row red cells are the ones with an odd column number. The red-black scheme allows you to think of each sub-generation as a separate parallel (forall) loop over the red (or black) cells, with no dependences between the iterations of the loop. Note that in the red-black scheme a fish or shark may end up moving twice in a generation. The rules that follow about selecting cells and resolving conflicts apply for each sub-generation.

Rules for Selecting a cell when multiple choices are possible

- If multiple cells are open for moving to for either a fish or a shark, or occupied for a shark to move to eat a fish, number the possible choices starting from 0, clockwise starting from the 12:00 position (i.e. up, right, down, left). Note that only cells that are unoccupied (for moves) or occupied by fish (for sharks to eat), are numbered. Call the number of possible cells p.
- Compute the grid cell number of the cell being evaluated. If the cell is at position (i,j) in the ocean grid with (0,0) the grid origin, and the grid is of size MxN, the grid cell number $C = i \times N + j$.
- The cell to select is then determined by $C \bmod p$. For example, if there are 3 possible cells to choose from, say up, down and left, then if $C \bmod p$ is 0 the selected cell is up from the current cell, if it is 1 then select down, and if it is 2 then select left.

Conflicts

- A cell may get updated multiple times during one generation due to fish or shark movement.
- If a cell is updated multiple times in a single generation, the conflict is resolved as follows:
    - If 2 or more fish end up in a cell, then the cell ends up with the fish with the greatest current age (closest to breeding - big fish win). The other fish disappear.
    - If 2 or more sharks end up in a cell, then the cell ends up with the shark with the greatest current starvation age (closest to starvation - more hungry sharks win), and the resulting shark gets the breeding age of the shark that had the greatest current starvation age. If 2 or more sharks have the same greatest starvation age, the resulting shark gets the greatest breeding age of the tied sharks. The other shark(s) disappear.
    - If a shark and a fish end up in a cell, then the cell ends up with a shark - the shark eats the

fish, so resets its starvation clock. The resulting shark gets the greatest breeding age of all the sharks that end up in the cell (i.e. ignore their previous starvation ages). Any other fish and/or shark(s) disappear.

For this project the game grid has finite size. The x-axis and y-axis both start at 0 (in the upper left corner of the grid) and end at limit-1 (supplied on the command line).

Part 1

Write a serial implementation of the above program in C. Name the source file `sharks-fishes-serial.c`.

Input

- Size of the grid
- Distribution of sharks and fishes in the following file format

```
x y type
1 3 fish
3 5 shark
```

- Shark and fish breeding ages
- Shark starvation age
- A number of timesteps (iterations)

Your program should take six command line arguments: the size of the grid (limit), the name of the data file, the shark and fish breeding ages, the shark starvation age, and the number of generations to iterate.

To be more specific, the command line of your program (e.g., for a sequential version) should be:

```
sharks-fishes <limit> <input file name> <shark breeding age>
<fish breeding age> <shark starvation age> <# of
generations>
```

A sample set of parameters for the command line and an input data file will be made available soon.

Output

At the end of program output a list of positions of sharks and fishes in the same file format as for the input data file.

A sample output file for the sample input data will also be made available soon.

Data structures

A 2-D grid of cells

```
struct ocean{
```

```
int type /* shark or fish or empty */
struct swimmer* occupier;
} ocean[MAX][MAX];
```

A linked list of swimmers

```
struct swimmer{
int type;
int x,y;
int age;
int last_ate;
int iteration;
swimmer* prev;
swimmer* next;
} *List;
```

At a high level, the logic of the serial (non-parallel) program is:

- Initialize ocean array and swimmers list
- In each time step, go through the swimmers in the order in which they are stored and perform updates

Part 2: OpenMP

Write an OpenMP implementation of the Sharks and Fishes program as in the Part 1, with the same rules. Name this source code `sharks-fishes-omp.c` . There are now some complications to consider:

- You need to distribute the 2D ocean grid across threads, in 1 or 2 dimensions. Each thread is responsible for a 2-D grid of ocean cells.
- For communication, each thread needs data from up to 4 neighboring threads.
- 2 challenges are potential for conflicts, and load balancing

Conflicts

- Border cells for a given thread (the ones just outside the cells a thread is responsible for) may change during updates due to fish or shark movement.
- Border cells need to be synchronized properly. Hence the update step involves communication between threads, but only at the borders of the grid assigned to each thread.

Load Balancing

- The workload distribution changes over time.
- A 2D block distribution may not be optimal, so you might want to experiment with 1D block distributions too.

Part 3: MPI

Write an MPI implementation of the Sharks and Fishes program as for OpenMP, and deal with the same problems. Name this source code `sharks-fishes-mpi.c` .

HINTS

The goal is not to write the most efficient implementations, but rather to learn parallel programming with OpenMP and MPI.

Figure out how you will decompose the problem for parallel execution. Remember that OpenMP synchronization between threads must be done carefully to avoid performance bottlenecks, and that MPI (at least the mpich implementation) does not have great communication performance so you will want to make message passing infrequent. Also, you will need to be concerned about load balancing.

WHAT TO TURN IN, AND WHEN

You must eventually submit the sequential and both parallel versions of your program (please use file names that make it obvious which files correspond to which version, as described above) and the times to run the parallel versions on input data to be give later, for 1, 2, 4 and 8 processes).

You also must submit a short report about the results (1-2 pages) that explains:

- what decomposition was used
- how was load balancing done
- what are the performance results, and are they what you expected

You will turn in the serial version and either the OpenMP or MPI parallel version at the first due date, with the short report, and then the serial version again (hopefully the same) and the other parallel version at the second due date, with an updated report. I will send email to each of you, telling you which parallel version you should turn in first. Please do the parallel versions in the order that you are assigned them, to aid the software engineering study being done on you during the programming assignments.

RUNNING OpenMP on tau/ceti

To run with OpenMP, you need to add the Sun compiler to your path (typically done in your .cshrc file):

```
set path=(/opt/SUNWhpc/bin $path)
```

The Sun C compiler that understands OpenMP is then invoked with mpcc (if you really want to use C++, use mpCC instead).

To compile your OpenMP program, use the compiler flags "-xO3 -xopenmp=parallel" (e.g., `mpcc -xO3 -xopenmp=parallel your_file.c`).
Note that OpenMP compiler optimization is level 3 whether you explicitly set it or not, and you will get a compiler warning you if you don't use -xO3.

To choose the number of threads that OpenMP will use at runtime, set the OMP_NUM_THREADS environment variable before running your program. This means that you don't have to recompile to change the number of threads.

For example, to run the OpenMP hello world example from http://www.llnl.gov/computing/tutorials/openMP/exercise.html):

```
$ mpcc -xO3 -xopenmp=parallel omp_hello.c -o omp_hello
$ setenv OMP_NUM_THREADS 8
$ ./omp_hello
Hello World from thread = 0
Number of threads = 8
Hello World from thread = 4
Hello World from thread = 7
Hello World from thread = 1
Hello World from thread = 3
Hello World from thread = 2
Hello World from thread = 6
Hello World from thread = 5
```

Note that because the default compiler optimization level is 3 for OpenMP codes, if you compile your sequential code without optimization level 3, you may see a performance boost that's due to compiler optimization, not parallelism. So please also compile your serial code with -xO3 to show your performance results

RUNNING MPI with MPICH on the red/blue cluster

To run MPI, you need to set a few environment variables:

```
setenv MPI_ROOT /usr/local/stow/mpich-1.2.7
setenv MPI_LIB  $MPI_ROOT/lib
setenv MPI_INC  $MPI_ROOT/include
setenv MPI_BIN $MPI_ROOT/bin
# add MPICH commands to your path (includes mpirun and the C
compiler that links in the MPI library, mpicc)
set path=($MPI_BIN $path)
# add MPICH man pages to your manpath
if ( $?MANPATH ) then
     setenv MANPATH  $MPI_ROOT/man:$MANPATH
else
     setenv MANPATH  $MPI_ROOT/man
endif
```

The number of processes/processors the program will run with is specified as part of the mpirun command with the –np switch (you will instead probably use the pbsmpich command to run the MPI job with the Linux cluster scheduler - see the cluster documentation for more details).

Software Engineering Study

The instructions for setting up your accounts for the high performance computing software engineering study discussed in class are here .

Templates, sample code, makefiles, etc. are available here .

ADDITIONAL RESOURCES

For additional OpenMP information, see http://www.openmp.org (OpenMP API specification, and a lot more). For more on the Sun OpenMP implementation, see http://docs.sun.com/app/docs/doc/819-0501 .

For additional MPI information, see http://www.mpi-forum.org (MPI API) and http://www-unix.mcs.anl.gov/mpi (for MPICH).

For more information about using the Maryland cluster PBS scheduler, MPI, etc., see http://www.umiacs.umd.edu/research/parallel/classguide.htm . The scheduler has recently been updated with a new frontend with the same set of commands as PBS, so you need to add the path to the new scheduler commands at the front of your path:

```
set path = (/opt/UMtorque/bin $path)
```

Last updated Saturday, 15 October 2005 10:53 PM